HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

2016

Automatisierte Unterstützung der Studierenden beim Lösen von SQL-Aufgaben durch Baumvergleich und Integration in ein Werkzeug zur automatisierten Bewertung

Masterarbeit

Erken Bollenhagen

02.12.2016

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

.....

(Ort, Datum und Unterschrift)

Autor	Erstprüfer	Zweitprüfer
Erken Bollenhagen	**Prof. Dr. Carsten Kleiner**	**Prof. Dr. Felix Heine**
Hochschule Hannover Fakultät IV Abteilung Informatik Ricklinger Stadtweg 120 30459 Hannover	Hochschule Hannover Fakultät IV Abteilung Informatik Ricklinger Stadtweg 120 30459 Hannover	Hochschule Hannover Fakultät IV Abteilung Informatik Ricklinger Stadtweg 120 30459 Hannover

INHALTSVERZEICHNIS

1 EINLEITUNG

Diese Arbeit wurde als Abschlussarbeit für den Masterstudienganges der Angewandten Informatik an der Hochschule Hannover realisiert. Sie umfasst die Analyse, Konzeption und Implementierung einer Weiterentwicklung des bestehenden automatisierten SQL-Bewertungssystem aSQLg (Kurzform für „automated SQL grader“) für Studenten der Hochschule Hannover.

1.1 MOTIVATION

Für den Studiengang der angewandten Informatik an der Hochschule Hannover existiert für verschiedene Lehrveranstaltungen, in denen SQL-Kenntnisse vermittelt werden, das System aSQLg. Es dient der Vereinfachung der Abgaben und Bewertungen von studentischen Übungsaufgaben für die Datenbankabfragesprache SQL von Orade. Im System aSQLg werden die Abgaben automatisch auf Korrektheit überprüft und im einheitlichem Maß mit Punkten bewertet (diese Funktion wird im Folgenden als Grader bezeichnet), die zu einem reduzierten zeitlichen Aufwand gegenüber manueller Überprüfung und fairer Bewertung beitragen. Aktuell erhält der Student als Rückmeldung nur die Bewertung zu seiner Abgabe in Form von erreichten Punkten. Es gibt jedoch es noch keine Rückmeldung zu den gemachten Fehlern und Empfehlungen zur Herleitung der korrekten Lösung. In diesem Punkt liegt die Stärke einer manuellen Überprüfung und Bewertung durch Professoren, Dozenten sowie Tutoren (im weiteren Text als Instruktoren zusammengefasst).

Die Hauptaufgabe dieser Masterarbeit ist daher die Weiterentwicklung von aSQLg, um dadurch im Fall von inkorrekten studentischen Abgaben rechnergestützt hilfreiche Hinweise zu generieren, anhand derer die Studenten dann selbständig die Lösung finden sollten. Als Ziel der Erweiterung ist der Einsatz im Tutorensystem für die Studierenden vorgesehen, insbesondere zum selbstorganisierten Überprüfen der korrekten Formulierung der SQL-Statements. Es erfolgt eine Analyse und Evaluierung der Graphentechnologie GReQL für einen möglichen Einsatz in dieser Masterarbeit. Es folgt eine Analyse, ob Baum- oder Graphvergleiche sich besser eignen zur Untersuchung der Unterschiede des studentischen SQL-Statements mit der Musterlösung zur Generierung von Lösungshinweisen.

Diese Masterarbeit geht in Richtung Forschungsarbeit und schließt den Aspekt Erforschung und Erprobung mit Hilfe eines Prototyps ein, um herauszufinden, welche der Ansätze und Ideen geeignet und welche weniger geeignet sind. Ergänzt wird die Masterarbeit durch ein separates Handbuch, welches sich an potentielle zukünftige Entwickler richtet zur Weiterentwicklung des Systems. In diesem werden technische Details und Empfehlungen aufgelistet, die über den Inhalt der Masterarbeit hinausgehen.

1.2 AUFGABENSTELLUNG

Die genaue Ausschreibung für diese Masterarbeit lautet:

„Statische Prüfung der Korrektheit von SQL-Aufgaben basierend auf Baumvergleichen (intern): In dieser Arbeit soll eine mögliche Erweiterung für das Werkzeug aSQLg zur automatisierten Prüfung von SQL-Statements entworfen und realisiert werden. Die Idee dabei ist, sowohl das Statement der Musterlösung als auch das Statement des Studierenden in eine Baum-/Graph-Repräsentation zu übersetzen und dann mithilfe von Ähnlichkeiten dieser Bäume/Graphen sowohl die Korrektheit der Lösung wie auch ggfs. Hinweise zur Verbesserung bei nicht korrekten Lösungen zu erzeugen. Eine solche Ähnlichkeitsprüfung von Graphen kann möglicherweise mithilfe der existierenden Graph-Anfragesprache GReQL erfolgen oder auch auf andere Art und Weise. In der Arbeit sollen verschiedene Möglichkeiten solcher Repräsentationen und der zugehörigen Ähnlichkeitsoperationen verglichen werden. Dabei soll sowohl die Möglichkeit der Korrektheitsprüfung wie auch die Möglichkeiten zur Unterstützung der Studierenden bei der Lösungsfindung berücksichtigt werden. Die beste dieser Varianten soll dann implementiert und in das Werkzeug aSQLg integriert werden. “

1.3 AUFBAU DER ARBEIT

Die Masterarbeit ist in folgende Arbeitsschritte unterteilt:

Ausgangssituation	Es erfolgt die Beschreibung des aktuellen Standes vom System aSQLg mit seinen Komponenten und der Prüfung hinsichtlich der Projekteignung.
Evaluierung verwandter Arbeiten	Es werden thematisch ähnlichen Veröffentlichungen und Technologien vorgestellt, darunter GReQL. Anschließend wird die Eignung für die Ähnlichkeitsüberprüfung über Graph-/Baumvergleiche mit möglichen Algorithmen evaluiert.
Konzept	Das Konzept umfasst die Konzeption der notwendigen Änderungen am übernommenen System als Erweiterung, sowie der Beschreibung des Lösungsansatzes zur Generierung der Lösungsempfehlungen. Anschließend erfolgt eine erste Einschätzung des Konzeptes.
Implementierung	Der Abschnitt Implementierung beschreibt die Details zur Umsetzung vom Konzept und ergänzt die Besonderheiten der jeweiligen Komponenten.
Tests	Die Testergebnisse werden als Auszug vorgestellt, um einen Eindruck der Stärken und Schwächen bezüglich der Findung der verschiedenen Fehler, Qualität der Lösungsempfehlungen und der Laufzeit zu erhalten.
Bewertung	In der Bewertung wird eine kritische Evaluierung durchgeführt. Dies erfolgt anhand des Konzepts, der Implementierung und daraus resultierenden Tests.
Weiterentwicklung	In diesem Abschnitt werden neue Ideen und Ansätze sowie Verbesserungsvorschläge für eine potentielle zukünftige Weiterentwicklung vorgestellt.
Zusammenfassung	Zum Abschluss wird die Umsetzung dieser Arbeit anhand der Ergebnisse zusammengefasst, bewertet und ein Ausblick für mögliche weiterführende Arbeiten abgegeben.

2 AUSGANGSSITUATION

Wie eingangs dargestellt, stellt diese Masterarbeit eine Weiterentwicklung eines existierenden Projektes dar. Daher wird zunächst die Architektur mit den wichtigsten Komponenten beim Stand der Übernahme beschrieben. Die Zusammenfassung der wichtigsten Elemente basiert auf dem Hochschul-internen Projekt-Wiki [1] und der Analyse des Programmquelltexts.

2.1 ÜBERSICHT VON ASQLG

Die derzeitige Architektur ist umfangreich und komplex, daher gibt die nachfolgende Abbildung mit einem Ausschnitt der Hierarchie der Komponenten einen Überblick über die Struktur.

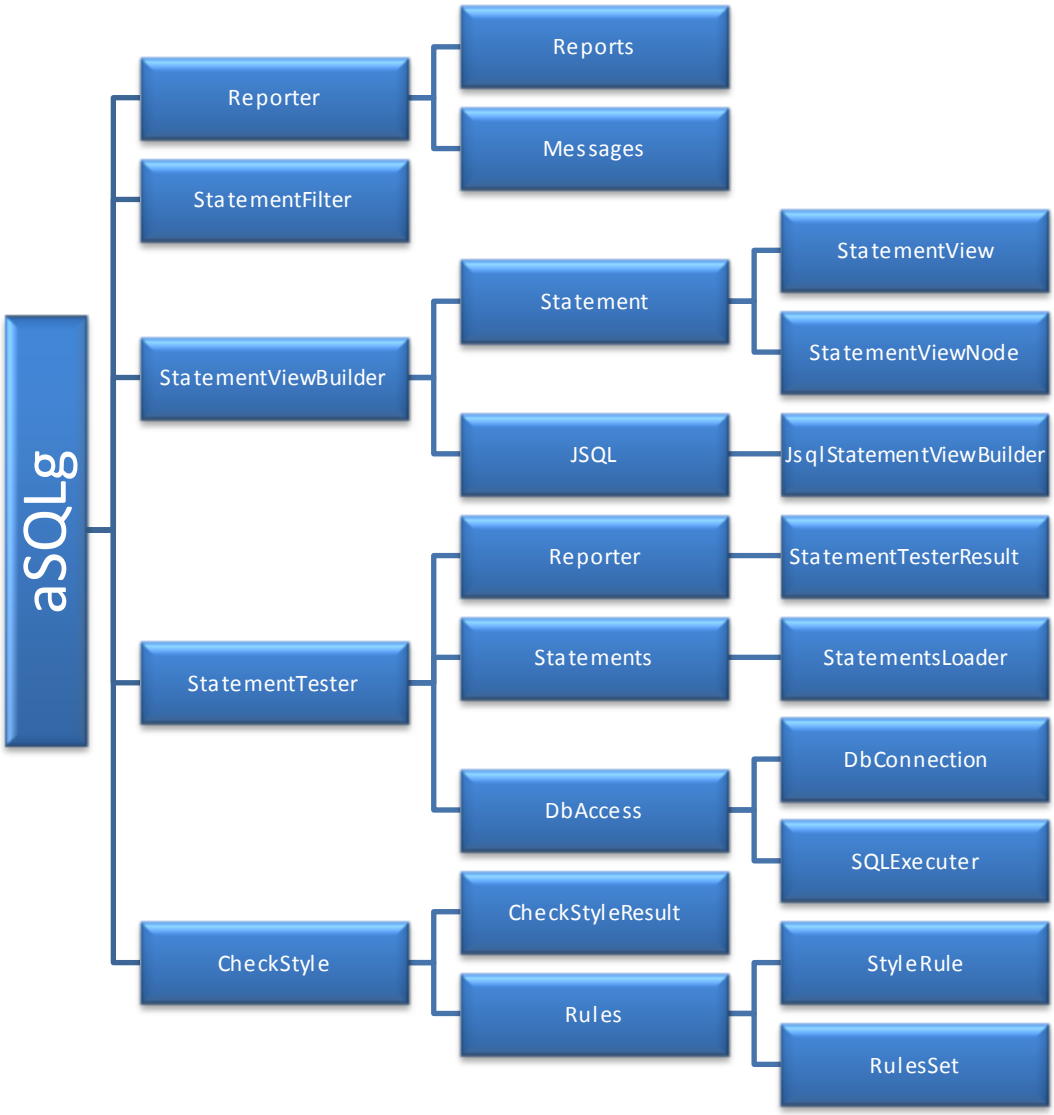

Abbildung 1: Ausschnitt der aSQLg-Komponenten (Horizontale Hierarchie von links nach rechts, beginnend mit aSQLg)

Die Details der vollständigen Architektur können dem separaten hochschulinternen Klassendiagramm [2] entnommen werden. Anschließend werden die wichtigsten Komponenten erklärt.

2.2 KOMPONENTEN VON ASQLG

2.2.1 ASQLG-ENGINE: DAS KERNMODUL VON ASQLG

Die Klasse *ASQLg-Engine* ist das Kernmodul von aSQLg und kontrolliert die Steuerung des Programmablaufs mit allen vorbereitenden und nachbereitenden Arbeitsschritten.

Es übernimmt dabei nacheinander folgende Aufgaben [3]:

1. Entgegennahme der Parameter für die Konfiguration und die Aufgabenstellung
2. Erzeugung des *StatementTester* zum Laden des Statements mit den gewählten Einstellungen und Dateien
3. Erstellung eines *Reporter* für die Erstellung von Berichten
4. Filterung der Statements mit dem *StatementFilter*
5. Testen der Statements mit dem *StatementTester*
6. Berechnung der Punkte vom Ergebnis
7. Dokumentation der Ergebnisse in einer Übergabedatei

2.2.2 STATEMENT-FILTER: SCHUTZ GEGEN BETRUGSVERSUCHE

Das System bietet zum Schutz gegen Betrugsversuche die Möglichkeit, für jede Aufgabe einen Statement-Filter in Form von Dateien zu nutzen, die wahlweise als White- oder Blacklist fungieren und die erlaubten bzw. verbotene Statementarten, Datenbanken, Tabellen, Spalten und Funktionen enthalten. Nicht erlaubte Elemente werden dadurch ausgefiltert, die bei der Bewertung dann keine Punkte geben.

2.2.3 STATEMENTTESTER: BEWERTUNG DES SQL-STATEMENT MIT PRÜFSCHRITTEN

Die studentische Abgabe wird mit dem *StatementTester* [4] zu Beginn mit der Musterlösung des Instruktors verglichen. Bei einer exakten Übereinstimmung wird die volle Punktzahl vergeben und die Überprüfung beendet. Bei einer Abweichung werden weitere Prüfschritte zur Bewertung des SQL-Statements unternommen, in der dann für die jeweiligen folgenden Bereiche die Teilpunkte für das Gesamtergebnis verteilt werden:

2.2.3.1 SYNTAXPRÜFUNG

Bei der Syntaxüberprüfung wird die syntaktische Korrektheit der SQL-Anfrage überprüft. Nur für ein korrekt ausführbares Statement gibt es die volle Punktzahl, ansonsten keine.

2.2.3.2 KOSTENPRÜFUNG

Bei der Kostenprüfung werden die benötigten Ressourcen für die Ausführung der Anfrage berechnet und mit den Anfragekosten der Musterlösung verglichen. Der Instruktor kann Maximalwerte festlegen, die als absoluter Wert oder als relativen Faktor zur Musterlösung angegeben werden. So kann zum Beispiel als Toleranzgrenze die 1,5 fachen Kosten der Musterlösung als Maximalwert ausgewählt werden. Bei der Bewertung gibt es für die Übereinstimmung der Anfragekosten die volle Punktzahl. Liegen die Anfragekosten im Toleranzbereich, sind also größer als in der Musterlösung, aber noch unter dem Maximalwert, dann erfolgt eine Verhältnisberechnung. Bei Überschreiten des Maximalwertes werden keine Punkte vergeben.

2.2.3.3 ERGEBNISPRÜFUNG

Bei der Ergebnisprüfung erfolgt eine Anfrage zur Datenbank für die Ausführung des SQL-Statements des Studenten und der Musterlösung. Die erhaltenen Ergebnisse werden dabei miteinander auf Abweichungen verglichen, darunter die Anzahl der Spalten, Spaltennamen, Dateitypen und Anzahl der Ergebniszeilen.

Anschließend werden die Ergebnisse miteinander verglichen, ob sie das gleiche Resultat liefern. Dafür werden die Statements des Studenten und des Instruktors auf der Datenbank ausgeführt und im ersten Teil der Abfrage über UNION zusammengelegt. Im zweiten Teil wird die Schnittmenge beider Statements durch INTERSECT gebildet, die mit der MINUS-Operation von SQL dem ersten Teil abgezogen wird. Übrig bleibende Zeilen stellen somit eine vorhandene Abweichung der Ergebniszeilen dar. Es werden nur Punkte für exakte Übereinstimmung vergeben.

```
String checkStatement =
    "(SELECT x.* FROM (" + stud + ") x UNION SELECT x.* FROM (" + instr + ") x" +
    ") MINUS (" +
    "SELECT x.* FROM (" + stud + ") x INTERSECT SELECT x.* FROM (" + instr + ") x)";
```

Abbildung 2: Test auf gleiches Ergebnis durch Vergleich der Ergebniszeilen auf bleibende Zeilen

2.2.3.4 BEWERTUNG DES STILS

Um den Stil zu bewerten wird die Lesbarkeit und Verständlichkeit vom SQL-Statement überprüft. Hier prüft das Modul *CheckStyle* mit Hilfe von Dateien für Regeln (*RuleSet*) die SQL-Statements auf guten Stil und sammelt für die Bewertung die Informationen über die verletzten bzw. erfüllten Regeln. Die Bewertungsskala reicht von 0 bis 100%, analog dem Prozentsatz der erfolgten Regeln entspricht.

2.2.3.5 AUSFÜHRUNG DER PRÜFUNG

Um die Funktion von Orade zur Überprüfung zu nutzen, wird eine Verbindung zur Datenbank mit Hilfe der Klasse *DbConnection* benötigt. Man erhält dadurch die Möglichkeit des Syntax- und Kosten-Check sowie der Ergebnisprüfung.

Alternativ lässt sich laut der Projektwiki-Beschreibung [4] in ASQLg auch per ZQL-Parser die Syntax überprüfen, wobei in diesem Fall keine Datenbankverbindung benötigt wird.

2.2.3.6 ZQL-PARSER

ZQL [5] ist ein SQL-Parser für Java, der auf dem *JavaCC* (einem Java Parser Generator [6]) basiert. ZQL parst SQL-Statements und erstellt die Java-Strukturen zur Representation der Abfragen und kann die jeweiligen Teile der Abfrage extrahieren, z.B. SELECT über die Funktion *getSelect*, FROM über *getFrom*. ZQL enthält einen SQL-Ausdrucks-Bewerter „SQL-Expression-Evaluator“, mit dem man einfach SQL-Ausdrücke im Parser bewerten kann. Der Funktionsumfang vom ZQL-Parser wird anhand der Beispiele aus dem SQL-Tutorial von James Hoffman [7] wiedergegeben.

2.2.3.7 ABLAUF DER PRÜFUNG

Der Ablauf von aSQLg ist detailliert im hochschulinternen aSQLg-Ablaufdiagramm [8] dargestellt und lässt sich wie folgt zusammenfassen:

1. Die studentische Abgabe wird eingereicht.
2. Die Konfiguration von aSQLg wird geladen.
3. Die Statements werden in gewählter Konfiguration und Bewertungspunkten geladen.
4. Im StatementTester erfolgt für jedes Statement die Überprüfung mit den Prüfschritten.
    - Durch die Nutzung des StatementFilters werden zuerst die nicht erlaubten Elemente herausgefiltert.
    - Falls das studentische Statement mit dem des Instruktors übereinstimmt, gibt es die volle Punktzahl. Ansonsten werden die weiteren Prüfschritte durchgeführt und eine Datenbankverbindung dafür aufgebaut.
    - Falls beim Syntaxcheck das SQL-Statement fehlerhaft ist, erhält der Benutzer keine Punkte und es geht weiter zum *StyleCheck*. Ansonsten erhält er die volle Punktzahl und es folgen weitere Überprüfungen.
    - Bei der Kostenprüfung wird bei Übereinstimmung der Anfragekosten des Studenten mit denen des Instruktors die volle Punktzahl vergeben. Bei Überschreitung werden bis zum maximal gesetzten Limit Teilpunkte vergeben. Wird dieses Limit der Kostenprüfung gar überschritten, werden keine Punkte vergeben und die nachfolgende Ergebnisprüfung wird nicht durchgeführt. Dies dient dem Schutz gegen eine mögliche Überlastung der Datenbank bei ineffizienten Anfragen.
    - Als letzte Überprüfung werden im *StyleCheck* die gewählten Regeln überprüft und der prozentuale Anteil der erfüllten Regeln als Punkte vergeben.
5. Nach der Überprüfung erfolgt die Berechnung der Punkte und die Erstellung des Reports.

2.2.4 REPORTER

Der *Reporter* dient zur Sammlung der Informationen, z.B. für den Grader bei der Auswertung der Prüfschritte. Er nutzt diese Daten dann für spätere Ausgaben und Berichte in wählbaren Formaten. Verwendet werden die Nachrichten durch die vorhandene *Messages*-Klasse“. Nachrichten können z.B. Fehler, Warnungen oder Infos sein, die während der Überprüfung auftreten.

2.2.5 KLASSEN FÜR DIE BAUMSTRUKTUR DES SQL-STATEMENTS

Die Objektsicht auf ein SQL-Statement in Form einer Baumstruktur erfolgt durch die Klasse *StatementView*, die die Knoten des Statementbaumes durch die Klasse *StatementViewNode*. Das Interface *StatementViewBuilder* liest das SQL-Statement ein, parst dieses und erzeugt daraus das passende *StatementView*. Dies nutzt jeweils den passenden *StatementViewGenerator* für die verschiedenen Statement-Typen (z.B. SELECT, INSERT oder UPDATE), in der die jeweiligen Knoten (*StatementViewNode*) durch den passenden *StatementViewNodeGenerator* erzeugt werden.

Die folgende Tabelle stellt eine Übersicht der zugehörigen Klassen für die Baumstruktur zu einem SQL-Statement dar:

Klasse	Funktion / Aufgabe
StatementView	SQL-Statement in Form einer Baumstruktur (Statementbaum).
StatementViewNode	Knoten des Statementbaums.
StatementViewBuilder	Das Interface *StatementViewBuilder* dient zum Einlesen und Parsen eines SQL-Statements und dem Erzeugen des entsprechenden *StatementView*.
StatementViewGenerator	Das Interface *StatementViewGenerator* wird vom *StatementViewBuilder* genutzt für die jeweiligen verschiedenen Statement-Typen zur Erstellung des passenden Statementbaums, zum Beispiel *InsertViewGenerator* bei einem INSERT-Statement.
StatementViewNodeGenerator	Zur Erstellung des passenden Knoten der Klasse *StatementViewNode* wird das Interface *StatementViewNodeGenerator* genutzt, zum Beispiel die Implementierung des *ExpressionVisitor* für einen Knoten mit verschiedenen Ausdrücken wie die Arithmetische Funktionen oder Datentypen.

2.2.6 JSQL

Die *StatementViewBuilder*-Implementierung *JsqlStatementViewBuilder* nutzt den JSQL-Parser [9] (dieser basiert auf dem *JavaCC*) und erzeugt einen Statementbaum [10] in Form der Klasse *StatementView* aus einem Oracle SQL-Statement, das hierarchisch in die passenden Java-Klassen übersetzt.

Die Hierarchie wird über die SQL-Schlüsselwörter unter Nutzung des Visitor-Pattern [11] angewendet, welches für jede der jeweilig möglichen Parameter-Ausprägungen (Statement-Typen: z.B. SELECT, INSERT oder UPDATE) eine eigene Methode *visit* mit den spezifischen Details nutzt, die der jeweilige Teil des SQL-Statement besitzen kann.

```
void visit(Insert statement) {
    //Generierung Knoten eines Insert Statements
}
```

```
void visit(Update statement) {
    //Generierung Knoten eines Update Statements
}
```

Der JSQL-Parser selbst besitzt den Nachteil, dass die Position der Knoten im SQL-Statement nicht mit einbezogen wird. Als Lösung wird im *JsqlStatementViewBuilder* zum Baum vom JSQL-Parser ein zweiter Baum über den *NodePositionCalculator* für die Position erzeugt und über ein Merge der finale Baum generiert, in der jeder Knoten syntaktisch und semantisch eingeordnet ist. Auf dem ersten Blick klingt das möglicherweise nicht elegant, dies funktioniert jedoch bei der Analyse ohne erkennbare Probleme. Aus diesem Blickwinkel betrachtet stellt der *JsqlStatementViewBuilder* eine Art Wrapper für den JSQL-Parser dar, der mit Implementierungsdetails und Korrektur der fehlenden Position an die Bedürfnisse des Projekts angepasst ist.

Im Verlauf der Arbeit wird der Begriff JSQL und JSQL-Parser als ein Synonym für den JSQL-Parser zusammen mit dem *JsqlStatementViewBuilder* als eine gemeinsame Einheit genannt.

2.3 BEWERTUNG VOM PROJEKTSTAND

2.3.1 ALLGEMEIN

Bei der Bewertung des aktuellen Projektstandes soll herausgefunden werden, ob es die geeignete Grundlage bietet, auf der die Masterarbeit aufbauen kann. Daher steht insbesondere JSQL im Vordergrund, da es mit den Klassen der Bäume und Knoten als Basis für Baum- bzw. mit Anpassung für Graphvergleiche verwendet werden könnte. Weniger relevant ist die Untersuchung und Darstellung der Funktion als Grader, sie wird deshalb nicht weiter detailliert analysiert. Bei einer ersten Betrachtung ergibt das Projekt eine gute und ausbaufähige Grundlage, die in Verbindung mit den eigenen Konzeptideen als Erweiterung erfolgversprechend kombiniert werden könnte. Besonders der JSQL-Abschnitt vermittelt den Eindruck einer hohen Kompatibilität mit der Fragestellung der Masterarbeit. Die Richtigkeit dieser ersten Einschätzung zeigt die spätere Auswertung des Konzepts und der Implementierung.

2.3.2 UNTERSCHIEDE ZWISCHEN PARSER DER DATENBANKSPRACHE SQL

Der Umgang mit verschiedenen SQL-Parsern, wie dem JSQL-Parser, ZQL-Parser und der Oracle SQL-Datenbank, beinhaltet oft durch mögliche abweichende Interpretationsunterschiede in der Überprüfung der SQL-Syntax ein Potential für Fehler und Probleme. Dies sollte in der weiteren Bearbeitung kritisch begleitet werden. Die SQL-Datenbank von Oracle stellt die native SQL-Syntax dar und ist die Referenz. Der Umfang der SQL-Syntax des ZQL-Parser ist übereinstimmend mit der aus dem zugehörigen ZQL-Tutorial [12], was nur einen Ausschnitt der Syntax bedeutet. Der JSQL-Parser arbeitet ohne ein vorhandenes Datenbankschema und überprüft das Statement vor der Generierung des Baumes nur auf die allgemeine Einhaltung der Struktur der SQL-Syntax. Dies kann Vor- und Nachteil zugleich sein, denn Fehler durch eine Abweichung vom Schema werden zwar nicht durch JSQL gefunden, könnten aber durch Grader erkannt werden. Als Vorteil von JSQL kann die Flexibilität und mögliche Offline-Überprüfung gesehen werden, die ohne Zugriff auf die Datenbank oder Speicherung des Schemas durchgeführt werden kann.

2.3.3 HILFESTELLUNG BEI SYNTAXFEHLER

Auf ungültige SQL-Statements mit Syntaxfehler (z.B. falsche Position, Fehlen oder falsche Schreibweise eines Schlüsselwortes) sollte erwartungsgemäß mit der passenden Fehlermeldung reagiert werden. Laut der Beschreibung im ProjektWiki [4] wird im ZQL-Parser beim ersten auftretenden Syntaxfehler dieser aufgezeigt und eine 100% Fehlerrate für die Bewertung zugewiesen. Bei der Nutzung der SQL-Datenbank von Oracle zur Überprüfung kann die datenbankinterne Fehlermeldung verwendet werden, die hier natürlich die native SQL-Syntax verwendet. Beim JSQL-Parser wird bei einem entdeckten Syntaxfehler zwar kein Baum generiert, dafür allerdings eine eigene Fehlermeldung. In Tests hat sich jedoch gezeigt, dass bei speziellen Problemfällen weder ein Baum noch eiine Fehlermeldung erzeugt werden. Dies geschieht durch teilweise fehlerhafte bzw. unvollständige Abbildung der SQL-Syntax, bei der JSQL mit einer *NullPointerException* abbricht. Die in der Untersuchung gefundenen JSQL-Fehler werden im Abschnitt „Fehler in JSQL“ genauer beschrieben und detailliert im Handbuch dokumentiert, die daraus resultierende Korrektur von JSQL ist allerdings nicht Bestandteil dieser Masterarbeit.

3 EVALUIERUNG VERWANDTER ARBEITEN

In diesem Kapitel werden verwandte Arbeiten vorgestellt, in denen potentielle einzusetzende Technologien und Algorithmen untersucht werden. Nachfolgend wird das SQL Lernsysteme *ÜPS* vorgestellt und in ausgewählten Aspekten, die für die Masterarbeit relevant sind, kritisch betrachtet. Das Problemfeld der Semantikfehler wird anschließend detailliert erläutert und die Graph-Anfragesprache GReQL mit ihren Komponenten vorgestellt und bewertet. Abschließend werden die Unterschiede der Datenstrukturen Baum und Graph und vorhandene Algorithmen für die Graph-/ Baumvergleiche auf ihren Einsatz untersucht.

3.1 „ÜPS – EIN AUTORENFREUNDLICHES TRAININGSSYSTEM FÜR SQL-ANFRAGEN“

Die Publikation „ÜPS – Ein autorenfreundliches Trainingssystem für SQL-Anfragen“ [13] von Marianus Ifland, Michael Jedich, Christian Schneider und Frank Puppe von der Universität Würzburg (2014) beschreibt kurz zusammengefasst ihr eigenes SQL-Trainingssystem, das Überschneidungen mit der Aufgabenstellung der Masterarbeit vorweist.

Das beschriebene Trainingssystem besteht aus freiwilligen Übungs- sowie Pflichtaufgaben. Der Lernerfolg wurde über ein Semester vor allem anhand des Auftretens semantischer Fehler im Pflichtteil ermittelt. Ein großer Teil der Evaluation betrachtet die pädagogische Seite, wie die Studenten das Trainingssystem annahm und welche Verbesserungen der studentischen Lernerfolge damit erzielt wurden. Wissenschaftlich sind die Ergebnisse kritisch zu sehen, weil es keine parallele Vergleichsgruppe gab, in der der Lernerfolg ohne Teilnahme am *ÜPS* erhoben wurde.

Ein interessanter Teil für die Masterarbeit ist der Abschnitt zum Tutorsystem mit dem Konzept eines 3-Ebenen-Feedbacks. Die erste Ebene ist die Fehlermeldung der Datenbank bei einer syntaktischen, nicht validen Anfrage, also einem Parse-Error. Die zweite bei syntaktisch valider Anfrage ist die Ermittlung der ähnlichsten Musterlösung (Tutoren können mehrere Musterlösungen eintragen) über die *Levenshtein-Distanz*. Anschließend wird für die studentische Anfrage und der ausgewählten Musterlösung ein Parsebaum erstellt, in der die Unterschiede als Fehlermeldungen angezeigt werden. In der dritten Ebene werden die Ergebnisse ausgeführt und miteinander verglichen, wobei bei identischem Inhalt die Aufgabe als gelöst gilt. Zusätzlich können sich hier die Studierenden die Ergebnis-Tabellen der ausgeführten Anfragen anschauen und miteinander vergleichen.

Die Idee mehrere Musterlösungen zu verwenden hat den Vorteil einer gewissen Flexibilität. Es bedeutet jedoch Mehrarbeit für den Instruktor sich über bei jeder Aufgabe viele alternative Lösungswege Gedanken zu machen. Zudem entfällt die Flexibilität bei Mischung aus zwei mehr Kombinationen der Musterlösungen, da es bei der ausgewählten Musterlösung dieses strenger mit dem Parsebaum vergleicht.

Da der Schwerpunkt bei dieser Arbeit auf dem lernenden Trainieren einer korrekten SQL Anfrage liegt, können sich daraus Anregungen zur Lösungsfindung für die Masterarbeit ergeben und entdeckte Parallelen als Bestätigung der verfolgten Richtung bewertet werden.

3.2 „Semantic Errors in SQL Queries: A Quite Complete List“

Die schon 10 Jahre alte, aber immer noch gültige Publikation „Semantic errors in SQL queries: A quite complete list“ [14] von Stefan Brass und Christian Goldberg behandelt das Thema von Semantikfehlern in SQL-Statements, die aus der Auswertung von studentischen Zwischentests und Übungen entstammen. Während Syntaxfehler ungültige SQL-Statements darstellen, sind Semantikfehler zwar syntaktisch gültige SQL-Statements, aber es wird nicht das erwartete Ergebnis (Effektivität) oder auf die gewünschte Art (Effizienz) berechnet.

Bei der Erstellung der SQL-Statements durch die Studenten werden häufig durch Unkenntnis der Musterlösung beim Ausprobieren unnötige Elemente in das Statement aufgenommen und abschließend nicht wieder entfernt, obwohl diese nicht mehr gebraucht werden, weder in der Ausgabe, noch zur Beschaffung, Filterung oder Weiterverarbeitung der Daten. Diese Statements führen neben einer Verschlechterung der Lesbarkeit auch zu längeren Laufzeiten, da der SQL-Optimizer nicht alles Unnötige automatisch entfernt. Teilweise führen diese semantischen Fehler auch zur Berechnung eines falschen Ergebnisses. Es gibt aber auch Beispiele, wo der Einsatz der persönlichen Präferenz unterliegt, wie zum Beispiel die Nutzung von expliziten Tupel-Variablen für die genutzte Tabelle, obwohl der Name der Spalte einmalig ist, sowie der Einsatz vom optionalen Schlüsselwort AS. Ein komplett unterschiedliches, kürzeres und besser lesbares Statement entsteht durch das Entfernen nicht.

Die Autoren Brass und Goldberg teilen die Semantikfehler in verschiedene Kategorien ein:

- Unnötige Verkomplizierung
- Ineffiziente Formulierung
- Verletzung von Standard-Patterns
- Viele Duplikate
- Mögliche Laufzeitfehler
- Stilüberprüfungen

3.2.1 Unnötige Elemente und Verkomplizierung

Falsche Ergebnisse

- Überflüssige Spalten und Tabellen und Vertauschen der Reihenfolge in Vergleichen bzw. JOIN können die Logik der Anfrage verfälschen und damit zum falschen Ergebnis führen. Es können dadurch sogar inkonsistente Bedingungen wie `X=1 AND X=2` entstehen, welche zum leerem Ergebnis führen.

Unnötige Elemente: SELECT

- Die Auflistung von Spalten im SELECT-Abschnitt, die einen konstanten Wert in allen Zeilen beinhalten, z.B. durch Nutzung in der WHERE-Bedingung zur Filterung nach einem speziellen Wert ist unnötig mit Ausnahme der Fälle, wo der Wert für die weitere Verarbeitung genutzt wird.
- Die Angabe DISTINCT sollte nur erfolgen, wenn es notwendig ist. Bei der Nutzung von Schlüsseln kann es normal keine Duplikate geben.
- Duplikate von identischen Spalten sind ebenfalls unnötig.

Unnötige Elemente: FROM

- Wenn nur der Schlüssel des Fremdschlüssels selbst von einer Tabelle benötigt wird, so ist kein JOIN notwendig.

Unnötige Elemente: WHERE

- Unnötig kompliziert sind implizite, wiederholte oder inkonsistente Unterbedingungen, wenn sie durch wahr/falsch ausgetauscht werden könnten und noch äquivalent bleiben. Es kommt vor, dass getestete Bedingungen im WHERE-Abschnitt eigentlich Bedingungen auf die Relation (CONSTRAINT) sind, die besser im JOIN platziert wären.
- Ein Vergleich auf NULL (A=NULL) sollte vermieden werden, da es die Werte NULL und Unbekannt haben kann. In manchen Systemen kann das zum Syntaxfehler führen.
- Unnötige Verallgemeinerungen der Vergleichsoperatoren, genannt sei hier die Verwendung von „>=“ an einer Stelle, an der „=“ mehr Sinn macht, wie zum Beispiel im Ausschnitt „WHERE SAL >= (SELECT MAX(SAL) …“.
- Ein LIKE, das keine Wildcards („%“ / „_“) enthält, würde besser mit einem Vergleich auf Gleichheit („=“) überprüft werden.
- In EXISTS-Unterabfragen sind die SELECT-Listen nicht so wichtig, daher sollten sie einfach gehalten werden, wie durch z.B. „*“, „1“ oder ein einzelnes Attribut.
- IN/EXIST-Bedingungen aus Unterabfragen können teilweise auch durch Vergleiche im Hauptteil ausgetauscht werden.

Aggregations-Funktionen

- Bei Aggregations-Funktionen wie MIN und MAX ist ein DISTINCT überflüssig.
- COUNT(*), also die Berechnung der Anzahl ohne Argument, wird gegenüber einem Argument bevorzugt, wenn es äquivalent ist (also ohne DINSTINCT) und nicht NULL sein kann.

GROUP BY

- Ein GROUP BY ist unnötig, wenn gezeigt werden kann, dass jede Gruppe nur aus einer Zeile besteht oder es nur eine einzige Gruppe insgesamt gibt.
- Ebenso unnötig ist das GROUP BY, wenn das genutzte Attribut nicht im SELECT oder HAVING genutzt wird.
- Ein GROUP BY kann durch DINSTINCT ausgetauscht werden, wenn genau die SELECT-Attribute im GROUP BY aufgelistet werden und darin keine Aggregations-Funktionen genutzt werden. Es ist damit kürzer und besser zu lesen.

HAVING, UNION (ALL), ORDER BY

- Für HAVING gilt das gleiche wie für WHERE. Bedingungen, die unter WHERE möglich sind, sollten dort auch besser platziert werden.
- UNION ALL kann durch ein einzelnes Statement mit OR im WHERE-Abschnitt ausgetauscht werden, wenn die gleiche Liste bei SELECT und FROM sowie beidseitig exklusive WHERE-Bedingung vorliegt. Ähnliche Bedingungen gelten für UNION.
- ORDER BY ist unnötig, wenn gezeigt werden kann, dass das genutzte Attribut nur einen möglichen Wert haben kann

3.2.2 Ineffiziente Formulierung

Die Datenbanksprache SQL ist zwar eine deklarative Sprache, in der die Beschreibung des Ziels im Vordergrund steht und nicht der Weg der Berechnung. Die Nutzer sollten jedoch dem System bei der Ausführung helfen, damit der SQL-Optimizer diese möglichst zu einer kosteneffizienten Anfrage optimieren kann. Das Ziel ist Laufzeit möglichst gering zu halten und dabei so wenig Speicher wie möglich zu benötigen.

- Ineffizientes HAVING: Eine Bedingung, die nur GROUP BY Attribute und keine Aggregationsfunktion nutzt, kann unter WHERE oder HAVING erfolgen, wobei die Überprüfung unter WHERE kostengünstiger ist.
- Ineffizientes UNION: Ein UNION sollte durch UNION ALL ausgetauscht werden, wenn man zeigen kann, dass die Ergebnisse beider Statements immer disjunkt sind und keine Duplikate ergeben.

3.2.3 Verletzung von Standard-Patterns

- Fehlende JOIN-Bedingung.
- Wenn eine EXISTS-Unteranfrage keine Verbindung zu einer Variablen der äußeren Anfrage besitzt, ist sie entweder global wahr oder falsch und stellt damit ein sehr ungewöhnliches Verhalten dar. Diese unkorrelierten EXISTS-Unteranfragen sind meist fehlende JOIN-Bedingungen.
- Die SELECT-Abfrage mit einer Unteranfrage benutzt keine Variable der Unterabfrage. Ein möglicher Auslöser könnte ein Tippfehler bzw. eine Verwechslung sein.
- Eine Bedingung in einer Unterabfrage, die nur Variablen von der Hauptabfrage nutzt, ist ungewöhnlich und kann teilweise nach oben verschoben werden.
- Eine Nutzung von HAVING ohne ein GROUP BY ist ungewöhnlich und besitzt entweder nur ein Resultat oder keins.
- Ein Indikator kann ein DISTINCT bei der Nutzung der Funktionen SUM und AVG sein, da hier Duplikate sehr wahrscheinlich und gewollt sind.
- Wenn Wildcards („%“ und „_“) bei einem Vergleich „=“ enthalten sind, ist i.d.R. eigentlich das Schlüsselwort LIKE gemeint.

3.2.4 Viele Duplikate

- Für manche Aufgaben werden Duplikate im Ergebnis einer Anfrage benötigt, jedoch erschweren diese die Lesbarkeit des Ergebnisses und sind häufig ein Indikator für andere Fehler wie zum Beispiel ein fehlendes JOIN. Je nach der Entscheidung, ob Duplikate enthalten sein dürfen, sollte ein Blick auf die Spaltenelemente, Schlüssel, DISTINCT, Gruppierung und JOIN erfolgen.

3.2.5 Möglichkeit von Laufzeitfehler

Laufzeitfehler werden während der Ausführung des Statements vom Datenbankmanagementsystem entdeckt. Es kann vorkommen, dass je nach Zustand der Datenbank das Statement ausgeführt wird oder ein Laufzeitfehler auftritt.

- Falls nur ein Wert bei Unteranfragen durch z.B. „A=(SELECT…)“ als Ergebnis erwartet wird, kann bei mehr als einem Wert ein Laufzeitfehler in PL/SQL auftreten.
- Dies gilt ebenfalls bei SELECT INTO, das mehr als ein Tupel zurückliefert.
- Wenn Ergebniszeilen NULL sein können oder Aggregatfunktionen eine leere Eingabe besitzen können, so ist die Überprüfung wichtig, um einen Laufzeitfehler zu vermeiden. In Embedded-SQL wird dafür eine Indikatorvariable mit einem Integer-Wert zur Überprüfung festgelegt. In SQL repräsentiert NULL entweder ein unbekanntes Attribut oder eine ungeeignete Information. Dies stellt eine Abweichung zu Programmiersprachen wie zum Beispiel C dar, in der NULL für einen Zeigerwert genutzt wird, der auf keine Speicherposition hinweist [15].
- Laufzeitfehler treten auch durch problematische Typkonvertierungen auf, da z.B. manchmal Strings implizit in Nummern konvertiert werden.
- Probleme wie Division durch Null sind bei Datentypfunktionen nur schwer zu verhindern. Selbst Versuche wie WHERE `Y<>0 AND X/Y>2` sind unsicher, da SQL keine spezifische Evaluierungsreihenfolge garantiert, so dass man nicht von der Erfüllung der Bedingung ausgehen kann.

3.2.6 Stilüberprüfungen

Zusätzlich zu den Fehlern gibt es noch Überprüfungen des Stils, in denen folgendes empfohlen wird:

- Eine Überschattung durch Tupel mit gleichem Namen in Unterabfragen sollte vermieden werden.
- Auf Tupel des äußeren Statements sollte nicht ohne seinen Namen in der Unteranfrage zugegriffen werden, z.B. „A“ anstelle „X.A“.
- Eine IN-Unteranfrage mit Zugriff auf Variablen der äußeren Abfrage könnte eventuell besser durch eine EXISTS-Unterabfrage ersetzt werden.
- Eine Vielzahl von unnötigen Klammern kann die Lesbarkeit des Statements verschlechtern.
- Zwischen verschiedenen Datenbankmanagementsystemen ist eine gewisse Portabilität von SQL-Statements wünschenswert, wobei man einen guten Mittelweg aus kurzem Statement und Effizienz wählen sollte.

3.2.7 Zusammenfassung der Fehler

In einer verwandten Arbeit „Do you know SQL? About Semantic Errors in Database Queries“ (2008) [16] fasst Christian Goldberg die Semantikfehler zusammen und listet die 10 am häufigsten auftretenden bei der Untersuchung der studentischen Abgaben auf. Diese werden in der nachfolgenden Abbildung gezeigt. Auch wenn die Inhalte sich sehr ähnlich sind, die Nummern der Semantikfehler beider Publikationen unterscheiden sich etwas voneinander.

Rank	Ratio	Semantic Error
1.	14.5 %	Error 27: Missing join condition
2.	12.8 %	Error 37: Many duplicates
3.	10.7 %	Error 6: Unnecessary join
4.	7.5 %	Error 1: Inconsistent condition
5.	5.8 %	Error 17: Unnecessary argument of `COUNT`
6.	5.4 %	Error 8: Implied, tautological or inconsistent subcondition
7.	4.6 %	Error 2: Unnecessary `DISTINCT`
8.	4.5 %	Error 26: Inefficient `UNION`
9.	4.2 %	Error 5: Unused tuple variable
10.	3.6 %	Error 3: Constant output column

Abbildung 3: Die 10 am häufigsten auftretenden Semantikfehler von Studenten in der Auswertung

3.2.8 FAZIT

Die Auflistung bietet eine brauchbare Übersicht von unterschiedlichen Fehlermöglichkeiten, die man bei der Nutzung von SQL machen kann. Eine Integration als Anmerkung oder Leseaufgabe in den Unterrichtsübungen wird daher als sinnvoll eingestuft. Als Semantikfehler werden diejenigen Abweichungen bezeichnet, die zur Berechnung falscher Ergebnisse führen oder bei richtigem Ergebnis jedoch ungenutztes Optimierungspotential bzw. schlechten Stil haben.

Zusammengefasst: Welche Arten von Fehler gibt es?

Art des Fehler	SQL-Syntax	Fehlermöglichkeiten
Syntaxfehler	inkorrekt	• Abbruch des SQL-Parsers
Semantikfehler	korrekt	• Berechnung des falschen Ergebnis • Ungenutzte Optimierungspotential • Schlechter Stil

Hier ließe sich darüber diskutieren, was eigentlich Fehler sind und ob man diese zusammen als Fehler gleichsetzt oder lieber klarer differenzieren möchte.

Im Gegensatz zur Aufgabenstellung der Masterarbeit erfolgt in der Arbeit von Goldberg [14] [16] kein Vergleich zwischen der Studentischen Abgabe und der Musterlösung des Instruktors, sondern eine reine Analyse auf die Abgabe des Studenten. Einige Beispiele der Semantik-Fehler und Optimierungen können in der Masterarbeit durch den Vergleich mit einem gut konstruierten Referenz-Statement vom Instruktor bereits gefunden werden. Für viele der Untersuchungen auf ein einzelnes Statement wird bei der Analyse die genaue Kenntnis des SQL-Schemas zu der Abfrage benötigt, andere sind unabhängig vom Schema untersuchbar.

Da diese Publikation erst spät der Masterarbeit zur Verfügung stand und nicht ein Teil der Planung war, wird es als Ergänzung genutzt zur Definition von Fehlern, deren Erkennung und was dem Studenten als Lösungsempfehlung mitgeteilt wird. Ebenso dient es zur späteren Auswertung des Projektergebnisses, welche Fehler erkannt und welche übersehen werden und was noch weiterentwickelt werden muss. In der Masterarbeit erfolgt in den Lösungsempfehlungen keine explizite Trennung zwischen Fehler und ungenutztem Potential, wird jedoch als Thema offen gehalten, falls dies zur Verbesserung der Qualität der Lösungsempfehlung führt und somit einen Mehrwert für den Studenten besitzt.

3.3 EVALUIERUNG GREQL

In diesem Abschnitt wird die Technologie GReQL [17] auf der Basis von Veröffentlichungen der Universität Koblenz-Landau mit seinen Komponenten und Funktionen vorgestellt. Anschließend wird dazu eine Evaluierung durchgeführt, ob sie für die Ähnlichkeitsüberprüfung der SQL-Statements durch Graphvergleiche geeignet ist.

3.3.1 BESCHREIBUNG DER KOMPONENTEN

Zur nachfolgenden Evaluierung ist eine fundierte Kenntnis der Technologie von Vorteil, daher erfolgt eine Zusammenfassung der wichtigsten Informationen zu GReQL und seinen zugehörigen Komponenten.

3.3.1.1 JGRALAB UND GREQL

Die textuelle, ausdrucksbasierte Anfragesprache GReQL (Graph Repository Query Language) ist für den flexiblen Zugriff auf die Eigenschaften von TGraphen konzipiert und ein Bestandteil vom Graphenlabor.

Das Graphenlabor [18] [19] ist ein Projekt der Universität Koblenz-Landau und dient zur Verarbeitung von TGraphen. In den 90er-Jahren wurde für die Programmiersprache C++ das GraLab mit der ersten Version von GReQL in C++ erschaffen. In 2006 folgte die Java-Klassenbibliothek JGraLab für GReQL als Java-Variante, die von manchen daher gern auch GReQL2 bezeichnet wird. Hinweis: Bei der weiteren Verwendung des Begriffs GReQL bezieht es sich auf die Version für Java.

Ein eigenes Wiki für JGraLab [20] mit einer schrittweisen Erklärung ist neben dem Projektarchiv [21] verfügbar.

3.3.1.2 TGRAPHEN

TGraphen [22] [23] sind spezielle Graphen und eine Entwicklung für JGraLab. Sie bestehen wie alle Graphen aus einer Knoten- und Kantenmenge, sowie einer Inzidenzfunktion, welche jeder Kante ein Knotenpaar zuordnet.

TGraphen besitzen folgende Eigenschaften:

Eigenschaft	Erklärung
Typisiert	Jeder Knoten und jede Kante weisen einen Typ auf
Attributiert	Jeder Knoten und jede Kante kann über ein oder mehrere Attribute unterschiedlicher Ausprägung verfügen
Gerichtet	Jede Kante verbindet einen Start- mit einem Endknoten
Geordnet	Es existiert eine Ordnung auf Knoten- und Kantenmengen. Eine Ordnung im azyklischen Graphen bedeutet eine Anordnung der Knoten in eine Sequenz bzw. Reihenfolge, besonders im Kontext der topologischen Anordnung

3.3.1.3 SYNTAX VON GREQL

GReQL hat Parallelen zu SQL, die Ausdrücke sind allerdings durch die Konstruktion „FROM WITH REPORT“ („FWR“) aufgebaut anstelle von „SELECT FROM WHERE”.

Schlüsselwort	Beschreibung
FROM	Variablendeklarartion: Deklaration der relevanten Graph-Elemente, möglich ist die Auswahl zwischen Knoten, Kanten oder Variablen (String, Zahlen oder boolesche Werte)
WITH	Optionale Bedingung: Spezifiziert zusätzliche Constraints (/Bedingungen) für die deklarierten Graph-Elemente.
REPORT	Ergebnisbeschreibung: Beschreibung des Aussehen/Aufbau vom Query-Ergebnis

Die dadurch erstellbaren Anfragen sind flexibel und bieten viele Funktionen, z.B. quantifizierte Ausdrücke, reguläre Pfadausdrücke, komplexe Prädikate (mit Existenz- und/oder Allquantor), Abfragen von Grapheigenschaften (Grad, Test auf Zyklen, Knoten/Kanten-Typ), mathematische Basisfunktionen (Summe, Schnitt, Anzahl) und verschachtelte GReQL-Anfragen als Unteranfragen, sowie Aggregation von Informationen.

Mit dem folgenden Musterbeispiel wird der Aufbau einer GReQL-Anfrage gezeigt:

Zeile	Code
1	`FROM caller, callee: V{Identifier}`
2	`WITH`
3	`    caller`
4	`         --> {IsDeclaratorIn}+`
5	`         <-- {IsCompoundStatementIn}`
6	`         <-- {IsStmtIn}*`
7	`         <-- {IsFunctionNameIn}`
8	`    callee`
9	`REPORT`
10	`    caller.name as Caller,`
11	`    callee.name as Callee`
12	`END`

Als Erklärung, der eben genannten Beispielanfrage, lassen sich die Zeilen wie folgt beschreiben:

Zeile	Beschreibung
1	Variablen caller + callee sind an die Menge aller Knoten vom Typ Identifier gebunden
4	Ein oder mehr ausgehende Kanten der Klasse *IsDeclaratorIn*
5	Eine eingehende Kante der Klasse *IsCompoundStatementIn*
6	Mehrere oder keine eingehende Kanten der Klasse *IsStmtIn*
7	Eine eingehende Kante der Klasse *IsFunctionNameIn*
9	Auswertung

Das folgende Beispiel übersetzt ein einfaches SQL-Statement in ein GReQL-Statement:

```
SELECT Func_Name
FROM Func_Plus
WHERE Target_Name = “CheckErrOut”
```

```
FROM callee:V{Method}, caller:V{Method}
WITH callee.name = ‘CheckErrOut’
          AND callee <-- {calls} caller
REPORT caller.name
END
```

Die nachfolgende Grafik aus der Publikation „Reverse Engineering Using Graph Queries“ [24] von Jürgen Ebert und Daniel Bildhauer (2010) gibt eine beispielhafte Übersicht der GReQL-Syntaxsprachelemente.

```
0, 123                          // integer literals
0.0, -2.1e23                    // double literals
true, false                     // boolean literals
"hugo", "ab\n"                  // string literals
v                               // variable expression
let x := 3 in x + y             // let-expression
x + y where x := 3              // where-expression
sqr(5)                          // function application
not true                        // unary operator expression
b and c, x > 0                  // binary operator expressions
v.name                          // value access
x > 5 ? 7 : 9                   // conditional expression

// quantified expressions
exists v:V{Method} @ outDegree{IsDeclarationOfInvokedMethod}=0
forall e:E{IsCalledByMethod} @ alpha(e) = omega(e)

// FWR expression
from caller, callee:V{MethodDeclaration}                //(1)
with caller <--{IsBodyOfMethod} <--{IsStatementOf}*     //(2)
            <--{IsDeclarationOfInvokedMethod} callee    //(3)
report caller, callee end
```

Abbildung 4: GReQL - Übersicht der Syntax und Sprachelemente [24]

3.3.1.4 Überprüfung der Eigenschaften in GReQL

In GReQL könnten viele Eigenschaften im Graph überprüft werden. Dazu zählen die Attributwerte von Knoten, Kanten, Strukturen im TGraphen oder aggregierte Informationen. Der besondere Fokus der GReQL-Statements liegt aber auf der Auswertung der Kanten-Eigenschaften im Graphen.

3.3.1.5 GReQL-Schemata

Zur Nutzung von JGraLab arbeitet man mit einem vorher definierten Schema für die TGraphen, [23] dem TGraph-Schema. In diese werden die Daten für die späteren Anfragen übertragen. Es können nur Knoten- und Kantentypen konform zum spezifizierten TGraph-Schema verwendet werden. Anschließend kann man u.a. mit GReQL die TGraphen über beliebige Anfragen analysieren und auswerten.

Viele der Arbeiten über GReQL befassen sich mit dem Thema „Code Reverse Engineering”. Hier sollen über die Auswertung von Quelltexten die Beziehungen und Abhängigkeiten zu anderen Klassen in großen Projekten leichter erkennbar gemacht werden und somit das Modifizieren vereinfachen. Ein speziell dafür passendes Metamodell definiert alle Knoten und Kantentypen, die zur Repräsentation von Java-Quelltext nötig sind.

3.3.2 Bewertung von GReQL

3.3.2.1 Pro GReQL

Durch JSQL liegt das SQL-Statement bereits repräsentiert in Form eines Statementbaums vor. Dieser kann in die Struktur der Graphen, genauer gesagt in die interne Repräsentation der TGraphen für GReQL, transformiert werden kann, um darauf die Analysen und Untersuchungen durchzuführen. Das liegt daran, dass Bäume spezielle Graphen sind, also sich Bäume immer mit Graphen darstellen lassen und sich somit in diese Richtung gut transformieren lassen. In die Gegenrichtung, vom Graph zum Baum, geht es nur bei Erfüllung der strengeren Eigenschaften als Baum.

Die Prüfung der Kanten-Verbindungen kann direkt in beide Richtungen betrachtet werden. Zum Beispiel ist hier der Zugriff auf den jeweiligen Elternknoten einfach, der wiederum im JSQL-Statementbaum aktuell nicht enthalten ist. Allgemein ermöglicht GReQL sehr flexible Anfragen mit umfangreichen Funktionen. Durch die Angabe der Beschreibung des gewünschten Ergebnisses lassen sich möglicherweise mit weniger Aufwand bzw. andere Algorithmen für Suche und Vergleiche entwickeln. Die Anfragesprache GReQL wird als effizient bei der Ausführung der Analysen in Bezug auf Laufzeit und Speicher beschrieben, was im Vergleich zu individuell entwickelten Algorithmen möglicherweise ein Vorteil sein könnte. GReQL legt stark den Fokus auf den Vergleich von Kanten, man kann also leicht gezielt nach speziellen ausgewählten Pfaden in den TGraphen suchen.

3.3.2.2 Contra GReQL

Der Fokus der Überprüfung der Masterarbeit liegt auf der Untersuchung, welche Bereiche identisch sind und welche voneinander abweichen. Bei GReQL werden primär die Kanten-Eigenschaften eines Graphen untersucht, also z.B. welche Knoten wie zueinander stehen. Es wurde für GReQL keine Funktion oder Algorithmus entdeckt, der die Unterschiede, Gemeinsamkeiten und Ähnlichkeiten von zwei Graphen auswertet. Falls dies der Fall ist, müsste zu den einzelnen Abfragen weiter kombiniert algorithmisch um die einzelnen Abfragen herum gebaut werden. Z.B. sind häufige Vergleiche zwischen den TGraphen von Vergleichen mit Knoten-Beziehungen für die Pfade mit GReQL aus erster Einschätzung erschwert und verkompliziert. Die Wiederholung vollständig neuer Auswertungen in jedem Arbeitsschritt könnte es ein Nachteil gegenüber anderen möglichen Algorithmen sein.

Bevor die Abfragen genutzt werden können, steht ein größerer Integrations- und Entwicklungsaufwand für das TGraphen-Schema an, dadurch wird die Integrierung der Bibliothek in das Projekt mit samt der Anpassung für die Transformation der Daten aus dem Statementbaum in den TGraphen notwendig. Ein solcher Aufwand ist nur durch einen deutlich verbesserten Nutzen gerechtfertigt, bisher wurde jedoch kein ausschlaggebender Vorteil oder besonderer Lösungsansatz mit GReQL und TGraphen entdeckt, die andere bzw. eigene Ansätze mit Graphen und Bäumen nicht ebenso bieten würden. Trotz der Strukturähnlichkeit der Graphsprache mit FROM-WITH-REPORT zur SQL-Syntax ist damit nicht automatisch ein Vorteil verbunden für den Vergleich von Unterschieden, Gemeinsamkeiten und Ähnlichkeiten zwischen zwei Graphen.

3.4 Baum- vs Graphvergleich

Eine der Aufgaben dieser Masterarbeit ist die Entscheidung, ob die Datenstruktur Baum oder Graph besser zum Vergleich der beiden SQL-Statements und Findung der Unterschiede geeignet ist. Vor der Entscheidung zur Eignung wird zunächst der Unterschied zwischen den beiden Varianten vorgestellt.

3.4.1 Definition und Unterschiede zwischen Bäumen und Graphen

Um die Unterschiede zwischen Bäumen und Graphen zu erläutern, werden beide Datenstrukturen kurz mit ihren besonderen Eigenschaften vorgestellt.

3.4.1.1 Graphen

Graphen besitzen eine Menge an Knoten und Kanten und existieren mit verschiedenen Ausprägungen. Je nach Typ gibt es folgende mögliche Eigenschaften:

- Die Kanten sind gerichtet oder ungerichtet
- Der Graph besitzt Zyklen oder ist zyklenfrei
- Die Knoten sind zusammenhängend oder nicht zusammenhängend
- Die Kanten sind gewichtet oder ungewichtet
- Die Kanten können geordnet oder ungeordnet sein (Reihenfolge der Indexierung)

Die Eigenschaften werden in der folgenden Abbildung visuell dargestellt:

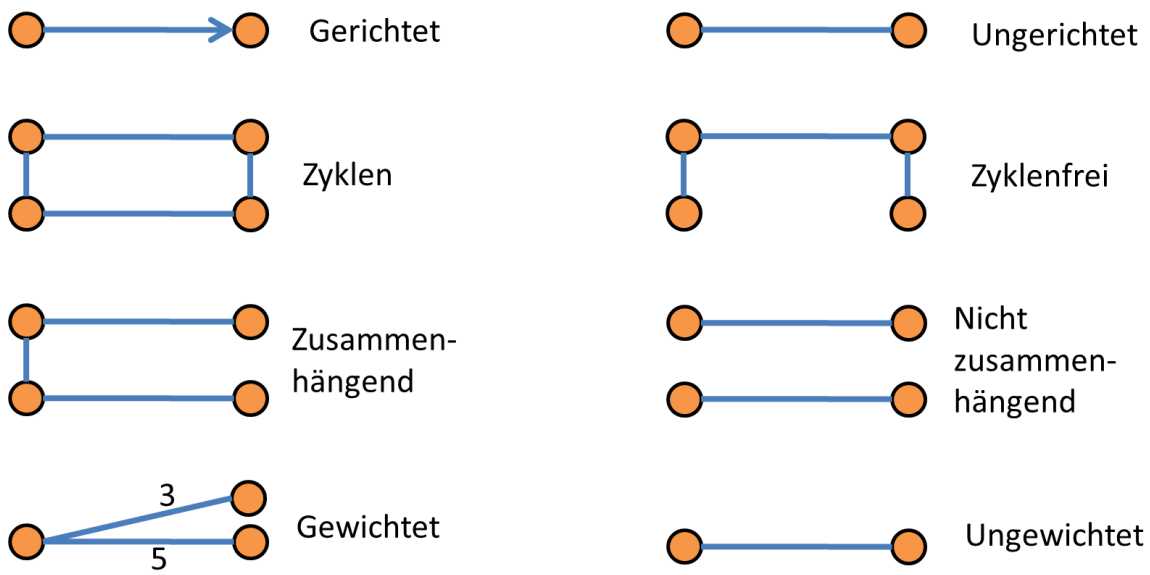

Abbildung 5: Beispiel verschiedener Ausprägungen von Kanten und Knoten in Graphen

3.4.1.2 Baum

Ein Baum besitzt ebenfalls eine Menge an Knoten und Kanten, ist jedoch ein spezieller Typ von Graph, in dem die die Eigenschaften zusammenhängend und zyklenfrei erfüllt sein müssen.

Es gibt ebenfalls verschiedene Ausprägungen von Bäumen, für das Projekt wäre der Einsatz eines gerichteten Wurzelbaumes von Vorteil. Dieser ist durch gekennzeichnet durch einen Wurzelknoten als oberstes Element zum Start, von dem aus ein eindeutiger Pfad von der Wurzel zu jedem Knoten existiert. Dadurch lässt sich eine Monohierarchie abbilden. Einen Teilbaum erhält man hier, indem man einen im Baum tiefer stehenden Knoten als neuen Wurzelknoten für den Teilbaum auswählt.

3.4.2 Vergleich der Eigenschaften von Graphen und Bäumen

Für die Fragestellung, ob Graphen oder Bäume besser für die Repräsentation eines SQL-Statements und deren Auswertung geeignet sind, werden die hierfür relevanten Eigenschaften analysiert.

Ein einzelnes SQL-Statement weist einen eindeutigen Beginn und ein klares Ende auf sowie eine feste Reihenfolge der Schlüsselwörter und Werte. Wie sieht es für diese Eigenschaften beim Vergleich der Knoten, Kanten, Teilbäume und Teilgraphen auf Unterschiede, Gemeinsamkeiten und Ähnlichkeiten aus?

Eigenschaft	Wirkung
Wurzelknoten	Mit einer Wurzel wird der Anfang vom Statement festgelegt. Ohne sie könnte man mitten im Statement sein oder man müsste durch die Richtung den Start berechnen.
Zusammenhängend	Ein einzelnes SQL-Statement besteht aus einem zusammenhängenden Text, deren Auswertung nur im Ganzen Sinn macht.
Zyklen	Es darf keine Zyklen oder Polyhierarchie existieren, ansonsten kann von der Wurzel zu dem ausgewählten Knoten kein eindeutiger Pfad bestimmt werden.
Gerichtet	Die Kanteneigenschaft Gerichtet wird zur Darstellung der Hierarchie der Knoten von Elternknoten zu Kindknoten und außerdem als Teil der Reihenfolge benötigt.
Gewichtet	Die Beziehung der Knoten von Eltern zu Kind besitzen keine Unterschiede zu anderen in der Wertigkeit. Daher fällt die Wahl auf ungewichtete Kanten.
Ordnung	Die Reihenfolge der Kindknoten über eine Ordnung (geordnete Kanten) zusammen mit gerichteten Kanten ist wichtig, da es dadurch die Reihenfolge des Auftretens im Statement repräsentiert wird. Ohne Ordnung oder Richtung würde bei mehreren Kindern diese Information verloren gehen oder müsste extra im Knoten selbst gespeichert werden.

Optimal wäre somit ein Graph mit folgenden Eigenschaften: ein Wurzelknoten als Beginn, zusammenhängend als ein gesamtes Statement, zyklenfrei, mit gerichteten, ungewichteten und geordneten Knoten, welches als Wahl der Datenstruktur einem Baum entspricht.

Durch die vorliegende Hierarchie im Baum und geordnete Teilbäumen kann man hier von jedem Punkt aus die Bereiche für die Auswertung gut mit denen des anderen Baumes vergleichen. Eine Abweichung dieser Eigenschaften erschwert die Suche und Vergleich auf identische, abweichende und ähnliche Bereiche. Die Verwendung von Bäumen als Datenstruktur hat sich im ÜPS-SQL-Trainingssystem [13] als praxistauglich erwiesen und bestätigt somit die eigenen Überlegungen zugunsten von Bäumen. Da durch JSQL bereits das SQL-Statement im gewünschten Baum erhalten ist, fällt die Wahl der Datenstruktur auf diese Art Baum mit den genannten Eigenschaften. Im Gegensatz zu GReQL müssen die Algorithmen für die Suche und Vergleiche selbst implementieren werden, dafür hat man jedoch volle Kontrolle und Flexibilität.

3.4.3 VERGLEICH ZUM SQL-REFERENZ-STATEMENT

Die Syntax mit den Regeln für den Aufbau eines SQL-Statements wird von Oracle durch Referenz-Statements dargestellt. Das nachfolgende Beispiel zeigt das Referenz-Statement der Datenbanksprache Oracle-SQL für ein SELECT-Statement [25].

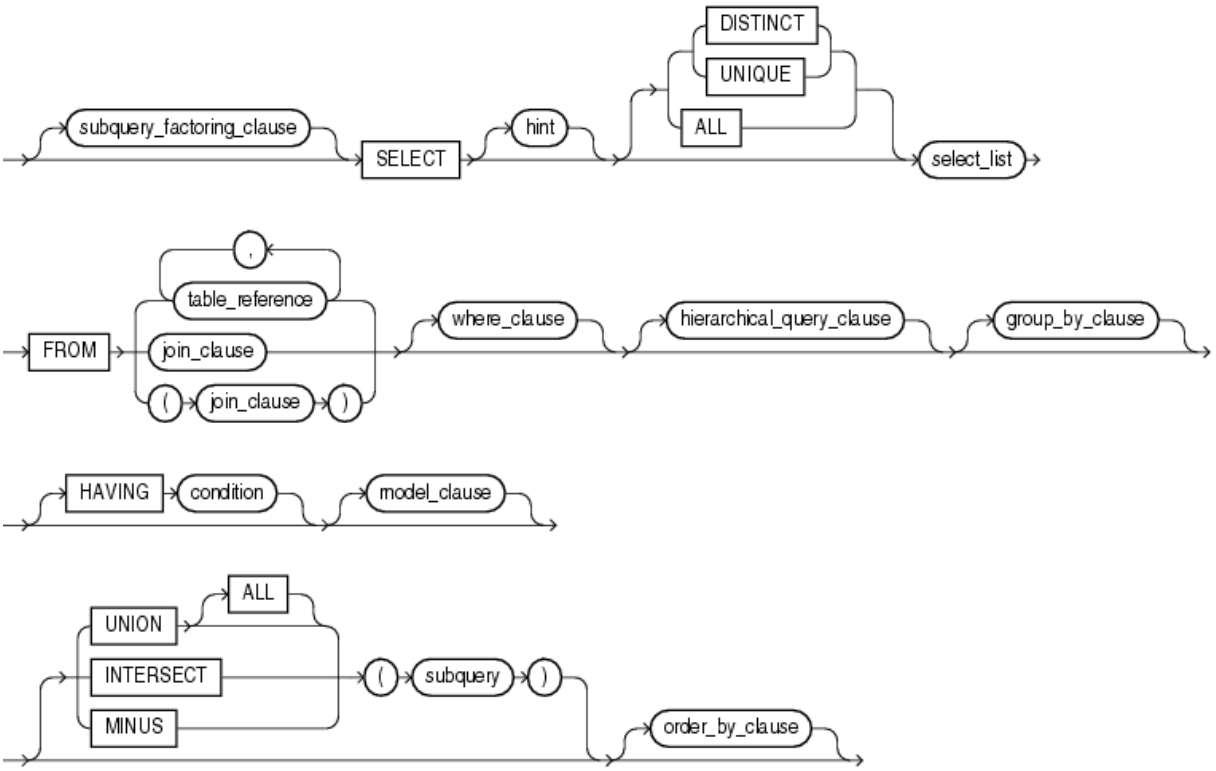

Abbildung 6: Beispiel Referenz-SELECT-Statement für Oracle SQL

Diese Referenz gilt für jede Ausprägung aller möglichen SELECT-Statements in SQL, d.h. man kann damit jedes gültige SELECT-Statement erzeugen. Da diese Referenz Zyklen enthält, würde sie nicht durch die Baumstruktur dargestellt werden können. Diese Darstellungsform für die Referenz wird auch Syntaxgraph genannt.

In der Praxis benötigen wir jedoch nur eine einzelne konkrete Instanz eines SQL-Statements, welches zyklenfrei ist und sich in Form eines Baumes darstellen lässt, obwohl nach den Regeln der Referenz erstellt wird. Genauer betrachtet ist das erzeugte Statement eine Liste von Buchstaben und Zeichen. Der wirkliche Baum wird erst durch die Einteilung der Listenelemente in Knoten erzeugt, mit logischer Einordnung der Typen für die Schlüsselwörter und Werte sowie Unterordnung in eine Hierarchie unter Einhaltung der Reihenfolge durch geordnete Kanten.

Dies geschieht durch den JSQL-Parser mit dem Statementbaum, dessen Struktur nebenbei auch genau den beschriebenen gewünschten Eigenschaften entspricht.

3.5 Recherche Algorithmus

Benötigt werden abschließend noch Algorithmen, die für die Graph- und Baumvergleiche der SQL-Statements eingesetzt werden können. Gesucht ist ein Algorithmus, welcher die Übereinstimmenden Knoten, Teilgraphen bzw. Teilbäume findet und markiert (*Mapping*), so dass die Übereinstimmungen und Unterschiede für die weitere Analyse verwendet werden können. Es folgt eine Vorstellung potentieller Algorithmen samt Bewertung, alternativ bietet sich die Entwicklung eines eigens dafür konzipierten Algorithmus an.

3.5.1 Edit Distance / Levenshtein-distanz

Der Algorithmus *Edit Distance*, auch *Levenshtein-Distanz* genannt, wird primär beim Vergleich von String-Werten genutzt. Gemäß dem Autor Hangjun Xu in der Publikation "An Algorithm for Comparing Similarity Between Two Trees: Edit Distance with Gaps" (2014) [26] arbeitet er dabei nur mit den drei Operationen Umbenennen, Löschen und Hinzufügen. Dabei wird mittels dieser Operationen der Wert solange angepasst, bis er dem Ziel entspricht. Die Anzahl der dafür nötigen Schritte werden dabei mitgezählt, je weniger Schritte, desto ähnlicher sind sich beiden Elemente. Dieser klassische Ansatz des Algorithmus mit String-Vergleichen lässt sich auch für die Baumsuche weiter entwickeln, wobei die Operationen dann auf die Knoten oder gar Teilbäume angewendet werden, um die ähnlichsten Gegenstücke zu suchen.

Dieser Algorithmus birgt allerdings mehrere Nachteile, z.B. ist die Ähnlichkeitsprüfung der Teilbäume bei größeren Bäumen schnell kostspielig, weil beständig Knoten und Teilbäume erstellt und verändert werden müssen und zwar für jeden der möglichen Vergleiche.

Äußerst fraglich ist, ob bei dieser Arbeitsweise die Ziele Effizienz und Effektivität erfüllt werden können, auch stellt sich die Frage, ob und wie genau der optimale Weg gefunden wird bei der Berechnung der Schrittzahl für Teilbäume. Außerdem ist die Aussagekraft einer Schrittanzahl hinsichtlich der Ähnlichkeit und Genauigkeit in Zweifel zu ziehen. Daher wird vom Einsatz dieses Algorithmus Abstand genommen.

3.5.2 Subtree-Kernel

Searles beschreibt den *Subtree-Kernel-Algorithmus* in der Publikaition „The Similarity Graph: Analyzing Database Access Patterns Within A Company“ [27] für Graph- und Baumvergleiche, der mit Hilfe von Fingerprints (ähnlich einem Hash-Wert) nach identischen Knoten, Teilgraphen und Teilbäumen sucht. Der Algorithmus stellt eine Weiterentwicklung des *Weisfeiler-Lehman-Algorithmus* dar, welcher ursprünglich zum Isomorph-Test auf Graphen entwickelt wurde. Dieser wird für den *Subtree-Kernel-Algorithmus* weiterentwickelt, um die Fingerprints zu berechnen.

Für diesen Algorithmus wird die Datenstruktur Baum zur Stärke bei den Vergleichen. Die Fingerprints werden mit einer passenden Funktion erst aus den Knoten und anschließend in Iterationen BottomUp die Teilbäume berechnet und in einer Liste von einzigartigen Fingerprints (Lookup-Tabelle) gespeichert, mit der bei einmaliger Berechnung schnell und einfach überprüft werden kann, ob diese Knoten oder Teilbäume im anderen Baum existieren.

Beim Vergleich einzelner Knoten werden die identischen Knoten direkt gefunden, bei leichten Abweichungen erfolgt hingegen keine Übereinstimmung, da es nur die Unterscheidung zwischen identisch und nicht identisch gibt. Auch bei Vorhandensein mehrfacher gleicher Knoten kann durch

die binäre Einteilung nicht direkt entschieden werden, welcher Knoten die bessere Wahl ist. Bei Teilbäumen wird das Problem stärker deutlich, denn bei kleinen Abweichungen wird der gesamte Teil als unterschiedlich bewertet. So wird z.B. bei Zwischenknoten im Pfad, zusätzliche Subknoten und aufwärts im Teilbaum ein komplett anderer Fingerprint erzeugt.

Die Überprüfung funktioniert hier also nur bis zum ersten auftretenden Unterschied wie gewünscht. Eine Bewertung der Ähnlichkeit von der Wurzel bis zum untersten Blatt, wie in der Masterarbeit gefordert, ist über rein binäre Bewertung nicht praktikabel. Hier lassen sich nur einzelne Knoten sowie kleine Teilbäume nahe den Blättern finden. Gesucht ist jedoch eine flexible Methode mit einem konkreten Ähnlichkeitswert auf der gesamten Baumstruktur. Daher entspricht der Algorithmus insgesamt nicht den geforderten Anforderungen.

3.5.3 Eigener Algorithmus

Auf der Basis der vorangegangenen Analysen und Bewertungen bietet es sich an, mit dem gewonnenen Wissen und den gesammelten Ideen ein eigenes Konzept für den Baumvergleich zu erstellen, das auf die als praxistauglich erscheinenden Elemente zurückgreift. Das Ziel ist, die Stärken als Ideen aufzugreifen und die vorhandenen Schwächen zu mindern oder gar durch Verbesserungen, Ersatz oder Kombinationen zu lösen. Einer der Vorteile eines eigenen Algorithmus besteht auf der speziell abgestimmten Anpassung an die vorliegenden Datenstruktur bzw. der Daten. Da die Knoten in unserem Statementbaum dank JSQL über mehrere klar definierte Attribute verfügen, würde dadurch auch eine Ähnlichkeitsüberprüfung über diese Attribute möglich sein. Eine Funktion würde hier zu zwei Knoten einen Wert zwischen 0 und 1 für die Übereinstimmung berechnen. Im Falle von übereinstimmenden Werten, können bei der Entscheidung spezielle Attribute als Sekundärwahl verwendet werden. Dadurch werden variable Abstufungen ermöglicht im Gegensatz zum binären Vergleich durch z.B. Fingerprints. Dieser Algorithmus hätte in einer ersten Einschätzung mehr Aussagekraft als eine Schrittanzahl bei Transformation.

3.5.4 Entscheidung

Aus der Analyse von GReQL ist kein Konzept entstanden, dass gegenüber Konzepten auf der Basis von Baumvergleichen deutliche Vorteile aufweist. Da im Baumvergleich mehr Potential gesehen wird und sich diese Datenstruktur sich in ÜPS [13] als praxistauglich erwiesen hat, wird für die Masterarbeit eine Lösung mit einer Baumstruktur präferiert.

Bei der Auswahl des Algorithmus wird der eigene Algorithmus bevorzugt, der auf der Grundlage der Bewertung vorhandener Algorithmen entwickelt wird. Es wird im nachfolgenden Kapitel Konzept ausführlich vorgestellt.

4 KONZEPT

Die Weiterentwicklung von aSQLg beinhaltet schwerpunktmäßig die Untersuchung der Unterschiede zwischen den Statements der studentischen Abgaben auf Abweichungen zur Musterlösung dar, aus denen die Lösungsempfehlungen generiert werden.

Dieses Kapitel beschreibt die Idee des erstellten Konzepts mit seinen Arbeitsschritten und Komponenten. Die zentralen Komponenten Mapping, Auswertung und Lösungsempfehlung werden im Detail vorgestellt, anschließend erfolgt eine Einschätzung des Konzepts als Bewertung.

4.1 ÜBERSICHT UND VORSTELLUNG DER GRUNDIDEE

Zur Beschreibung des Konzepts werden die wichtigsten Arbeitsschritte in der nachfolgenden Abbildung in einem vereinfachten Modell eingeteilt:

Abbildung 7: Grobe Einteilung des Konzepts in 4 Arbeitsschritte

Beginnend mit der Statementprüfung wird geschaut, ob im Fall von leeren oder identischen Statements die Überprüfung nicht durchgeführt werden muss. Bei der Überprüfung werden die zunächst die Statementbäume aus den beiden Statements generiert, im Falle eines Fehlers dabei wird dieser gemeldet und die Überprüfung beendet. Das darauffolgende Mapping basiert auf dem Erkennen und Kenntlichmachen der Gemeinsamkeiten sowie Unterschiede der Knoten beider Bäume. Die Knotenattribute werden mitsamt seinem Pfad im Baum für die Vergleiche der Knotenähnlichkeit genutzt, bei der nach gleichen oder sehr ähnlichen Knoten gesucht wird und bei ausreichender Übereinstimmung miteinander gemappt werden. Anschließend werden die gewonnenen Informationen aus dem Mapping für die einzelnen Knoten in der Auswertung extrahiert und gesammelt. Durch verschiedene Untersuchungen werden diese in höherwertige Informationen weiterverarbeitet. Abschließend werden die ausgewerteten Informationen dann als Lösungsempfehlungen umwandelt als Nachrichten an den Student und falls notwendig Instruktor.

Die folgende Tabelle fasst die Arbeitsschritte zusammen:

Arbeitsschritt	Erklärung
Statementprüfung	Prüfung, ob die Statements leer oder identisch sind. Trifft dies zu, ist keine Suche nach Lösungsempfehlungen notwendig.
Baumgenerierung	Erstellung der Bäume aus den SQL-Statements durch JSQL. Falls beim Parsen der SQL-Statements ein Fehler auftritt, so erfolgt eine Fehlermeldung und die Überprüfung wird beendet.
Mapping	Vergleich der Knoten der Bäume. Durch das Mapping werden übereinstimmende Knoten kenntlich gemacht.
Auswertung	Aus dem Mapping erhaltene Informationen werden extrahiert und durch verschiedene Untersuchungen weiter verarbeitet.
Nachrichten	Aus den gewonnen Analyseergebnissen werden die Nachrichten für die Lösungsempfehlungen generiert

Aus dem existierenden Projekt wird die Struktur für die Statementbäume samt der Knoten verwendet, ebenso der JSQL-Parser für die Baumgenerierung. Andere Elemente wie der Grader werden nicht eingesetzt. Für den Baum als gewählte Datenstruktur existiert noch kein komplett zufriedenstellender Algorithmus, Ansätze für eine eigenständige Entwicklung sind aber gegeben.

4.2 Mapping

Im Mapping-Schritt werden die beiden Statementbäume miteinander verglichen, die ähnlichsten Knoten im anderen Baum gesucht und miteinander gemappt. Ziel ist dadurch die Kenntlichmachungen der Gemeinsamkeiten und indirekt auch der Unterschiede der Bäume. Dieser Abschnitt beschreibt den Ansatz des eigenständig entworfenen Mapping-Algorithmus, anschließend wird der Vergleich zur Berechnung der Knotenähnlichkeit erklärt.

4.2.1 Mapping-Algorithmus

Der Mapping-Algorithmus wird für die Suche nach den ähnlichsten Knoten in zwei Stufen aufgeteilt. Im ersten Schritt werden die in beiden Bäumen exakt übereinstimmenden Pfade durchlaufen und dort nach identischen bzw. sehr ähnlichen Knoten über eine Ähnlichkeitsberechnung geprüft und ab einem festgelegten Grenzwert als übereinstimmend gewertet und somit miteinander gemappt.

Im zweiten Schritt werden alle übrigen ungemappten Knoten des Studentenbaumes mit denen des Instruktorbaumes jeweils als Knotenpaar über die Ähnlichkeitsberechnung verglichen. Ab einem gewählten Grenzwert als Mindestanforderung werden die Kandidatenpaare in einer nach dem Ähnlichkeitsergebnis sortierten Kandidaten-Liste aufgenommen. Diese Liste mit den ähnlichsten Mapping-Kandidaten besteht jeweils aus einem Studentenknoten, einem Instruktorknoten und dem dafür berechneten Ähnlichkeitsergebnis. Im Falle einer identischen Bewertung für die Ähnlichkeit der Kandidaten wird als sekundäre Sortierung das Auftreten im Baum (Preorder) verwendet. Die Liste wird anschließend mit den ähnlichsten Kandidaten nacheinander zum Mapping verwendet und danach noch vorhandene Elemente aus der Liste entfernt, die einen der beiden zuvor gemappten Knoten enthält. Die folgende Abbildung fasst die Funktion des 2-Schritte-Mappings zusammen:

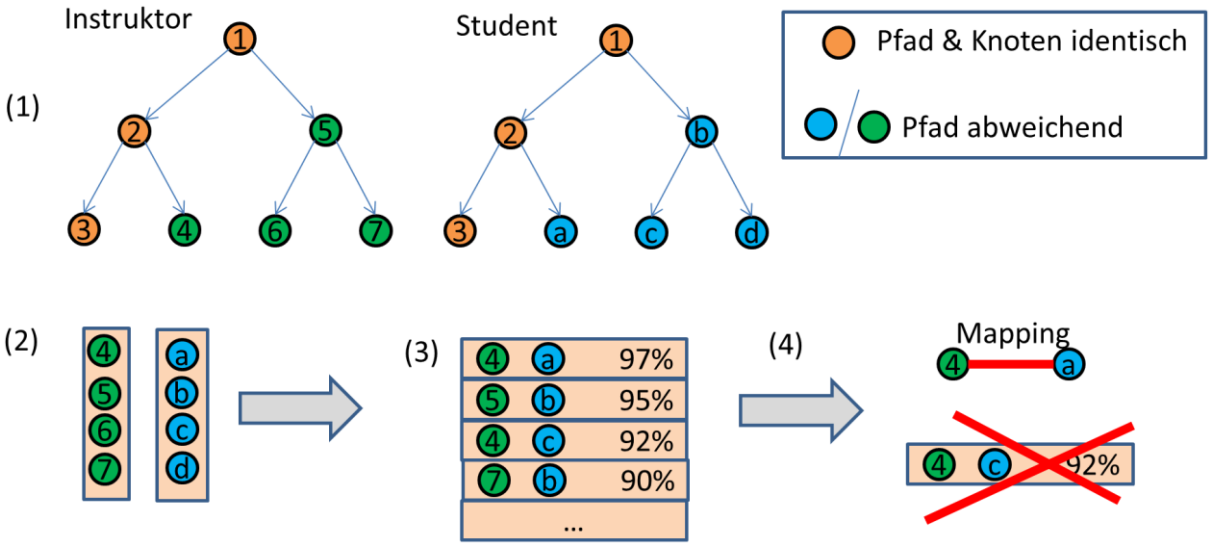

Abbildung 8: Vereinfachte Darstellung des 2-Schritte Mappings

Der erste Mapping-Schritt ist in (1) zu sehen, der zweite in (2) bis (4).

(1) Baum wird im ersten Schritt entlang identischem Pfad mit gleichen Knoten gemappt.

(2) im zweiten Schritt werden die ungemappten Knoten werden aus beiden Bäumen gesammelt.

(3) Für die beiden Listen wird eine Kandidatenliste gebildet, bewertet und sortiert.

(4) Mapping des besten Kandidaten und Entfernen anderer Kandidaten mit den gemappten Knoten.

Abschließend erfolgt nach dem Mapping eine zusätzliche Kenntlichmachung des Knotenstatus und der Teilbäume. Hier werden die einzelnen Knoten und komplett gemappte Teilbäume deklariert, was die spätere Auswertung erleichtert, da der Fokus in der Auswertung auf der Untersuchung von ungemappten Knoten und Teilbäumen liegt.

4.2.2 Berechnung der Knotenähnlichkeit

Ziel ist es, beim Mapping die ähnlichsten Knoten für das Mapping zu identifizieren. Dafür wird eine Berechnung des Wertes für die Ähnlichkeit zwischen zwei Knoten entworfen. Der berechnete Wert reicht von 0 als keine Ähnlichkeit bis 1 als komplett identisch. Die Funktion dafür nutzt die enthaltenen Attribute der Knoten sowie ihrer Pfade im Baum. Nicht nur die Gesamtbewertung liegt zwischen 0 und 1, sondern ebenfalls auch die der einzelnen Teilergebnisse der Attribute, jedoch mit unterschiedlichen Anteilen an der Gesamtwertung. Die durch JSQL generierten Knoten der Klasse *StatementViewNode* werden nach einem festen Schema aufgebaut, deren Attribute sich dadurch gut vergleichen lassen. Diese werden im folgenden Beispiel vorgestellt:

Attribut	**Erklärung**	**Beispiel**	**Anteil zur Bewertung**
Level	Entspricht der Ebene im Baum	5	2%
Column	Beschreibt, an welcher Zeichenstelle der Knoten im SQL-Statement enthalten ist. Falls kein Bezug auf den Ort gegeben ist, wird der Wert -1 vergeben	23	2%
Origin	Enthält den Namen des genutzten Visitors mit der zum Node dazugehörigen Kategorie	ExpressionVisitor>StringValue>Value	35%
StringRe-presentation	Enthält das Schlüsselwort aus dem SQL-Statement. Leer, wenn der Knoten nur für die Struktur des Baumes zuständig ist.	Musterstadt	20%
SyntaxType	Enthält die Syntax-Kategorie: *Statement / Keyword / Attribute / Group / Function / Category / SubArea / SetOperation / None*	Attribute	15%
SemanticType	Enthält die Semantic-Kategorie, z.B. FROM, WHERE, DISTINCT, isNull, OR, ...	Value	15%
SubStatement	Enthält die Substatements	Null	1%
Nodes	Enthält die Liste seiner Kindknoten	Null	0-5% Malus
Pfad	Neues Attribut für den Pfad im Baum von der Wurzel bis zu diesem Knoten.	Liste von Strings	10%

Zur wesentlichen Identifikation und Einteilung in die passenden Kategorie der Knoten ist primär das Knotenattribut *Origin* geeignet, welches sich mit Hilfe vom *SyntaxType* und *SemanticType* präziser bestimmen und einteilen lassen. Diese erhalten bei der Gewichtung einen hohen Stellenwert. Über die *StringRepresentation* erhält man weitere wichtige Informationen, zum Beispiel die Namen der genutzten Tabellen, Spalten oder Werte. Anteilig geringer fließen zusätzlich die Attribute *Level* sowie *Column* für die Position ein, dazu ein mögliches vorhandenes *SubStatement* und die Übereinstimmung im Falle von vorhandenen Kindknoten.

Während alle Attribute außer den Kindknoten ein Ergebnis von 0 bis 1 ergeben, ist der Einfluss der Kindknoten hier in Form eines Malus – folglich ein Abzug an der Gesamtwertung mit maximal 5%. Wenn beide Knoten keine Kinder haben, dann gibt es keinen Abzug. Hat ein Knoten Kinder und der andere nicht, so gibt es die maximalen 5% Malus. Als Alternative wäre auch statt direkt 5% ein ansteigendes 1 % pro Kind bis zum Maximalwert von 5% denkbar. Haben beide Kinder, so wird die Ähnlichkeit zwischen diesen berechnet und abhängig der Abweichung ein Abzug im Bereich von 0-5% für die Gesamtwertung erfolgt.

Als Zusatz zu den existierenden Attributen wird als neues Attribut für jeden Knoten der Pfad im Baum (Liste der Origin-Attribute) als Kontext gespeichert und fließt dann anteilig für die Bewertung beim Vergleich der Ähnlichkeit mit ein. Zur Berechnung der Ähnlichkeit zwischen den Pfaden wird eine Formel gesucht. Durch den Vergleich der Listen beider Knoten erhält man die die Gesamtanzahl der Listenelemente sowie die Menge der übereinstimmenden Elemente der Pfade, quasi der „Treffer“. Das Ziel ist es, eine gutes Verhältnis zu finden, in der gefundene Gemeinsamkeiten belohnt werden und Abweichungen Abzüge erhalten, jedoch beides nicht zu stark. Dabei sollen übereinstimmende, ähnliche und stark abweichende sich klar voneinander unterscheiden. Als Formel wird dafür folgendes gewählt:

Pfadähnlichkeit = (2* Treffer) / (Anzahl(Liste1) + Anzahl(Liste2))

Die daraus resultierende Berechnung würde folgende Beispiele ergeben:

	#1	#2	#3	#4	#5	#6	#7	#8	#9
Anzahl Liste 1	4	6	5	5	20	3	6	9	3
Anzahl Liste 2	4	5	4	5	15	3	4	5	2
Treffer	4	5	4	4	12	2	3	4	1
Wertung	100%	90,9%	88,9%	80%	68,6%	66,7%	60%	57,1%	40%

Hier wären auch alternative Formeln denkbar, z.B. aufgrund häufiger Ähnlichkeiten nahe der Wurzel eine unterschiedliche Gewichtung für die Reihenfolge nutzen. Die tiefer liegenden Knoten erhalten hier höhere Anteile an der Gesamtbewertung. Ebenfalls Einfluss hätte der Vergleich, auf welcher Höhe die jeweiligen Knoten liegen, wobei hier dann relativ bzw. absolut gesehen ein ähnlicher Level mit einer höheren Bewertung bevorzugt wird.

4.3 AUSWERTUNG

Die Auswertung beginnt damit, die ungemappten Knoten der ausgewerteten Statementbäume für die spätere Auswertung zu sammeln und durch Kenntnis der verschiedenen Ausprägungen die relevanten Informationen zu extrahieren. Dadurch werden die Informationen in viele kleine Teile aufgeteilt, die durch spätere Analysen zu größeren komplexeren Informationen zusammengesetzt und damit zur Erstellung von Lösungsempfehlungen genutzt werden.

Effektiv bedeutet es, durch das Mapping, die Kenntnis von fehlenden und überflüssigen Knoten zu erhalten, welche durch Prüfung auf An- oder Abwesenheit im anderen Baum erfolgt. Durch die Aggregation in höherwertige Informationen kann man zusätzlich auch die Einteilung für Verwechslungen vornehmen. Ebenfalls lassen sich in den höherwertigen Informationen Ähnliche zu Gruppen zusammenfassen. Dabei wird also versucht, aus mehreren einzelnen Nachrichten gefiltert eine aussagekräftige aggregierte Nachricht mit hohem Informationsgehalt zu erzeugen.

Die nachfolgende Abbildung beschreibt vereinfacht die Auswertung der ungemappten Knoten durch Analysen und Erstellung von Lösungsempfehlungen.

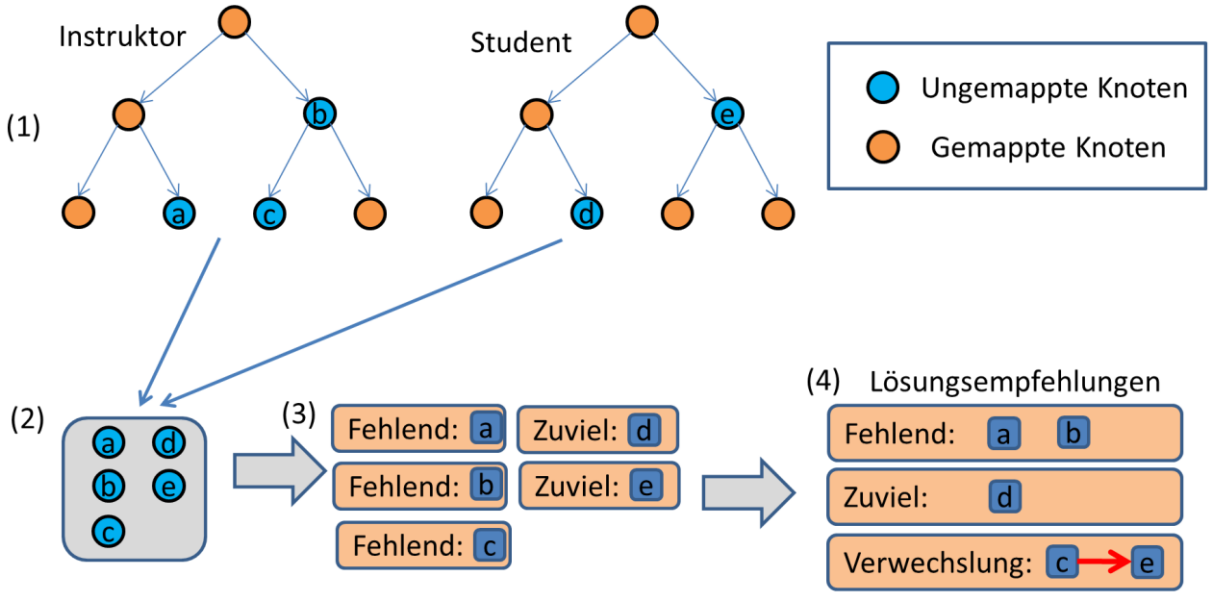

Abbildung 9: Vereinfachte Darstellung der Auswertung von ungemappten Knoten durch Analyse zu Lösungsempfehlung

(1) Die Unterschiede der Bäume sind durch das Mapping kenntlich gemacht.

(2) Sammlung der ungemappten Knoten in einer Liste

(3) Generierung der *CollectorItems* aus den Elementen der Liste

(4) Weiterverarbeitung der Elemente mit Aggregation zu den Lösungsempfehlungen

Als Inspiration der Idee, in viele kleine Bausteine aufzuteilen und dann wieder systematisiert die gewonnen Informationen zu größeren Einheiten zusammensetzen, dient das Prinzip *Divide-and-Conquer* und kurze Einblicke in das Thema *Complex Event Processing*. Dabei ist es sehr wichtig darauf zu achten, ein geeignetes Verhältnis zwischen der Fragmentierung und Zusammensetzung der richtigen Abschnitte zu finden.

4.4 KONZEPT DER LÖSUNGSEMPFEHLUNG FÜR DIE STUDENTEN

Bei der Konzipierung der Lösungsempfehlung für den Studenten stellen sich im Vorfeld nicht nur technische Fragen, die bei der Planung berücksichtig werden sollten.

Beispielhaft genannt seien:

- Was ist der Umfang der SQL-Syntax?
- Welche Fehler sind typisch?
- Wie sollen die Lösungsempfehlungen aussehen?
    - Was ist die richtige Menge bzw. Detailgrad an Informationen, ohne zu viel von der Lösung preiszugeben
    - Welche Lösungsempfehlungen sind für den Studenten aufschlussreich, welche sind ohne zielführende Aussage?
    - Wie sollen die Lösungsempfehlungen dargestellt werden?

4.4.1 ÜBERSICHT DES SQL-UMFANGS VON JSQL

JSQL bildet einen Ausschnitt der wichtigsten Elemente der SQL-Syntax nach, die zum Erlernen der SQL-Syntax wichtig sind. Der Funktionsumfang von JSQL wird als ausführliche Tabelle im Anhang unter „Übersicht JSQL“ aufgelistet und dient als Referenz für die Implementierung. Zusammenfassend sind folgende Schlüsselwörter von SQL in JSQL enthalten:

- Die verschiedenen Statement-Typen wie SELECT, DELETE, ALTER, DROP, INSERT, CREATE, UDPATE und TRUNCATE
- Die verschiedenen JOIN-Arten (FULL, NATURAL, SIMPLE, RIGHT, LEFT, INNER, OUTER), sowie Sub-Statements
- Die Basis der Arithmetischen Operationen: Addition, Multiplikation, Division und Subtraktion
- Funktionen wie: COUNT, MIN, MAX, AVG, SUM, NVL, ROUND, TRUNC und REPLACE
- Vergleiche wie kleiner/größer/gleich (<, >, <= ,>= , =),Ungleich (!= bzw. <>), LIKE, BETWEEN, IS (NOT) NULL
- Sortierung, Gruppierung und Filterung: DISTINCT, ORDER BY, GROUP BY, HAVING, LIMIT, TOP
- Verbindungen: OR, AND, UNION, UNION ALL
- Überprüfungen: IN, ALL, ANY und CASE WHEN THEN

4.4.2 LISTE MÖGLICHER SQL-STATEMENT-FEHLER

Zur Festlegung möglicher Lösungshinweise ist vorweg eine Sammlung von potentiellen Fehlern der studentischen SQL-Statements hilfreich, welche sich in die Kategorien der Syntax- und Semantik-Fehler einteilen lassen:

4.4.2.1 SYNTAX-FEHLER

Beim Parsen des Statements durch JSQL werden allgemeine Syntax-Fehler gegenüber der SQL-Struktur gefunden. Syntax-Fehler in Abhängigkeit zum verwendeten Schema werden nicht gefunden, wobei dies der Grader übernehmen könnte. Gründe für Syntaxfehler sind meist Verletzung der SQL-Syntax-Regeln, wie zum Beispiel:

- Ein Schlüsselwort wird falsch geschrieben
- Syntax eines Schlüsselworts wird falsch benutzt
- Schlüsselwort oder Wert ist zu viel, fehlt oder ist in der falschen Reihenfolge
- Reservierte Worte oder unerlaubte Zeichen werden benutzt
- SQL-Syntax wird nicht eingehalten, z.B. Klammern nicht geschlossen

Da die Syntaxprüfung Aufgabe des Parsers ist und nicht Teil des Baumvergleichs, sind Syntaxfehler hier weniger im Fokus der Analyse, welche Fehler möglich sind.

4.4.2.2 SEMANTIK-FEHLER

Die Semantik-Fehler lassen sich auf folgende Arten untersuchen: durch Überprüfung auf Nichteinhaltung der Semantikregeln und durch Unterschiede im Baumvergleich.

Nichteinhaltung von Semantischer und Stilistischer Regeln

Beispiele von Semantik-Fehlern sind bereits umfangreich durch die Publikation „Semantic errors in SQL queries: A quite complete list“ beschrieben. Diese Liste enthält richtige Fehler, ungenutzte Optimierungsmöglichkeiten und Stilregeln. Die Überprüfung dieser lässt sich überwiegend durch eine reine Analyse auf das Statement mit Kenntnis des SQL-Schemas durchführen.

Unterschiede zwischen den Bäumen als Abweichung zur Musterlösung

Die zweite Kategorie umfasst die Unterschiede zwischen dem SQL-Statement des Studenten mit der Musterlösung des Instruktors als Referenzstatement. Die Unterschiede werden durch den Baumvergleich ausgewertet und sind ein Kernthema in den Analysen dieser Masterarbeit. Die meisten Fehler aus der ersten Kategorie können durch ein gut konstruiertes Referenz-Statement vom Instruktor bereits gefunden werden, da meist die Abweichungen zur Musterlösung meistens auch zu einer falschen Lösung führen. Als mögliche Fehler in den studentischen Statements sind nachfolgende Beispiele aufgeführt:

- Falscher Statement-Typ: SELECT statt INSERT durch Aufgabenverwechslung
- Ein Element ist in einem der Bäume, aber nicht im anderen vorhanden.
    - Resultiert aus Nennung überflüssiger Elemente bzw. fehlender Elemente
    - Kann auch durch einen alternativen Ansatz des Studenten erfolgen
- Verwechslung von Elementen:
    - Tabellen und Spalten: als Ganzes oder Reihenfolge z.B. bei LEFT JOIN
    - Werten: Zahlendreher, Grenzbereiche für Vergleiche mit kleiner/größer (gleich)
    - Verwendung von falschen Datentypen: String ‚42‘ statt Long 42

Aber nicht jede Abweichung von der gewünschten Musterlösung stellt auch wirklich eine falsche Lösung dar. Hier stellt sich die Frage, welche Abweichungen zwischen den Bäumen als echte Fehler deklariert werden und welche als äquivalente Alternativen in einem akzeptierten Toleranzbereich liegen. Folgende Beispiele zeigen Alternativen, die keine Fehler darstellen:

- (>0 AND <42) anstelle von (>=1 AND <=41)
- SUM(value)/COUNT(value) anstelle AVG(value)
- (X=1 OR X=8 or X=42) anstelle von X IN (1,8,42)
- JOIN der Tabellen über „FROM A,B WHERE A.X=B.Y“ statt über „FROM A JOIN B ON A.X=B.Y“

Eines der Ziele ist es, solche tolerierbaren Alternativen über spezielle Regeln der Analyse zu erfassen, um diese nicht als Fehler zu bewerten.

4.4.3 Aufbau der Nachricht für die Lösungsempfehlung

Durch den Vergleich der beiden Bäume erhält man die erwünschten Informationen über die Abweichungen: Elemente, die nur im Studentenbaum auftreten, werden als "Überflüssig" deklariert, nur im Instruktorbaum auftretende als "Fehlend". Diese Einzelinformationen werden weiter verarbeitet in höherwertige in Form von Gruppierungen. Durch diesen zusätzlichen Arbeitsschritt wird die Zuordnung zur Kategorie „Verwechslung“ ermöglicht. Anschließend werden diese für die Ausgabe zur Lösungsempfehlung aufbereitet.

Zur Festlegung des Aufbaus der Lösungsempfehlungen wird die Zielgruppe definiert, Ziele gesetzt und überlegt, welche Elemente zur Darstellung sinnvoll und welche weniger sinnvoll sind. Die Zielgruppe für das System sind diejenigen Studenten, die die benötigten theoretischen Konzepte bereits erlangt haben, um die Aufgaben lösen zu können. Daher sollen die Lösungsempfehlungen kein System mit einem Erklärfokus zum Erlernen der SQL-Sprache oder ein Nachschlagewerk sein, sondern eine Arbeitshilfe, mit der bereits erlerntes Wissen überprüft werden soll. Ebenfalls ist es nicht vorgesehen, dass der Student die fertige Musterlösung erhält, sondern lediglich Hinweise, wie er die Musterlösung auf der Grundlage seines eingereichten Statements findet. Die Hinweise müssen auch nicht größere Teilabschnitte des Statements der Musterlösung enthalten oder in der richtigen Reihenfolge stehen. Ziel ist die richtige Balance zu finden aus unterstützenden Hinweisen und studentischem Engagement bei der Lösungsfindung. Eine gute Übersicht der Lösungsempfehlungen soll durch gezielte Hinweise mit klaren Hinweisen entstehen, was bedeutet diese komprimiert in wenigen Zeilen mit möglichst wenig Text und unnötigen Zeichen darzustellen. Es soll nicht überfüllt wirken und eine klare Trennung von Schlagwörtern, Werten und Formatierung enthalten. Eine einheitliche, kurze stichpunktartige Struktur bietet sich für diesen Zweck besser an als ausgeschriebene Sätze. Eine Vielzahl von Lösungsempfehlungen mit unzureichendem Informationsgehalt wie zum Beispiel „Eine Spalte fehlt“ oder „Ein Schlüsselwort fehlt“ sind nicht zielführend für den Studenten und zu vermeiden. Besser sind spezifischere Informationen wie zum Beispiel „Die Spalte price fehlt“ oder „DISTINCT fehlt“, um damit genauer das eigene Statement auf die Abweichungen mit den eigenen Notizen der SQL-Syntax zu untersuchen. Wenn der Student ein Statement abgibt, in dem viele Abweichungen existieren, so sind gebündelte Informationen in übersichtlicher Darstellung für das Konzept angebracht, um schnell und kompakt die Lösungsempfehlungen nachvollziehen zu können. Die Unterteilung der möglichen Abweichungsmerkmale sind „Fehlend“ sowie „Zuviel“ beim Auftreten nur innerhalb eines Baume und „Verwechslung“ bei Auftreten ähnlicher Elemente in beiden Bäumen, jedoch mit leichten Abweichungen.

Beispiel einer einzelnen fehlenden Spalte „BSP1“:

Fehlend: Spalte[BSP1]

Beispiel mehrere überflüssiger Spalten „BSP2“ und „BSP3“ zusammen geblockt:

Zuviel: Spalte[BSP2, BSP3]

Beispiel Verwechslung von Funktionen, die geblockt zusammengefasst werden:

Verwechslung: Funktion[COUNT, SUM] statt [AVG]

4.5 Einschätzung des Konzepts

Für die Bewertung des vorgestellten Konzepts erfolgt eine Einschätzung vom Mapping, der Erkennung der Abweichungen und der Ausgabe.

4.5.1 Einschätzung des Mappings

Das Mapping der einzelnen Knoten sollte recht zuverlässig beim Auffinden und Zuweisen der gleichen sowie sehr ähnlichen Knoten im anderen Baum sein. Die Attribute lassen sich hier gut miteinander vergleichen.

Als Absicherung gegen mögliche Fehler bei der Analyse attributgleicher Knoten ist folgender Schritt gedacht: Beim Auftreten von Knoten mit identischen Attributen wird durch den Vergleich des Pfades der Unterschied in der Bewertung getroffen. Im seltenen Falle von identischen Geschwisterknoten ist die Entscheidung durch die frühere Position im Baum beim Mapping priorisiert.

Der gewählte Ansatz ist einer von mehreren möglichen Varianten, z.B. wäre auch ein Ansatz auf Basis des Mappings von Teilbäumen möglich oder über die Knoten mit den enthaltenen Schlüsselwörtern bzw. Werten.

4.5.2 Einschätzung der Auswertung und Erkennung von Unterschieden

Die Erkennung von Unterschieden in den SQL-Statements sollte prinzipiell gut funktionieren bei kleineren wie auch größeren Abweichungen. Besonders gut sollten Standardfehler wie z.B. überflüssige oder vergessene Elemente, Typverwechslung und Wertvertauschungen erkannt werden. Die Auswertung ist abhängig von den Informationen, also den ungemappten Knoten, die das Mapping liefert. Die noch durchzuführenden Praxistests werden Klarheit darüber bringen, wie gut dieses Zusammenspiel funktioniert oder ob dadurch eine Anfälligkeit für kleine Abweichungen mit größerem Effekt entstehen kann. Das Konzept basiert vorrangig auf einer Untersuchung von Existenz und Abwesenheit, wodurch Vor- und Nachteile entstehen können. Ein Vorteil wäre die Flexibilität der akzeptierten Reihenfolge der einzelnen Elemente, wenn zum Beispiel `a=2 AND b=3` genauso gültig wie `b=3 AND a=2` ist. Durch diese Unabhängigkeit der Reihenfolge und der Richtung in Prüfung der Existenz gibt es jedoch den Nachteil, dass bestimmte semantische Fehler nicht erkannt werden könnten. Dazu gehört u.a. die nicht überprüfe Reihenfolge wie z.B. für GROUP-BY und ORDER-BY bei mehreren Elementen oder die Abweichung von `a=2 AND b=3` mit `„a=3 AND b=2`, da in beiden alle Elemente [a,b,2,3 und AND] vorhanden sind für das Mapping. Als ähnliches Beispiel wären arithmetische Operationen wie `(A*B)+C` statt `A*(B+C)`. Zusammen mit dem existierenden GRADER sichert jedoch aSQLg im Gesamtablauf die Überprüfung der arithmetischen Korrektheit durch den Vergleich der Ergebnisse während der Ausführung beider Statements. Das Konzept beinhaltet bisher keinen konkreten Lösungsansatz für die Erkennung von Äquivalenzen. Falls der Student also `>0` und der Instruktor `>=1` benutzt, würde hier aktuell eine Abweichung erkannt werden, obwohl bei LONG-Werten diese gleichwertig sind. Dies würde hingegen von einem Instruktor in kürzester Zeit als Äquivalenz erkannt werden. In der Analyse wäre es vielleicht möglich, durch gezielte Sonderregeln nach solchen Übereinstimmungen zu suchen. Als Beispiel für eine solche Sonderregel wäre wie im oben beschriebenen Beispiel die Suche nach ähnlichen Operatoren für die Kategorie „Größer/Kleiner/BETWEEN“ und passender LONG-Werte.

4.5.3 Einschätzung der Darstellung der Lösungsempfehlung

Die Ausgabe der Lösungsempfehlung bietet durch die Gruppierung, Einteilung in Kategorien (Fehlend, Zuviel und Verwechslung) und einer einheitlichen Formatierung des Textes eine gute Übersicht und lässt sich dadurch gut vergleichen.

Bei größeren Abweichungen der Bäume von längeren Statements könnte die Auflistung jedoch umfangreicher werden, sie ist durch die Gruppierung jedoch in den Zeilen limitiert auf die Anzahl der verschiedenen Typen (z.B. JOIN, Funktionen, Werte, Spalten oder Tabellen).

Da der Zeilen- und Spaltenindex nicht für jeden Knoten im Baum vorhanden ist, wurde er im Konzept bei der Ausgabe der Lösungsempfehlung nicht weiter berücksichtigt. Diese wären vom Instruktorstatement für den Studenten auch unwichtig, da er dieses nicht einsehen kann. In seinem eigenen Statement kann das zwar nützlich sein im Fall von Mehrfachverwendung, würde jedoch den Text der Ausgabe verlängern, daher wurde im Konzept darauf verzichtet.

5 IMPLEMENTIERUNG

Auf der Grundlage des Konzepts wird die technische Umsetzung mit der Architektur entwickelt. Das Ziel dieses Projekts ist die Erweiterung vom Java-Projekt aSQLg als zusätzliches paralleles Modul, so dass es neben dem Grader genutzt werden kann. Die Komponenten wie der Grader sollen in ihrer Funktion nicht durch das Projekt verändert werden, daher wird bei der Nutzung vorhandener Bestandteile darauf geachtet, die bestehende Architektur möglichst unverändert beizubehalten.

5.1 ÜBERSICHT ARCHITEKTUR

In der Implementierung erfolgen die Details und Ergänzungen der Umsetzung des Konzeptes. Auf Basis der 5 Arbeitsschritte werden die Eigenschaften und Besonderheiten der enthaltenen Komponenten der Arbeitsschritte beschrieben. In der folgenden Abbildung werden die Komponenten kurz vorgestellt:

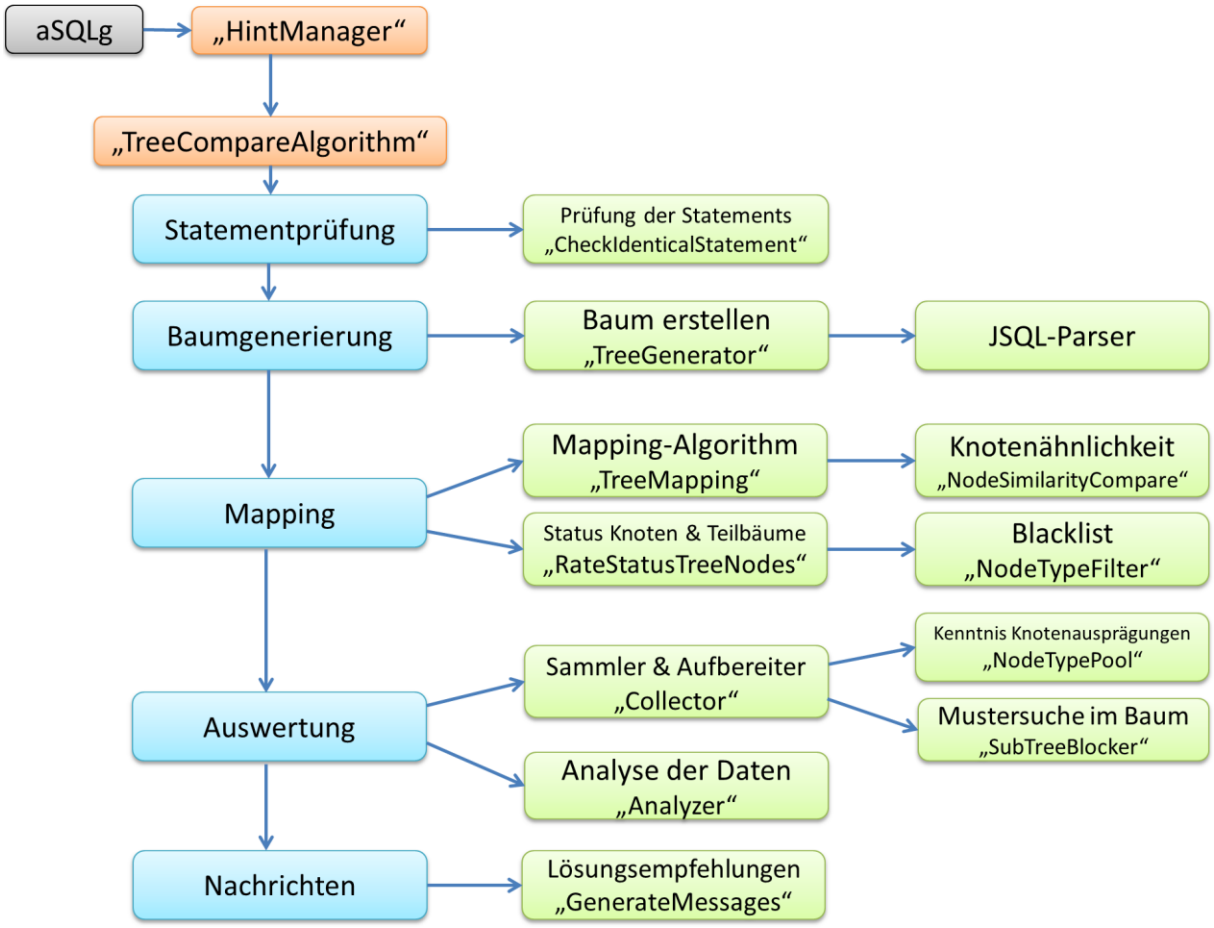

Abbildung 10: Übersicht der Architektur und wichtigsten Komponenten

Die einzelnen Komponenten erhalten einen eigenen Arbeitstitel, der in der späteren Beschreibung als Name für die Klassen verwendet wird.

5.2 Übersicht der neuen Komponenten

5.2.1 HintManager und TreeCompareAlgorithm

Der *HintManager* ist der Einstiegspunkt der Entwicklung und wird von aSQLg aufgerufen. Übergeben wird dabei die Liste von zu untersuchenden Statementpaaren des Studenten und Instruktors für die jeweiligen Aufgaben (von z.B. einem Aufgabenblatt mit mehreren Aufgaben). Für jedes dieser Paare wird vom *HintManager* der *TreeCompareAlgorithm* getaufter Algorithmus ausgeführt und die dadurch erhaltenen Lösungsempfehlungen und Fehler dem Statementpaar als Nachricht für den Studenten und evtl. dem Instruktor hinzugefügt.

Der *TreeCompareAlgorithmus* stellt somit das Zentrum dar und steuert den Ablauf der einzelnen Arbeitsschritte. Dieser wurde als Schnittstelle gewählt mit dem Hintergrund, später verschiedene Algorithmen für weitere Datenbanksprachen neben Oracle SQL anzubieten, zum Beispiel Microsoft ACCESS oder MySQL.

5.2.2 Erweiterung der StatementViewNode

Im Projekt werden, wie bereits erwähnt, die Klassen *StatementView* für den Statementbaum und *StatementViewNode* für die Knoten verwendet. Es gibt für die Knoten bisher kein Attribut, welches zur Speicherung der Vergleichsergebnisse im Mapping und für den Pfad möglich ist. Aus diesem Anlass wird die Entscheidung getroffen, die Klasse *StatementViewNode* um ein neues Attribut zu erweitern, welches die notwendigen Attribute als Sammlung enthält und in der folgenden Tabelle aufgezeigt werden.

Attribut	**Erklärung**
StatementViewNode mapping_node	Die Referenz auf den gemappten Knoten
double mapping_factor	Bewertung der gemappten Nodes: Von 0 bis 1

int mapping_status	Der Status der Node für die Analyse des Baumzustands.
	MAPPING_UNDEFINED = 0; // ist noch nicht geprüft worden
	MAPPING_UNIQUE = 1; // einzigartig → wichtig
	MAPPING_PLACEHOLDER = 2; // SubTree enthält mind. 1x Unique
	MAPPING_DELETEABLE = 3; // Knoten für Trim freigegeben

int nodeNumber	Erhält eine Nummer zur Identifikation. Verteilung ist Preorder. Wird zum Beispiel für den Vergleich der Position im Baum benutzt.
ArrayList<String> pathList	Enthält die Origin-Attribute der Knoten bis hin zur Wurzel. Wird als Kontext für den Pfad verwendet, z.B. für die Knotenähnlichkeit.

Alternativ hätte man auch nach der Baumgenerierung eine Transformation der Knoten durch Vererbung durchführen können und somit die Klasse *StatementViewNode* unverändert lassen.

5.2.3 STATEMENTPRÜFUNG UND BAUMGENERIERUNG

Neben dem regulären Durchlauf mit Generierung der Lösungsvorschläge kann auch die Situation auftreten, dass bei der Überprüfung der Statements durch die Komponente *CheckIdenticalStatement* oder beim Generieren des Statementbaums *TreeGenerator* ein Fehlerfall auftritt. Die folgende Abbildung beschreibt diese Fälle.

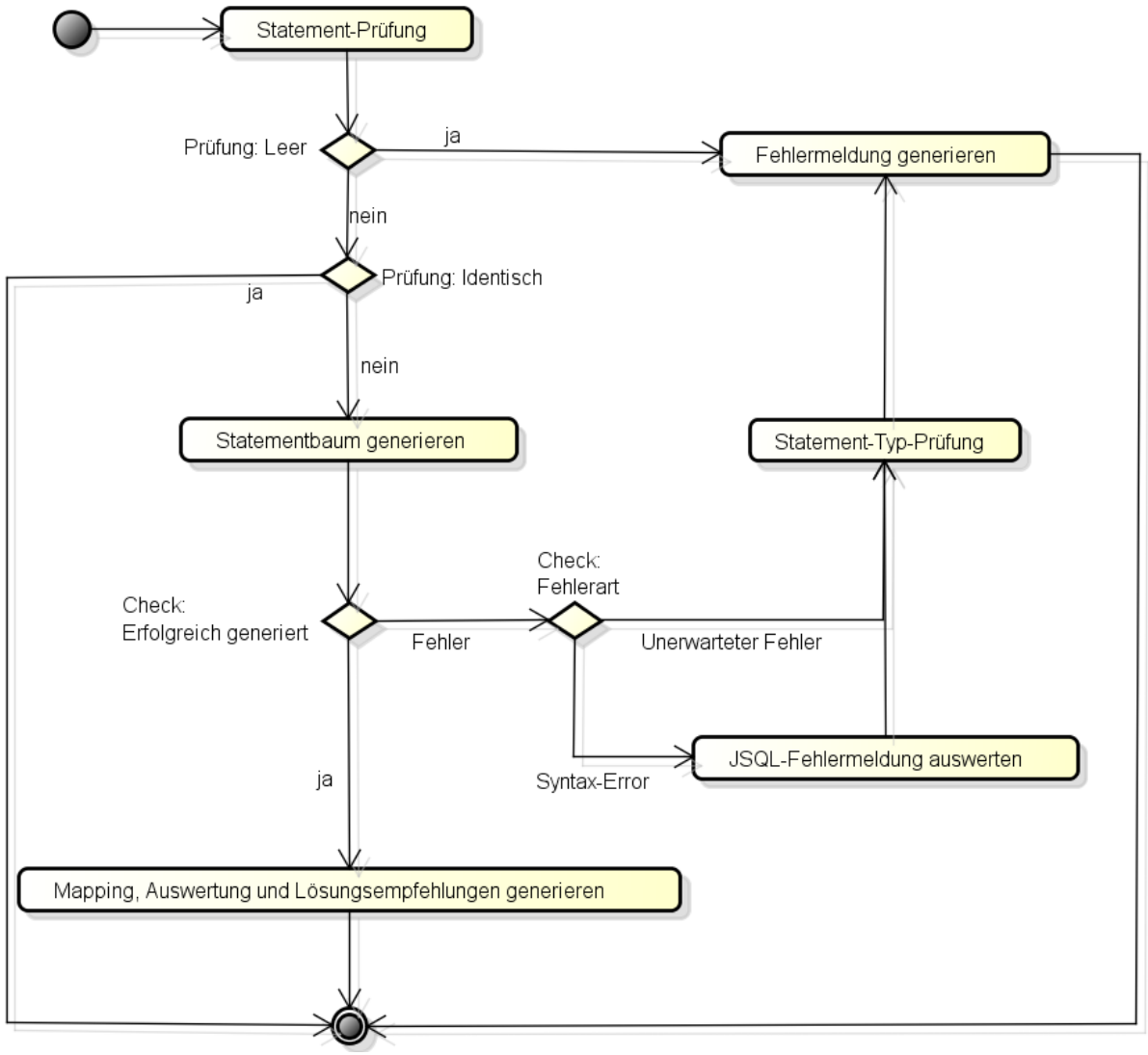

Abbildung 11: Unterschied Ablauf zwischen Normalfall und Fehlerfall

Die Statements werden zu Beginn 2-stufig überprüft, zuerst ob eines leer ist und danach ob beide identisch sind. Trifft eines davon zu, so wird die Überprüfung beendet, bei einem leeren Statement erhält der Student eine Fehlermeldung. Für den Vergleich werden die SQL-Statements gefiltert (LowerCase, doppelte Leerzeichen sowie Leerzeichen am Anfang und Ende entfernen), um kleine Abweichungen und die häufig daraus resultierende fehlerhafte Erkennung als Fehlerquelle zu reduzieren.

Im nächsten Schritt wird bei der Erstellung des Statementbaumes durch JSQL überprüft, ob dieser erfolgreich erstellt wurde. Enthält das Statement Syntax-Fehler, die von JSQL erkannt werden, erfolgt die Meldung von JSQL mit der entsprechenden Auswertung und Aufbereitung für die Fehlermeldung. Bei einem unerwarteten Abbruchsfehler durch eine *NullpointerException* wird von JSQL kein Baum und ebenso keine Meldung erzeugt, jedoch eine eigene Fehlermeldung für den Studenten. Im Falle

eines Fehlers bei der Baumgenerierung wird ein Vergleich der Statementtypen über den String durchgeführt und bei einer Abweichung eine Fehlermeldung generiert. Im Normalfall erfolgt die Untersuchung über die Knoten.

5.2.4 MAPPING

Im Arbeitsschritt Mapping wird der eigentliche Mapping-Algorithmus von anderen Teilkomponenten unterstützt.

Mapping-Algorithmus

Der bereits im Konzept vorgestellte Mapping-Algorithmus erhält den Arbeitstitel *TreeMapping*, welcher aber keinen Bezug zum gleichnamigen Algorithmus *TreeMap* bzw. *TreeMapping* von Ben Shneiderman [28] besitzt. Dieser nutzt die vorgestellte Funktion zur Berechnung der Knotenähnlichkeit *NodeSimilarityCompare* für den Vergleich und Mapping der identischen und ähnlichen Knoten.

Mapping-Status der Knoten

Mit *RateStatusTreeNodes* werden die Bäume durchwandert und für die Knoten der Mapping-Status in Abhängigkeit seiner Kindknoten gesetzt. Dies dient der Vereinfachung der späteren Auswertung. Der Mapping-Status erhält eine Unterteilung vom Initialwert UNDEFINED (noch unbewertet) zu UNIQUE (einmalig und daher nicht gemappt), PLACEHOLDER (darunter liegende Teilbäume besitzen mindestens einen einmaligen Knoten) und DELETEABLE (selbst DELETABLE und dazu keine Kinder oder alle Kinder DELETABLE). Diese Unterteilung ermöglicht die nicht mehr benötigten Teilbäume (mit dem Status DELETEABLE) zu identifizieren und dadurch schneller die abweichenden Knoten (UNIQUE) für die Analyse zu finden.

Blacklist

Beim setzen des Mapping-Status werden die Knoten zusätzlich überprüft, ob diese Elemente auf der Blacklist *NodeTypeFilter* stehen. Es wird der Status der unerwünschten Knoten markiert, welche nicht in der Untersuchung verwendet werden sollen. Dies sind oft nicht für die Untersuchung relevante Knoten, die nur für die Struktur zuständig sind.

Optional: TRIM

Für den Fall, dass keine einzige Überprüfung mehr in der Auswertung auf die löschbaren Teilbäume der Bewertung DELETEABLE durchgeführt werden sollen, können diese optional durch *TrimTrees* aus dem Baum entfernt werden statt nur durch eine Markierung gekennzeichnet zu werden. Der Vorteil wäre, dass nur noch die Elemente mit unterschiedlichen Knoten in den Bäumen bleiben, da diese Unterschiede die interessanten Bereiche für die Analyse darstellen. Vor dem Hintergrund noch weitere Analysen auf die Bäume durchführen zu können, ist die Funktion in der aktuellen Version deaktiviert und auf die Markierung der Knoten reduziert.

Überprüfung der Bäume auf vollständiges Mapping

Nach dem Mapping werden die Statementbäume überprüft, ob diese vollständig gemappt wurden und damit identisch sind. In diesem Fall wird keine Auswertung durchgeführt und die Untersuchung beendet. Dies geschieht nur bei einer Abweichung, also aufgrund der vorhandenen ungemappten Knoten.

5.2.5 AUSWERTUNG

Die Auswertung besteht aus den primären Komponenten *Collector*, der für das Sammeln und Aufbereiten der Daten aus den gemappten Bäumen zuständig ist, und dem *Analyzer* zur Analyse und Verarbeitung der Daten für die Lösungsempfehlung. Diese werden mit ihren Aufgaben, Besonderheiten und den unterliegenden Komponenten vorgestellt.

5.2.5.1 COLLECTOR

Der *Collector* hat die primäre Aufgabe, die relevanten Daten aus den Bäumen zu sammeln und aufzubereiten für die nachfolgende Analyse. Dies wird in mehreren Schritten unter Hilfe seiner Teilkomponenten durchgeführt. Er beginnt zunächst durch eine Überprüfung der Root-Knoten beider Bäume auf die Übereinstimmung des Statementtyps (z.B. SELECT, INSERT oder DROP). Im Falle einer Abweichung wird für den Studenten eine passende Fehlermeldung generiert. Durch den anschließenden *SubTreeBlocker* werden die Teilbäume nach zusammenhängenden Mustern ungemappter Knoten durchsucht und jeweils daraus ein kombiniertes *CollectorItem* erstellt. Im nächsten Schritt sucht der *Collector* aus beiden Bäumen die einzelnen ungemappten Knoten, welche die Unterschiede der beiden Statementbäume darstellen. Mit Hilfe des *NodeTypePool* extrahiert dieser die relevanten Informationen der einzelnen Knoten und erstellt daraus ein *CollectorItem*, welches Objekte mit Gruppierungsmöglichkeiten sind und im späteren Abschnitt vorgestellt werden. Abschließend werden mögliche Duplikate aus den gesammelten Informationen entfernt.

NodeTypePool

Der *NodeTypePool* beinhaltet das komplette Wissen für die einzelnen verschiedenen Ausprägungen der Knotentypen und deren relevanten, verwertbaren Informationen, die der *Collector* zur Untersuchung in Objekten der Klasse *CollectorItem* sammelt.

SubTreeBlocker

Da mit dem *NodeTypePool* nur Informationen einzelner Knoten umgewandelt werden können, werden mit Hilfe des zusätzlichen Moduls *SubTreeBlocker* ganze Teilbaummuster untersucht und mehrere Knoten zusammen geblockt in ein *CollectorItem* abgelegt. Dazu zählen ungemappte Tabellennamen mit den Spaltennamen (also [Tabelle]+[Spalte] → [Tabelle.Spalte]), sowie GROUP-BY mit den Spalten. Das Ziel von diesem Schritt ist es, die Aufteilung einer zusammenhängenden Ansammlung von Knoten in viele verschiedene kleine Teile zu reduzieren. Dies ist zur Verbesserung der Übersicht der Ausgabe gedacht, um mehr die zusammenhängenden Bereiche aus dem Statement beizubehalten, diese aber gleichzeitig analysieren und gruppieren zu können.

CollectorItem

Die Klasse *CollectorItem* wird für die Sammlung und Verarbeitung der Daten in der Auswertung verwendet. Sie wurde mit der Möglichkeit erstellt, mehrere Werte nach Kategorien eingeteilt zu sammeln und ähnliche Elemente aus beiden Bäumen zu gruppieren. Ebenfalls soll dadurch bezweckt werden, Einzelinformationen genauso wie kombinierte Informationen für aussagekräftige Lösungsempfehlung vorzubereiten.

Es sind folgende Attribute vorhanden:

- Group: Die Hauptgruppe, zum Beispiel JOIN, COLUMN, ORDER, etc
- Groupable: Entscheidet, ob das *CollectorItem* gruppierbar ist oder nicht.
- Source: Aus welchem Statementbaum stammt das Schlüsselwort oder Wert

  - aus dem Studentenbaum, was die Kategorie "Zuviel" (bzw. NEEDLESS) wäre
  - aus dem Instruktorbaum, was als Kategorie "Fehlend" (bzw. MISSING) gilt.
  - aus beiden, was der Kategorie "Verwechslung" (bzw. MISTAKE) entspricht.
- Für die Kategorie der beiden Bäume (NEEDLESS/MISSING) gibt es je folgende Attribute:
  - Subgroup: die Untergruppe, z.B. JOIN_LEFT, ALLCOLUMNS, ASC, etc.
  - Category: Die Kategorie ist hauptsächlich für die Ausgabe in Form von: *Kategorie[Werte].*
  - List: Enthält die Liste der verschiedenen zusammengefassten Werte.

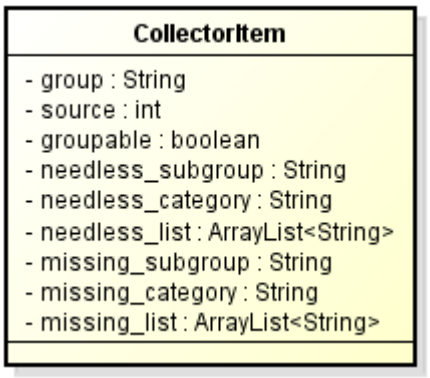

Abbildung 12: Die Attribute der Klasse CollectorItem zum Sammeln und Gruppieren der Daten in der Auswertung

5.2.5.2 ANALYZER

Im *Analyzer* werden die durch den *Collector* gesammelten *CollectorItem*s analysiert und in mehreren Schritten weiter in Richtung Lösungsempfehlung verarbeitet. Dabei wird versucht, aus mehreren Nachrichten gefiltert eine größere aussagekräftige Nachricht mit höherem Informationsgehalt zu erzeugen.

- Zur Gruppierung werden im ersten Schritt die aus demselben Statementbaum stammenden *CollectorItem* der gleichen Haupt- und Untergruppegruppe zu einem größeren *CollectorItem* zusammengeführt. Dies geschieht über Vergleiche, ob die Attribute *Source*, *Group*, und die jeweilige *Subgroup* identisch sind.
- Im zweiten Schritt werden die aus beiden Statementbäumen stammenden *CollectorItem*s miteinander über die Haupt- und Untergruppe für beide Ursprünge (*NeedlessSubGroup* und *MissingSubGroup*) miteinander verglichen und bei Übereinstimmung zusammengeführt. Wenn nun Elemente aus beiden Statementbäumen im *CollectorItem* vorhanden sind, so wird es in die neue Kategorie der Verwechslung eingeteilt, statt wie zuvor in Fehlend oder Zuviel.
- Im dritten Schritt werden nun auch die ähnlichen Gegenstücke der übrigen Kategorien Fehlend bzw. Zuviel miteinander über die Hauptgruppe zu Verwechslungen kombiniert.

Der Grund für diese Aufteilung in mehrere Schritte ist die Priorisierung der Kombinationen, um gezielt erst größere gleiche Gruppen zu erstellen und danach beim Rest die Ähnlichen zusammenzufassen. Abschließend wird präventiv eine Ausgabe von fragwürdigen Angaben verhindert, indem gespiegelte Duplikate entfernt werden, falls diese durch unvorhergesehene Auswertungen in die Auflistung geschafft haben. Ein Beispiel dafür wäre:

„Verwechslung: Statement[SELECT] statt Statement[SELECT]"

5.2.6 Lösungsempfehlungen und Nachrichten

Nachdem die Auswertung abgeschlossen ist, wird aus der Liste an *CollectorItem* jeweils die entsprechende Lösungsempfehlung in Form der eigenen Klasse *HintMessage* erzeugt. Diese werden in der Klasse *HintMessageCollection* gesammelt, zusammen mit der Liste der Fehler. Ebenso enthalten ist der Status der Überprüfung, ob dieser korrekt durchgelaufen ist, die Statements identisch oder leer waren, ob der JSQL-Parser einen Fehler gemeldet hat oder unerwartet ohne Fehlermeldung abgebrochen wurde. In der einzelnen *HintMessage* ist neben der Nachricht auch die Art der Nachricht (Zuviel/Fehlend/Verwechslung) und der Empfänger (Student oder Instruktor) notiert. Diese Vorgehensweise ist vor dem Hintergrund konzipiert worden, die Ausgabe zu einem späteren Zeitpunkt individuell formatiert anzupassen und je nach Typ gruppieren zu können.

In aSQLg werden Nachrichten der Klasse *Message* bzw. *MessageCollection* verwendet, welche zwar einen ähnlichen Ansatz verfolgen, jedoch genug Abweichung von der *HintMessage* sowie *HintMessageCollection* besitzen, damit die Transformation getätigt werden muss. Da sich das neue Projekt in das existierende aSQLg als neues Modul integriert, werden im *HintManager* die Liste der *HintMessage* in die Klasse *Message* transformiert und zu den Nachrichten der Statements eingetragen.

5.3 Einschätzung der Implementierung

Die Implementierung erfolgt auf Basis des Konzepts und hat die einzelnen Arbeitsschritte mit Subkomponenten für die einzelnen Aufgaben erweitert. Das Konzept geht auf, die Unterschiede der Bäume werden untersucht und über die Auswertung erfolgreich zur Generierung von Lösungsempfehlungen genutzt. Ein Vorteil der Implementierung ist auch die Unabhängigkeit von einer Datenbank für die Vergleiche der Statements. Das Mapping hat in der Entwicklung einen guten Eindruck gemacht. Nur eine einzelne falsch gemappte Klammerung ist während der Implementierung aufgefallen. Das vollständige Wissen zu den einzelnen Knoten mit den Typen und deren Inhalt ist in der Klasse *NodeTypePool* gesammelt, mit dem die Daten aus den Knoten extrahiert werden und somit als Bindeglied zwischen Knoten und Informationen zur Weiterverarbeitung fungiert. In der Implementierung ist der Bereich der semantischen Prüfung jedoch noch im ausbaufähigen Zustand. Dies resultiert u.a. daraus, dass das Wissen aus der Publikation für die Liste der Semantischen Fehler erst spät in der Implementierung eingeflossen ist. Es fehlt ebenso eine konkrete umfangreiche Lösung zur Bildung und Vergleich von Äquivalenzen bezüglich der Teilstatements. Im nachfolgenden Kapitel wird dieses Thema durch die Tests genauer analysiert.

6 TESTS

Das Testen der Implementierung erfolgt durch die Überprüfung mit einer eigenen Testsammlung von Statement-Paaren, die ihren Fokus auf die verschiedenen Elemente der SQL-Syntax legen.

Die Sammlung enthält zum Beispiel folgende Tests:

- Tests mit korrekter Syntax zur Analyse der Lösungsempfehlungen. Ziel ist es, die Schwachstellen der Implementierung zu finden
- Tests mit inkorrekter Syntax zur Überprüfung, ob der Fehler gefunden wird und welche Nachricht für den jeweiligen Fehler erzeugt wird
- Spezielle Tests zur Suche nach unvollständiger SQL-Interpretation und Problemen durch JSQL
- Spezielle Tests zur Analyse und Einschätzung der Laufzeit für die jeweiligen Fälle sowie Suche nach vorhandenen Grenzen und Problemfällen

6.1 ERGEBNIS-BEISPIELE

Um einen Eindruck für die Resultate der durchgeführten Tests zu bekommen, erfolgt die Vorstellung eines ausgewählten Auszugs an Beispielen aus der Testsammlung.

6.1.1 NORMALE TESTFÄLLE

Test-Nr	#1
Statement Stud.	SELECT DISTINCT name, price, stocks FROM products WHERE stocks>1 ORDER BY price
Statement Instr.	SELECT id, name, price FROM products WHERE stocks>0 ORDER BY price DESC
Erwartete Meldungen	Zuviel: DISTINCT Verwechslung der Spalten: stocks statt id gefunden Verwechslung: Wert 1 statt 0 gefunden Verwechslung: Sortierkriterium ASC statt DESC gefunden
Ausgabe	Zuviel: SELECT[DISTINCT] Verwechslung: Spalte[stocks] statt Spalte[id] Verwechslung: Long[1] statt Long[0] Verwechslung: ORDER[ASC] statt ORDER[DESC]

Das erste Beispiel zeigt einen einfachen Test mit Standardfehlern, bei dem überflüssige und fehlende Elemente korrekt erkannt werden. Diese werden zu gemeinsamen Auflistungen der Kategorie Verwechslungen kombiniert.

Test-Nr	#2
Statement Stud.	SELECT * FROM L LEFT JOIN R on L.id=R.id WHERE x = 42
Statement Instr.	SELECT * FROM L RIGHT JOIN R ON L.id=R.id WHERE x = '42'
Erwartete Meldungen	Verwechslung des JOIN:LEFT-statt RIGHT Verwechslung des Datentyp: Long 42 statt String '42'
Ausgabe	Verwechslung: JOIN[LEFT JOIN] statt JOIN[RIGHT JOIN] Verwechslung: Long[42] statt String[42]

Das zweite Beispiel zeigt erneut die erfolgreiche Auswertung von Standardfehlern, wobei hier anstelle ganzer Elemente hier der Typ der Abweichung der Elemente erkannt und in der Lösungsempfehlung ausgegeben wird.

Test-Nr	#3
Statement Stud.	SELECT product, SUM(price)/COUNT(price) FROM sells WHERE category = '%tool_%'
Statement Instr.	SELECT product, AVG(price) FROM sells WHERE category LIKE 'tool_%' GROUP BY product
Erwartete Meldungen	Verwechslung von Durchschnittsfunktion von price: SUM/COUNT statt AVG Verwechslung des Vergleichsoperator = statt LIKE und Abweichung im Ausdruck Fehlende Gruppierung über die Spalte product
Ausgabe	Fehlend: GROUPBY[product] Zuviel: Division[/] Verwechslung: Funktion[SUM,COUNT] statt Funktion[AVG] Zuviel: Klammern[()] Zuviel: Spalte[price] Verwechslung: Vergleich[=] statt Vergleich[LIKE] Verwechslung: String[%tool_%] statt String[tool_%]

Der dritte Test erkennt zwar die einzelnen Abweichungen, listet diese jedoch diesmal teilweise in getrennte Lösungsempfehlungen auf. Obwohl der Vergleichsoperator und der Ausdruck auch getrennt voneinander Sinn machen, führt die Aufteilung der Durchschnittsfunktion auf 4 Nachrichten zu einer verringerten Übersichtlichkeit für den Studenten. Die Nennung der Funktionen hat hier Priorität für das Verständnis der Lösungsempfehlung. Die Details mit Nennung der überflüssigen Klammern oder der Spalte price sind eher nachrangig.

Bei der Durchschnittsfunktion ist anzumerken, dass die Funktion AVG Null-Werte ausschließt und somit nur mit äquivalent mit SUM/COUNT wäre bei zusätzlicher passender Bedingung in WHERE. Dies zeigt, wie sich der reine Vergleich auf Baumunterschiede und die semantische Überprüfung mit

zusätzlichem Hintergrundwissen sich voneinander unterscheiden kann. Außerdem wird deutlich, dass die Komplexität der Überprüfung sich deutlich erhöhen kann, wenn solch ein Wissen in die Überprüfung mit einfließen soll.

Test-Nr	#4
Statement Stud.	SELECT * FROM products WHERE x>=0 AND x<=5
Statement Instr.	SELECT id, name FROM products WHERE X BETWEEN 0 AND 5
Erwartete Meldungen	Verwechslung der Spalten: * statt id und name. Verwechslung beim Vergleich: [x>=0 AND x<=5] anstatt [X BETWEEN 0 AND 5]
Ausgabe	Verwechslung: Spalte[*,x] statt Spalte[id,name] Zuviel: Verkettung[AND] Verwechslung: Vergleich[>=,<=] statt Vergleich[BETWEEN] Fehlend: Auflistung[Komma]

Der vierte Test zeigt erneut das Problem, wenn ein Statement-Abschnitt auf mehrere Zeilen der Lösungsempfehlungen aufgeteilt wird, abhängig vom jeweiligen Typ.

Die Auflistung mit dem Komma zeigt hier auch erneut, dass bestimmte Informationen geringere Prioritäten haben und teilweise aus anderen Informationen ableitbar sind, im Test #4 sind dies mehrere vorhandene Abweichungen der Spalten. Eine verbesserte Erkennung von äquivalenten Elementen würde Sinn machen, um auch Alternativen vom Studenten zu akzeptieren und nicht als Fehler zu bewerten Es müssten daher kleine tolerierte Abweichungen in den Alternativen speziell gesucht werden, so wäre z.B. die Angabe `<6` mit Datentyp Long anstelle von `<=5` als gleichwertig akzeptierbar, während alle anderen Unterschiede wie `<4` weiterhin als Abweichung verbleiben.

6.1.2 Testfälle mit Problemen in der Erkennung von Fehlern

Test-Nr	#5
Statement Stud.	SELECT * FROM L LEFT JOIN R on L.id=R.id WHERE 2+x>y GROUP BY g ORDER BY a, o
Statement Instr.	SELECT * FROM R RIGHT JOIN L ON R.id=L.id WHERE x>2+y GROUP BY g,a ORDER BY o
Erwartete Meldungen	Verwechslung der Position der Spalte [a] in ORDER-BY statt GROUP-BY
Ausgabe	Verwechslung: JOIN[LEFT JOIN] statt JOIN[RIGHT JOIN]

Der Test Nummer 5 zeigt die Schwäche des Mappings mit der fehlenden Berücksichtigung der Reihenfolge. Während im zweiten Test nur der JOIN-Typ unterschiedlich ist, ist hier die Reihenfolge für Tabelle und JOIN -Typ zusammen umgedreht und damit wieder äquivalent und damit korrekt. Dies wird jedoch in diesem Test nicht erkannt und daher als Abweichung bewertet. Der Fehler durch die falsche Position der Spalte a in ORDER BY statt GROUP BY wird hier fälschlicherweise nicht erkannt.

Test-Nr	#6
Statement Stud.	SELECT a.nr, a.name FROM artikel a WHERE EXISTS (SELECT * FROM auftrag af WHERE a.nr=af.nr)
Statement Instr.	SELECT a.nr, a.name FROM artikel a JOIN auftrag at ON a.nr=at.nr
Erwartete Meldungen	Subquery mit EXIST anstelle JOIN
Ausgabe	Zuviel: Ausdruck[EXISTS] Zuviel: Statement[SubQuery,SELECT] Zuviel: Spalte[*] Verwechslung: AS[af] statt AS[at] Fehlend: JOIN

Im Test Nummer 6 wird gezeigt, wie bei einer Unteranfrage EXISTS statt des eleganteren JOIN verwendet wird und sich diese Abweichung dadurch auf mehrere Nachrichten verteilt. Das Problem ist, dass abweichende Elemente der Unteranfrage komplett aufgelistet und diese nicht sinngemäß als ein gesamtes aggregiertes Element zum Vergleich genutzt werden.

6.1.3 Fehler ohne Baumvergleich

Die bisherigen Beispiele zeigen Fälle von gültiger Syntax, bei denen die Bäume gemappt und ausgewertet werden können. Dieser Abschnitt zeigt verschiedene Beispiele mit Fehlern, die einen Baumvergleich verhindern, und beschreibt, welche Lösungsempfehlungen daraus generiert werden.

Test-Nr	#7
Statement Stud.	SELECT DISTINCT 123, 234, 456 AS example FROM tblx WHERE (X=1 AND Y=2)
Statement Instr.	DROP TABLE tblx
Erwartete Meldungen	Verwechslung StatementType: SELECT statt DROP
Ausgabe	- Falscher Statementtyp: [SELECT] statt [DROP]

Der 7. Test zeigt ein Beispiel, bei dem der Student das falsche Statement für die Aufgabe eingereicht hat und dies zu unterschiedlichen Statementtypen führt. Dies wird auch bei gültiger Syntax erkannt und eine passende Lösungsempfehlung erzeugt.

Test-Nr	#8
Statement Stud.	SELCT a form b were x=2
Statement Instr.	SELECT a from b where x=2
Erwartete Meldungen	Syntaxfehler durch Rechtschreibfehler - SELCT statt SELECT - form statt FROM - were statt WHERE
Ausgabe	Gefunden: [<S_IDENTIFIER> SELCT] in Zeile [1] Spalte[1]. Erwartet: [DROP,WITH,DELETE,SELECT,INSERT,UPDATE,REPLACE,TRUNCATE,ALTER,EXEC,EXECUTE,(] Falscher Statementtyp: [SELCT] statt [SELECT]

Im Test Nummer 8 liegen mehrere Rechtschreibfehler vor. Der dadurch entstandene Syntaxfehler wird durch den Parser erkannt und eine passende Lösungsempfehlung generiert. Da die Statementtypen sich dadurch unterschieden, wird hierfür ebenfalls eine entsprechende Nachricht erzeugt.

Test-Nr	#9
Statement Stud.	SELECT * FROM tblx WHERE x='A' OR x='B' OR x='C'
Statement Instr.	SELECT * FROM tblx WHERE x IN ('A', 'B', 'C')
Erwartete Meldungen	+IN statt Liste[= & OR]
Ausgabe	- <NullPointerException> Fehler beim erzeugen des Instruktor-Baums. Bitte Instruktor melden (@Student) - <NullPointerException> Fehler beim erzeugen des Instruktor-Baums. (JSQL-Parser unvollständig?) (@Instruktor)

Der 9. Test zeigt ein Problem mit Elementen aus der SQL-Syntax, die zu unerwarteten Abbrüchen (mit *NullpointerException*) im JSQL-Parser führen. Dies weist auf die unvollständige Abbildung der SQL-Sprache hin. Durch den Abbruch sind weder Informationen vorhanden, an welche Stelle das Problem auftritt, noch wird ein Statementbaum generiert, nicht mal ein unvollständiger. Da die Ursache des Problems im Statement des Instruktors liegt, erfolgt in diesem Fall die spezielle Meldung an den Studenten, sich an den Instruktor zu wenden. Der Instruktor erhält ebenfalls eine Mitteilung über den Fehler.

Test-Nr	#10
Statement Stud.	SELECT a FROM b WHERE x='c'
Statement Instr.	SELECT d,e FROM f WHERE y='g' OR y=z
Erwartete Meldungen	Verwechslung der Spalten [a,b,x] statt [d,e,y,z] Verwechslung der Tabellen: [b] statt [f] Verwechslung des Hochkommata mit Apostroph und Inhalt [c] statt [g]
Ausgabe	Nicht vorhanden

Test Nummer 10 enthält ebenfalls ein Problem mit unerwarteten Abbrüchen, jedoch diesmal in kritischem Ausmaß durch ein unerwartetes Sonderzeichen ausgelöst. Es wird weder eine JSQL-Fehlermeldung erzeugt oder ein Statementbaum generiert noch eine Exception erzeugt. Der Algorithmus beendet für dieses Statement komplett die Untersuchung. Die genaue Ursache und Lösung konnte bisher nicht ermittelt werden.

6.2 Untersuchung zur Erkennung von Semantikfehler

Zur Untersuchung der Semantikfehler wird die bereits erwähnte zusammengefasste Liste der Semantikfehler [16] von Christian Goldberg verwendet. Es erfolgt eine Beurteilung, ob und wie gut diese Fehler erkannt werden.

Rank	Semantic Error
1	Fehlende JOIN-Bedingung
2	Viele Duplikate
3	Unnötiger JOIN
4	Inkonsistente Bedingung
5	Unnötiges Argument von COUNT
6	Implizite, tautologische (Aussagenlogik ergibt immer wahr) oder inkonsistente Unterbedingung
7	unnötiges DISTINCT
8	Ineffizientes UNION
9	Unbenutzte Tupel-Variablen
10	Konstante Spalte in der Ausgabe

Die Liste der Semantikfehler bezieht sich auf die Auswertung eines einzelnen Statements mit Kenntnis des SQL-Schemas. Durch den Vergleich des Studentenstatements mit der Musterlösung des Instruktors als Vorgabe werden die vorhandenen Informationen ohne Kenntnis des SQL-Schemas durchgeführt. So entstehen andere Vor- und Nachteile durch die Abweichung dieser beiden Auswertungen.

Rank#1: Fehlende JOIN-Bedingung

Der häufigste Fehler der Auswertung gehört beim klassischen JOIN zu den gut erkannten Fehlern der Kategorien Fehlend und Überflüssig. Durch die Vorgabe der Musterlösung wird der Unterschied auch dann sicher gefunden, auch wenn der Student eine Alternative wie Unterabfrage nutzt.

Rank#2: Viele Duplikate

Mit vielen Duplikaten sind doppelt bzw. mehrfach gleiche Ergebniszeilen gemeint. Das Ergebnis der Abfragen wird hier nicht geprüft, dies erfolgt nur im Grader. Eine Prüfung der Schlüssel für die Spalten durch das SQL-Schema wird auch nicht durchgeführt, dafür werden aber die Unterschiede für die jeweiligen Spaltenelemente, DISTINCT, JOIN und z.B. Gruppierung durch den Vergleich mit der Musterlösung gefunden.

Rank#3: Unnötiger JOIN

Die Vorgabe des Instruktors trifft die Entscheidung, ob ein JOIN unnötig ist. Vom JOIN im Rank#1 unterscheidet sich der dritthäufigste Fehler nur in der Kategorie der Fehlermeldung, also zwischen Fehlend und Überflüssig.

Rank#4: Inkonsistente Bedingung

Bei der Überprüfung werden die Bedingungen nicht auf Konsistenz überprüft, sondern nur, ob diese im anderen Statementbaum vorhanden sind oder nicht. Falls die Inkonsistenz durch eine vertauschte Reihenfolge der gleichen Elemente erfolgt, so wird dies ebenfalls nicht erkannt.

Rank#5: Unnötiges Argument von COUNT

Dieser Fehler wird ebenfalls durch die Kategorie Überflüssig gut gefunden.

Rank#6: Implizite, tautologische oder inkonsistente Unterbedingung

Diese Fehler sind wie Rank#4 zu bewerten und werden nicht in der Logik, sondern nur auf Vorhandensein überprüft.

Rank#7: Unnötiges DISTINCT

Dieser Fehler wird ebenfalls in der Kategorie Überflüssig aufgeführt.

Rank#8: Ineffizientes UNION

Durch Vorgabe des Instruktors, ob UNION oder UNION ALL zu nutzen ist, lässt sich dieser Fehler leicht als Baumunterschied finden.

Rank#9: Unbenutzte Tupel-Variablen

Zwar werden überflüssige Spalten und Tabellen gefunden, jedoch zählt die Verwendung von Alias und optionalen Tupeln für Tabellen und Spalten zu den Schwachstellen der aktuellen Umsetzung. Da sich im Statementbaum die jeweiligen Knoten zu denen ohne Alias sehr voneinander unterscheiden, erfolgt hier kein Mapping. Dies wird dann bei kleinsten Unterschieden als Abweichung bewertet und in die Lösungsempfehlungen aufgenommen.

Rank#10: Konstante Spalte in der Ausgabe

Dieser Fehler wird während des Vergleiches mit dem Instruktorstatement gefunden, eine explizite Suche nach diesem Fall erfolgt nicht.

6.3 Fehler in JSQL

Die durchgeführten Tests zeigen die Fehler und Lücken vom JSQL-Parser bzw. dem *JsqlStatementViewBuilder* mitsamt den Visitor-Klassen. Die Auslöser sind ignorierte Schlüsselwörter, unvollständige Abschnitte des SQL-Umfangs, kleinere Rechtschreibfehler und Sonderzeichen. Die nachfolgende Tabelle gibt einen Ausschnitt Überblick. Die vollständige Liste der JSQL-Probleme wird im separaten Handbuch aufgezeigt.

Ausdruck	Fehler-Beschreibung
NOT	Schlüsselwort **NOT** ausserhalb von **IS NOT NULL** wird vom JSQL-Parser ignoriert und nicht in dem Baum aufgenommen.
IN	Schlüsselwort **IN** wird ignoriert: „*WHERE x **IN** (1,2,3)*“
PERCENT	Schlüsselwort **PERCENT** wird ignoriert: „*SELECT TOP 5 **PERCENT***“
REFERENCES **FOREIGN KEY**	Schlüsselworte **REFERENCES** und **FOREIGN KEY** werden im **CREATE**-Statement ignoriert. Fälschlicherweise liefert die Abfrage zum Knoten des Fremdschlüssel “*getValue("isPrimaryKey").equals("true")*” einen vorhandenen Primärschlüssel, eine Funktion für Fremdschlüssel existiert nicht. Der Teilbaum für **REFERENCES** ist nicht vorhanden. „*CREATE TABLE tblx (id int PRIMARY KEY, fk INT, name varchar(255) NOT NULL, nickname varchar(255) UNIQUE, **FOREIGN KEY** (fk) **REFERENCES** tblfk(id))*“
ALTER	Es fehlt ein *AlterViewGenerator.java*, bisher wird nur „***ALTER** TABLE tbl **ADD** col type*“ unterstützt, es fehlt jedoch die Unterstützung für **DROP**, **ALTER** und **MODIFY**.
AS	Der *AliasSolver* ignoriert die ersten beiden Alias „a“ und „b“, wenn sie nochmal ungültig verwendet werden. Besser wäre eine Fehlermeldung vom JSQL-Parser anstelle eines unvollständigen Baumes. „*SELECT **a, b** FROM (SELECT fk_id as **a**, COUNT(fk_id) as **b**)*“
Sonderzeichen	Zum Abbruch des JSQL-Parser ohne Fehlermeldung oder Statementbaum führen nicht vorgesehene Sonderzeichen wie zum Beispiel ein alternatives Apostroph [’] oder Zeichen wie „$crazy{1%}“ anstelle eines Spaltennamens.

6.4 Analyse der Laufzeit

In den vorangegangenen Tests stand die Effektivität des Prüfablaufes im Vordergrund, ob die abgegebenen Fehlermeldungen den Erwartungen entsprechen. In diesem Abschnitt erfolgt mit der Laufzeitanalyse die Untersuchung der Effizienz für die Testsammlung, in der eine Einschätzung zur allgemeinen Laufzeit und Suche nach Grenzen sowie Problemfälle erfolgt.

Als Testsystem dient ein Intel i7-4790K (4x 4.40 Ghz) mit 32GB Ram, auf dem unter Windows7 die Tests in der Entwicklungsumgebung NetBeans mit Java8 durchgeführt werden. Die Testsammlung wird als Liste nacheinander durchgeführt, die Normalfälle und damit ein großer Teil der Sammlung liegen jeweils bei einer Ausführungszeit von 1-8ms. Durch die JavaVM ist die Ausführung des ersten Tests langsamer, benötigt danach durch Optimierungen jedoch meist nur noch einen Bruchteil der Zeit bei erneuter Ausführung. Die Zeitangaben der verwendeten Zeitmessung sind nur Annäherungswerte und basieren aus mehrfacher Ausführung, um Verfälschungen durch Hintergrundprogramme zu senken. Eine exakte Analyse der Laufzeit würde z.B. durch diverse Java-Profiler erfolgen können, wird jedoch in dieser Masterarbeit nicht durchgeführt. Für die Laufzeitanalyse sind die Extremfälle mit aufwändigeren Statementpaaren interessant, um u.a. die Grenzen der Überprüfungsdauer aufzuzeigen und Angriffe auf die Verfügbarkeit zu simulieren. Zu den Extremfällen gehören z.B. sehr lange Statements mit vielen verschiedenen Elementen der SQL-Syntax und mit stark verschachtelten Unterabfragen.

6.4.1 Wichtige Komponenten bei der Laufzeitmessung

Für die Laufzeitmessung werden folgende Komponenten beobachtet: der JSQL-Parser für die Baumerzeugung, das Mapping und der Auswertungsschritt mit Nachrichtenerzeugung. Die Untersuchung ergab, dass der Auswertungsschritt in keinem Testfall relevante Anteile an der Laufzeit hat und minimal blieb. Im Großteil der Fälle hat JSQL den primären Anteil an der Gesamtdauer, in einigen wenigen das Mapping. Beim JSQL-Parser hat die Baumgenerierung keinen zusammenhängenden Effekt zwischen den Statements, jedoch beim Mapping. Fälle mit höherer Laufzeit wären zwei sehr große, unterschiedliche Bäume mit vielen verschiedenen Knoten. Sind die Bäume sehr ähnlich oder ein Baum sehr klein, so ist die Laufzeit sehr gering. Da der Instruktor kaum ein riesiges, künstlich verlängertes Statement verwendet, treffen hier eher selten zwei solcher Statements aufeinander und die Gefahr für so einen Worstcase ist daher eher gering. Im Mapping und in der Auswertung werden nur syntaktisch korrekte Statements untersucht, da der JSQL-Parser nur für diese die Statementbäume erstellt. Bei der Analyse der Laufzeitmessung vom JSQL-Parser ist auffällig, dass fehlerhafte Statements in der Regel eine höhere, fallweise sogar deutlich verlängerte Dauer besitzen als gültige Statements. Außerdem zeigt der Zeitaufwand deutliche Unterschiede je nach Art des Fehlers und der auftretenden Position. Zum Beispiel benötigen bei starken Verschachtelungen die inneren Fehler tendenziell länger als die äußeren. Besonders auffällig ist die erhöhte Laufzeit von einem stark verschachtelten Statement mit leeren Klammern „()“ in der innersten Unterabfrage gegenüber anderen Fehlern.

6.4.2 Beispiele von Problemfällen

Test-Nr	#11
Statement Stud.	SELECT P.PNO, P.DESCRIPTION FROM PHOTO P WHERE EXISTS(SELECT * FROM BELONGSTO B1, SUBTOPIC S1 WHERE B1.MID=S1.MID AND B1.SID=S1.SID AND B1.PNO=P.PNO AND S1.NAME='Nina') AND EXISTS(SELECT * FROM BELONGSTO B2, SUBTOPIC S2 WHERE B2.MID=S2.MID AND B2.SID=S2.SID AND B2.PNO=P.PNO AND S2.NAME='Lisa')
Statement Instr.	SELECT P.PNO, P.DESCRIPTION FROM PHOTO P, BELONGSTO B1, SUBTOPIC S1, BELONGSTO B2, SUBTOPIC S2 WHERE B1.MID=S1.MID AND B1.SID=S1.SID AND B2.MID=S2.MID AND B2.SID=S2.SID AND B1.PNO=P.PNO AND S1.NAME='Nina' AND B2.PNO=P.PNO AND S2.NAME='Lisa'
Ausgabe	Zuviel: Ausdruck[EXISTS] Zuviel: Statement[SubQuery,SELECT] Zuviel: AlleSpalten[*] Zuviel: JOIN[SIMPLE JOIN]
Laufzeit	Dauer JSQL: 3ms Dauer Mapping: 52ms Dauer Auswertung und Nachrichten: 0ms

Der Test Nummer 11 aus der Sammlung für die Semantifehler zeigt ein gleichwertiges Beispiel, bei dem der Instruktor die Lösung eleganter mit einem JOIN statt EXIST auf eine Unterabfrage gelöst hat. Es stellt einen der seltenen Fälle dar, wo das Mapping eine deutlich höhere Laufzeit gegenüber JSQL vorweist.

Test-Nr	#12
Statement Stud.	SELECT DISTINCT *, a as A,aa as AA,aaa as AAA,aaaa as AAAA FROM (SELECT b as B, bb as BB, bbbb as BBB, bbbb as BBBB FROM (SELECT c as C, cc as CC, ccc as CCC, cccc as CCCC FROM (SELECT d as D, dd as DD, ddd as DDD, dddd as DDDD FROM (SELECT e as E,ee as EE,eee as EEE,eeee as EEEE FROM (SELECT f as F, ff as FF, fff as FFF, ffff as FFFF FROM (SELECT g as G, gg as GG, ggg as GGG, gggg as GGGG FROM (SELECT h as H, hh as HH, hhh as HHH, hhhh as HHHH FROM (SELECT i as I, ii as II, iii as III, iiii as IIII FROM (SELECT j as J, jj as JJ, jjj as JJJ, jjjj as JJJJ FROM correct WHERE x0=null) WHERE x1=1) WHERE x2=2) WHERE x3=3) WHERE x4!=4) WHERE x5>=5 OR x5 IS NULL) WHERE x6<=6 and x6 IS NOT NULL) WHERE x7<7 OR 7=7) WHERE x8>8 AND 8>8) WHERE x9!=9 OR 9=9 ORDER BY notexist DESC
Statement Instr.	SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM b WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2
Ausgabe	Zuviel: SELECT[DISTINCT] Verwechslung: Spalte[*,notexist,x9,x8,x7,x6,x5,x4,x3,x2,x1,x0,jjjj,jjj,jj,j,iiii,iii,ii,i,hhhh,hhh,hh,h,gggg,ggg,gg,g,ffff,fff,ff,f,eeee,eee,ee,e,dddd,ddd,dd,d,cccc,ccc,cc,c,bbbb,bb,b,aaaa,aaa,aa] statt Spalte[a,x] Zuviel: Auflistung[Komma] Zuviel: AS[A,JJJJ,JJJ,JJ,J,IIII,III,II,I,HHHH,HHH,HH,H,GGGG,GGG,GG,G,FFFF,FFF,FF,F,EEEE,EEE,EE,E,DDDD,DDD,DD,D,CCCC,CCC,CC,C,BBBB,BBB,BB,B,AAAA,AAA,AA] Verwechslung: Tabelle[correct] statt Tabelle[b] Verwechslung: Long[1,9,8,7,6,5,4,3] statt Long[2] Vergleich[!=,>,<,<=,>=] statt Vergleich[=] Verkettung[OR,AND] Zuviel: Ausdruck[IS NULL,IS NOT NULL] Zuviel: ORDER[DESC] Fehlend: Statement[SubQuery,SELECT]
Laufzeit	Dauer JSQL: 1200ms (ca.20ms Student und ca. 1180ms Instruktor) Dauer Mapping: 2288ms Dauer Auswertung und Nachrichten: 1ms

Der 12. Test zeigt eines der Beispiele, bei dem die Grenzen für JSQL und für das Mapping nahezu erreicht werden. Interessant zu beobachten ist die deutlich voneinander abweichende Dauer der 2 Statements. Während in der das längere Statement des Studenten mit vielen verschiedenen Elementen mit 20ms noch im akzeptablen Bereich liegt, erreicht das einfache, aber stark durch

Unterabfragen verschachtelte und von Wiederholungen der gleichen Elemente gekennzeichnete Instruktorstatement mit 1180ms eine langsam problematische Laufzeitdauer.

Test-Nr	#13
Statement	SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM (SELECT a FROM () WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2) WHERE x=2
Laufzeit	Nach 60 Minuten abgebrochen

Im 13. Test wird das problematische Statement aus Test Nr. 12 mit dem Fehler von leeren Klammern ergänzt, für welches JSQL auch selbst nach einer Stunde keinen Statementbaum erzeugen konnte und daher manuell abgebrochen wird. Eine Ausgabe lag deshalb auch nicht vor.

6.5 TEST-FAZIT

Die Testreihen zur Überprüfung der Implementierung haben Stärken und Schwächen bei der Fehlererkennung in Syntax und Semantik deutlich gemacht sowie Fehler in JSQL und Grenzen der Laufzeit.

Die Stärken der Umsetzung liegen eindeutig in der zuverlässigen und schnellen Erkennung von Standardfehlern. Ebenso wird der überwiegende Teil der semantischen Fehler aufgrund der Vorgabe durch das Instruktorstatement implizit als Unterschied zwischen den Bäumen erkannt.

Die Nachteile liegen im Detail der strukturell anspruchsvolleren Anfragen mit z.B. Relevanz der Reihenfolge. Spezielle Testreihen widmen sich daher gezielt den verschiedenen Schwächen, die auf die verschiedenen Bereiche Syntax, Semantik und Laufzeit gerichtet sind und diese aufzeigen sollen.

Für den Großteil der Syntaxfehler ist durch JSQL eine Fehlermeldung vorhanden. In einigen Situationen zeigen sich aber auch die systembedingten Schwächen vom JSQL-Parser und die Fehlermeldung trotz klarer Abweichung ausbleibt. Dies wird z.B. ausgelöst durch unerwartete Abbrüche mit einer *NullpointerException*, die abgefangen und mit einer eigenen Fehlermeldung mitgeteilt wird. Zum kritischen Abbruch sind spezielle Sonderzeichen der Auslöser, in denen sogar die gesamte Prüfung ohne Warnung übersprungen wird und in der Zukunft genauer untersucht werden sollte.

7 BEWERTUNG

In diesem Kapitel wird primär das Ergebnis des Konzepts und der Implementierung mit den Testergebnissen evaluiert. Es werden die Stärken und Schwächen des Systems kritisch analysiert und aufgezählt. Auf Basis dieser Auswertung erfolgt für eine mögliche Zukunft des Projekts im Folgekapitel Weiterentwicklung eine Liste an Vorschlägen für Korrekturen sowie Ideen für Verbesserungen bezüglich der Schwächen.

7.1 ERFÜLLUNG AUFGABENSTELLUNG

Zunächst wird jedoch anhand der Aufgabenstellung aus Kapitel 1.2 geschaut, ob die Bestandteile des Projekts erfüllt sind. Die einzelnen Punkte der Aufgabenstellung werden betrachtet und in der folgenden Tabelle eingeschätzt:

Teil der Aufgabenstellung	Ergebnis
[…]mögliche Erweiterung für das Werkzeug aSQLg zur automatisierten Prüfung von SQL-Statements[…] Die beste dieser Varianten soll dann implementiert und in das Werkzeug aSQLg integriert werden	Das Projekt wurde in aSQLg als zusätzliches Modul neben dem Grader integriert.
[…]Statement der Musterlösung als auch das Statement des Studierenden in eine Baum-/Graph-Repräsentation zu übersetzen.	Durch JSQL wird das Statement in einen Statementbaum übersetzt, der im Projekt verwendet wird.
[…]mithilfe von Ähnlichkeiten dieser Bäume/Graphen sowohl die Korrektheit der Lösung wie auch ggfs. Hinweise zur Verbesserung bei nicht korrekten Lösungen zu erzeugen.	Bäume werden miteinander verglichen und Lösungsempfehlungen aus den Abweichungen erzeugt.
Eine solche Ähnlichkeitsprüfung von Graphen kann möglicherweise mithilfe der existierenden Graph-Anfragesprache GReQL erfolgen oder auch auf andere Art und Weise.	Analyse von GReQL wurde durchgeführt, jedoch dagegen entschieden und eine andere Lösung gewählt.
[…]verschiedene Möglichkeiten solcher Repräsentationen und der zugehörigen Ähnlichkeitsoperationen verglichen werden.	Die Strukturen Baum und Graph wurden miteinander auf ihren Einsatz verglichen. Zudem wurden auch Algorithmen für Baumvergleiche analysiert, jedoch für die Entwicklung eines eigenen entschieden.
Dabei soll sowohl die Möglichkeit der Korrektheitsprüfung wie auch die Möglichkeiten zur Unterstützung der Studierenden bei der Lösungsfindung berücksichtigt werden.	Die Korrektheit wird bereits durch den Grader geprüft und die Syntax der SQL-Struktur wird durch den JSQL-Parser nochmals geprüft. Fehler in syntaktisch Korrekten Statements ist durch eine Überprüfung in den Tests enthalten.

7.2 Einschätzung der Stärken

7.2.1 Ergebnis des Projekts

Das Werkzeug aSQLg wurde erfolgreich um ein zusätzliches lauffähiges Modul erweitert, welches eine automatisierte Prüfung von Studentenstatements mit der Musterlösung des Instruktors durchführt und Lösungsempfehlungen für die Auswertung der Unterschiede erstellt. Der Student erhält so eine nützliche Hilfe, mit der er sich der Musterlösung eigenständig annähern kann, jedoch ohne die vollständige Lösung zu erhalten. Die Lösungsempfehlungen werden in einer strukturierten Ausgabe präsentiert, die über keine Duplikate verfügen. Es sind nicht nur Lösungsempfehlungen bei syntaktisch korrekten Statements mit Abweichungen, sondern auch Fehlermeldungen für SQL-Syntaxfehler vorhanden. Die Ausführungszeit für die Berechnung der Lösungsempfehlungen ist im Allgemeinen als schnell zu bewerten. Der aktuelle Stand ersetzt zwar nicht vollständig den Tutor, jedoch kann es ihn in seiner Korrekturarbeit entlasten und außerdem den Studenten beim eigenständigen und überprüften Lernen der SQL-Sprache helfen als Unterstützung. Bei der Entwicklung wurde möglichst wenig im Projekt verändert, zu den geringfügigen Änderungen gehört die Anpassung der Klasse *StatementViewNode* für den Baumvergleich.

7.2.2 Erkennung von Fehler

Für einen großen Teil der Syntaxfehler wird durch JSQL eine Fehlermeldung erzeugt. In Kombination mit dem Grader in aSQLg, der die Anfragen im Zusammenspiel mit der Datenbank überprüft und ausführt, sollten gute Ergebnisse erzielt werden. Eine aufeinander abgestimmte Zusammenarbeit ist in der Entwicklung nicht erfolgt, sie ist jedoch vorstellbar. Die Erkennungsrate für syntaktisch korrekte SQL-Statements durch den Baumvergleich ist für die Standardfehler mit klaren Abweichungen der Kategorien Überflüssig und Fehlend besonders gut. Diese werden durch die ungemappten Knoten gut erkannt. Durch den Vergleich des studentischen Statements mit der Musterlösung werden viele der vorgestellten Semantikfehler implizit als Unterschied zwischen den Bäumen gefunden.

7.2.3 Ausgabe der Nachrichten

Die Lösungsempfehlungen und Fehlermeldungen sind übersichtlich aufgebaut und besitzen eine Struktur, die es ermöglicht, die enthaltenen Werte kompakter Form zu gruppieren. In den Lösungsempfehlungen wird die Lösung nicht kommentarlos wiedergegeben, sondern nimmt Bezug auf die Schlüsselwörter, verwendete Datentypen und enthaltene Werte. Die von JSQL erzeugte Fehlermeldung ist weniger als direkte Lösungsempfehlung geeignet und benutzt mehrere Zeilen für die Fehlermeldung, in einigen Fällen wurde das gleiche Zeichen sogar mehrfach hintereinander in jeder Zeile wiederholt. Diese Fehlermeldung wird ausgewertet, gefiltert und etwas komprimierter in einer übersichtlicheren, übersetzten Variante in einer Zeile dargestellt.

7.2.4 Einschätzung Laufzeit

Die Testergebnisse zeigen, dass für normale Statements die Laufzeit sich auf einem schnellen und daher einsetzbaren Niveau befindet. Die Statements mit längeren Ausführungszeiten mussten eigens gesucht und dafür erstellt werden.

7.3 Einschätzung der Schwächen

7.3.1 Projektstand

Aufgrund des Zeitlimits in der Masterarbeit konnten nicht alle geplanten Elemente in vollem Umfang in der Implementierung Einzug erhalten. Die Liste für die Weiterentwicklung und Verbesserungsvorschläge besitzt Potential und wären Optionen für eine zukünftige Version.

Das Projekt könnte zwar schon in der Praxis als Lernhilfe für die Studenten eingesetzt werden, momentan sind aber einige Teilbereiche bei den Laufzeiten und in der Erkennung für spezielle Problemfälle noch nicht zufriedenstellend bearbeitet. Die Integration dieser noch zu optimierenden Elemente in eine nachfolgende Version würde sowohl den Lernerfolg als auch die Akzeptanz bei den Studenten erhöhen. Diese Schwachstellen werden in den folgenden Abschnitten erläutert.

7.3.2 Laufzeit und Defizite in JSQL

Der JSQL-Parser verfügt über diverse Defizite, z.B. ist nicht für alle Fehler eine Fehlermeldung möglich, da es unerwartete Abbrüche mit einer *NullPointerException* geben kann – hier wird weder ein Baum noch eine Fehlermeldung erzeugt. Durch spezielle Sonderzeichen kann sogar die gesamte Prüfung ohne Warnung abgebrochen werden.

Die Laufzeit von JSQL für die Baumerzeugung von zu künstlich stark verlängerten Statements ist ein kritisches Problem, ebenso die Vervielfachung der Laufzeit mit speziellen Fehlern. Dies bedeutet eine Schwachstelle, da es einen Angriff auf die Verfügbarkeit darstellt.

Obwohl der JSQL-Parser in der Regel für den Großteil der Laufzeit verantwortlich ist, wird in den Tests gezeigt, dass in manchen Aufgabenstellungen das Mapping länger benötigt kann. Hier spielt es keine Rolle, dass dies seltener auftritt, die höhere Laufzeit ist der entscheidende potentielle Flaschenhals.

7.3.3 Schwächen in der Erkennung

Die größte Schwäche bei der Erkennung von Abweichungen resultiert aus der zu geringen Verwertung des Kontexts im Mapping und in der Auswertung. Dadurch kann es Fälle geben, in denen das Mapping die falsche Wahl trifft, ohne weitere Informationen zu nutzen. Auch mit einer binären Einteilung in sehr ähnlich bzw. identisch und abweichend durch einen festgelegten Grenzwert ist es schwer, immer die korrekte Wahl zu treffen. Da die Auswertung abhängig von den Informationen ist, die das Mapping liefert, können dadurch Anfälligkeiten für kleine Abweichungen mit größerem Effekt entstehen.

7.3.3.1 Semantische Fehler

Die Liste der Semantikfehler hat gezeigt, dass diese Gruppe ein weites Feld an Abweichungen umfasst. In diesem Abschnitt liegt der Fokus primär auf den semantischen Fehlern, die zu einer Berechnung eines falschen Ergebnisses führen und deren Bedeutung der Wertigkeit im Zusammenhang mit anderen Elementen, in der Reihenfolge sowie in der Arithmetik liegt.

Eine echte semantische Überprüfung dazu ist nur in Ansätzen vorhanden. Das Problem resultiert aus dem erstellten Konzept, dass primär auf Existenzprüfung sowie nur auf die existierenden Kategorien Fehlend/Zuviel/Verwechslung basiert und dabei aber zu wenig auf den Kontext setzt.

7.3.3.2 Reihenfolge und Arithmetik

Zwar kann die Flexibilität durch eine beliebige Reihenfolge der Elemente etwas positives sein, jedoch nur eingeschränkt, wenn dadurch die Erkennung von Fehlern nicht erfolgt.

Es ist dadurch z.B. möglich, dass ein fehlendes Element in Abschnitt A übersehen wird, weil es dafür in einem anderen Abschnitt B zu viel ist, obwohl es im ganz anderen Kontext steht.

Als komplexeres Beispiel zur Unterscheidung von Klammern mit **AND/OR** beziehungsweise Arithmetischen Operationen (***/+/-)** kombiniert wären die Vergleiche zwischen mehreren WHERE-Bedingungen. Dadurch wird deutlich, dass das Erkennen von bestimmten semantischen Abweichungen keine triviale Aufgabe darstellt, sondern als eigener Themenkomplex zu betrachten ist.

In der folgenden Tabelle sind mehrere Beispiele aufgeführt, in denen die ersten beiden zwar ähnlich sind, jedoch Unterschiede in der Bedeutung haben. Der Vergleich des ersten und letzten Beispiels wirkt auf den ersten Blick zwar abweichend, ist semantisch jedoch identisch.

#1	WHERE X > (A+B-C)*(D +(E+F)*(E+F)) AND (X>G OR X > (H*H))
#2	WHERE X >(A+B-C)*(D +(E+F)*(E+F)) AND X>G OR X > (H*H)
#3	WHERE (X>G OR X > (H*H)) AND X > ((F+E)*(E+F)+D)* (B-C+A)

Ein ähnlich gelagerter, potentiell übersehener Fehler würde in der Situation eintreten, dass ein LEFT oder RIGHT JOIN nicht korrekt erkannt wird, wenn der Student die umgedrehte Reihenfolge für entweder LEFT/RIGHT-JOIN oder die gewählten Tabellen verwendet. Erneut sind die ersten beiden Beispiele abweichend und das erste und letzte übereinstimmend.

#1	FROM L LEFT JOIN R ON L.id=R.id;
#2	FROM R LEFT JOIN L ON L.id=R.id;
#3	FROM R RIGHT JOIN L ON L.id=R.id;

7.3.3.3 Klammern

Das Thema Klammern gehört eigentlich auch wie die Reihenfolge zur Erkennung von Kontexten, hat jedoch die Besonderheit, immer als Paar aufzutreten. In der aktuellen Lösung wird jede Klammer einzeln für sich betrachtet und mit einer anderen statt einem Paar verglichen, daher kann hier eine Vermischung erfolgen. Da aber derzeit nur gemeldet wird, dass Klammern fehlen oder überflüssig sind und nicht welche Klammern bzw. an welcher Stelle, ist dieses Problem in der Praxis im Projekt nicht relevant. Wenn es also mehr Klammern in der Musterlösung gibt, ist dies ein Hinweis auf mögliche fehlende im Studentenstatement, wobei es hier keine Rolle spielt, an welcher Stelle diese stehen und welche miteinander gemappt werden.

7.3.3.4 Unterabfragen

Derzeit wird der Baum als ein großes Statement gewertet und nur der Knoten für Unterabfragen gewertet, der für die Struktur zuständig ist. Das bedeutet, es wird derzeit nicht unterschieden, ob das Schlüsselwort aus dem Hauptteil oder der Unterabfrage stammt. Dies kann zu Problemen durch Vermischung führen und sollte in den Vergleichen mehr differenziert werden.

7.3.3.5 ALIAS

In der aktuellen Lösung sind ALIAS-Elemente ein Problem, was aus der Abweichung der Knoten resultiert. Die verwendeten ALIAS-Werte werden nicht miteinander im Statementbaum referenziert bzw. aufgelöst. Aus der aktuellen Sicht der Knoten sind die verwendeten ALIAS komplett eigene neue Spalten. Ein Beispiel für verschiedene Verwendungen von ALIAS wäre der Vergleich zwischen folgenden beiden Statements:

```
SELECT movie.id, m.moviename AS name FROM movie m WHERE name='bsp'

SELECT id, moviename FROM movie WHERE moviename ='bsp'
```

In dem Beispiel sticht im WHERE die Spalte name besonders als Problem heraus.

7.3.3.6 ÄQUIVALENZEN

Äquivalente Abschnitte in Statements zu erkennen und als gleichwertig zu bewerten ist eine wünschenswerte Fähigkeit. Die Bildung dieser Äquivalenzen ist in der aktuellen Implementierung nicht vorhanden.

Eine Integration der Suchen nach speziellen Mustern in einem Baum ist aufwändig in der Implementierung, wie z.B. die Suche im *SubTreeBlocker*. Da diese Funktion nicht von Beginn an Teil des ursprünglichen Konzepts war, findet hier für jede Untersuchungen ein separater Baumdurchlauf zur Mustersuche statt. Der Aufwand der Integration solcher Suchen nach speziellen Mustern ist allein in einem einzigen Baum bereits höher als erwartet. Das Prinzip der Musteridentifizierung gleichzeitig so in beiden Bäumen einzusetzen, um Äquivalenzen zu finden, wird daher als nicht empfehlenswert eingestuft.

7.3.3.7 INFORMATIONSVERLUST

Durch die Auswertung der Knoten und Überführung der enthaltenen Informationen in die *CollectorItems* durch den *Collector* gehen mehrere Informationen zum Knoten verloren. Der gewählte Datentyp String für die Werte der Listen steht auch in der Kritik, da der Bezug zum Knoten verloren geht und dadurch der Kontext in der Auswertung nicht mehr enthalten ist.

Wenn ganze Bereiche abweichend sind, so erfolgt eine Aufteilung für jeden einzelnen Knoten in das *CollectorItem* des jeweiligen Typs (Spalte, Wert, Datentyp, Funktion, etc). Dies besitzt zwar eine klare Struktur und lässt sich gut vergleichen, jedoch könnte eine Ausgabe des gesamten zusammenhängenden abweichenden Blocks mehr zur Übersicht beitragen und den Studenten schneller den Bezug zu seinem Statement jeweils finden.

7.4 FAZIT DER EVALUIERUNG

Die vorliegende Arbeit bietet eine solide Basis als Grundlage für eine Weiterentwicklung des übergeordneten Themenkomplex "Computergestütztes Tutorensystem". Dass dieses Projekt aber noch nicht komplett abgeschlossen ist, zeigen die dargestellten Bewertungsergebnisse. Vor allem an den ausgewerteten Schwachstellen aus der Analyse sollten nachfolgende Arbeiten ansetzen mit Verbesserungen, Korrekturen, Optimierungen, um das vorhandene Potential voll ausschöpfen zu können. Die Evaluierung macht aber auch deutlich, dass besonders bei der Erkennung komplizierter Fehlermöglichkeiten wie bei speziellen Semantikfällen oder Äquivalenzen die persönliche Bewertung durch den Tutor nicht gänzlich zu ersetzen ist.

8 WEITERENTWICKLUNG

Die Auswertung der Testergebnisse und die Evaluierung geben Aufschluss darüber, bei welchen Aufgabenstellungen die Schwachstellen des Konzepts liegen und wo Probleme in der Realisierung auftreten. Aus den daraus gewonnenen Erkenntnissen und Erfahrungen basieren die Ideen der folgenden Ausarbeitungen als Vorschläge zur Korrektur, Verbesserung und Optimierung. Diese waren in dem vorgegebenen Zeitrahmen der Masterarbeit nicht mehr umsetzbar, sie bieten sich aber für die Implementierung in zukünftige Versionen bei einer Fortführung des Projektes an.

8.1 VERBESSERUNG DER LAUFZEIT VON PROBLEMFÄLLEN

Bei der Überprüfung von Worstcase-Szenarien wird gezeigt, dass sich gezielt Statements konstruieren lassen, die mit zu langen Laufzeiten einen Angriff auf die Verfügbarkeit bedeuten und somit ein Sicherheitsrisiko darstellt. Problematische Laufzeiten entstehen durch sehr lange, stark verschachtelte Statements, die durch Fehler zusätzlich verlängert werden können. Zwischen den einzelnen Fehlern gibt es wie bereits erwähnt ebenfalls große Unterschiede. Dieses Problem sollte definitiv Priorität haben, um eine gewisse Anwendungsreife zu erhalten, die nicht durch Wartungsarbeit am Server wegen Auslastung hervorgerufen wird.

Ein Lösungsansatz für dieses Problem wäre die Einführung eines gewählten Zeitlimits (Timeout) bzw. ein Zeichenlimit für das Studentenstatement. Das Zeichenlimit kann mehrstufig sein, oben durch einen absoluten Grenzwert limitiert und darunter eine relative Länge als Faktor zum Instruktor-Statement, ähnlich wie im Grader. Da die Länge bei strukturell komplexen Statements nicht immer der ausschlaggebende Faktor für die Laufzeit ist, ist das Zeitlimit die geeignetere Methode als Lösung des Problems.

Nur bedingt geeignet wäre hingegen die Variante, in der versucht wird, alle problematischen Fehler wie die leeren Klammern abzufangen. Dies kann bei Einzelfällen wirksam sein, jedoch kaum zu realisieren, den gesamten Umfang aller Problemfälle dadurch abzudecken. Nachteilig ist auch die Möglichkeit dadurch ungewollt bestimmte Statements auszuschließen.

8.2 ALTERNATIVE BEI SYNTAX-FEHLER

Mit den Testergebnissen wurde ebenfalls deutlich gezeigt, dass der JSQL-Parser nicht jedes SQL-Statement wie erwartet bearbeiten kann und für manche Fehler keine Fehlermeldung liefert. Neben möglichen Korrekturen von JSQL wäre die zusätzliche Integration eines alternativen Parsers als Fallback-Mechanismus, der bei kritischen JSQL-Fehlern einspringt.

Ohne diese Nachbesserung erhalten die Studenten mit dem aktuellen Stand von JSQL nicht den vollen Funktionsumfang für die Fehlermeldungen und Lösungsempfehlungen.

Als Unterstützung gegen den Umgang mit der Problematik der Sonderzeichen, kann man eine Überprüfung integrieren, die nach Abweichungen aus einem erlaubten Bereich der ASCII-Zeichen sucht und eine entsprechende Fehlermeldung dafür erzeugt.

8.3 Neues Konzept: SubTreeGrouping

Im gesamten Projektverlauf taucht immer wieder die Frage der Fehlerdefinition mit ihren verschiedenen Aspekten auf. Besonders die Grenzziehung zwischen gleich und abweichend ist in denjenigen Fällen schwierig umzusetzen, die nur unter Einbeziehung des Kontexts zu einer klaren Entscheidung führen. Das für diese Fragestellung neu erarbeitete Konzept bietet hier einen verbesserten Lösungsansatz. Eine sehr wichtige Erkenntnis aus der Schwachstellenanalyse ist, dass hier eine Suche nach dem richtigen Fragmentierungsgrad stattfindet, in denen die Größe für die Einteilung der Informationen weder zu komplex, noch zu einfach sein darf. Dies hat starken Einfluss auf der Effektivität des Mapping, der Auswertung und der Ausgabe. Die aktuelle Variante besitzt einen zu starken Fragmentierungsgrad mit frühem Verlust wichtiger Kontextinformation, die später in der Auswertung nützlich gewesen wären. Hier setzt das neue Konzept als Idee an und versucht dieses Defizit besser zu machen.

8.3.1 Zusätzlicher Schritt vor dem Mapping

Wie der gewählte Name *SubTreeGrouping* andeutet, ist das wichtigste neue Element die Einteilung in Gruppen. Vor dem Mapping-Schritt werden die Teilbäume und Knoten in den jeweiligen Bäumen in einer oder mehreren Gruppen zugeordnet, wobei jede Gruppe eine Typisierung bzw. Klassifikation (SELECT, GROUP-BY, Funktion, etc) erhält und wiederum Teilbäume, Knoten und andere Gruppen beinhalten kann, ähnlich einer Mehrfach-oder geschachtelten Indexierung. Die Aufteilung in Gruppen wird über den Typ des jeweiligen Knoten entschieden, wobei eine Auswahl fester Trennpunkte möglich wären (z.B. über das Attribut *Origin* oder *StringRepresentation*) oder eine Liste mit Regeln für erlaubte Knotentypen zum jeweiligen Knoten- bzw. Gruppentyp. Eine mögliche kleinere Gruppe wäre die Kombination aus der Tabelle, Spalte und möglichem Alias. Eine größere Gruppe wie zum SELECT-Abschnitt kann mehrere dieser kleineren Gruppen enthalten, dazu noch einzelne Knoten für das DISTINCT.

8.3.2 Erweiterung des Mapping und Berechnung der Ähnlichkeit

Im verbesserten Mapping-Schritt wird wie bisher zuerst der identische Pfad überprüft, da diese i.d.R. keine Abweichung zur Musterlösung darstellen. Im weiteren Verlauf der Untersuchung wird die der reguläre Vergleich der restlichen einzelnen, ungemappten Knoten ausgeweitet auf die Gruppen, die untereinander auf Ähnlichkeit überprüft werden. Dies stärkt die bevorzugte Suche innerhalb gleichen Kontexts über denselben Gruppentyp und zusammenliegender Teilbäume. Das Mapping selbst wird dadurch aber auch komplexer, da zusätzlich zu den Knoten nun ebenfalls auch die Gruppen gemappt werden können. Hier sollte man sich ein Konzept überlegen, dass bei der Bewertung von Knoten und Gruppen nicht nur die Ähnlichkeit, sondern auch die Größe der Gruppe in den Auswahlkriterien berücksichtigt. Als Beispiel wäre eine sehr große Gruppe zu bevorzugen, die sehr knapp unter der Ähnlichkeitsberechnung von einer sehr kleinen Gruppe mit enthaltenen überschneidenden Knoten liegt. Die Entscheidungskriterien sind vielschichtig angelegt und sollten daher auch in unterschiedlichen Ansätzen erprobt werden. Das Mapping arbeitet derzeit mit einer binären Trennung (in gemappt/ungemappt), also auf Grundlage einer Entscheidung über einen festgelegten Grenzwert, könnte aber durch eine Abstufung profitieren. Die neue Unterteilung trennt im Mapping zwischen gleich (ab dem hohem Grenzwert) und ähnlich (zwischen niedrigem und hohem Grenzwert). Das bedeutet einige der derzeit nicht gemappten Knoten würden nun gemappt werden, jedoch mit der Kennzeichnung „ähnlich“, was die Auswertung vereinfacht und verbessert.

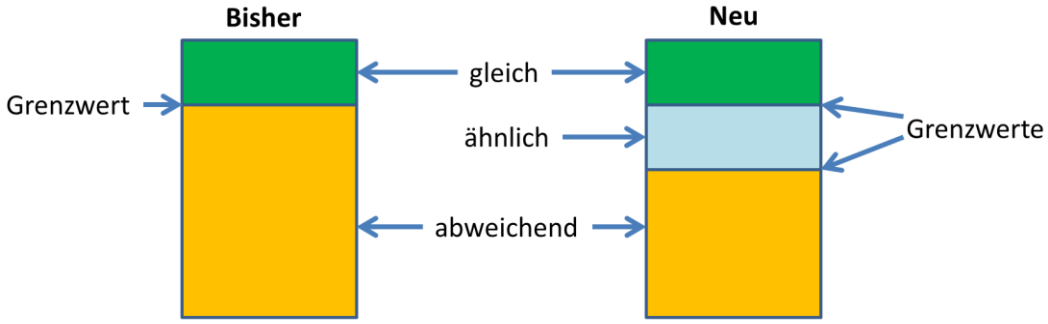

Abbildung 13: Darstellung der Einteilung mit Abstufung in gleich, ähnlich und abweichend

In der Umsetzung dieser Abstufungsalternative kann der neue untere Grenzwert für ähnlich auch für das gesamte Mapping verwendet werden und die Trennung in gleich, ähnlich und identisch in der Auswertung durch den *Collector* erfolgen.

Für die Berechnung der Ähnlichkeit zwischen Knoten und Gruppen sollten Maßnahmen getroffen werden, die ein erneutes Berechnen zwischenspeichert und so die wiederholte Berechnung verhindert. In Anlehnung an die Idee der Fingerprints, könnte man Lookup-Tabellen für Knoten und Gruppen erstellen, in denen man nachschlagen kann, ob der Wert schon vorhanden ist, anstelle ihn in jedem Vergleich komplett neu zu berechnen.

8.3.3 Neue Auswertungsmöglichkeiten

8.3.3.1 Neues Kategorie: Fehlplatziert

Bei der Umwandlung der gemappten Bäume sollen die Kontextinformationen mit übertragen werden (Gruppen und Pfad), um für die Weiterverarbeitung durch die anschließende Auswertung zur Verfügung zu stehen. Durch diese zusätzlichen Informationen wird mehr Flexibilität bei der Analyse erreicht, dadurch kann neben der Unterteilung in Zuviel/Fehlend/Verwechslung um die neue Kategorien „Fehlplatziert“ ergänzt werden. Diese wird derzeit beim Mapping häufig als Übereinstimmung bewertet anstelle eines Fehlers, was verdeutlicht, welcher Qualitätssprung sich hier für die Ergebnisse anbietet.

Ein derzeit nicht als Fehler erkanntes Beispiel wäre die Verschiebung eines Elements von ORDER-BY in das GROUP-BY, welches durch diesen Ansatz erkannt wird.

Student = „[…] GROUP BY a, b ORDER BY c“

Instruktor = „[…] GROUP BY a ORDER BY b,c“

Aktuell werden nur andere Fehler erkannt, wie die Vertauschung der Schlüsselwörter ORDER-BY anstelle GROUP-BY bzw. umgekehrt oder wenn das Schlüsselwort selbst fehlt. Ein fehlendes Attribute, dass zu einem der beiden gehören würde, würde derzeit in fehlende, überflüssige oder verwechselte Spalten aufgelistet werden.

8.3.3.2 Alternative zur bisherigen Mustersuche

Die Gruppen als neues zentrales Element ersetzen den *SubTreeBlocker*, der eine gezielte Suche nach festgelegten Mustern von zusammenhängenden ungemappten Teilbäumen für ein gemeinsames *CollectorItem* durchführt. Die bisherige Mustersuche hat den Nachteil, die einzelnen Muster manuell mit hohem Entwicklungsaufwand zu integrieren und bei der Suche für jedes einzelne, vorgegebene Muster den gesamten Baum zu durchlaufen.

Das Prinzip der Musteridentifizierung in dieser Art gleichzeitig in beiden Bäumen einzusetzen, um Äquivalenzen zu finden, wurde daher als nicht empfehlenswert eingestuft.

Durch die neuen Gruppen lassen sich hier die Muster deutlich einfacher auswerten und für die Überprüfungen der Liste der Semantikfehler und Äquivalenzen verwenden. Auch die Reihenfolge innerhalb der Gruppe lässt sich für die Überprüfung leichter verwenden und differenzierter analysieren.

Aus der Liste der Semantikfehler stammt ein Beispiel für die Überprüfung von Regeln, z.B. dass bei den Funktionen MIN und MAX nie ein DISTINCT benötigt wird. Weitere Beispiele zeigen, dass welche Äquivalenzen durch die Gruppen nun besser überprüft werden können:

x BETWEEN a AND b	**=**	**x>=a AND x<=b**	
x BETWEEN a AND b	**=**	**x>=a-1 AND x<b+1**	**(Bei Datentyp LONG)**
>=1	**=**	**>0**	**(Bei Datentyp LONG)**
X<Y	**=**	**Y>X**	
FROM L LEFT JOIN R ON L.id=R.id	**=**	**FROM R LEFT JOIN L ON L.id=R.id**	

Dies ist jedoch keine konkrete Lösung für komplexere Kombinationen aus Arithmetischen Funktionen. Zum Beispiel werden folgende Angaben auf diese Art weiterhin nicht ausgewertet:

X+1<Y	**!=**	**Y+1>X**

Dieser Bereich gehört zu den komplizierteren Problemen der Auswertung und wiederum einen eigenen, umfangreichen Themenbereich dar.

8.3.3.3 Äquivalenzen in einem eigenen Sammlung

Ähnlich wie der *NodeTypePool* mit der Sammlung des Wissens über die einzelnen Knotenausprägungen und ihrer Werte, können die einzelnen gleichwertigen Überprüfungen in einem *EquivalencePool* für einen Vergleich gesammelt werden.

8.3.4 Nachfolger zum CollectorItem

Die Überarbeitung des Konzepts stellt neue Anforderungen an die genutzten Elemente in der Auswertung. Das *CollectorItem* in seiner jetzigen Form wird sehr wahrscheinlich durch einen Nachfolger zu ersetzen sein. Dieser sollte mehr mit dem Kontext, den verwendeten Knoten und Gruppen sowie den neuen Funktionen zusammen arbeiten und nicht für die enthaltenen Werte eine Liste des Datentyps String verwenden.

8.3.5 Verbesserte Ausgabe

Die Verbesserungen erfolgt nicht nur für das Mapping und der Auswertung, sondern hat auch Einfluss auf die Ausgabe. Neue Möglichkeiten entstehen und bestehende werden optimiert.

8.3.5.1 Neuer Fehlertyp „Fehlplatziert“

Während der Fehlertyp „Verwechslung“ sich auf Fehler wie z.B. falsche Spalte, Tabelle, Datentyp oder Funktion bezieht, unterscheidet sich der neue Typ „Fehlplatziert“ durch den Fokus auf die Nennung der Position, so dass zu einem Element jetzt die falsche und korrigierte Position angegeben wird.

„Fehlplatziert: Spalte[Movie.budget] in [ORDER-BY] statt [GROUP-BY]“

8.3.5.2 Ursprung des Fehlers verdeutlichen durch Kontext

Eine Erweiterung der Fehlermeldung betrifft besonders diejenigen Spaltenelemente, die mehrfach und gleichzeitig an mehreren Stellen (SELECT, FROM und WHERE oder Unterbereiche) auftreten können. Die Nennung erfolgt mit detaillierter Angabe der Quelle des Fehlers als Hilfe.

„ Fehlend in [WHERE]: Spalte[Movie.budget]“

8.3.5.3 Ausgabe des zentralen Elements einer Gruppe

Wenn Elemente gänzlich fehlen oder überflüssig sind und daher auch die Knoten komplett in dieser Gruppe ungemappt sind, so stellt sich in solchen Fällen die Frage, ob alle Inhalte der Gruppe aufgelistet werden müssen. Dies wird derzeit gemacht und die Elemente auf die jeweiligen Typen (Spalte, Funktion, Vergleich, etc) verteilt.

Durch die Gruppen kann man aber das zentrale Element darin auswerten, z.B. das Schlüsselwort BETWEEN aus dem Abschnitt `x BETWEEN 1 AND 5`, welches dann stellvertretend für die gesamte Lösungsempfehlung verwendet wird. Allgemein wären hier die Schlüsselwörter meist im Zentrum und die Werte eher sekundär. Z.B. das Schlüsselwort IN in dem Abschnitt `IN(1,2,3)` oder das ORDER-BY, in dem die Spalte und Sortierreihenfolge eher zweitrangig sind, wenn der gesamte Abschnitt fehlt.

8.4 Neues Konzept für ALIAS

Wie in der Evaluierung beschrieben ist die Verwendung von ALIAS eine der Schwachstelle. Dies ähnelt zwar der Kategorie der Äquivalenzen, ist jedoch durch gewisse Besonderheiten ein eigenständiges Thema. Das folgende Beispiel aus der Evaluierung zeigt mehrere Verwendungen von ALIAS für Tabellen und Spalten:

```
SELECT movie.id, m.moviename AS name FROM movie m WHERE name='bsp'

SELECT id, moviename FROM movie WHERE moviename ='bsp'
```

Das Ziel der Überarbeitung ist es, diese Übereinstimmungen nicht mehr als Abweichungen oder Fehler zu bewerten. Als Lösungsvorschlag ist die Einführung eines *AliasPool*, der am besten vor dem Mapping die Bäume analysiert und die Tabellen und Spalten mit ihren ALIAS-Namen sammelt.

Eine Variante wäre, bei den Vergleichen im Mapping die Synonyme im *AliasPool* nachzuschlagen und mit den möglichen Optionen zu vergleichen. Dies müsste allerdings bei jedem Vergleich geschehen. Daher wäre die alternative Variante, sich nach dem Aufbau des *AliasPool* sich auf einen dieser möglichen Werte festzulegen und die anderen gleichwertigen Optionen in den Bäumen dadurch zu ersetzen.

8.5 KLAMMERN

In der Analyse des Mappings der Statementbäume wurde unter grafischer Darstellungshilfe sichtbar, dass es auch Fälle mit Mapping von falschen Klammern auftreten können. Dieser Fehler stammt aus Geschwisterknoten, die quasi identische Attribute besitzen, aber von denen nur dem einen ein Kindknoten für das Leerzeichen anhängt. Die falsche Wahl für die Klammern ist zwar unerwünscht, hat aber in der aktuellen Version keinen Einfluss auf das Ergebnis. Die Ursache liegt darin, dass es im allgemeinen Mappingschritt ohne Differenzierung erfolgt, in der man den Vorwurf machen könnte zu sehr in die Kategorie der Greedy-Algorithmen zu gelangen und weniger das als Zuordnungsproblem zu behandeln. Der Vorschlag zur Verbesserung wäre, im ersten Schritt die gesamten Klammern aus beiden Bäumen mitsamt der Reihenfolge und dem Typ (geöffnete, geschlossene Klammer) zu sammeln. Beim Mapping sollten dann am besten immer die Paare aus geöffneter und geschlossener Klammer gemappt werden. Mit Hilfe des neuen Gruppen-Konzepts und der Betrachtung der Reihenfolge, könnte so die Zufriedenheit aller konkurrierenden Elemente erreicht werden und die fehlenden sowie überflüssigen leichter identifiziert werden.

8.6 ERWEITERUNG DER NACHRICHTEN UND AUSGABE

Die Erweiterung der Ausgabe mit den Nachrichten für die Lösungsempfehlungen behandelt keine Fehler, hier geht es um ebenso wichtige Punkte wie Übersichtlichkeit, Lesbarkeit und Lernhilfe für den Studenten.

8.6.1 AKTUELLER STAND

Die Klasse in aSQLg für die Nachrichten besteht aus einer Liste an Nachrichten, die neben dem eigentlichen Text der Nachricht zusätzlich noch einen Empfänger (Student/Instruktor), Nachrichtentyp (Info, Warnung, Fehler und fatalem Fehler) und optional den Verweis auf die Zeile und Spalte aufweisen. Für die Auswertung der Lösungsempfehlung wird in der eigenen Nachrichtenklasse ein ähnliches, jedoch angepasstes Konzept genutzt, in dem die Nachricht einen Status der Auswertung enthält. So weiß das System, ob eine normale Durchführung des Algorithmus erfolgte, ein Abbruch durch leere Statements, keine Untersuchung aufgrund identischer Statements oder ein Fehler durch Syntaxfehler bzw. kritischem JSQL-Fehler. Als Vorbereitung wird hier die Kategorie der Lösungsempfehlung ebenfalls eingetragen, was mehr Möglichkeiten und Flexibilität für eine übersichtliche Ausgabe ermöglicht, z.B. durch Gruppierungen der gleichen Kategorie, in der nur einmal diese Kategorie genannt wird mit anschließender Auflistung der enthaltenden Nachrichten. Dies führt zu einer geringeren Anzahl verwendeter Zeichen bei gleicher Menge an Informationen, die dadurch auch noch übersichtlicher wirkt. Ein Beispiel für die Gruppierung in die verschiedenen Kategorien zeigt die folgende Tabelle:

Verwechslung	Long[234] statt Long[345]
Zuviel	AS[aliasname]
Fehlend	- Verkettung[OR] - Spalte[X,Y] - Vergleich[=] - Klammern[()]

Zur Nutzung der einfachen Nachrichtenklasse von aSQLg für die Lösungsempfehlungsnachricht stehen Möglichkeiten zur einfachen Transformation in der Implementierung bereit.

8.6.2 Prioritäten

Bei sehr großen Abweichungen kann die Liste der Lösungsempfehlungen so umfangreich werden, dass sie für den Studenten unübersichtlich wird und keine eindeutige nützliche Lernhilfe mehr darstellt. Ein Mittel zur Prävention wäre die Einführung eines Prioritätenlevels für die Nachrichten. Dies dient als Auswahlhilfe und könnte bei Überschreitung gewählter Grenzen für die Nachrichtenanzahl die geringer bewerteten Nachrichten ausblenden bzw. zugunsten einer Detailansicht zurückhalten.

Als Beispiele mit geringer Priorität könnten fehlende ALIAS oder Klammern sein, höher bewertet wären wichtige Schlüsselwörter. Alternativ lässt sich hier auch eine Differenzierung der verschiedenen Semantikfehler durchführen, die zwischen echten Fehlern in den Ergebnissen und nicht genutztem Potential sowie schlechtem Stil trennt.

8.6.3 Neue Fehlerkategorien

Durch die Liste der Semantikfehler erfolgt die Einteilung in Fehler mit falschen Ergebnissen, ungenutztem Optimierungspotential und schlechten Stil. Diese können als neue Fehlerkategorien neben Verwechslung/Zuviel/Fehlend für die Lösungsempfehlungen mit unterschiedlich hohem Prioritätenlevel einfließen. Die Trennung sollte für den Studenten auch kenntlich gemacht werden, damit der Student auch zwischen Optimierungsvorschlag und Fehler besser trennen kann.

8.6.4 Alternatives Konzept für die Ausgabe

Die Darstellung der Lösungsempfehlungen basiert bisher auf einer formatierten Auflistung von Texten. Als ein alternatives bzw. zusätzliches Konzept bietet sich die grafische Darstellung für die Auswertungsergebnisse an, entweder als Unterstützung der Ausgabe oder als gänzliche ersetzende Alternative.

8.6.4.1 Farbiger Text mit Auswahl

Als Erweiterung der Nachrichten können zusätzliche Informationen als Kontext mit einfließen, die bereits vorhandenen Elemente sollen mehr genutzt werden, wie die Zeile und Spalte. Bisher fehlt deren Verwertung in der Auswertung, stattdessen werden sie nur bei der Berechnung der Knotenähnlichkeit verwendet.

Für die Nutzung in der Ausgabe gibt es je nach eingesetzter Technologie mehrere Möglichkeiten, darunter die Idee spezieller Formatierungen, in der die einzelnen Schlüsselwörter anhand der Spalte und Zeile durch unterschiedliche Farben hervorzuheben, je nachdem ob hier dafür Lösungsempfehlungen oder keine vorliegen. Ergänzen kann man dies mit einer Verknüpfung der Auswahl, in der durch die ausgewählte Lösungsempfehlung dann der fehlerhafte Bereich hervorgehoben wird (Formatierung anpassen: Schriftgröße, Schriftart, Fett, Farbe) und umgekehrt bei Auswahl des fehlerhaften Bereichs dann die Lösungsempfehlung. Je nachdem, ob dies eine Weboberfläche ist oder ob diese für Touchscreens geeignet sind, entscheidet sich die Auswahl, ob diese über ein Hover bzw. Mouseover-Effekt oder einem Klick erfolgt.

8.6.4.2 Grafische Ausgabe des Statementbaum

Über einen Durchlauf der Bäume kann man über das Attribut *StringRepresentation* das Statement mit den genutzten Schlüsselwörtern und Werten in der vorhandenen Reihenfolge rekonstruieren. Für diese Werte wäre es auch mit einfachen Mitteln möglich, einen kleineren, kompakten Baum zu erstellen, in dem nur die im Studentenstatement genutzten Wörter enthalten sind. Die einzelnen Knoten werden in mehreren Stufen eingefärbt, je nachdem, wie sehr der Studentenbaum hier dem Instruktorbaum ähnlich ist, wobei dafür ein sehr gutes Mapping die Voraussetzung ist. Als ergänzende Option könnte bei Auswahl des farbigen Knoten bzw. Teilbaumes die jeweilige erzeugte Lösungsempfehlung zu diesem Bereich angezeigt werden.

Die folgende Abbildung zeigt eine vereinfachte Darstellung des sehr kompakten Statementbaumes mit farbiger Markierung der Knoten für die Ähnlichkeit seiner Teilbäume und einer Auswahl von Elementen für das Anzeigen der Lösungsempfehlungen.

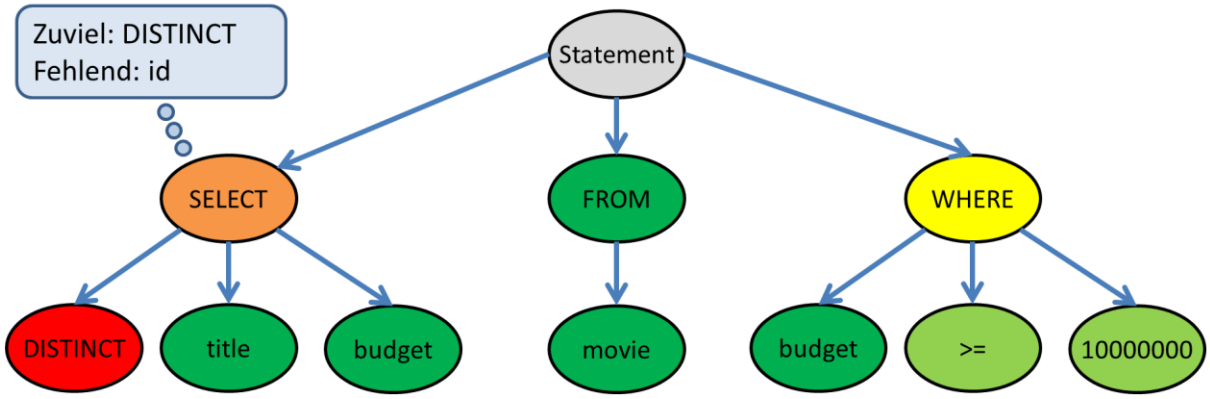

Abbildung 14: Beispiel einer grafischen Ausgabe des Statementbaums mit farbiger Markierung und Auswahl

In diesem Vorschlag werden die Knoten durch unterschiedliche Farben für die Ähnlichkeiten mit der Musterlösung des Tutors dargestellt. Eine mögliche Farbwahl wäre die als identisch bewerteten Knoten bzw. Teilbäume in Dunkelgrün darzustellen, die sehr ähnlichen in Hellgrün, ähnliche in Gelb, bei minimaler Ähnlichkeit in Orange und komplett abweichend Knoten und Teilbäume in Rot.

Die Darstellung des Baums kann je nach eingesetzter Technologie als einfaches statisches Bild erfolgen oder auch als interaktive Abbildung, z.B. durch *D3.js*, die mit dem Statement und den erstellten Lösungsempfehlungen verknüpft ist. Hier ist auch zu überlegen, ob der von JSQL erstellte Baum wie gewünscht mit der *StringRepresentation* aufgebaut und so übernommen werden kann. Oder ob eine alternative kompaktere Form mit einem kleineren Baum durch Bündelungen besser lesbar und damit auch mehr geeignet wäre.

Eine weitere ergänzende Idee betrifft die Ausgabe der abweichenden Bereiche, bei dessen Auswahl die entsprechende Teilbaumausgabe aus dem Instruktor-Statement zu nutzen. Hier ist es jedoch wichtig, dass es Limits gibt, um nicht zu viel von der Musterlösung zu verraten.

Kurz zusammengefasst umfasst der Vorschlag eine Ausgabe des Statementbaumes durch die Schlüsselwörter und Werte, die abhängig von der Abweichung farblich markiert werden und bei der Auswahl die zugehörigen Lösungsempfehlungen anzeigen oder limitierte Ausschnitte aus dem Instruktor-Statement.

9 ZUSAMMENFASSUNG

9.1 AUFGABENSTELLUNG UND THEMA

E-Learning ist in vielen Studiengängen längst zum Standard geworden, hingegen jedoch seltener die Kombination von automatisierter Bewertung studentischer Aufgaben mit individueller Fehleranalyse, die auf das Lernen aus den eigenen Fehlern ausgerichtet ist. Die vorliegende Masterarbeit „Automatisierte Unterstützung der Studierenden beim Lösen von SQL-Aufgaben durch Baumvergleich und Integration in ein Werkzeug zur automatisierten Bewertung“ im Studiengang Angewandte Informatik an der Hochschule Hannover setzt sich mit diesem Aufgabenfeld – Prüfen, Bewerten, Kommentieren und Anleiten beim Üben- im Tutorensystem für SQL auseinander.

9.2 ZUSAMMENFASSUNG DER ARBEITSSCHRITTE UND INHALTE

Eine bereits im Studiengang vorhandene Plattform aSQLg zur Überprüfung von Korrektheit und zur Punktbewertung von SQL-Aufgaben soll um die Aufgabenfelder „Vergleichen mit der Musterlösung, Fehleranalyse und Lösungshinweise erstellen“ erweitert werden. Hierfür war besonders in den Arbeitsschritten Konzept-Entwurf und Realisierung eigene Forschungs- und Entwicklungstätigkeit notwendig, da nach der Sichtung vorliegender Arbeiten zu Teilaspekten der Aufgabenstellung sich z.B. keine überzeugenden Algorithmen anboten. Diese umfangreiche Evaluierung verwandter Arbeiten zu Beginn diente der kritischen Wertung existierender Trainingssysteme und der Entscheidungsfindung, welche Elemente sich für die Erweiterung eignen, aber auch der Fehlerdefinition bei Semantikfehler. Aufgrund dieser Recherchen fiel die Entscheidung, für die weitere Bearbeitung einen eigenen Algorithmus zu entwickeln und beim Vergleich die Baumstruktur zu verwenden, die auch vom vorhandenen SQL-Parser JSQL aus dem Statement erzeugt wird.

Der Entwurf für das Konzept fokussiert sich für den Algorithmus auf primär fünf Arbeitsschritte, in denen die studentische Arbeit mit der Musterlösung des Instruktors verglichen und bewertet wird.

1. Zu Beginn als kleinster Arbeitsschritt die Statementprüfung
2. Übertragung der Daten der SQL-Statements in die Baumstruktur durch JSQL
3. Kenntlich machen der Unterschiede der beiden Bäume durch ein eigenentwickeltes Mapping
4. Auswertung der Baumunterschiede durch diverse Analysen
5. Ergebnisse als Nachricht zu den Lösungsempfehlungen generieren

Besonders für den 3. und 4. Arbeitsschritt war bei der technischen Umsetzung viel eigene Entwicklungsarbeit erforderlich. Dies betrifft die Kernbereiche der Masterarbeit mit dem Mapping und der Auswertung.
Das Mapping setzt dann ein, wenn nach einer Eingangsüberprüfung der Statements die Erstellung beider Statementbäume erfolgreich abgeschlossen ist. Um das studentische Statement mit der Musterlösung des Instruktors (Professor / Dozent / Tutor) vergleichen zu können, werden zuerst die völlig übereinstimmenden Pfade beider Bäume durchlaufen und über eine Ähnlichkeitsberechnung über die Attribute und des Pfades auf identische bzw. sehr ähnliche Knoten überprüft. Ab einem festgelegten Grenzwert gelten die verglichenen Knoten als übereinstimmend und werden miteinander gemappt. In dem zweiten Durchlauf werden die übrigen Knoten beider Statements paarweise gegenübergestellt und gesammelt, um diese der sortierten Ähnlichkeitsberechnung nach miteinander zu mappen.

Für die weitere Verarbeitung der Mappingergebnisse in der anschließenden Auswertung und Herausgabe der Lösungsempfehlungen sind nur die Angaben zu den Unterschieden von Interesse. Aus ihnen wird ermittelt, welche abweichenden und somit fehlerhaften Angaben im studentischen Statement aus einer Verwechslung resultieren, welche fehlten und was zuviel war. In den abschließenden Lösungsempfehlungen werden die konkreten Hinweise diesen drei Kategorien zugeordnet. Mit dieser Unterstützung kann der Studierende dann eigenständig aus den Lösungsempfehlungen seines Statements sich zum Statement der Musterlösung erarbeiten.

9.3 Untersuchung der Ergebnisse

Eine erste Einschätzung des Konzeptes ließen Stärken für Standardfehler von fehlenden sowie überflüssigen Elementen und Schwächen gegenüber speziellen Semantikfehlern erwarten. Deshalb schlossen sich umfangreiche Testreihen an, in denen konstruierte studentische Aufgaben mit der vom Instruktor vorgegebenen Musterlösung verglichen wurden. In der Masterarbeit wird eine Auswahl vorgestellt, die der Klärung von unterschiedlichen Fragestellungen dienen. Getestet werden Statements mit den folgenden Zielen:

- Korrekte Syntax zur Analyse der Lösungsempfehlungen
  - Suche nach Schwachstellen im Konzept und der Implementierung
  - Analyse der gefundenen und übersehenen Semantikfehler
- Inkorrekter Syntax zur Überprüfung der Fehlermeldungen
  - Suche nach unvollständiger SQL-Interpretation und Problemen durch JSQ
- Analyse und Einschätzung der Laufzeit
  - Untersuchung der Standardsituationen, Suche nach Grenzwerten und Problemfälle

Die Ergebnisse der Tests bestätigten die erste Einschätzung, brachten aber auch in der Konzeptionierung nicht vorhersehbare Probleme zu Tage. Hervorzuheben ist, dass das Projekt in der aktuellen Fassung sicher und zuverlässig und unter kurzer Laufzeitdauer die Standardfehler erfasst. Ebenso werden aufgrund der klaren Vorgabe der Musterlösung viele semantischen Fehler als Unterschied erkannt. Werden die zu vergleichenden Statement komplexer und strukturell anspruchsvoller, verlieren die Angaben aus der Auswertung ihre Detailschärfe, auch die Laufzeitdauer z.T. nimmt exponentiell bei gewissen Aufgabenstellungen zu. Besonders bei Anfragen, wo z.B. die Reihenfolge der Werte eine Rolle spielt, kann das System nicht mit genügender Berücksichtigung inhaltlicher Bezüge operieren. Durch die Tests wurden auch die systembedingten Schwachpunkte deutlich wie Syntaxfehler, für die es keine Fehlermeldungen gab, oder spezielle Sonderzeichen, die den regulären Prüfablauf für eine Aufgabe vollständig unterbricht.

9.4 Bewertung und Ausblick

Die vorgestellten Testergebnisse liefern über die Stärken-Schwächen Analyse hinaus konkrete Anhaltspunkte, an welchen Punkten das Konzept abgeändert oder erweitert werden sollte, um auch in umfangreichen und verschachtelten Statement in kurzer Laufzeit zu zuverlässigen Vergleichsergebnissen zu kommen. Hierzu stellt die vorliegende Masterarbeit ein überarbeitetes Konzept für das Projekt vor, das auf eine bessere Nutzung des vorhandenen Potentials zielt. Hier würden auf der Ebene des Mappings die Vergleichsoptionen durch Kontext und Gruppen neue Möglichkeiten erhalten und auch in die Auswertung mit einfließen können. Die Erkennung für Fehler von korrekten Elementen an falschen Positionen sollte dadurch verbessert werden und gibt die Ergebnisse in die neue Fehlerkategorie "Fehlplatziert" weiter. Darüber hinaus werden Lösungen für

spezielle semantische Fragen sowie die Frage der Laufzeitfehler vorgelegt, die in einer weiterführenden Arbeit wiederum in einer Testreihe evaluiert werden sollten.

Den Tutor kann es bisher noch nicht vollständig ersetzen, jedoch ist es vorstellbar ihn in den Tutorien zur Entlastung zu unterstützen. Nach einer Weiterentwicklung des Projekts würde der Umfang der Hilfe entsprechend steigen können. Der Blick auf die die Studierenden ist genauso bedeutsam, wie die Frage, ob sie von dem Einsatz des Projektes profitieren würden. Für den Einsatz als lernunterstützendes Bewertungssystem zum Einüben korrekter Syntax und Semantik spricht die gute und sichere Fehlererkennung bei den gängigen Standardsituationen. Die Abweichungen wegen kategorisiert und mit detaillierten Hinweisen so aufbereitet, dass sich daraus dem Studenten die Lösung bei entsprechender Nachbereitung erschließen sollte. Die Masterarbeit stellt im Kapitel 8 verschiedene Möglichkeiten der Repräsentation der Lösungshinweise vor.

10 QUELLENVERZEICHNIS

[1] *ProjektWiki aSQLg,* https://proanvil.inform.hs-hannover.de/redmine/projects/webcatsql/wiki.

[2] *Klassendiagramm aSQLg,* https://proanvil.inform.hs-hannover.de/redmine/attachments/download/2254/concept_asqlg_0_2_0.png.

[3] *Kernmodul aSQLg,* https://proanvil.inform.hs-hannover.de/redmine/projects/webcatsql/wiki/Concept_webcatsqlcore.

[4] *Konzept aSQL-Grader,* https://proanvil.inform.hs-hannover.de/redmine/projects/webcatsql/wiki/Concept_grading.

[5] *ZQL,* http://zql.sourceforge.net/.

[6] *JavaCC,* https://javacc.java.net/.

[7] *SQL-Tutorial Hoffman,* http://www.dbbm.fiocruz.br/class/Lecture/d17/sql/jhoffman/sqltut.html.

[8] *aSQLg-Grader-Ablaufdiagramm,* https://proanvil.inform.hs-hannover.de/redmine/attachments/download/2262/testfaelleimablaufdiagramm.jpg.

[9] *JSQL-Parser,* http://jsqlparser.sourceforge.net/.

[10] *Beispiel JSQL-Statementbaum,* https://proanvil.inform.hs-hannover.de/redmine/attachments/download/2874/statement_baum.jpg.

[11] *VisitorPattern,* https://en.wikipedia.org/wiki/Visitor_pattern.

[12] *ZQL-Tutorial,* http://zql.sourceforge.net/sqltut.html.

[13] M. Ifland, M. Jedich, C. Schneider und F. Puppe, „ÜPS – Ein autorenfreundliches Trainingssystem für SQL-Anfragen," Universität Würzburg, 2014.

[14] S. Brass und C. Goldberg, „Semantic Errors in SQL Queries: A Quite Complete List," Journal of Systems and Software (Volume 79, Issue 5), 2006.

[15] *Indikatorvariable für die Behandlung von NULL verwenden,* http://dcx.sybase.com/1101/de/ulc_de11/es-development-sectc-3774482.html.

[16] C. Goldberg, „Do you know SQL? About Semantic Errors in Database Queries," Martin-Luther-Universität Halle, 2008.

[17] *GReQL Kurzbeschreibung,* Universität Koblenz-Landau: https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/research/graph-technology/GReQL.

[18] *GraLab/JGraLab Kurzbeschreibung,* Universität Koblenz-Landau: https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/research/graph-technology/JGraLab.

[19] D. Bildhauer, „Auswertung der TGraphanfragesprache GReQL 2," Universität Koblenz-Landau, 2008.

[20] *JGraLab-Wiki,* Universität Koblenz-Landau: https://github.com/jgralab/jgralab/wiki.

[21] *JGraLab Projekt,* Universität Koblenz-Landau: https://clojars.org/de.uni-koblenz.ist/jgralab.

[22] *TGraphen Kurzbeschreibung,* Universität Koblenz-Landau: https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/teaching/ss2011/fpfg.

[23] M. Sattel, „Entwicklung eines TGraph-basierten Modell-Vergleichsverfahren," 2010.

[24] J. Ebert und D. Bildhauer, „Reverse Engineering Using Graph Queries," in *Graph Transformations and Model-Driven Engineering*, Springer-Verlag, 2010.

[25] *Oracle SQL Reference Example,* https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_10002.htm.

[26] H. Xu, „An Algorithm for Comparing Similarity Between Two Trees: Edit Distance with Gaps," 2014.

[27] R. Searles, „The Similarity Graph: Analyzing Database Access Patterns Within A Company," University of Delaware Newark.

[28] B. Shneiderman, B. B. Bederson und M. Wattenberg, „Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies," University of Maryland.

11 ANHANG

11.1 HANDBUCH

Das Handbuch zur Masterarbeit ist als separates Dokument vorhanden.

11.2 ÜBERSICHT JSQL

Es folgt eine Übersicht für die möglichen Inhalte der Baumstruktur in Form eines *StatementView*, die direkt aus dem internen ProjektWiki übernommen wird.

https://proanvil.inform.hs-hannover.de/redmine/projects/webcatsql/wiki/Concept_statementviewbuilder_jsql

Syntax	Semantic	Bedeutung
Keyword		Es handelt sich um ein durch SQL definiertes Schlüsselwort. (Schlüsselwörter werden nur als solche erkannt, wenn sie an der erwarteten Stelle stehen. Ansonsten gibt es bereits vom Parser einen Fehler.
	SELECT	SELECT Schlüsselwort
	Open	Öffnendes Element
	Close	Schließendes Element
	Separator	Ein Element, das andere in eine Aufzählung trennt (z.B. ,)
	Addition	Plus-Zeichen
	Division	Divisions-Zeichen
	Multiplication	Multiplikations-Zeichen
	Substaction	Minus-Zeichen
	AND	AND Operator
	OR	OR Operator
	BETWEEN	BETWEEN Operator
	EqualsTo	Gleichheits-Zeichen
	GreaterThan	Größer-Als-Zeichen
	GreaterThanEquals	Größer-oder-gleich-Zeichen
	IN	IN Operator
	isNull	IS NULL Ausdruck
	LIKE	LIKE Operator
	MinorThan	Kleiner-Als-Operator
	NotEqualsTo	Ungleich Operator
	CASE	CASE Ausdruck
	WHEN	WHEN Ausdruck
	THEN	THEN Ausdruck
	ELSE	ELSE Ausdruck
	END	END Ausdruck
	ALL	ALL Ausdruck
	ANY	ANY Ausdruck
	DISTINCT	DISTINCT Ausdruck
	INTO	INTO Ausdruck

	FROM	FROM Ausdruck
	ON	ON Ausdruck für JOINs
	FULL JOIN	FULL JOIN Ausdruck
	NATURAL JOIN	NATURALL JOIN Ausdruck
	SIMPLE JOIN	SIMPLE JOIN Ausdruck (,)
	RIGHT JOIN	LEFT JOIN
	INNER JOIN	OUTER JOIN
	JOIN	
	WHERE	WHERE Ausdruck
	ORDER BY	
	ASC	ASC Schlüsselwort für Aufsteigend sortierte ORDER BY Ausdrücke
	DESC	DESC Schlüsselwort für Absteigend sortierte ORDER BY Ausdrücke
	GROUP BY	GROUP BY Ausdruck
	HAVING	HAVING Ausdruck
	LIMIT	LIMIT Ausdruck
	TOP	TOP Ausdruck
	UNION	UNION Schlüsselwort
	CREATE TABLE	CREATE TABLE Schlüsselwort
	DELETE	DELETE Ausdruck
	DROP	DROP Statement
	DropType	Typ des Objekts, das gelöscht werden soll. (z.B. TABLE, VIEW, ...)
	INSERT	INSERT Statement
	VALUES	VALUES vom INSERT oder REPLACE Statement
	TRUNCATE	TRUNCATE Keyword (Deleting contents of a table)
	TABLE	TABLE Keyword wie es bei TRUNCATE benutzt wird
	REPLACE	REPLACE Keyword
	SET	SET Keyword
	SetEqualSymbol	= Zeichen, wenn es in einem REPLACE oder UPDATE Statement verwendet wird
	UPDATE	UPDATE Statement
	Connector	Verbindet verschiedene Elemente (z.B. Tabellenname und Spaltenname in: table.column)
Attribute		Gibt an, das es sich um einen Wert handelt, der vom Anfragesteller in das Statement eingesetzt wurde.
	ColumnIndex	Bei der Angabe handelt es sich um die Nummer einer Spalte.
	Column	Bei der Angabe handelt es sich um einen Spaltennamen.
	ColumnAlias	Es handelt sich um den Alias für eine Spaltenangabe oder einen Spaltenausdruck.
	Value	Gibt einen Wert im Statement an. (Beispielsweise 5 oder eine Zeichenkette)
	Table	Bei der Angabe handelt es sich um den Namen einer Tabelle.
	TableAlias	Bei der Angabe handelt es sich um den Alias Namen einer Tabelle.
	Schema	Bei der Angabe handelt es sich um ein Schema (aus dem die Tabelle stammt)
	LimitValue	Bei der Angabe von Limit verwendeter Wert
	TopValue	Bei der Angabe von TOP verwendeter Wert
	DataType	Bei der Angabe handelt es sich um einen Datentypen (Aus einem CREATE

		TABLE Statement)
	ColumnArgument	Bei der Angabe handelt es sich um ein Argument für eine Spalte. (z.B. die 32 in VARCHAR
	ColumnSpecifier	Bei der Angabe handelt es sich um eine einschränkende Angabe für die Spalte (z.B. NOT NULL)
	CreateTableIndex	Bei der Angabe handelt es sich um die Definition eines Indexs. (Nicht funktionsfähig)
	CreateTableOption	Bei der Angabe handelt es sich um eine Angabe die als Option für CREATE TABLE angegeben wurde. (Der Inhalt ist nur ein String)
	ElementName	Name eines Datenbankobjekts. Dies kann eine Tabelle oder ein anderes Objekt innerhalb der Datenbank sein.
	DropParameter	Weiterer Parameter eines DROP Statements.
Category		Verwendet, wenn einzelne Elemente zu besonderen Typ gehören, obwohl sie auch einem allgemeinen Typ angehören.
	ColumnExpression	Zeichnet einen Ausdruck als Spaltenangabe aus.
	OrderByElement	Ein Element das in einem OrderBy Ausdruck steht
	SelectExpressionItem	Das Element ist Teil eines Select-Ausdrucks
	JoinRightItem	Rechtes Element eines JOIN Ausdrucks
Group		Es handelt sich um einen Knoten der lediglich eine Gruppierende Funktion hat und mehrere andere Knoten umfasst.
	Function	Umfasst alle Elemente, die zu einer Funktion gehören
	Value	Umfasst einen Wert und seine syntaktischen Elemente (bspw. einen String in Hochkommata)
	Arithmetic	Umfasst arithmetische Ausdrücke mit ihren Parametern
	Logical	Umfasst logische Ausdrücke und ihre Parameter (z.B. AND, OR, =, !=, <, >, etc)
	Conditional	Umfasst Ausdrücke für situationsabhängige Kommandos (CASE, WHEN)
	Expressions	Umfasst mehrere Ausdrücke
	AllTableColumns	Umfasst den Namen einer Tabelle und die Angabe .*
	SelectItems	Umfasst eine Gruppe von Spaltennamen, die selektiert werden sollen.
	JoinElements	Umfasst die Angaben verschiedener Tabellen, die mit einem JOIN verbunden werden.
	JoinExpression	Umfasst die Angabe ON und alle JOIN Ausdrücke
	ColumnExpression	Fasst die Angaben zu einer Tabellenspalte zusammen. (Umfasst Schemaname, Tabellenname, Spaltenname)
	TableExpression	Fasst Angaben zu einer Tabelle zusammen
	SubArea	Umfasst einen SubArea Bereich mit Umklammerungen.
Subrea		Bezeichnet einen größeren Bereich, der separat geparst werden könnte. (z.B. Subselects)
	Select	Gibt an, das es sich um ein SubSelect handelt.
	Join	(Bisher nicht Implementiert)
Function		Es handelt sich um eine Funktion und deren Parameter.
	FunctionName	Die Angabe ist der Name einer Funktion.
Format		Formatierungen im original Statement.
	LineBreak	Stellt einen Zeilenumbruch dar.
	Space	Stellt ein Leerzeichen dar.