Bachelorarbeit

Konzepte zur Erstellung datenbewusster Tests anhand eines Fallbeispiels aus dem Unternehmenssektor

Guided Creation of Data-Aware Test Cases Based on a Real World Business Use Case

von

Frank Blechschmidt

Potsdam, Juli 2014

Betreuer

Prof. Dr. Hasso Plattner

Dr. Matthias Uflacker, Thomas Kowark, Keven Richly,

Ralf Teusner, Arian Treffer

Enterprise Platform and Integration Concepts

Kurzfassung

Kontinuierlich wachsende Datenmengen in Unternehmenssoftware beeinflussen zunehmend die Anwendungsentwicklung. Die frühzeitige Analyse von Datenbankinteraktionen hinsichtlich ihrer Performance ist deshalb bereits im Entwicklungsprozess essentiell, um die hohen Kosten von Nachbesserungen im operativen Geschäft zu vermeiden. Die durch den Programmfluss und die Parameter geprägten Anfragen an die Datenbank erfordern dabei passende Testwerte, welche auch die Randfälle der Anfrageausführung abdecken.

In dieser Bachelorarbeit stelle ich deshalb verschiedene Ansätze vor, die Entwicklern bereits während der Implementierung relevante Testdaten vorschlagen. Die unmittelbare Bereitstellung repräsentativer Testdaten in Echtzeit hilft, potentielle Stellen der Datenbankinteraktion, die die Skalierbarkeit beeinflussen, frühzeitig zu entdecken und zu beheben. Die vorgestellten Ansätze werden dazu in der praktischen Umsetzung eines Fallbeispiels aus dem Unternehmenssektor evaluiert.

Abstract

The continuously growing mass of data in enterprise applications has an increasing impact on the development process. Therefore it is essential to analyze the performance of database interactions during the development phase and therewith reducing the costs caused by the correction of defects in the operating phase. Database queries, which are driven by the control flow and their parameters, need appropriate test values, which also cover edge cases of the query execution.

This bachelor thesis describes approaches to generating suggestions of relevant test data for developers. The immediate provision of representative test data in real time helps to find and fix potential scalability issues in database interaction. The presented approaches are evaluated by the practical implementation of a business use case.

Danksagung

An dieser Stelle möchte ich mich recht herzlich bei all denjenigen bedanken, die mich während meines Studiums und insbesondere bei der Erstellung dieser Bachelorarbeit unterstützt und motiviert haben.

Mein besonderer Dank gilt den Betreuern des Bachelorprojektes: Thomas Kowark, Keven Richly, Ralf Teusner und Arian Treffer. Mit zahlreichen Ideen und Feedback und stets offenen Türen haben sie das Arbeiten am Projekt und die Erstellung dieser Arbeit bereichert.

Überdies danke ich meinen Freunden und Kommilitonen, die dieses Studium zu mehr als einer langweiligen Vorlesung gemacht haben.

Schlussendlich gilt mein Dank meiner Familie. Ihre stetige Unterstützung und Motivation haben mich während des Studiums begleitet, selbst wenn der "Große" nicht allzu oft die Heimreise antrat.

Hasso Plattner Institut

Eigenständigkeitserklärung

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, 14. Juli 2014

_____

(Frank Blechschmidt)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungverzeichnis

AST	Abstract Syntax Tree
BVC	Boundary Value Coverage
DSE	Dynamic Symbolic Execution
ER	Entity-Relationship
ERIC	Enterprise-Ready IDE Concepts
ERP	Enterprise Resource Planning
IDE	Integrated Development Environment
IT	Information Technology
JSON	JavaScript Object Notation
LC	Logical Coverage
SQL	Structured Query Language
UI	User Interface

1. Einleitung

In vielen Bereichen, vor allem bei Geschäftsanwendungen, dienen relationale Datenbanken als essentielle Persistenzschicht für die anfallenden Daten. Bei der Entwicklung solcher Anwendungen spielt neben der Validität auch die Performance eine wichtige Rolle. Wichtigster Indikator dafür ist die tolerierbare Wartezeit, die sich laut psychologischen Studien schon nach 2 Sekunden negativ auf die Aufmerksamkeit der Nutzer auswirkt, sodass sie in ihrem Denkprozess unterbrochen werden [Nah04]. Ein zweiter wichtiger Aspekt ist die Entwicklung anhand von Echtdaten. Sie wird als Best Practice betrachtet [Pla13], da sie die Charakteristiken der realen Welt als Maßstab nutzt. Das frühzeitige Betrachten der Performance einer Anwendung auf Echtdaten liegt so in der Verantwortung des Entwicklers und soll von Anfang an in den Entwicklungsprozess einfließen.

Um Informationen aus der Datenbank in der Anwendung zu nutzen, erfolgt der Zugriff durch das Einbetten von in SQL geschriebenen Anfragen. Die eingebetteten Datenbankanfragen haben zumeist variable Bestandteile und Parameter, für die für Performance-Messungen und -Analysen passende Testwerte ausgewählt werden müssen. Dies wird jedoch häufig durch riesige Datenmengen in unverständlich benannten Relationen und Attributen zusätzlich erschwert. Mit dem Ziel der Vereinfachung werden in dieser Bachelorarbeit verschiedene Ansätze diskutiert, die das Auswählen relevanter Testwerte durch sinnvolle Vorschläge anhand der Daten und Metainformationen aus der Datenbank unterstützen.

Die vorgestellten Ansätze bilden einen essentiellen Teil in einer Reihe von Konzepten für Entwicklungsumgebungen, die bei der Entwicklung von Geschäftsanwendungen assistieren. Sie werden in Kapitel 3 erläutert. Anschließend wird in Kapitel 4 die Einbettung von SQL in die Programmiersprache der Anwendungen untersucht, um auf deren Basis die in Kapitel 5 vorgestellten Algorithmen zur Vorschlagsgenerierung von Testdaten darzulegen. In Kapitel 6 wird ergänzend die administrative Architektur für Testdaten und -systeme betrachtet. Abschließend werden in Kapitel 7 die vorgestellten Ansätze im Rahmen der Umsetzung einer praxisnahen Geschäftsanwendung evaluiert.

2. Verwandte Forschungsarbeiten

Relevante Testdaten sind in verschiedenen Phasen im Entwicklungsprozess einer Anwendung von hoher Wichtigkeit. Um Geschäftsanwendungen zu testen, gibt es neben dem Mittel, die Datenbankanbindung zu mocken [TZX10] auch die Möglichkeit, sie direkt mit einzubeziehen. Bei diesem Ansatz werden die Eingabewerte der Tests mit dem Zustand der Datenbank in Verbindung gebracht. Dazu gibt es verschiedene Vorgehensweisen.

2.1. Eingabewerte zum Testen von Datenbankanwendungen

Mit dem Erzeugen relevanter Eingabewerte für funktionale Anwendungstests hat sich in den letzten Jahren eine Reihe von Forschungsprojekten beschäftigt. Der Fokus der nachfolgend beschriebenen Ansätze liegt dabei, im Kontrast zu dieser Bachelorarbeit, auf der Abdeckung und Zweigüberdeckung von Anwendungstests durch Einbeziehung des Status der Datenbank.

Im Sektor Datenbankanwendungstests bietet das AGENDA-Framework [CDF+00, Cha04, CDF+04, DFC05, CSF08] eine Palette an Werkzeugen für das funktionale Testen. Es nutzt die Metainformationen der Datenbank in Kombination mit Voreinstellungen vom Entwickler, um eine Testdatenbank synthetisch zu erzeugen. Mithilfe dieser Datenbank wird die Quellcodeabdeckung der funktionalen Tests erhöht. Im Gegensatz dazu, liegt der Schwerpunkt dieser Bachelorarbeit auf dem nicht-funktionalen Testen der Performance der Anwendung anhand von Echtdaten. Dabei werden ähnlich dem AGENDA-Framework auch die Metainformationen der Datenbank mit einbezogen.

Auf Basis von Microsofts Testing-Framework Pex für die .Net-Plattform [TDH08] entwickelten Pan et al. mehrere Erweiterungen [PWX11b, PWX11a], die die Quellcodeabdeckung durch Einbeziehung der Datenbank und ihrer Daten erhöhen. Dabei werden mittels Dynamic Symbolic Execution (DSE) [CGP+06, GKS05] sowohl die Variablen, die in SQL-Statements einfließen, als auch ihre Änderungen im Programmfluss nachverfolgt, um daraus passende Eingaben zu generieren [PWX11b]. Durch die zusätzlichen Anforderungen

an Logical Coverage (LC) [AOH03] und Boundary Value Coverage (BVC) [KLPU04] werden die gefunden Variablen im Zusammenhang mit Bedingungen innerhalb der Anwendung betrachtet, wodurch gegebenenfalls weitere Eingabewerte erzeugt werden.

Der Ansatz der symbolischen Ausführung ist auch in die Implementierung der Web-IDE eingeflossen, um die zu betrachtenden Abhängigkeiten von Datenbankzugriffen zu ermitteln. Ebenso findet in den Ansätzen dieser Bachelorarbeit zur Ermittlung der verschiedenen Ausprägungen von SQL-Statements eine Betrachtung von Bedingungen innerhalb des Kontrollflusses statt.

2.2. Liveanalyse von Datenbankanwendungen

Das Monitoring ist eine alternative Möglichkeit, die Performance einer Datenbankanwendung zu ermitteln. Softwarelösungen wie New Relic[1] betten eigene Komponenten in Anwendungen ein, um Metriken aus dem laufenden Betrieb aufzunehmen und zu analysieren. Sie können dann unter anderem die langsamsten SQL-Statements mitsamt ihren Parametern dem Entwickler anzeigen. Für dieses Verfahren ist es allerdings erforderlich, dass SQL-Statements vollständig erfasst und ihre variablen Bestandteile mit Werten gefüllt sind. Die vorgestellte Entwicklungsumgebung ermöglicht es hingegen, schon in der Entwicklungsphase auch partielle Datenbankabfragen zu analysieren und mit verschiedenen Werten zu testen. Eine Erweiterung der vorgestellten Algorithmen, Parameter aus den Ausführungsdaten zu extrahieren, würde beide Ideen kombinieren. So könnten häufig genutzte Werte aus dem Betrieb für Vorschläge von Testdaten zur Weiterentwicklung der Anwendung genutzt werden.

[1] `http://newrelic.com/`

3. ERIC: Eine IDE für daten- und performancebewusste Entwicklung

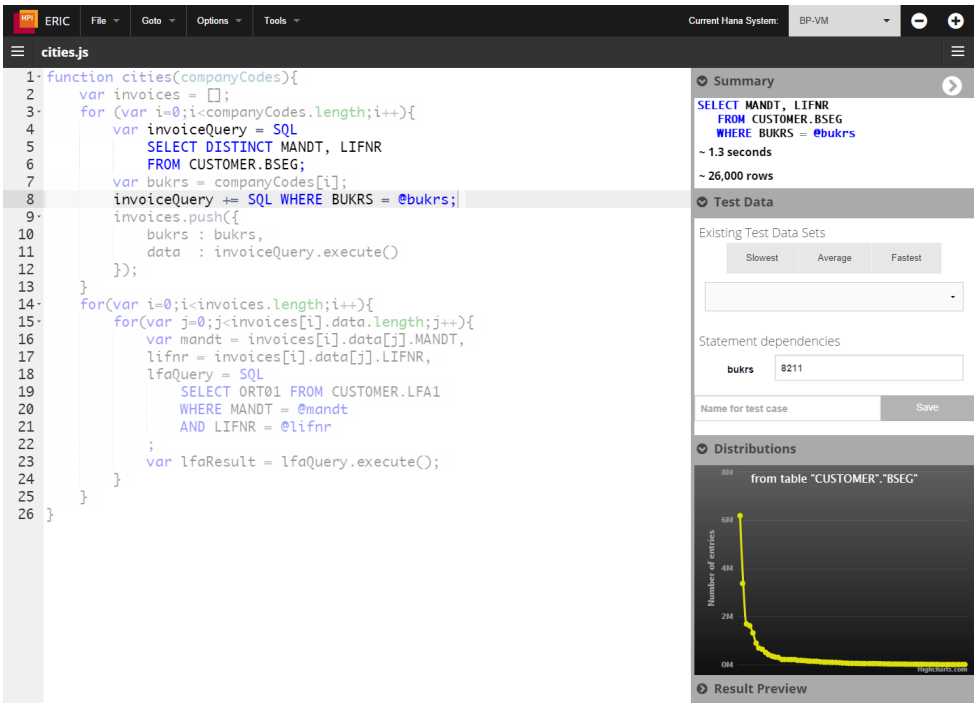

Abbildung 1: Screenshot der Web-IDE »ERIC«

Die im Rahmen des Bachelorprojektes “Modern Computer-Aided Software Engineering” entwickelte Web-IDE namens »ERIC« (Abbildung 1) vereint eine Reihe von Konzepten zur Entwicklung von Geschäftsanwendungen mit dem Fokus auf der besseren Integration von Informationen aus Datenbanken. Besonders die Schärfung des Bewusstseins für Daten und Datenmengen sowie das vorausschauende Entwickeln in Hinblick auf die Skalierung der Anwendung soll gefördert werden. Grundlage dafür bietet die Einbettung von SQL in die Programmiersprache der Geschäftsanwendung [Hor14] (mehr Details dazu in Kapitel 4). Durch das Parsen des Quellcodes [Hor14] und der darin enthaltenen SQL-Statements [Sch14] werden die Voraussetzungen geschaffen, Analysen und Visualisierungen der Datenabfragen durchzuführen. Unter anderem kann anschließend eine Abschätzung über die Laufzeiten und Ergebnisgrößen von

SQL-Statements gegeben werden, sowie eine Vorschau der Ergebnisse und die Verteilung der Daten in den angefragten Spalten. Für die Berechnung der abgeschätzten Laufzeiten und Ergebnisgrößen stehen zwei Verfahren zur Auswahl: beim Sampling [Exn14] werden mithilfe von Teilmengen der Relationen ungefähre Größen hochgerechnet und durch den Ansatz des Machine Learnings [Mue14] können Ergebnisse vergangener Anfragen als Grundlage der Berechnung genutzt werden.

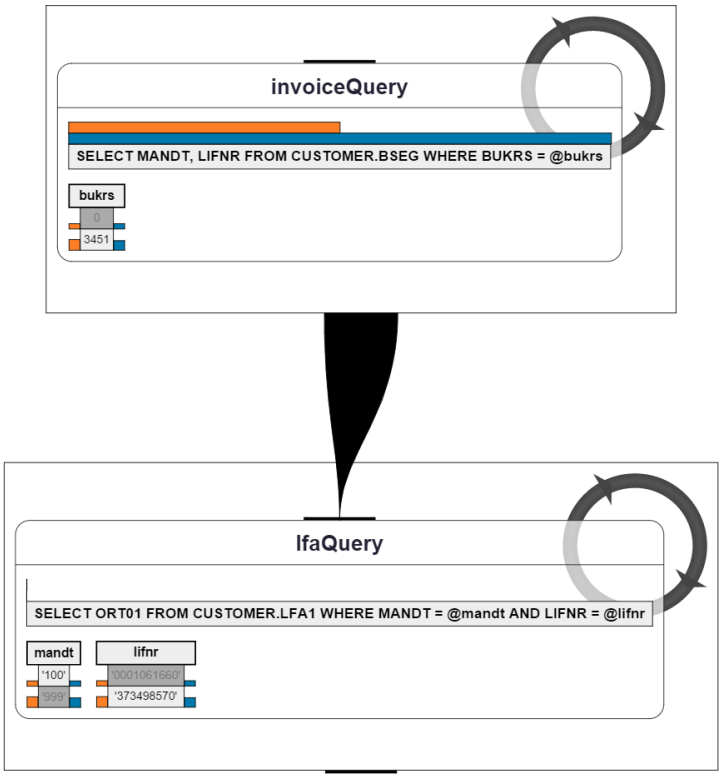

Abbildung 2: Visualisierung des Kontrollflusses

Zusätzlich ist es möglich, den Verlauf des Kontrollflusses im Zusammenhang mit den Datenanfragen zu visualisieren [Fra14] (siehe Abbildung 2) um die Ursachen von Performance-Problemen ausfindig zu machen.

Die betrachteten Features für die Analysen von SQL-Statements haben dabei zwei Voraussetzungen: das SQL-Statement muss komplett erfasst und dessen variable Parameter müssen testweise mit Werten belegt sein.

```
1   var customer = request.body.customer,
2       negative = request.body.negative,
3       filter = [],
4       stmt = "SELECT * FROM CUSTOMER.BSEG";
5   if(negative)
6     filter.push("BSEG.XNEGP = 'X'");
7   if(customer)
8     filter.push("BSEG.KUNNR = '" + customer + "'");
9   if(filter.length > 0)
10    stmt += " WHERE " +  filter.join(" and ");
```

Code-Beispiel 1: Variablen nehmen Einfluss auf die SQL-Query und -Parameter

Schon einfache Algorithmen, wie im Code-Beispiel 1, lassen das SQL-Statement auf Basis von Programmvariablen variieren (`negative`) und belegen die SQL-Parameter in Abhängigkeit von beispielsweise Sitzungsdaten, Formularen oder Anfrageparametern mit unterschiedlichen Werten (`customer`). Dadurch ist es für den Entwickler schwer abzuschätzen, welche Testwerte repräsentativ sind oder sogar Randfälle darstellen und die Antwortzeit der Anwendung in die Höhe treiben. Deshalb ist es wichtig, sinnvolle Testwerte und Kombinationen von Testwerten zu nutzen, die die verschiedenen Szenarien innerhalb der Anwendung abdecken. Mit der Theorie und möglichen Algorithmen zum Vorschlagen dieser Daten beschäftigt sich diese Bachelorarbeit und diskutiert sie in den folgenden Kapiteln.

4. Verknüpfung von SQL mit dem Anwendungskontext

Mit der Einbettung von SQL-Anfragen in andere Programmiersprachen, zum Beispiel in JavaScript[2], treffen zwei unterschiedliche Konzepte aufeinander: die imperative Programmiersprache der Anwendung und die deklarative Abfragesprache der Datenbank.

Häufig fließen dabei Informationen aus dem Kontext der Anwendung in das SQL-Statement ein, zum Beispiel als Parameter für Filterbedingungen. Ebenso

[2]ECMAScript Language Specification: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`

können sie sich durch die Steuerung des Kontrollflusses auf das Erstellen von SQL-Statements auswirken. Im Folgenden werden deshalb die verschiedenen Varianten von SQL-Statements, deren Erstellung sowie Zusammenhänge mit Quellcodevariablen dargestellt.

4.1. Dynamische Erstellung von SQL-Statements

SQL-Statements können auf verschiedenste Weisen in den Quellcode einer Anwendung integriert werden, die sich vor allem durch die Stärke der Bindung von SQL-Statements und dem umliegenden Anwendungskontext unterscheiden. Grundlegend kann man drei Arten differenzieren:

Statische SQL-Statements

```
1    var stmt = "SELECT *
2    FROM CUSTOMER.BSEG LEFT OUTER JOIN CUSTOMER.LFA1
3    ON BSEG.MANDT = LFA1.MANDT AND BSEG.LIFNR = LFA1.LIFNR";
4    stmt.execute();
```

Code-Beispiel 2: Statisches SQL-Statement eingebettet im Quellcode

Statische SQL-Statements bleiben über den Kontrollfluss hinweg unverändert und sind unabhängig vom umliegenden Kontext der Anwendung (siehe Code-Beispiel 2). Sie können für Anfragen genutzt werden, die jederzeit dieselben Informationen aus der Datenbank auslesen (zum Beispiel das Auflisten aller Kunden). Diese Datenbankanfragen sind für Analysen leicht aus dem Quellcode herauszuparsen und müssen nicht verändert oder mit Testwerten ergänzt werden.

Prepared SQL-Statements

```
1    var stmt = con.prepareStatement("
2    SELECT *
3    FROM CUSTOMER.BSEG LEFT OUTER JOIN CUSTOMER.LFA1
4    ON BSEG.MANDT = LFA1.MANDT AND BSEG.LIFNR = LFA1.LIFNR
5    WHERE BSEG.KUNNR = ?");
6    stmt.setInt(1, 23342341).execute();
```

Code-Beispiel 3: Prepared Statements eingebettet im Quellcode

Der Anwendungsfall von Prepared Statements ist das mehrfache Ausführen derselben Anfrage mit verschiedenen Parameterwerten. Dabei werden die variablen Stellen mit Fragezeichen versehen und vor der Datenabfrage explizit gesetzt. Die im Code-Beispiel 3 gezeigte Variante setzt dabei in Zeile 6 einen konstanten Wert (23342341) für `BSEG.KUNNR` ein. Häufig kommen diesen Informationen aus der Anfrage vom Nutzer oder Sitzungsdaten und sind damit nicht als Konstanten im Quellcode erfasst. Das SQL-Statement ist somit abhängig von den zu setzenden Parameterwerten, wird jedoch nicht strukturell verändert.

Dynamische SQL-Statements

```
1   var customer = request.body.customer,
2       customerTo = request.body.customerTo,
3       stmt = "SELECT *
4         FROM CUSTOMER.BSEG LEFT OUTER JOIN CUSTOMER.LFA1
5         ON BSEG.MANDT = LFA1.MANDT AND BSEG.LIFNR = LFA1.LIFNR
6         WHERE ";
7   if(customer && !customerTo){
8     stmt += "BSEG.KUNNR = '" + customer + "'";
9   }
10  if(customer && customerTo){
11    stmt += "BSEG.KUNNR BETWEEN '" + customer +
12            "' AND '" + customerTo + "'" ;
13  }
```

Code-Beispiel 4: Der Kontrollfluss verändert dynamische SQL-Statements

Dynamische SQL-Statements werden erst zur Laufzeit in Abhängigkeit vom Kontrollfluss des Programms erstellt. So ist es möglich, Variablen aus der Anwendung sowohl zur Anpassung des SQL-Statements zu nutzen, als auch als Parameter für die Abfrage. Es entsteht eine enge Bindung des Kontrollflusses an das resultierende SQL-Statement, wodurch die Variabilität steigt, aber auch mit zunehmender Komplexität das Lesen und Verstehen der Anwendung erschwert wird. Im Code-Beispiel 4 verändert sich das SQL-Statement durch das Setzen bzw. Nicht-Setzen von Anfrageparametern durch den Nutzer. Dabei kann der Nutzer entscheiden, ob er die Informationen für eine konkrete Kundennummer (Zeile 5 bis 7) oder für ein Intervall von Kundennummern (Zeile 8 bis 11) abruft. In beiden Fällen gibt es einen konstanten Part der Anfrage (Zeile 1 bis

4) und es fließen die Anfrageparameter in das SQL-Statement ein (`customer`, `customerTo`).

Im folgenden Abschnitt werden diese Beziehungen auf Basis von Variablen aus dem Kontext der Anwendung ausführlicher untersucht.

4.2. Verknüpfung von SQL-Parametern und Quellcodevariablen

Aus dem Code-Beispiel 4 geht bereits deutlich hervor, dass Kontextvariablen einen Einfluss auf das SQL-Statement und vorrangig dessen Parameter haben. Allerdings wurden in den vorherigen Beispielen SQL-Teile stets nur als Zeichenketten in der Anwendung genutzt. Durch diese Umwandlung verlieren sie ihre Komfortfunktionen (z.B. Autovervollständigung und Syntaxüberprüfung) und bergen zeitgleich das Risiko von Fehlern, zum Beispiel durch vergessene Leerzeichen, ohne die das finale SQL-Statement nicht valide wäre. Um die Nachteile dieser Konkatenation von Zeichenketten abzuschaffen, kann der Quellcode in die Syntax nach Horschig [Hor14] übertragen werden (vgl. Code-Beispiel 5). Durch die Einbettung der Datenbankanfragen nach dem Prinzip von Language Boxes [DT13] werden die Konzepte von SQL und der umgebenen Sprache miteinander kombiniert und es entstehen Synergien, die die Entwicklung der Anwendung vereinfachen und den Quellcode leichter verständlich machen, zum Beispiel durch die explizite Verknüpfung von Quellcodevariablen mit SQL-Statements und -Parametern.

```
1   var customer = request.body.customer,
2       customerTo = request.body.customerTo,
3       stmt = SQL[CUSTOMER]
4         SELECT *
5         FROM BSEG LEFT OUTER JOIN LFA1
6         ON BSEG.MANDT = LFA1.MANDT AND BSEG.LIFNR = LFA1.LIFNR;
7   if(customer && !customerTo){
8     stmt += SQL WHERE BSEG.KUNNR = @customer;
9   }
10  if(customer && customerTo){
11    stmt += SQL WHERE BSEG.KUNNR BETWEEN @customer AND @customerTo;
12  }
```

Code-Beispiel 5: Darstellung des Code-Beispiels 4 in der Syntax nach [Hor14]

Die einzelnen SQL-Blöcke im Code-Beispiel 5 beginnen mit einer `SQL`-Anweisung, auf die der eigentliche SQL-Text folgt und mit einem Semikolon abgeschlossen wird. In Zeile 3 wird zusätzlich das Schema `CUSTOMER` für das Statement `stmt` festgelegt. Markant dabei ist die Verwendung des Zeichens `@`. Durch diese Anweisung wird der zum Zeitpunkt der Definition des SQL-Statements aktuelle Wert der Variable `customer` fest im SQL-Statement gesetzt.

Um eine Wiederverwendung von SQL-Blöcken zu ermöglichen, können sie ähnlich wie Funktionen als SQL-Templates [Hor14] formuliert werden.

```
1  var toleranceDays = request.body.toleranceDays,
2      percentageRate = request.body.percentageRate,
3      getFunctionParameters = SQL[CUSTOMER](xskr1)
4        REGUP.BUDAT, REGUP.WSKTO, REGUP.BLDAT,
5        :xskr1, @toleranceDays, @percentageRate;
6  getFunctionParameters(11).execute();
```

Code-Beispiel 6: SQL-Templates ermöglichen Wiederverwendung

Neben dem `@` kann so zusätzlich `:` zur Referenzierung dienen. Der Unterschied liegt darin, dass der Wert des Parameters (`xskr1`) erst beim Aufruf des SQL-Templates festgesetzt wird, die Werte von `toleranceDays` und `percentageRate` hingegen zum Zeitpunkt der Definition des SQL-Statements. Das SQL-Template `getFunctionParameters` kann anschließend mittels der Anweisung `+=` einem existieren SQL-Statement angefügt werden.

Eine solche Wiederverwendung ist z.B. nützlich bei SQL-Statements, die in einer Schleife verwendet werden und ein oder mehr variable Bestandteile haben. In diesem Fall wird die Anfrage nur einmal als SQL-Template definiert und kann innerhalb der Schleife mit den entsprechenden Parameterwerten aufgerufen werden.

An diesen Code-Beispielen wird deutlich, dass Variablen häufig ein SQL-Statement verändern und in dieses an verschiedenen Stellen einfließen. Im Folgenden wird deshalb eine Unterscheidung der Variablenverbindung zwischen dem eingebetteten SQL und dem umliegende Quellcode vorgenommen.

4.3. Differenzierung von Kontrollfluss- und SQL-Statement-Abhängigkeiten

Neben dem direkten Einfließen von Variablen in ein SQL-Statement (vgl. Kapitel 4.2), können diese auch (nur) als Abhängigkeit im Kontrollfluss auftreten.

```
1   var noDunning = request.body.noDunning,
2       customer = request.body.customer,
3       selection = request.body.selection;
4   var stmt = SQL[CUSTOMER]
5     SELECT @selection
6     FROM BSEG LEFT OUTER JOIN LFA1
7     ON BSEG.MANDT = LFA1.MANDT AND BSEG.LIFNR = LFA1.LIFNR;
8   if(noDunning){
9     stmt += SQL WHERE BSEG.MANST = 0;
10  }
11  if(customer){
12    stmt += SQL WHERE BSEG.KUNNR = @customer;
13  }
```

Code-Beispiel 7: Verschiedene Arten der Abhängigkeit von Variablen

Im Code-Beispiel 7 sind alle drei Möglichkeiten der Einflussnahme auf SQL-Statements durch Variablen dargestellt. Sie können direkt in das SQL-Statement eingebunden werden (`selection`), das SQL-Statement in der Struktur manipulieren (`noDunning`) oder beides zugleich vornehmen (`customer`).

Schon bei diesen einfachen Algorithmen ist es für Entwickler schwer, relevante Testwerte zu finden, um Analysen des resultierenden SQL-Statements zu ermöglichen, ohne z.B. die Inhalte der Relationen der zugrundeliegenden Datenbank zu kennen. Zudem können die Relationen kryptische Werte in unverständlich benannten Spalten enthalten. Beispielsweise bedeutet in einem SAP ERP-System der Eintrag `R` in der Spalte `BSEG.ZLSCH`, dass eine Rechnung mittels Euroüberweisung beglichen wurde.

Zusätzlich stellt die Variation an Verknüpfungen von Variablen und SQL-Statements eine Herausforderung für das Vorschlagen passender Testdaten dar, denn nicht immer können Informationen aus der Datenbank genutzt werden. Aus diesem Grund werden in Kapitel 5 verschiedene Lösungsstrategien und

Ansätze erörtert, die den Entwickler unterstützen, repräsentative Testwerte durch passende Vorschläge zu finden, um aussagekräftige Analysen auf SQL-Statements zu ermöglichen.

5. Generierung von Vorschlägen für Testdaten

Es gibt verschiedene Strategien dem Entwickler repräsentative Testwerte vorzuschlagen, die die Variationspunkte eines SQL-Statements beeinflussen. Als Grundlage dafür dient neben dem Quellcode auch das Datenbanksystem mit den enthaltenen Echtdaten. In den folgenden Beispielen werden Unternehmensdaten aus einer SAP-Infrastruktur genutzt. Die Integration solcher Datenbanksysteme und die Administration der genutzten Testdaten werden in Kapitel 6 ausführlicher behandelt.

Neben der Betrachtung der Charakteristiken von Daten innerhalb der Datenbank und dem Kontext aus dem Quellcode ist vor allem die Verknüpfung mit Analyseergebnissen, vorrangig den Laufzeitmessungen, ein Kriterium für die Generierung der Vorschläge. Mittels Auswahl unterschiedlicher, vorgeschlagener Testwerte ist es dem Entwickler möglich, konkrete Ausprägungen von SQL-Statements nachzuvollziehen und auf Grundlage der Auswertung der Messungen gegebenenfalls Optimierungen durchzuführen bis das gewünschte Performance-Verhalten erreicht ist. Doch zuerst müssen die Variablen ermittelt werden, die einen Einfluss auf das SQL-Statement haben.

5.1. Bestimmung der zu betrachtenden Variablen

Kapitel 4.3 zeigte bereits auf, dass Variablen auf unterschiedliche Weisen auf ein SQL-Statement einwirken können. Dies schlägt sich auch auf die Vorschlagsgenerierung nieder. Um die Variablen, die auf ein SQL-Statement Einfluss nehmen, zu bestimmen, dient die Schnittstelle `javascriptParser.getSqlQueryAtPosition()` des Quellcode-Parsers [Hor14]. Sie liefert ein Objekt mit zwei wichtigen Attributen zurück: `dependencies` und `variables`. Dabei enthält `dependencies` alle Varia-

blen, von denen das SQL-Statement im Kontrollfluss abhängig ist, wohingegen `variables` alle Variablen umfasst, die innerhalb des SQL-Statements auftreten. Darüber hinaus können Variablen auch in beiden Listen vorkommen (die Verknüpfung erfolgt in diesem Fall anhand des Attributs `uniqueName`). Die Elemente in `variables` enthalten zusätzlich Kontextinformationen, die eine Zuordnung zu Spalten innerhalb einer Relation ermöglichen, und so die Grundlage für die Generierung von Testdatenvorschlägen schaffen. In Abbildung 3 ist eine solche Datenstruktur beispielhaft für Code-Beispiel 7 dargestellt.

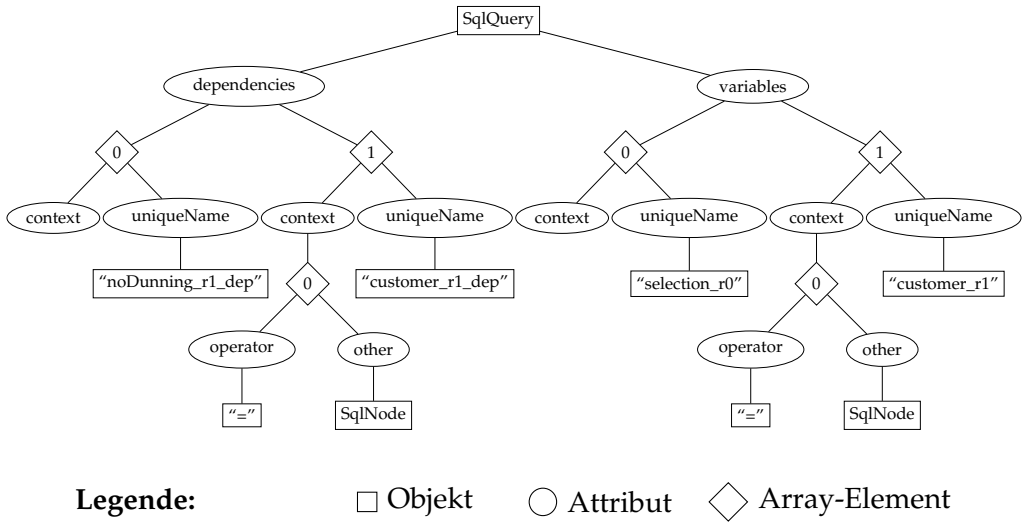

Abbildung 3: Datenstruktur zum Code-Beispiel 7

5.2. Vorschläge auf Basis von Datencharakteristiken

Sobald Variablen nicht nur die Gestalt eines SQL-Statements variieren, sondern auch darin einfließen, ist der erste Anhaltspunkt für das Vorschlagen relevanter Testwerte die Charakteristik der Datenbankinhalte. Für Selektionsfilter spielen dabei primär die Verteilung der Daten innerhalb der Relationen sowie die Anzahl ihrer unterschiedlichen Werteausprägungen eine Rolle. Im Gegensatz dazu werden für die Projektion, Sortierung und Gruppierung Metainformationen zu den Relationen der Datenbank berücksichtigt. Zweiteres wird in Kapitel 5.3 näher betrachtet.

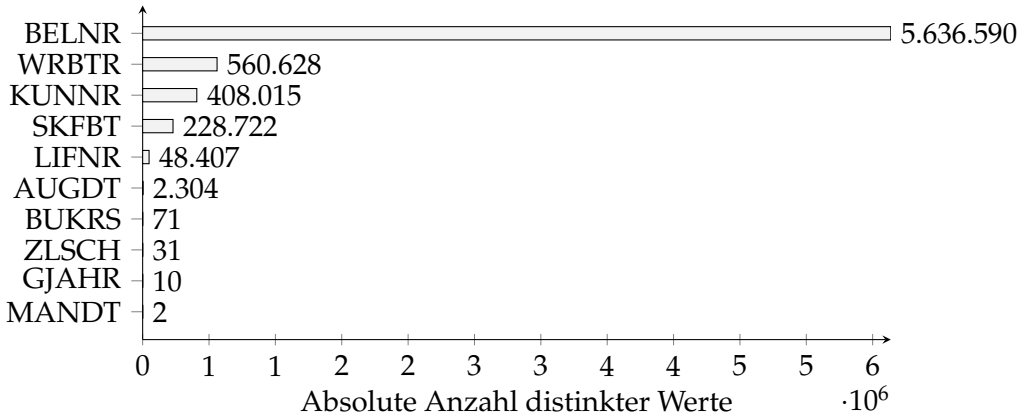

Abbildung 4: Verteilung distinkter Werten einer Auswahl von Spalten aus BSEG

Abbildung 4 zeigt eine Auswahl der 326 Spalten der BSEG-Relation mit der Anzahl deren distinkter Werte eines SAP-Systems. Die Relation enthält alle einzelnen Belegpositionen zu den Buchungsbelegen des Unternehmens. Eine Erläuterung zu den Bedeutungenn der einzelnen Spalten befindet sich im Anhang (Tabelle 7).

Besonders auffällig ist die starke Varianz der Anzahl von distinkten Werten im Vergleich der verschiedenen Spalten. Sollte die Anzahl der distinkten Werte einstellig sein (beispielsweise bei der Spalte MANDT), können dem Entwickler alle möglichen Ausprägungen in einem Auswahlmenü zur Verfügung gestellt werden. Damit wird gleichzeitig auch sichergestellt, dass nur Werte eingegeben werden können, die beim testweisen Ausführen des SQL-Statements ein Ergebnis zurückgeben. Auf der anderen Seite können sich Spalten jedoch über eine große Menge von verschiedenen Datenausprägungen erstrecken, wie zum Beispiel bei Belegnummern (BELNR), was eine einfache Auswahl passender Testdaten kompliziert gestaltet. Für diesen Fall werden Äquivalenzklassen anhand der Vorkommen der Werte erzeugt. Die Klassen von Bedeutung umfassen die drei häufigsten Werte, die drei seltensten Werte und drei Werte um den Median.

Für die häufigsten Werte wird eine aufsteigende Sortierung anhand der Anzahl ihres Vorkommens vorgenommen und die ersten drei selektiert. Sollten mehrere Werte dieselbe Anzahl an Vorkommen vorweisen, werden sie zusätzlich an-

hand ihrer Werte aufsteigend sortiert. Im Unterschied dazu wird bei der Klasse der seltensten Werten initial eine absteigende Sortierung vorgenommen. Für die Werte um den Median muss zuerst die Anzahl der verschiedene Werte ermittelt werden. Anschließend dient die Halbierung des sortierten Ergebnisses als Offset für die Bestimmung der drei vorzuschlagenden Werte. Die SQL-Statements dazu befinden sich als Code-Beispiel 13 im Anhang.

5.3. Vorschläge anhand von Metainformationen

Für Projektion, Sortierung oder Gruppierung sind weniger die Spalteninhalte wichtig, als vielmehr die Spalten an sich. Um Vorschläge für diese Art SQL-Statement-Variablen zu erzeugen, werden anhand der Metainformationen über Spalten der betrachteten Relation aus der SAP HANA-internen Systemtabelle `SYS.TABLE_COLUMNS` die Spaltennamen bestimmt (vgl. Code-Beispiel8).

```
1 SELECT COLUMN_NAME
2 FROM SYS.TABLE_COLUMNS
3 WHERE SCHEMA_NAME = '<schema>'
4 AND TABLE_NAME = '<table>'
```

Code-Beispiel 8: SQL-Statement für Metainformationen zu Relationen

Die notwendigen Informationen (Schema und Relation) werden dazu dem SQL-Statement entnommen.

Nutzung von Metainformationen für UI-Elemente

Die Metainformationen zu Spalten umfassen neben den Namen auch weitere nützliche Informationen, die die Eingabe von Testdaten unterstützen können. Beispielsweise können der Datentyp von Spalten (`DATA_TYPE_NAME`), die Erlaubnis der Eingabe von NULL-Werten (`IS_NULLABLE`) oder auch die maximale Länge für Eingaben in der betreffenden Spalte (`LENGTH`) für die Erstellung der Eingabefelder genutzt werden kann. So kann die Eingabe, die eine Spalte mit Datumsangaben referenziert, durch einen Kalender vereinfacht werden. Auch die Einschränkung auf Datentypen, z.B. Ganzzahlen, kann fehlerhafte Eingabe

durch den Entwickler verhindern. Eine weitere Unterstützung ist die Autovervollständigung von teilweise eingegebenen Werten. Mittels des SQL-Operators `LIKE` kann dazu nach Daten gesucht werden, die dem Muster der bisherigen Eingabe entsprechen. Diese Zusatzinformationen erhöhen zusammenfassend also den Komfort der Eingabe und können fehlerhafte Eingaben reduzieren.

5.4. Adaptive Vorschlagsgenerierung durch Laufzeitanalysen

Der in Kapitel 5.2 vorgestelle Ansatz zum Vorschlagen einzelner Testwerte stößt an seine Grenzen, sobald mehrere Parameter genutzt werden und diese voneinander abhängig sind. Im Code-Beispiel 4 aus Kapitel 4 werden beispielsweise offene Belege und deren Einzelpositionen gesucht. Die Variable `customer` gibt dabei die Kundennummer an, mit der die Rechnungen assoziiert sind. Wird zusätzlich ein Wert für `customerTo` festgelegt, erfolgt die Selektion anhand einer Bereichsanfrage. Man kann nun annehmen, dass die Kundennummern mit den größten Gesamtsummen auch die meisten Belege bzw. Belegposition haben. Tabelle 1 verdeutlicht das Gegenteil.

Kundennummer	Belege	Belegpositionen	Summe der Beträge
0000001201	12.555	19.449	72.160.054,92
0000003401	11.896	19.329	54.318.123,25
0000003501	12.380	24.571	52.813.094,78
0016207834	246	273	5.930.616.840,00
0015423702	29	31	5.306.744.980,00
0000001051	4.750	7.617	5.159.057.217,21

Tabelle 1: Auswahl an Einträgen der BSEG-Relation nach Kundennummer

Im oberen Teil der Tabelle sind die drei Kundennummern gelistet, die die meisten Belegpositionen umfassen. Der untere Teil enthält die Kundennummern mit den höchsten Gesamtsummen. Es gibt zwei Auffälligkeiten: die Belege haben im Durchschnitt nur sehr wenige Positionen und viele Belege bzw. Belegpositionen haben nicht automatisch eine hohe Gesamtsumme zur Folge. Würden nun Entscheidungen auf dieser Grundlage gefällt werden, z.B. zusätzliche Berechnun-

gen für Kunden mit einem Umsatzvolumen von über 1 Milliarde €, würden einfache Testwertvorschläge diesen Teilbereich nicht abdecken.

Diese noch recht einfache Abhängigkeit kann beliebig erweitert werden. Damit entstehen komplexe SQL- und Programmstrukturen, die durch das einfache Vorschlagen anhand von Charakteristiken einzelner Spalten nicht zwangsläufig die Randfälle aufzeigen, die der Entwickler sucht. Aus diesem Grund ist die Betrachtung von Messergebnissen aus Laufzeitanalysen ([Exn14], [Mue14]) eine sinnvolle Erweiterung, um die Genauigkeit bzw. den Nutzen der Vorschläge zu steigern. Die variablen Stellen von SQL-Statements werden dabei durch die vom Entwickler ausgewählten Werte gefüllt. Im nächsten Schritt werden nun diese atomaren Vorschläge kombiniert und mit dem dazugehörigen Ergebnis aus der Laufzeitanalyse verknüpft. Dies ermöglicht die Vergleichbarkeit verschiedener Konstellationen. Für die Erstellung der initialen Daten können zwei verschiedene Ansätze verfolgt werden.

Der offensichtliche Ansatz ist das Durchprobieren und Messen aller Kombinationen (Brute-Force). Der hohe Aufwand, besonders bei Relationen mit vielen Einträgen und distinkten Werten in den Spalten, stellt jedoch aufgrund der enormen Berechnungszeit ein großes Hindernis dar. Beispielsweise entstehen schon bei der Betrachtung von 3 Spalten mit jeweils 1000 distinkten Werten (typisch für z.B. Identifikationsmerkmale) eine Milliarde Kombinationen, die analysiert werden müssen.

Dem gegenüber steht der adaptive Ansatz, bei dem der Testdatenbestand kontinuierlich erweitert wird. Diese Variante speichert die Ergebnisse der Laufzeitanalysen mit den dazugehören Testdatenkonstellationen als Testdatensets. Sobald mehrere dieser Sets vorhanden sind, können dem Entwickler drei Vorauswahloptionen gegeben werden: das Testdatenset mit der höchsten Laufzeit, mit der geringsten Laufzeit und mit mittlerer Laufzeit.

Um für diese Methode eine Datengrundlage zu schaffen, stehen wiederum zwei Verfahren zur Auswahl. Mithilfe der in Kapitel 5.2 ermittelten Werte können alle möglichen Kombinationen von Tupelausbildungen erstellt werden. Dies entspricht einer Permutation der Werte aller betrachteten Spalten. Allerdings entsteht somit dasselbe Problem wie bei der Brute-Force-Methode: mit steigender

Anzahl an Spalten wächst die Anzahl der zu betrachtenden Tupel, selbst wenn diese keine Repräsentationen in der Relation vorweisen.

Aus diesem Grund werden die distinkten Kombinationen der betrachteten Spalten mit dem meisten Vorkommen innerhalb der Relation bestimmt (siehe Code-Beispiel 9). In Hinblick auf die Laufzeit der Analysen der einzelnen Kombinationen, wurde eine derzeitige Obergrenze von 50 zu betrachtenden Tupeln festgelegt. Sollten Spalten über mehrere Relationen hinweg betrachtet werden, so erfolgt die Bestimmung auf Basis der Verknüpfung der Relationen.

```
1 SELECT TOP 50 DISTINCT <list of columns>, COUNT(*) AS OCCURENCES
2 FROM <table>
3 GROUP BY <list of columns>
4 ORDER BY OCCURENCES DESC
```

Code-Beispiel 9: Bestimmung distinkter Tupel anhand gegebener Spalten

Die gefundenen Tupel bilden mit Ergebnissen der Laufzeitanalysen die initialen Testdatensets, die durch die explorative Eingabe weiterer Werte durch den Entwickler kontinuierlich in ihrer Genauigkeit verbessert werden können. Sollten neue Konstellationen von Testwerten vorliegen, werden diese als Sets mitsamt ihrer Laufzeit in der Datenbank der Web-IDE hinterlegt. Die Ermittlung der Vorschläge an Testdatensets mit höchster, mittlerer und langsamster Laufzeit erfolgt durch Sortierung und Filterung anhand der gespeicherten Laufzeit.

5.5. Vorschläge für Variablenwerte ohne Datenbankbezug

Nicht immer müssen Variablen in ein SQL-Statement einfließen, um darauf Auswirkungen zu haben. Ein typisches Beispiel ist die Verwendung einer Variable als Bedingung, z.B. für Verzweigungen, zu sehen im Code-Beispiel 7 in der Nutzung der Variable `noDunning`. Das Setzen der Variable entscheidet über die Ergänzung des Selektionsfilters um die Bedingung `BSEG.MANST = 0`. Sie hat jedoch keinen Bezug zu einem Attribut aus der Datenbank. Für solche Fälle wird der Programmkontext der Variable für die Vorschläge der dazugehörigen Testwerte genutzt. Abbildung 5 zeigt für das Code-Beispiel 7 die notwendige Erweiterung des Baumes aus Abbildung 3.

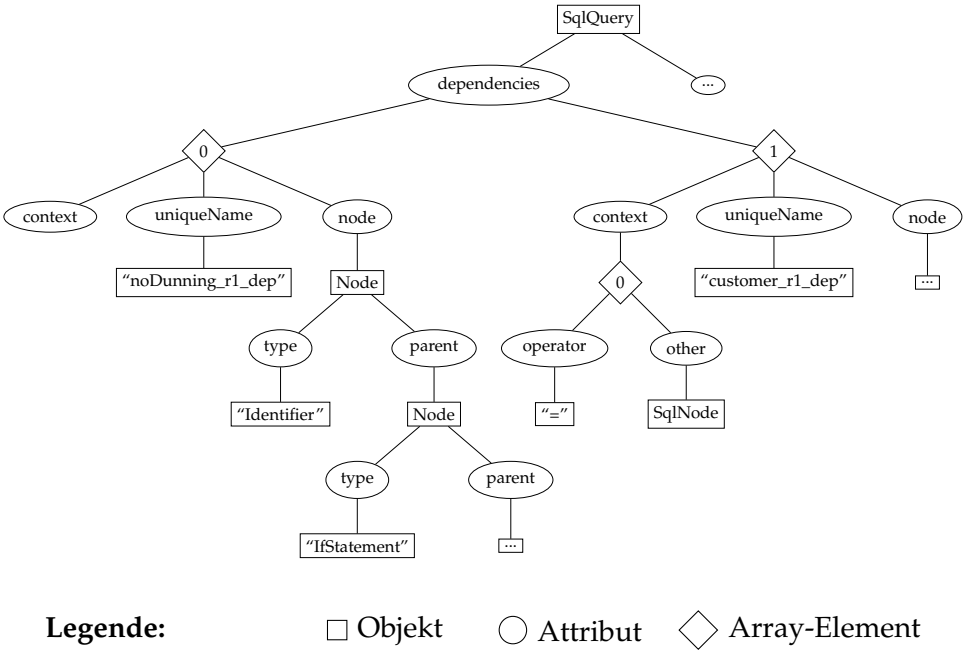

Abbildung 5: Erweiterung des Baumes aus Abbildung 3 für Variablen ohne Datenbankbezug

In den erkannten Abhängigkeiten eines SQL-Statements werden zusätzlich ihre Kontextinformationen als Knoten gespeichert. Diese dienen zur Einordnung innerhalb des vom Quellcodeparser erstellten AST [Hor14]. Durch Betrachtung der Typen der Vorfahren im Baum, in diesem Beispiel des Elternknotens vom Typ "IfStatement", können passende Vorschläge erzeugt werden. Ein Vorschlag für eine Variable umfasst immer 2 Werte: einer, der die assoziierte Bedingung erfüllt und einer, der sie nicht erfüllt. Tabelle 2 zeigt für verschiedene Knotentypen und Operatoren die Bestimmung der zwei Wertvorschläge bei Vergleichen mit Literalen. Für die Variable `noDunning` sind das `true` und `false`. Ein Vergleich, beispielsweise `if(amount > 5)` ergibt als Testwerte `5` (nicht erfüllend) und `5+1`, also `6` (erfüllend). In Abhängigkeit des Vergleichsoperators muss einer der beiden vorgeschlagene Werte angepasst werden um die Negation der Bedingung zu erreichen. Der Ansatz ist ähnlich dem Verfahren in Testing-Frameworks, um eine erhöhte Verzweigungsüberdeckung zu erreichen und unterstützt derzeit die drei primitiven Datentypen von JavaScript (Wahrheitswer-

te, Zahlen, Zeichenketten).

Elternknotentyp	Operator	Erfüllender Wert	Nichterfüllender Wert
IfStatement		`true`	`false`
UnaryExpression	`!`	`true`	`false`
BinaryExpression	`==`	`x`	`x+1`
		`x`	`x+'a'`
		`x`	`!x`
	`<`	`x-1`	`x`
	`<=`	`x`	`x+1`
	`>`	`x+1`	`x`
	`>=`	`x`	`x-1`

Tabelle 2: Testwertermittlung für Vergleiche einer Variable mit einem Literal `x`

5.6. Integration in die Web-IDE

Die in Kapitel 5.2 und 5.4 erzeugten Vorschläge werden in der zur Web-IDE gehörenden Datenbank gespeichert und in einer Sidebar im Frontend eingebunden (vgl. Abbildung 6). Durch das Auswählen einer Quellcodezeile mit SQL-Inhalt werden die Informationen zu dem dazugehörigen SQL-Statement aggregiert und aufbereitet (vgl. Kapitel 3) und schließlich in der Sidebar angezeigt.

Aus der in Kapitel 5.4 ermittelten Menge an Testdatensets werden zum einen die drei Sets mit der höchsten, geringsten und durchschnittlichen Laufzeit angeboten, zum anderen aber auch ein Menü mit allen gespeicherten Sets zur Auswahl gestellt. Die Wahl einer dieser Optionen füllt die Eingabefelder für die Testdaten automatisch mit den gespeicherten Werten und löst eine Analyse aus. Möchte der Entwickler weitere Testdatensets hinzufügen, so kann er diese benennen und direkt abspeichern. Die Verwaltung solchermaßen gespeicherten Daten wird im folgenden Kapitel behandelt.

Abbildung 6: Integration des Werkzeugs in die Sidebar der Web-IDE

6. Verwaltung von Testdaten und -systemen

An das Backend der Web-IDE ist eine eigene SAP HANA-Instanz angebunden. Darin werden die Testdatensets und Zugangsdaten für Testsysteme hinterlegt. Im Folgenden geht es um das zugrunde liegende Schema, die Integration verschiedener Testsysteme und das Pflegen der Testdaten, insbesondere im Hinblick auf deren Gültigkeit.

6.1. Datenschema für Testdatensets

Die größte Herausforderung beim Speichern der Testdatensets liegt in der Wiederzuordnung zu den Variablen in der vom Entwickler geöffneten Quellcodedatei. Der in der Web-IDE genutzte Quellcode-Parser [Hor14] nutzt symbolische Ausführung [Kin76] zur Bestimmung von Variablen und deren eventuell bereits festgelegten Werten. Die auf diesem Wege gefundenen Variablen bekommen eine eindeutige Kennzeichnung, die zum Identifizieren genutzt werden kann. Sollten die Variablen im Laufe des Programmflusses in mindestens ein SQL-Statement einfließen, werden sie als sogenannte Testvariablen betrachtet. Eine Menge von Testvariablen bildet zusammen mit dem Pfad der geöffneten Quell-

codedatei auf dem Server, der ermittelten Laufzeit und einem vom Entwickler angegebenen Namen ein Testdatenset. Diese Sets können wiederum in Beziehung mit dem Testsystem gebracht werden, auf dem die Performance-Analysen erfolgten, um den Einfluss der Systemauswahl auf die Laufzeiten der Testdatensets zu berücksichtigen. In Abbildung 7 ist das dazugehörige Datenschema dargestellt.

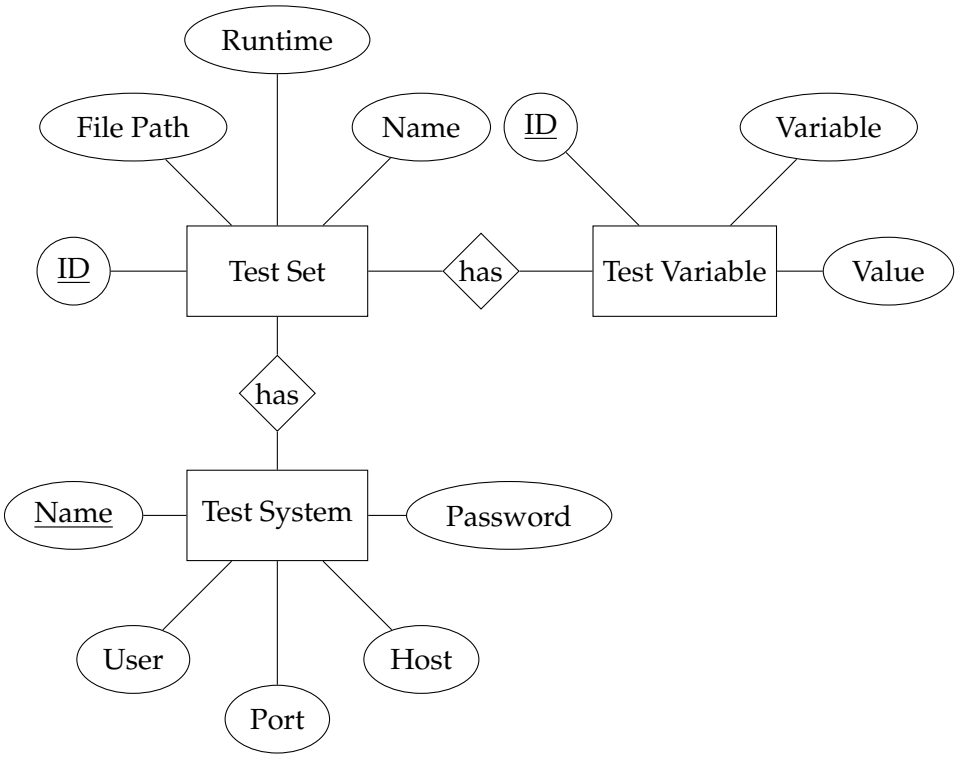

Abbildung 7: ER-Diagramm für Testdatensets und -systeme

6.2. Caching und Gültigkeit von Testdaten

Um nicht für jede Vorschlagsanfrage einen Datenbankzugriff durchzuführen, werden die Testdaten und Testdatensets sowohl im Frontend als auch im Backend gecached. Die Caches unterscheiden sich in der Hinsicht, dass es einen Frontendcache für jede Sitzung eines Entwicklers gibt, wohingegen der Backendcache global agiert. Eine Anfrage für Testdatenvorschläge zu einer

Spalte einer Relation wird dabei erst durch den Frontendcache zu beantworten versucht. Sollten dort keine passenden Daten vorliegen, wird eine Anfrage an das Backend ausgelöst. Dort versucht der Backendcache als erste Instanz, diese Anfrage zu beantworten. Sollten auch dort keine Daten vorliegen, werden die initialen Vorschläge mithilfe der in Kapitel 5 vorgestellten Algorithmen bestimmt und in den Caches hinterlegt.

Eine Herausforderung stellt das Überprüfen der Gültigkeit der Vorschläge und Testdatensets dar. Diese kann durch zwei Fälle beeinflusst werden: die Daten in der genutzten Datenbank werden verändert (erweitert, aktualisiert oder gelöscht) oder der Quellcode der Anwendung wird in einer Weise abgeändert, die die Variablen aus den SQL-Statements beeinflusst.

Für Ersteres gibt es Ansätze [HSN97], die kontinuierlich verändernde SQL-Statements nachverfolgen und dementsprechend ihren Algorithmus durch manuelles Mitzählen der Anzahl von Einträgen in der betrachteten Relation anpassen. Dieser Ansatz ist für die in Kapitel 5 beschriebenen Algorithmen nicht anwendbar, da sie nicht nur die Anzahl der Vorkommen innerhalb einer Relation betrachten, sondern darüber hinaus auch die Werteausprägungen und Analyseergebnisse für die Vorschlagsgenerierung berücksichtigen. Nichtsdestotrotz floss die Idee der Betrachtung von manipulierenden SQL-Statements mit in die Implementierung ein, indem ein Schwellenwert für Datenveränderungen festgelegt wurde (derzeit 10), ab dem eine Neubestimmung der Vorschläge und der initial berechneten Testdatensets erfolgt.

Die Invalidierung der Daten durch Veränderung des dazugehörigen Quellcodes wurde im Zuge dieser Arbeit nicht betrachtet, ist aber als Weiterentwicklung geplant. Die Herausforderung liegt in der Findung einer Granularität, auf der Änderungen zur Invalidierung führen, und gegebenenfalls diese Veränderung nachzuvollziehen. Legt man die Granularitätsstufe auf das gesamte Dokument fest, würde dies dazu führen, dass die Testdaten zu häufig als ungültig gekennzeichnet werden, obwohl das nicht zwangsläufig zutreffen muss. Wählt man eine feine Granularitätsstufe (zum Beispiel alle Änderungen, die nur Variablen betreffen, die in SQL-Statements einfließen), so ist eine komplexe Nachverfolgung der Änderungen am Dokument über dessen verschiedene Revisionen not-

wendig. Sollte ein Versionsverwaltungssystem für die Quellcodedateien genutzt werden, könnten dessen Informationen in der Nachverfolgung berücksichtigt werden.

6.3. Integration mehrerer Testsysteme

Typischerweise dient nicht nur ein System als Grundlage für Performance-Analysen einer Geschäftsanwendung, sondern eine Auswahl an Systemen mit unterschiedlichen Ausstattungsmerkmalen. Dies kann mehrere Gründe haben. In Hinblick auf die Daten können so Datenschutzbestimmungen eingehalten oder auch Datensätze mit verschiedene Charakteristiken getestet werden. Zum anderen kann so auch die Auswirkung verschiedener Systemkonfigurationen auf die Performance der Anwendung geprüft werden, wodurch Kosten gespart werden können, indem man eine passendes System für den Produktiveinsatz auswählt. Realisiert wird dies durch ein Menü in der Web-IDE (vgl. Abbildung 8) bei dem das gewünschte Testsystem ausgewählt oder aber auch neue hinzufügt oder existierende entfernt werden können. Die Zugangsdaten für die Systeme werden in der Datenbank der Web-IDE hinterlegt (vgl. Abbildung 7) und das derzeit vom Entwickler ausgewählte System in seinen Sitzungsdaten gespeichert.

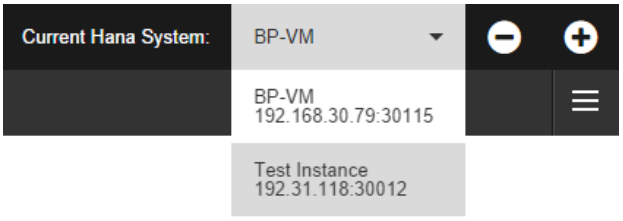

Abbildung 8: Auswahlmenü für verschiedene Datenbankserver

7. Fallbeispiel: Der Zahllauf

Im Bereich der Geschäftsanwendungen gibt es eine Reihe hochkomplexer Prozesse, die in der IT abgebildet werden. Einer davon ist der sogenannte Zahllauf, bei dem sich der Anwender eine Liste von offenen Zahlungen erst vorschlagen lässt, den Vorschlag überprüft und anschließend die Bearbeitung durch das System veranlasst. Dabei spielen insbesondere die Abhängigkeiten von Zahlungen, Lieferanten und Rabattverträgen eine wichtige Rolle, da sie im Idealfall in eine günstige Konstellation für das Unternehmen resultieren und dadurch Einsparungen bei den Ausgaben ermöglichen.

Aus diesem Grund hat die Implementierung des dazugehörigen Algorithmus viele Einflussfaktoren. Zusätzlich erschweren große Relationen mit stark abgekürzten Namen von Feldern und Werten, die als Eingabe, Zwischenspeicher und Ausgabe dienen, das Erstellen einer daten- und performancebewussten Umsetzung. Dies gilt besonders für domänenfremde Entwickler. Umso wichtiger ist es, bereits frühzeitig in der Entwicklung relevante Testdaten zu nutzen, die auch Randfälle der Implementierung abdecken und Engpässe aufzeigen. Dabei unterstützt die im Rahmen des Bachelorprojektes erstellte Web-IDE den Entwickler durch die in dieser Arbeit vorgestellten Ansätze zur Generierung von Testdatenvorschlägen.

In diesem Kapitel wird folgend exemplarisch die Implementierung des Zahllauf-Algorithmus mithilfe der Web-IDE vorgestellt. Dabei steht die von Nutzereingaben geprägte Verarbeitung der Suche nach Belegen im Fokus. Anschließend werden die vorgestellten Ansätze zur Testwertgenerierung anhand der Umsetzung des Fallbeispiels evaluiert.

7.1. Implementierung des Algorithmus in der Web-IDE

Die in Abbildung 9 dargestellte Filtereingabemaske des Zahllaufs hat zwei Bestandteile. Zum einen können allgemeine Parameter für die Suche festgelegt werden (z.B. das Ausführungsdatum (Run Date) und der Bereich der Lieferantennummern (Vendor)). Zum anderen dient die Eingabe in Form einer Ta-

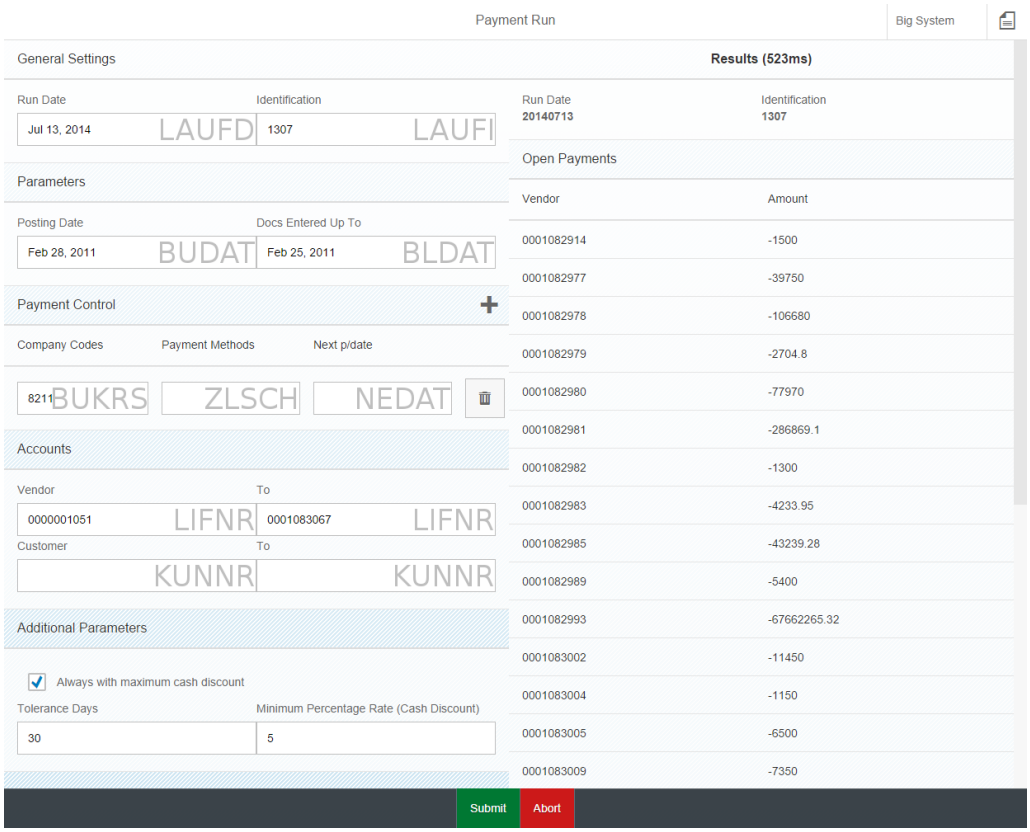

Abbildung 9: Eingabemaske des Zahllaufs

belle (Payment Control) zur Festlegung von zusammenhängende Suchkriterien. Jede Zeile der Tabelle besteht aus einem Filter nach Buchungskreis (Company Codes), Zahlmethoden (Payment Methods) und dem nächsten Ausführungsdatum (Next p/date) und entspricht damit einem eigenen Suchlauf.

Für die Implementierung des Frontends wurde das UI-Framework OpenUI5[3] von SAP genutzt. Der Fokus dieses Kapitels liegt jedoch auf der Backendimplementierung auf Basis der SAP XS-Engine[4].

[3] http://sap.github.io/openui5/
[4] http://help.sap.com/hana/SAP_HANA_XS_JavaScript_Reference_en/index.html

Der Algorithmus des Zahllaufs besteht aus 5 Schritten:

1. Parsen der JSON-Anfrage vom Frontend
2. Selektion von Belegpositionen und Speicherung in der REGUP-Relation
3. Berechnung von Zahldatum und Skonto
4. Speicherung der Beleginformationen in der REGUH-Relation
5. Senden einer JSON-Antwort

Die REGUP- und REGUH-Relationen dienen als datenbankinterne Zwischenspeicher für die ermittelten Ergebnisse. Zur Selektion wird die Relation BSIK als materialisierte Sicht auf die BSEG-Relation genutzt.

Die Eingaben des Nutzers in der Filtermaske wirken sich insbesondere auf die Selektion der Belegpositionen aus. Sie werden im ersten Schritt aus der JSON-Anfrage geparst (siehe Code-Beispiel 14 im Anhang) und fließen im Programmverlauf in das allgemeine SQL-Statement zur Selektion (siehe Code-Beispiel 16 im Anhang) ein.

Code-Beispiel 10 zeigt exemplarisch wie die Projektion des SQL-Statements um das Ausführungsdatum (`LAUFD`) ergänzt wird. Für das Identifikationsmerkmal (`LAUFI`) wird analog verfahren.

```
1  if(runDate){
2      insertRegup += SQL SELECT @runDate AS LAUFD;
3  } else {
4      insertRegup += SQL SELECT '' AS LAUFD;
5  }
```

Code-Beispiel 10: Ergänzung der Projektion um das Ausführungsdatum

Anschließend werden dem SQL-Statement das Buchungs- (`BUDAT`) und Erfassungsdatum (`BLDAT`) optional als Filter hinzugefügt (Code-Beispiel 11).

```
1  if(postingDate){
2      insertRegup += SQL WHERE AND BSIK.BUDAT = @postingDate;
3  }
4  if(docsEnteredDate){
5      insertRegup += SQL WHERE AND BSIK.BLDAT <= @docsEnteredDate;
6  }
```

Code-Beispiel 11: Einfügen zusätzlicher Filter

Für die Eingabe einer bzw. mehrerer Kundennummern (KUNNR) wird das bereits bekannte Code-Beispiel 5 aus Kapitel 4.2 genutzt und kann in leicht abgewandelter Form auch für Lieferantennummern (LIFNR) verwendet werden (Code-Beispiel 12).

```
1  if(vendor && !vendorTo ){
2      insertRegup += SQL WHERE AND BSIK.LIFNR = @vendor;
3  }
4  if(vendor && vendorTo ){
5      insertRegup += SQL WHERE AND BSIK.LIFNR BETWEEN @vendor
6        AND @vendorTo;
7  }
```

Code-Beispiel 12: Unterscheidung zwischen Einzel- und Bereichsfilter

Das komplexeste Konstrukt der Eingabemaske ist die Tabelle für Kriterien zur Suche anhand von Buchungskreisen (BUKRS), Zahlmethoden (ZLSCH) und dem nächsten Ausführungsdatums (NEDAT) des Zahllaufs. Die einzelnen Zellen einer Zeile bilden eine Konjunktion, wobei die Reihen untereinander jeweils disjunkt sind. Die Komplexität entsteht durch die vielen Variationsmöglichkeiten der Eingabe. So können Buchungskreise z.B. kommasepariert, mittels Klammern als Bereich oder beides kombiniert angegeben werden. Die Zahlmethoden werden mit Großbuchstaben abgekürzt und aneinander gereiht um die Angabe mehrerer Methoden zuzulassen.

Die Verarbeitung der Nutzereingaben aus der Tabelle erfolgt dazu zeilenweise (siehe Code-Beispiel 15). Im ersten Schritt wird die Angabe der Buchungskreise geparst. Zusammen mit den Zahlmethoden und nächsten Ausführungsdatum bildet sie einen Filter. Die gebildeten Filter der einzelnen Zeilen werden als Disjunktion dem SQL-Statement angefügt wird. Mit der anschließenden Ausführung des Statements ist der 2. Schritt des Algorithmus abgeschlossen.

Die nachfolgende Berechnung von Zahldatum und Skontobetrag erfolgt mithilfe einer SQLScript-Funktion. Dies geschieht im selbe Zuge mit der Ermittlung und Speicherung der Beleginformationen in der REGUH-Relation. Im letzten Schritt wird aus den Ergebnissen der REGUH-Relation die JSON-Antwort für den Anwender gebildet. Sie umfasst das Ausführungsdatum und Identifikationsmerkmal, sowie eine Liste mit Lieferantennummern und Gesamtbeträgen.

Die vielen optionalen Eingaben durch den Anwender bestimmen das Aussehen, die Komplexität und die Laufzeit des untersuchten SQL-Statements. In diesen Fällen ist es für Entwickler deshalb wichtig, relevante Testwerte zu haben, um früh in der Entwicklung Performance-Analysen zu ermöglichen. Für das vorgestellte SQL-Statement werden dazu im folgenden Kapitel die Ansätze dieser Arbeit zur Vorschlagsgenerierung evaluiert.

7.2. Evaluierung der vorgestellten Ansätze am Fallbeispiel

Die vorgestellten Ansätze werden in diesem Kapitel anhand des Fallbeispiels auf ihre Ausführungsdauer und vorgeschlagenen Testwerten verglichen. Als Grundlage dienen folgende mögliche Nutzereingaben zur Filterung der Ergebnisse des Zahllaufprogramms:

- Buchungsdatum (`BUDAT`)
- Belegdatum (`BLDAT`)
- Lieferantennummer (`LIFNR`)
- Zahlmethode (`ZLSCH`)
- Buchungskreis (`BUKRS`)

		Häufigste Werte					
Spalte	Distinkte Werte	Wert	Vorkommen	Wert	Vorkommen	Wert	Vorkommen
BUDAT	903	**20110228**	760	20110221	139	20110224	103
BLDAT	933	**20110216**	177	20101208	124	20101204	118
LIFNR	247	**1051**	3717	2671	1029	1082993	234
BUKRS	9	**3451**	4426	8211	816	4101	259
ZLSCH	6		5739	T	181	R	10

Tabelle 3: Ermittelte häufigste Werte der betrachteten Spalten

Tabelle 3 zeigt die im ersten Schritt ermittelten drei häufigsten Werte für die betrachteten Spalten (entsprechend dem Verfahren in Kapitel 5.2). Naheliegend ist die Auswahl der Werte mit dem jeweils höchsten Vorkommen (in der Tabelle hervorgehoben). Dies ist in diesem Fall nicht zielführend, da die so gewählte Konstellation von Werten keine Einträge in der Relation vorweist.

Mithilfe der adaptiven Erweiterung des Ansatzes (siehe Kapitel 5.4) wird deshalb die Kombination der Spalten betrachtet. Tabelle 4 zeigt die vier Eingabetupel aus den so erstellten initialen Testdatensets.

BUDAT	BLDAT	LIFNR	ZLSCH	BUKRS	Ergebnisse	Zeit (ms)
'20110228'	'20101208'	'0001082993'	' '	'8211'	231	51,973
'20110228'	'20101204'	'0001082993'	' '	'8211'	117	50,685
'20110221'	'20110216'	'0001063780'	' '	'3451'	59	49,280
'20110224'	'20110216'	'0001063780'	' '	'3451'	51	47,110

Tabelle 4: Gefundene Tupel für Testdatensets inklusive Ausführungszeiten

Die Anzahl der gefunden Tupel kann anhand der Verteilung der Daten innerhalb der Relation erklärt werden. Abbildung 10 zeigt das Verhältnis der Anzahl der distinkten Tupel zur ihrer Anzahl an Vorkommen in der Relation.

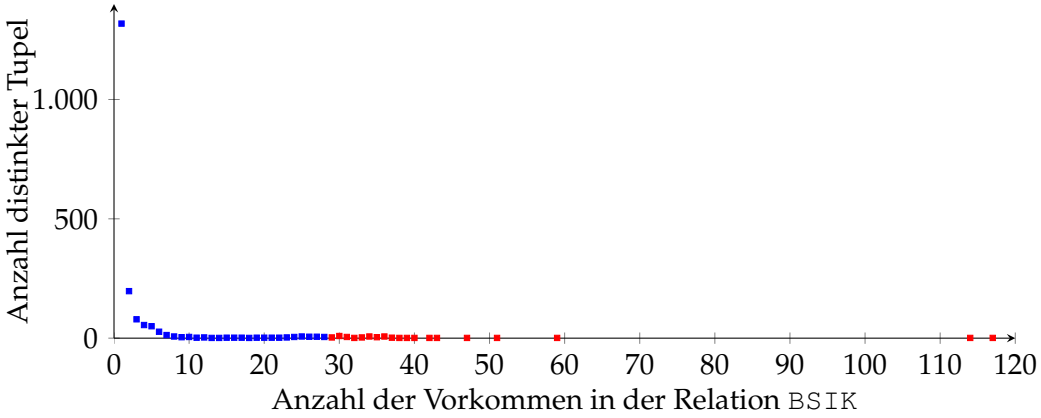

Abbildung 10: Verhältnis der distinkten Tupel zur Anzahl ihres Vorkommens

Besonders auffällig ist die starke Streuung. So gibt es beispielsweise 1318 Tupel, die jeweils nur in einem Datensatz der Relation vorkommen, wohingegen zwei Tupel über 100 Repräsentanten haben.

Rot hervorgehoben ist der Anteil, der vom adaptiven Ansatz als initiale Eingabe genutzt wird. Dies entspricht der absteigenden Sortierung anhand der Vorkommen sowie der festgelegten Obergrenze von 50 Tupeln. Zusammen mit dem blauen Anteil ergibt dies in der Summe 1860 Tupel, was dem kompletten Ein-

gaberaum entspricht. Nur vier Tupel der betrachteten roten Menge ergeben bei Ausführung des Zahllaufprogramms ein Resultat mit mindestens einem Datensatz. Dies sind die zuvor erwähnten Eingabetupel in Tabelle 4.

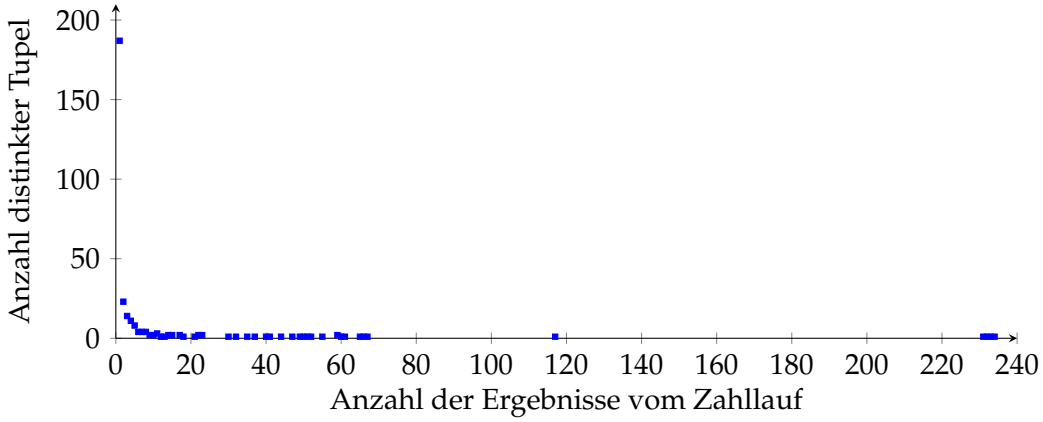

Abbildung 11: Verteilung der 301 möglichen Eingabetupel für den Zahllauf

Wird der komplette Eingaberaum für die Ausführung des Zahllaufprogramms genutzt, so haben nur 301 der 1860 Tupel ein Ergebnis mit mindestens einem Eintrag. Dies entspricht einem Anteil von 16 %. Ihre Verteilung (siehe Abbildung 11) verhält sich analog zu der Beobachtung über den gesamten, potentiellen Eingaberaum (Abbildung 10). Nur fünf der Tupel haben über 100 Datensätze in ihren Resultaten, wohingegen 187 Tupel jeweils nur einen Datensatz umfassen. Tabelle 5 zeigt exemplarisch diese fünf Tupel mit den größten Ergebnismengen, wovon zwei den ermittelten Eingabetupeln in Tabelle 4 entsprechen.

Tupel	BUDAT	BLDAT	LIFNR	ZLSCH	BUKRS	Ergebnisse	Zeit (ms)
1	’20110228’	’20101227’	’0001082993’	’ ’	’8211’	234	52,317
2	’20110228’	’20101220’	’0001082993’	’ ’	’8211’	233	51,689
3	’20110228’	’20101213’	’0001082993’	’ ’	’8211’	232	50,240
4	**’20110228’**	**’20101208’**	**’0001082993’**	**’ ’**	**’8211’**	**231**	**49,917**
5	**’20110228’**	**’20101204’**	**’0001082993’**	**’ ’**	**’8211’**	**117**	**49,280**

Tabelle 5: Eingabetupel mit den meisten Ergebnissen

Im Weiteren wird deshalb der Einfluss der gewählten Obergrenze auf die Anzahl der ermittelten Eingabetupel mit mindestens einem Ergebnisdatensatz untersucht. Dazu wird die Grenze stufenweise erhöht und mit der benötigten Laufzeit sowie der Menge an ermittelten Tupeln verglichen.

Die Messergebnisse (siehe Tabelle 6) zeigen, dass mit steigender Betrachtung von Tupeln auf der einen Seite die Anzahl der gefundenen Eingabetupel für Testdatensets steigt, auf der anderen Seite jedoch auch die Laufzeit der Analysen wächst. Aufgrund dessen ist es in Erwägung zu ziehen, dem Entwickler die Möglichkeit der manuellen Festlegung der Obergrenze zu geben. Er könnte so das Verhältnis zwischen dem Maximum an initial ermittelten Testdatensets und der für ihn vertretbaren Laufzeit selbst festlegen. Die optimale Parameterwahl bedarf dabei weiteren Untersuchungen der Beziehung zwischen der Anzahl der betrachteten Spalten, der Verteilung der Daten und der Laufzeit der Berechnungen.

Aus Gründen der Einfachheit und Geschwindigkeit des Feedbacks für den Entwickler wird die adaptive Lösung mit fester Obergrenze für den Einsatz in der Web-IDE favorisiert. Sie ermöglicht es, auf Grundlage der explorativen Erweiterung die Menge der initial ermittelten Testdatensets kontinuierlich zu erweitern und stellt somit ein Kompromiss aus Laufzeit und Menge an Testdaten dar.

Obergrenze	Anzahl gefundener Tupel	Laufzeit (sek)
50	4	0,06
100	5	0,12
500	50	0,59
1000	81	1,14
1500	236	1,77
2000	301	2,25

Tabelle 6: Auswirkung der Wahl der Obergrenze auf Eingabetupel und Laufzeit

8. Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurden Konzept und Implementierung der Vorschlagsgenerierung relevanter Testwerte für Performance-Analysen vorgestellt und evaluiert. Die verschiedenen Ansätze nutzen dabei neben den Datencharakteristiken innerhalb der Datenbank auch die Kontrollfluss- und Kontextinformationen der Anwendung. Eine adaptive Erweiterung ermöglicht das kontinuierliche Ergänzen der Testdaten durch den Entwickler. Zusammen mit der Integration mehrerer Testsysteme wurde so die Grundlage geschaffen für das performancebewusste Testen von Geschäftsanwendungen.

Die entwickelte Palette an Werkzeugen ist wichtiger Bestandteil der vom Bachelorprojekt erschaffenen Web-IDE, denn sie dient als Grundlage sowohl für Vorhersagen von Laufzeit und Ergebnisgrößen als auch für die Visualisierung von SQL-Statements.

Nachfolgend werden weitere Ideen vorgestellt, die die Ansätze in Zukunft ergänzen können.

8.1. Vorschläge auf Basis von Query-Plan-Analysen

Um den Einfluss bestimmter Parameter auf Abfrageausführungspläne zu ermitteln, können die Bordmittel des Datenbanksystems als Ergänzung genutzt werden. Die Analyse von SQL-Statements durch den SQL-Befehl `EXPLAIN PLAN` liefert eine Kostenaufschlüsselung der einzelnen SQL-Operatoren vor der eigentlichen Ausführung. Die Zuordnung der Kosten zu den Parametern mit den zuvor ermittelten Testwerten würde eine Auskunft über deren Gewichtung geben. Für eine Kostenanalyse inklusive Ausführung kann die SAP HANA interne Prozedur `PLANVIZ_ACTION` genutzt werden. Die Betrachtung einer solchen Analyse ist nicht Teil dieser Arbeit, kann aber in einer späteren Erweiterung die Präzision der Vorschläge von Testwerten erhöhen.

8.2. Einbeziehung von Vorwissen über das genutzte System

Soll ein bestimmtes System genutzt werden, z.B. SAP ERP, so kann Vorwissen über dessen Charakteristiken in den Vorschlägen zu Testdaten berücksichtigt werden. Ein Beispiel dafür sind Standardwerte, die in jeder Instanz des Systems verwendet werden (z.B. feste Benutzerkennungen), oder die Betrachtung besonderer Zeiträume, wie das Jahresende. Somit können Informationen aus dem Kontext des Systems die Vorschläge erweitern, um typische Szenarien abzudecken.

Schon jetzt unterstützen die Analysen von Datenbankinteraktionen mittels relevanter Testwerte die Entwickler von Geschäftsanwendungen. Sie helfen bereits während der Entwicklungsphase, Engpässe aufzudecken und zu beheben, um so kostenintensives Nachbessern zu vermeiden.

Literatur

[AOH03] Paul Ammann, A. Jefferson Offutt, and Hong Huang.
Coverage criteria for logical expressions.
In *ISSRE*, pages 99–107, 2003.

[CDF+00] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber.
A framework for testing database applications.
In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 147–157, New York, NY, USA, 2000. ACM.

[CDF+04] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker.
An agenda for testing relational database applications: Research articles.
Softw. Test. Verif. Reliab., 14(1):17–44, March 2004.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler.
Exe: Automatically generating inputs of death.
In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[Cha04] D. Chays.
Test Data Generation for Relational Database Applications.
PhD thesis, Brooklyn, NY, USA, 2004.
AAI3115007.

[CSF08] David Chays, John Shahid, and Phyllis G. Frankl.
Query-based test generation for database applications.
In *Proceedings of the 1st International Workshop on Testing Database Systems*, DBTest '08, pages 6:1–6:6, New York, NY, USA, 2008. ACM.

[DFC05] Yuetang Deng, Phyllis Frankl, and David Chays.
Testing database transactions with agenda.
In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 78–87, New York, NY, USA, 2005. ACM.

[DT13] Lukas Diekmann and Laurence Tratt.
Parsing composed grammars with language boxes.
In *Workshop on Scalable Language Specifications*, 2013.

[Exn14] Moritz Exner.
Abschätzung von Query Ergebnissen und Laufzeiten während der Implementierungsphase mit Hilfe von Sampling.
Bachelorarbeit, Hasso-Plattner-Institut, 2014.

[Fra14] Clemens Frahnow.
Entwickeln von Performance-Bewusstsein — Ein Benutzeroberflächen-Konzept für sofortiges Performance-Feedback.
Bachelorarbeit, Hasso-Plattner-Institut, 2014.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen.
Dart: Directed automated random testing.
In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[Hor14] Friedrich Horschig.
Direkte Integration von SQL in JavaScript.
Bachelorarbeit, Hasso-Plattner-Institut, 2014.

[HSN97] Banchong Harangsri, John Shepherd, and Anne H. H. Ngu.
Query size estimation using machine learning.
In *DASFAA*, pages 97–106, 1997.

[Kin76] James C. King.
Symbolic execution and program testing.
Commun. ACM, 19(7):385–394, 1976.

[KLPU04] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting.
Boundary coverage criteria for test generation from formal models.
In *ISSRE*, pages 139–150, 2004.

[Mue14] Malte Mues.
Vorhersage von SQL-Query-Charakteristika unter Verwendung von Machine Learning.
Bachelorarbeit, Hasso-Plattner-Institut, 2014.

[Nah04] Fiona Fui-Hoon Nah.
A study on tolerable waiting time: how long are web users willing to wait?
Behaviour & IT, 23(3):153–163, 2004.

[Pla13] Hasso Plattner.
A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases.
Springer Publishing Company, Incorporated, 2013.

[PWX11a] Kai Pan, Xintao Wu, and Tao Xie.
Database state generation via dynamic symbolic execution for coverage criteria.
In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM.

[PWX11b] Kai Pan, Xintao Wu, and Tao Xie.
Generating program inputs for database application testing.
In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 73–82, Washington, DC, USA, 2011. IEEE Computer Society.

[Sch14] Jasper Schulz.
Parsing and Merging Partial, Dynamic HANA SQL Queries.
Bachelorarbeit, Hasso-Plattner-Institut, 2014.

[TDH08] Nikolai Tillmann and Jonathan De Halleux.
Pex: White box test generation for .net.
In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[TZX10] Kunal Taneja, Yi Zhang, and Tao Xie.
Moda: Automated test generation for database applications via mock objects.
In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 289–292, New York, NY, USA, 2010. ACM.

A. BSEG- und BSIK-Erläuterung

BSEG- bzw. BSIK-Spalten	
MANDT	Mandant
GJAHR	Geschäftsjahr
ZLSCH	Zahlweg
BUKRS	Buchungskreis
AUGDT	Datum des Ausgleichs
LIFNR	Kontonummer des Lieferanten bzw. Kreditors
SKFBT	Skontofähiger Betrag in Belegwährung
WRBTR	Betrag in Belegwährung
KUNNR	Debitorennummer
BELNR	Belegnummer eines Buchhaltungsbeleges
MANST	Mahnstufe

Tabelle 7: Erklärung einer Auswahl an Spalten der BSEG- bzw. BSIK-Relation

B. SQL-Statements zur Bestimmung von Testwerten

```sql
-- Häufigste Werte
SELECT <column>, COUNT(<column>) AS OCCURENCES
FROM <schema>.<table>
GROUP BY <column>
ORDER BY OCCURENCES DESC, <column> ASC
LIMIT 3;

-- Seltenste Werte
SELECT <column>, COUNT(<column>) AS OCCURENCES
FROM <schema>.<table>
GROUP BY <column>
ORDER BY OCCURENCES ASC, <column> ASC
LIMIT 3;

-- Bestimmung der Anzahl an distinkten Werten
SELECT COUNT(DISTINCT <column>) AS OCCURENCES
FROM <schema>.<table>;

-- OCCURENCES_HALF = OCCURENCES / 2

-- Werte um dem Median
SELECT <column>, COUNT(<column>) AS OCCURENCES
FROM <schema>.<table>
GROUP BY <column>
ORDER BY OCCURENCES ASC, <column> ASC
LIMIT 3 OFFSET OCCURENCES_HALF;
```

Code-Beispiel 13: Bestimmung vorzuschlagender Testwerte anhand von Datencharakteristiken

C. Ausschnitte vom Algorithmus des Zahllaufs

```
var json = JSON.parse($.request.body.asString());
var runDate = json.runDate;
var identification = json.identification;
var postingDate = json.postingDate;
var docsEnteredDate = json.docsEnteredDate;
var paymentControl = json.paymentControl;
var vendor = json.vendor;
var vendorTo = json.vendorTo;
var customer = json.customer;
var customerTo = json.customerTo;
```

Code-Beispiel 14: Ermittlung der Nutzereingaben aus der JSON-Anfrage

```
var paymentControlFilter = SQL ;
for(var i = 0; i < paymentControl.length; i++){
    var line = paymentControl[i];
    var companyCodeFilter = SQL ;
    if (line.companyCodes){
        var companyCodes = parseCompanyCodes(line.companyCodes);
        if (companyCodes.singles)
            companyCodeFilter += SQL OR BSIK.BUKRS IN @companyCodes.singles;
        if (companyCodes.ranges){
            for (var j = 0; j < companyCodes.ranges.length; j++){
                companyCodeFilter += SQL OR BSIK.BUKRS BETWEEN
                  @companyCodes.ranges[j][0] AND @companyCodes.ranges[j][1];
            }
        }
    }
    var paymentMethodsFilter = SQL ;
    if (line.paymentMethods){
        paymentMethodsFilter += SQL
          AND BSIK.ZLSCH IN @line.paymentMethods;
    }
    if (line.nextPaymentDate){
        paymentMethodsFilter += SQL
          AND ADD_DAYS(ZFBDT, ZBD1T) < @line.nextPaymentDate;
    }
    paymentControlFilter += SQL
      OR (@paymentMethodsFilter AND @companyCodeFilter);
}
insertRegup += SQL WHERE AND @paymentControlFilter;
```

Code-Beispiel 15: Verarbeitung der Filter in Tabellenform

```
var insertRegup = SQL[CUSTOMER]
    INSERT INTO REGUP (MANDT, ZBUKR, LIFNR, BUKRS, BELNR, GJAHR, BUZEI,
    ZLSCH, BSCHL, HKONT, SAKNR, SHKZG, DMBTR, WRBTR, MWSKZ, ZFBDT,
    ZTERM, ZBD1T, ZBD2T, ZBD3T, ZBD1P, ZBD2P, SKFBT, SKNTO, WSKTO,
    DMBE2, PSWSL, PSWBT, BVTYP, SGTXT, SLAND1, POKEN, EMPFG, HBKID,
    BUDAT, BLDAT, VBLNR, KUNNR, LAUFD, LAUFI, XVORL, ZBFIX, WAERS)

    SELECT BSIK.MANDT, BSIK.BUKRS AS ZBUKR, BSIK.LIFNR AS LIFNR, BSIK.BUKRS,
    BELNR, GJAHR, BUZEI, ZLSCH, BSCHL, HKONT, SAKNR, SHKZG, DMBTR, WRBTR,
    MWSKZ, ZFBDT, BSIK.ZTERM, ZBD1T, ZBD2T, ZBD3T, ZBD1P, ZBD2P, SKFBT, SKNTO,
    WSKTO, DMBE2, PSWSL, PSWBT, BVTYP, SGTXT, LAND1 AS SLAND1,
    CASE
      WHEN (LFA1.CONFS = '' OR LFA1.CONFS IS NULL)
          AND (LFA1.LOEVM = '' OR LFA1.LOEVM IS NULL)
          AND (LFB1.CONFS = '' OR LFB1.CONFS IS NULL)
          AND (LFB1.LOEVM = '' OR LFB1.LOEVM IS NULL)
      THEN ''
      ELSE 'X'
    END as POKEN,
    CASE
      WHEN EMPFB <> '' THEN EMPFB
      WHEN LNRZB <> '' THEN LNRZB
      WHEN LNRZA <> '' THEN LNRZA
      ELSE ''
    END AS EMPFG,
    CASE
      WHEN BSIK.HBKID <> '' THEN BSIK.HBKID
      WHEN LFB1.HBKID <> '' THEN LFB1.HBKID
      ELSE ''
    END AS HBKID,
    CASE BUDAT
      WHEN '00000000' THEN NULL
      ELSE TO_DATE(BUDAT, 'YYYYMMDD')
    END AS BUDAT,
    CASE BLDAT
      WHEN '00000000' THEN NULL
      ELSE TO_DATE(BLDAT, 'YYYYMMDD')
    END AS BLDAT,
    BSIK.BELNR AS VBLNR, KUNNR
    FROM BSIK
    LEFT OUTER JOIN LFA1 ON BSIK.MANDT = LFA1.MANDT AND BSIK.LIFNR = LFA1.LIFNR
    LEFT OUTER JOIN LFB1 ON BSIK.MANDT = LFB1.MANDT AND BSIK.LIFNR = LFB1.LIFNR
      AND BSIK.BUKRS = LFB1.BUKRS
    WHERE LFB1.ZAHLS = '' AND (LFA1.SPERR = '' OR LFA1.SPERR IS NULL)
      AND (LFB1.SPERR = '' OR LFB1.SPERR IS NULL);
```

Code-Beispiel 16: Allgemeiner Teil des SQL-Statements für Belegpositionen