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 frühzeitig 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 database interactions regarding their performance already 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

TODO:

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.

Mein weiterer Dank gilt meiner Familie. Ihre stetige Unterstützung und Motivation haben mich während des Studiums begleitet, selbst wenn der “Große” nicht allzu oft die Heimreise antrat.

Schlussendlich bedanke ich mich bei x,x,Franziska Eller und meiner Mutter für das Korrekturlesen und die Tipps zur Fertigstellung dieser Arbeit.

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, 13. Juli 2014

_____

(Frank Blechschmidt)

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungverzeichnis

AST	Abstract Syntax Tree
ER	Entity-Relationship
ERIC	Enterprise-Ready IDE Concepts
ERP	Enterprise Resource Planning
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
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, S. 212], 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 im 2. Kapitel zusammengetragen werden und bei der Entwicklung von Geschäftsanwendungen assistieren. Anschließend wird die Einbettung von SQL in die Programmiersprache der Anwendungen untersucht, um auf deren Basis die im Kapitel 4 vorgestellten Algorithmen zur Vorschlagsgenerierung von Testdaten darzulegen. Im Kapitel 5 wird ergänzend die administrative Architektur für Testdaten und -systeme betrachtet. Abschließend werden die vorgestellten Ansätze im Rahmen der Umsetzung einer praxisnahen Geschäftsanwendung evaluiert.

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

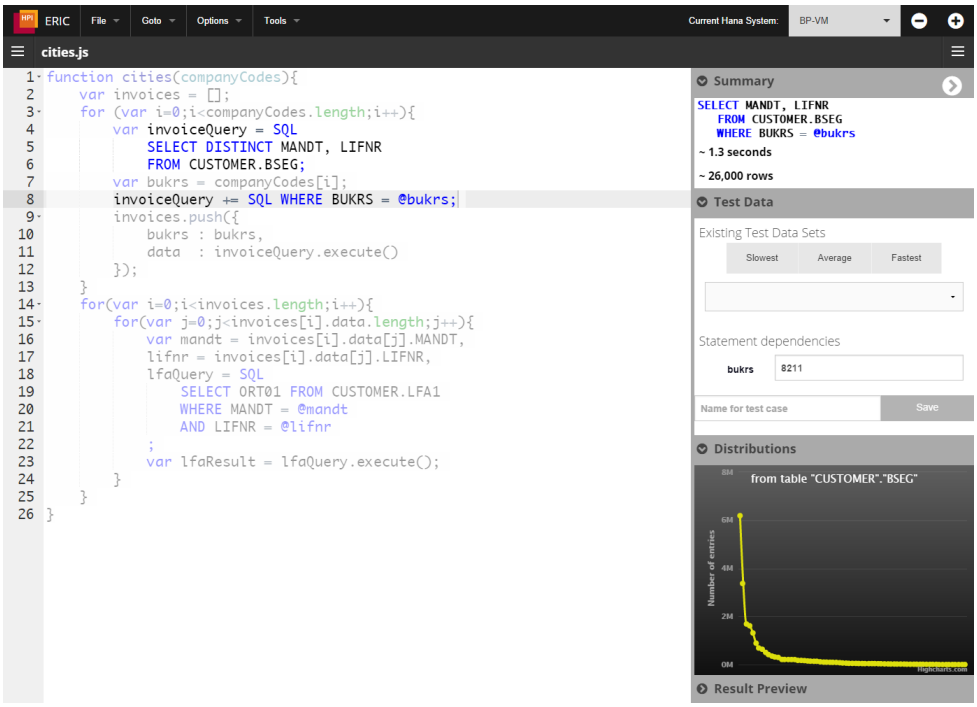

Abbildung 1: Screenshot der Web-IDE »ERIC«

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 3). 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: auf Basis von 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.

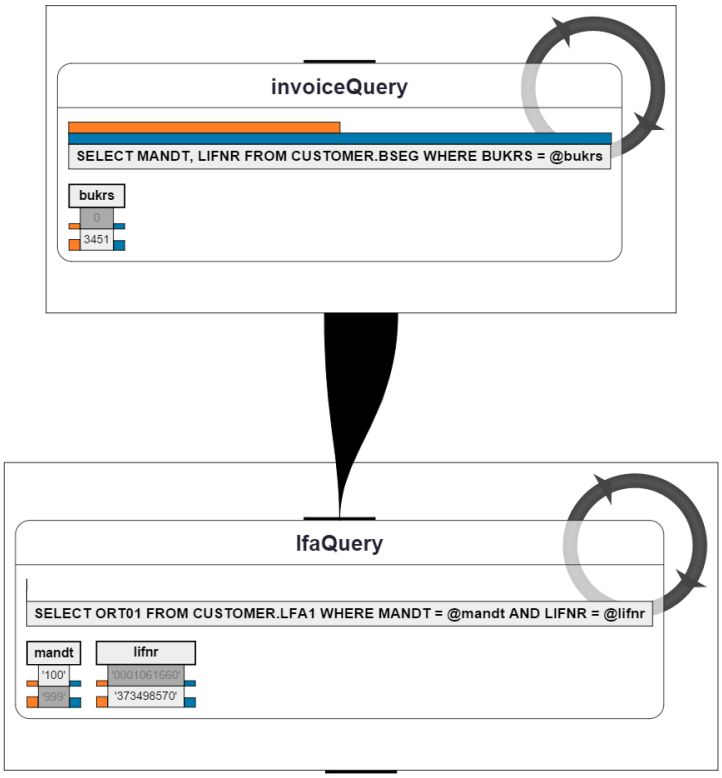

Abbildung 2: Visualisierung des Kontrollflusses

Zusätzlich ist es möglich den Verlauf des Kontrollflusses in 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   }
8   if(customer){
9     filter.push("BSEG.KUNNR = '" + customer + "'");
10  }
11  if(filter.length > 0){
12  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.

3. Verknüpfung von SQL mit dem Anwendungskontext

Mit der Einbettung von SQL-Anfragen in andere Programmiersprachen, zum Beispiel in JavaScript[1], 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

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

SQL-Statement ein, zum Beispiel als Parameter für Filterbedingungen. Ebenso 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 untersucht.

3.1. 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);
```

```
7     stmt.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 Anfrageparameter 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.

3.2. 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 [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 woraufhin 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 zu 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.

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

Neben dem direkten Einfließen von Variablen in ein SQL-Statement (vgl. Kapitel 3.2), 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 auf dem resultierenden SQL-Statement 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 im Kapitel 4 verschiedene Lösungsstrategien und Ansätze erörtert, die den Entwickler unterstützen, repräsentative Testwerte durch passende Vorschläge zu finden um aussagekräftige Analysen auf SQL-Statements zu ermöglichen.

4. Generierung von Vorschlägen für Testdaten

Um dem Entwickler repräsentative Testwerte, die die Variationspunkte eines SQL-Statements beeinflussen, vorzuschlagen, gibt es verschiedene Strategien. 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 im Kapitel 5 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.

4.1. Bestimmung der zu betrachtenden Variablen

Kapitel 3.3 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 Variablen, 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 Testdatenvorschläge schaffen. In Abbildung 3 ist eine solche Datenstruktur beispielhaft für Code-Beispiel 7 dargestellt.

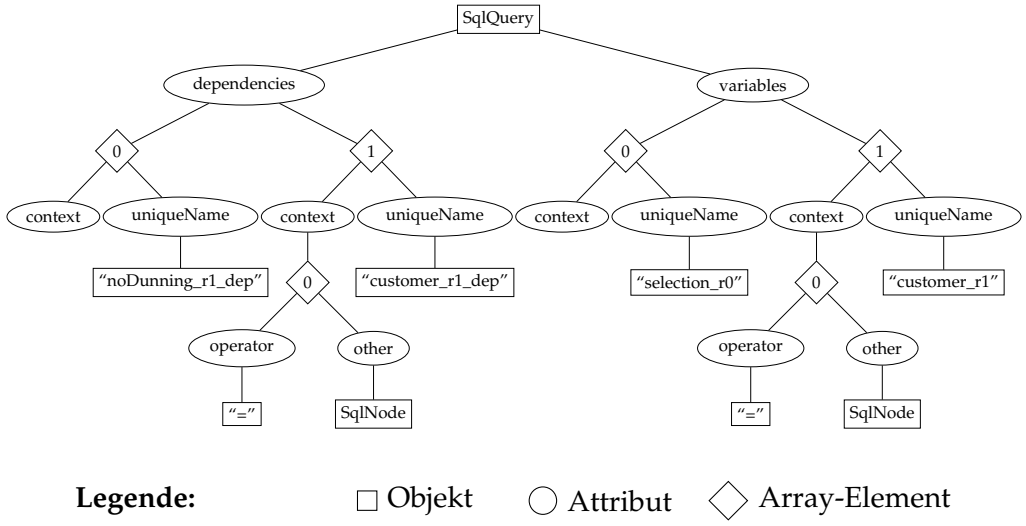

Abbildung 3: Datenstruktur zum Code-Beispiel 7

4.2. 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 spielt dabei primär die Verteilung der Daten innerhalb der Relationen eine Rolle, so-

wie die Anzahl ihrer unterschiedlichen Werteausprägungen. Im Gegensatz dazu werden für die Projektion, Sortierung und Gruppierung Metainformationen zu den Relationen der Datenbank berücksichtigt. Zweiteres wird im Kapitel 4.3 näher betrachtet.

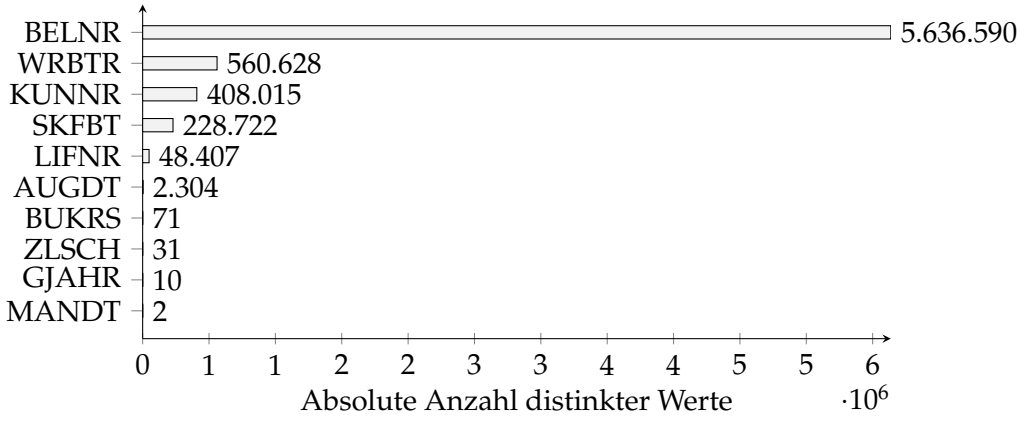

Abbildung 4: Verteilung distinkter Werten einer Auswahl von Spalten aus BSEG

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 der Bedeutung der einzelnen Spalten befindet sich im Anhang (Tabelle 6).

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 der Anzahl des Vorkommens eines Wertes vorgenommen und die ersten drei selektiert. Sollten mehrere Werte dieselbe Anzahl an Vorkommen vorweisen, werden sie zusätzlich anhand ihrer Werte aufsteigend sortiert. Im Unterschied dazu wird bei der Klasse der seltensten Werten initial ein 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.

4.3. 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 noch 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 eingegeben Werten. Mittels des SQL-Operators `LIKE` kann dazu nach Daten gesucht werden, die dem Muster der bisherigen Eingabe entsprechen. Diese Zusatzinformationen erhöhen zusammenfassend also die Komfortabilität der Eingabe und können fehlerhafte Eingaben reduzieren.

4.4. Adaptive Vorschlagsgenerierung durch Laufzeitanalysen

Der in Kapitel 4.2 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 3 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 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, z.B. zusätzliche Berechnungen 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 1 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: 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 im Kapitel 4.2 ermittelten Werte können al-

le 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 mit einer derzeitigen Obergrenze von 50 die distinkten Tupel der Kombinationen der betrachteten Spalten mit dem meisten Vorkommen innerhalb der Relation bestimmt (siehe Code-Beispiel 9). 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

Anschließend werden die gefundenen Tupel zur Analyse ihrer Laufzeit der Datenbankabfrage genutzt. Sie bilden somit 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.

4.5. Vorschläge für Variablenwerte ohne Datenbankbezug

Nicht immer müssen Variablen in ein SQL-Statement einfließen um darauf Auswirkungen zu haben. 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.

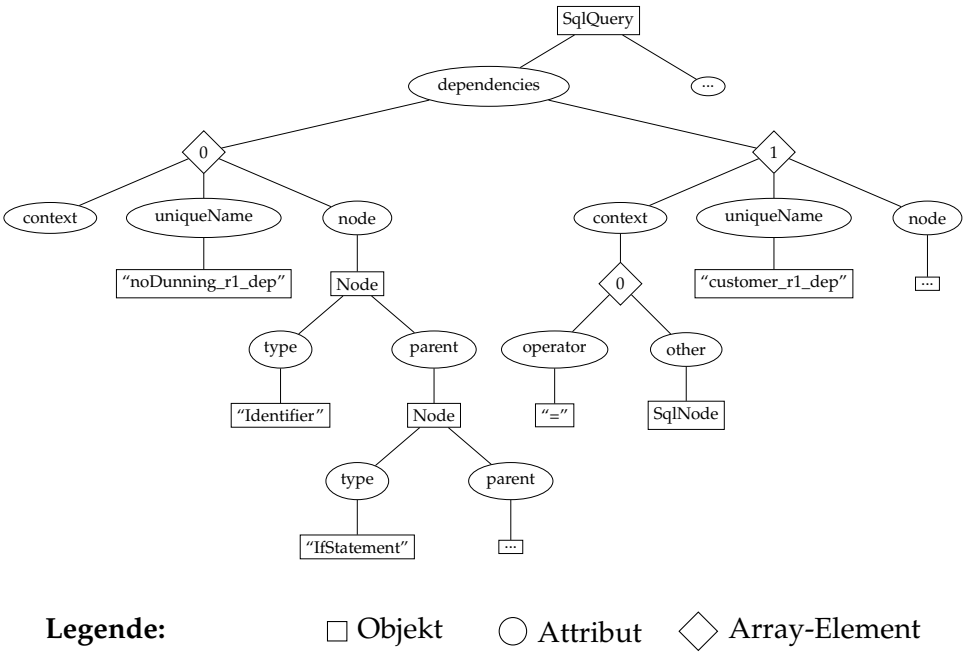

Abbildung 5: Erweiterung des Baumes aus Abbildung 3 für Variablen ohne Datenbankbezug

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.

Elternknotentyp	Operator	Erfüllender Wert	Nicherfüllender 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`

Der Ansatz ist ähnlich dem Verfahren in Testing-Frameworks um eine erhöhte Verzweigungsüberdeckung zu erreichen und unterstützt derzeit die drei primitiven Datentypen von JavaScript (Wahrheitswerte, Zahlen, Zeichenketten).

4.6. Integration in die Web-IDE

Die in Kapitel 4.2 und 4.4 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 2) und schließlich in der Sidebar angezeigt.

Aus der in Kapitel 4.4 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

5. 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 zugrundeliegende Schema, die Integration verschiedener Testsysteme und das Pflegen der Testdaten, insbesondere in Hinblick auf deren Gültigkeit.

5.1. 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 sog. Testvariablen betrachtet. Eine Menge von Testvariablen bildet zusammen mit dem Pfad der geöffneten Quellcodeda-

tei 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 Performanceanalysen erfolgten, um die Abhängigkeit der Systemauswahl auf die Laufzeiten der Testdatensets zu berücksichtigen. In Abbildung 7 ist das dazugehörige Datenschema dargestellt.

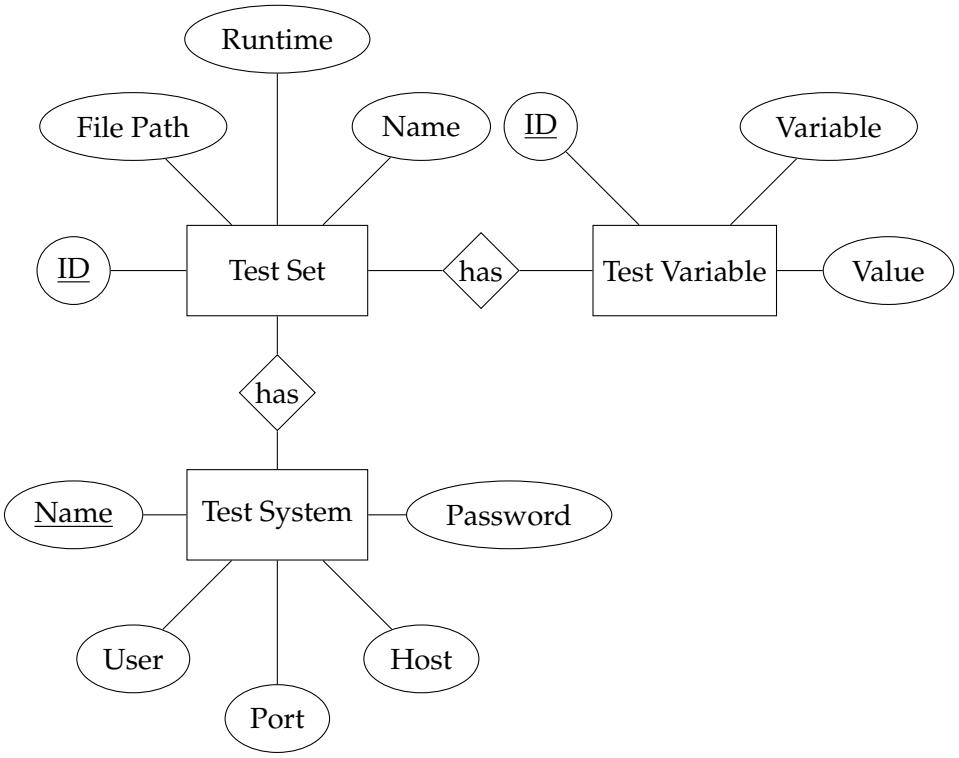

Abbildung 7: ER-Diagramm für Testdatensets und -systeme

5.2. 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 4 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 4 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 Schwellwert für Datenveränderungen festgelegt wird (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.

5.3. Integration mehrerer Testsysteme

Typischerweise dient nicht nur ein System als Grundlage für Performance-Analysen einer Geschäftsanwendung, sondern ein 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.

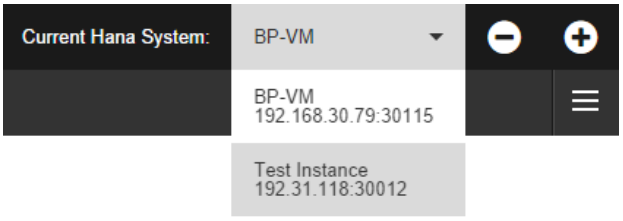

Abbildung 8: Auswahlmenü für verschiedene Datenbankserver

6. 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 und anschließend die Bearbeitung durch das System veranlasst. Dabei spielen die Abhängigkeiten von Zahlungen, Lieferanten und Rabattverträgen eine wichtige Rolle, da sie im Idealfall in einer günstigen 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 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 durch die in dieser Arbeit vorgestellten Ansätze zur Generierung von Testdatenvorschlägen.

In diesem Kapitel wird exemplarisch die Implementierung des Zahllauf-Algorithmus mithilfe der Web-IDE vorgestellt. Dabei steht die von Nutzereingaben geprägte Verarbeitung der Suchanfrage im Fokus. Auf dieser Grundlage wird anschließend eine Evaluierung der vorgestellten Ansätze zum Vorgeschlagen von Testwerten durchgeführt.

6.1. Implementierung des Algorithmus in der Web-IDE

Die in Abbildung 9 dargestellte Filtereingabemaske des Zahllaufs besteht zum einen aus allgemeinen Parametern für die Suche und zum anderen aus Eingabefeldern für Suchkriterien einzelner Rechnungen. Für die Implementierung des Frontends wurde das UI-Framework OpenUI5[2] von SAP genutzt, der Fokus die-

[2] `http://sap.github.io/openui5/`

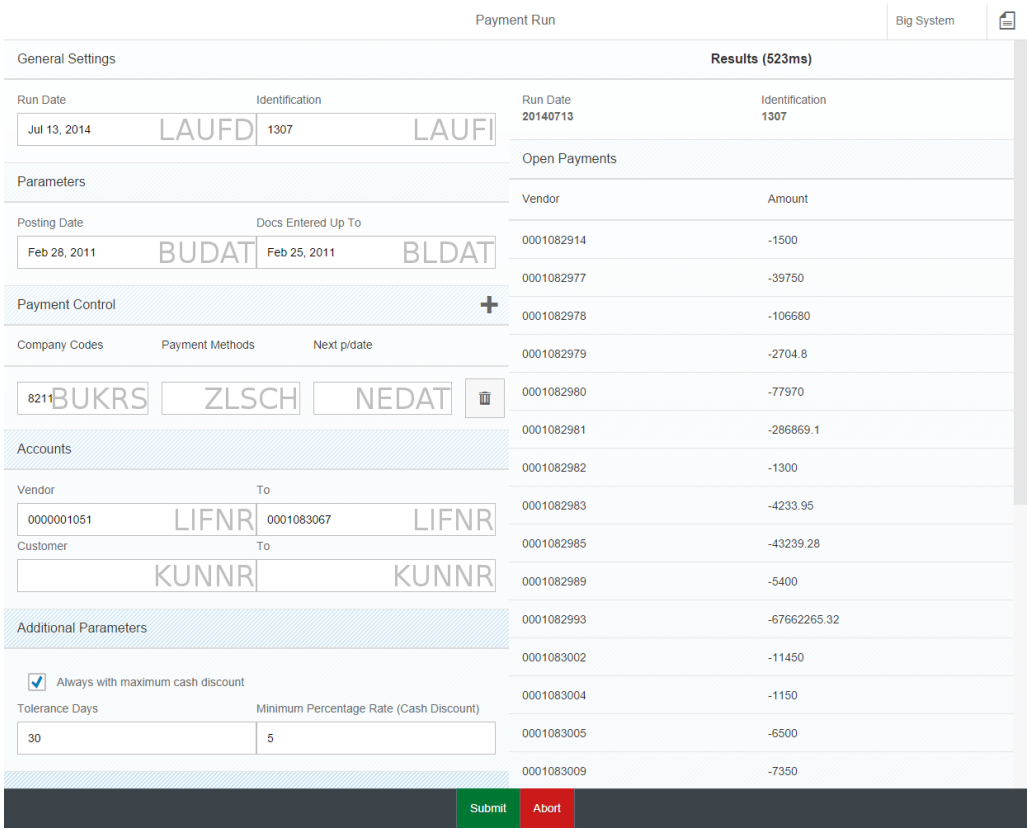

Abbildung 9: Eingabemaske des Zahllaufs

ses Kapitels liegt indessen auf der Backendimplementierung mittels der SAP XS-Engine[3].

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

[3] http://help.sap.com/hana/SAP_HANA_XS_JavaScript_Reference_en/index.html

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

Die Eingaben des Nutzers in der Filtermaske (aus dem 1. Schritt, siehe Code-Beispiel 14 im Anhang) wirken sich insbesondere auf die Selektion der Belegpositionen aus. Dem allgemeinen Teil des dazugehörigen SQL-Statements (siehe Code-Beispiel 16 im Anhang) werden durch die Eingaben weitere Filterkriterien hinzugefügt, die nachfolgend näher untersucht werden.

Das Code-Beispiel 10 zeigt exemplarisch, wie die Projektion des Ausführungsdatums (zu sehen in Abbildung 9 unter "General Settings") dem SQL-Statement ergänzt werden. Für das Identifikationsmerkmal (`LAUFI`) gilt dies analog.

```
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 optional die Filter für Buchungs- und Erfassungsdatum (unter "Parameters" in Abbildung 9) 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 (in Abbildung 9 unter "Accounts") wird das bereits bekannte Code-Beispiel 5 aus Kapitel 3.2 genutzt und kann in leicht abgewandelter Form auch für Lieferantennummern verwendet werden.

```
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, Zahlmethoden und dem nächsten Ausführungsdatums des Zahllaufs. Die einzelnen Zellen einer Reihe bilden eine Konjunktion, wobei die Reihen miteinander disjunkt sind. Die Komplexität entsteht durch die vielen Möglichkeiten 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 mehrere Methoden zuzulassen.

Die Verarbeitung der Nutzereingaben aus der Tabelle erfolgt zeilenweise. Für jede Zeile wird ein leeres SQL-Statement erstellt. Dieses wird im ersten Schritt mit den, mittels der Funktion `parseCompanyCodes()` ermittelten, Buchungskreisen gefüllt. Sollten Zahlmethoden und ein nächstes Ausführungsdatum angegeben sein, werden diese dem SQL-Statement ergänzt. Anschließend wird es einem allgemeinen, anfangs leeren SQL-Statement hinzugefügt. Sobald alle Zeilen der Tabelle abgearbeitet sind, wird der zusammengestellte Filter dem SQL-Statement `insertRegup` angefügt. Der Algorithmus zur Verarbeitung dieser Eingaben befindet sich im Anhang (Code-Beispiel 15).

Die Berechnung vom Zahldatum und Skontobetrag erfolgt mithilfe einer SQLScript-Funktion im selben Schritt mit der Ermittlung und Speicherung der Beleginformationen in der REGUH-Relation. Im letzten Schritt dienen der Ergebnisse in der REGUH-Relation für die JSON-Antwort an den Anwender. Sie enthält eine Liste mit Gesamtbetrag, Lieferantennummer, Ausführungsdatum Identifikationsmerkmal für jede erfasste Rechnung der REGUH-Relation.

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.

6.2. Evaluierung der vorgestellten Ansätze am Fallbeispiel

Zur Evaluation der vorgestellten Ansätze wurde ein vergleichendes Experiment auf Basis des Fallbeispiels durchgeführt. Als Grundlage dienten folgende mögliche Nutzereingaben zur Filterung der Ergebnisse des Zahllaufprogramms:

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

Die ermittelten Vorschläge nach dem im Kapitel 4.2 spezifizierten Verfahren sind in den Tabellen 7, 8 und 9 dargestellt (siehe Anhang). Naheliegend ist die Auswahl der häufigsten Werte der jeweiligen Spalten für eine Analyse der Datenbankanfrage, welche jedoch eine leere Ergebnismenge liefert. Deshalb wird in der adaptiven Erweiterung des Ansatzes (siehe Kapitel 4.4) die Erstellung initialer Kombinationen der ermittelten Testwerte als Testdatensets vorgenommmen. Tabelle 3 zeigt die zwei in diesem Schritt gefundenen Tupel, die beim Ausführen der Datenbankanfrage mindestens eine Ergebniszeile zurückgeben, mitsamt ihren Ausführungszeiten.

BUDAT	BLDAT	LIFNR	ZLSCH	BUKRS	Ergebnisse	Zeit (ms)
'20110228'	'20101208'	'0001082993'	' '	'8211'	231	49,917
'20110228'	'20101204'	'0001082993'	' '	'8211'	117	49,280

Tabelle 3: Gefundene Tupel für Testdatensets

Vergleicht man die ermittelten Testdatensets mit den initial vorgeschlagenen häufigsten Werten der einzelnen Spalten, fällt auf, dass die Werte des Buchungsdatums, der Lieferantennummer, der Zahlmethode und des Buchungskreises konstant bleiben und nur das Belegdatum in seinem Wert variiert. Des Weiteren ist zu beobachten, dass alle Werte aus den ermittelten Listen der häufigsten Werte stammen. Dies kann mit der Verteilung der Daten begründet werden, dargestellt in den Abbildungen 11 bis 15 (siehe Anhang). Die Grafiken zeigen das Verhältnis der Anzahl distinkter Werten innerhalb einer Spalte zu der Anzahl der Ausprägungen in den Einträgen der Relation. Beispielsweise haben 630 distinkte Werte in der Spalte `BUDAT` nur jeweils einen Eintrag, wohingegen ein einziger Wert 760 Repräsentationen in der Relation hat.

Dies zeigt sich besonders in den Abbildungen 11 (`BUDAT`), 12 (`BLDAT`) und 13 (`LIFNR`). Es wird deutlich, dass die Mehrheit der gefunden distinkten Werte in den Spalten meist nur in einem Eintrag der Relation genutzt werden. Dem gegenüber stehen wenige Ausnahmen, die eine hohe Anzahl von Repräsenationen innerhalb der Relation umfassen. Dies zeichnet sich auch in den ermittelten, vorgeschlagenen Werten um den Median ab (Tabelle 9), bei der die Werte für die Spalten `BUDAT`, `BLDAT` und `LIFNR` auch dort nur einen Eintrag vorweisen.

Ebenso zeigen die Verteilung der Daten in den Spalten `BUKRS` und `ZLSCH` (Abbildung 14 und 15) eine deutlich Spaltung. So ist zu erkennen, dass jeweils ein Wert im Großteil der Einträge in der Relation genutzt wird und somit die anderen distinkten Werte nur wenige Ausprägungen haben. Aufgrund der wenigen distinkten Werte ergibt dies eine Gerade in der Darstellung.

Ein Vergleich mit allen möglichen Wertkombinationen würde die Berechnung einer Permutation über alle Listen der distinkten Werte der fünf betrachteten Spalten erfordern. Aufgrund der Größe der einzelnen Listen (`BUDAT` 903, `BLDAT` 933, `LIFNR` 247, `ZLSCH` 6, `BUKRS` 9) ergäbe sich in der Gesamtsumme ein Liste mit über 11,2 Milliarden Einträgen. Des Weiteren hat nur ein Bruchteil dieser Kombinationen mindestens einen Repräsentanten in der Datenbank. In der Annahme, dass die Berechnung der Analyse pro Listeneintrag 10ms braucht, entspräche dies über 3 Jahre Berechnungen.

Aus diesem Grund fokussiert sich das Experiment auf den Vergleich der Ergebnisse folgender Ansätze:

1. Adaptiver Algorithmus mit fester Obergrenze (50)
2. Adaptiver Algorithmus ohne feste Obergrenze
3. Permutation der ermittelten häufigste, seltensten und mittleren Werte der Spalten

Dazu wird ermittelt, welcher der Ansätze die meisten der 5 Tupel mit den größten Ergebnismengen zurückgibt (zu sehen in Tabelle 4). Dies wird zusätzlich in Beziehung zu der Anzahl der betrachteten Tupel und die Laufzeit der Ermittlung betrachtet.

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 4: Eingabetupel für mit den meisten Ergebnissen

Die Ergebnisse (Tabelle 5) zeigen, dass die Einschränkung durch die Obergrenze den Umfang der gefunden Tupel reduziert. Die Betrachtung der Permutation ergibt zwar ein Tupel mehr, benötigt aber über eine halbe Minute für die Bestimmung, da mehr als 30.000 Tupel getestet werden.

	Tupel 1	Tupel 2	Tupel 3	Tupel 4	Tupel 5	Getestete Tupel	Laufzeit (sek)
mit Obergrenze				X	X	50	0,058
ohne Obergrenze	X	X	X	X	X	1.860	2,239
Permutation	X			X	X	30.618	34,352

Tabelle 5: Vergleich der evaluierten Ansätze

Der Grund für vollständige Abdeckung der Tupel der Variante ohne Obergrenze liegt in der Ermittlung der zu testenden Tupel. Die 1.860 bestimmten Tupel entsprechen allen distinkten Konstellationen der Werte der betrachteten Spalten.

Davon haben insgesamt nur 301 mindestens eine Ergebniszeile beim Ausführen des Programms, was einer Abdeckung des kompletten Eingaberaums entspricht. Die Festlegung einer Obergrenze entspricht also einer Einschränkung auf einen Subraum. Die Verteilung der 301 Tupel (siehe Abbildung 10) macht deutlich, dass der Großteil der Eingabekonstellationen nur eine Ergebniszeile aus der Datenbank zurückgeben (187 Tupel) und nur wenige Tupel eine größere Ergebnismenge abrufen (5 Tupel mit über 150 Ergebnissen, erfasst in Tabelle 4). Dies spiegelt die zuvor betrachtete Verteilung der Daten innerhalb der einzelnen Spalten wieder.

Aufgrund dessen ist es in Erwägung zu ziehen dem Entwickler die Möglichkeit der Festlegung einer Obergrenze zu geben. Dies bedarf weiteren Untersuchungen in Hinblick auf die Beziehung zwischen der Anzahl der betrachteten Spalten, dem Umfang an distinkten Werten und der Laufzeit der Berechnungen.

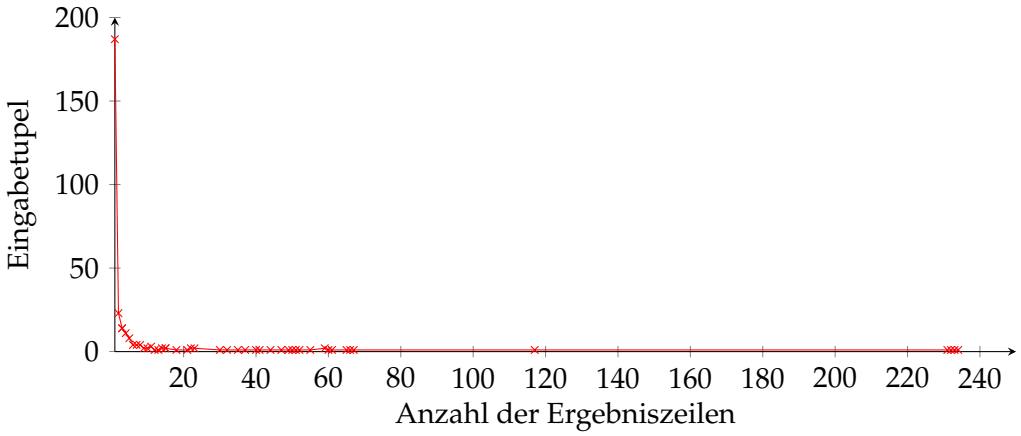

Abbildung 10: Verteilung der Anzahl von Ergebniszeilen der Eingabetupel

Die derzeitige adaptive Lösung ermöglicht es auf Grundlage der mit Obergrenze ermittelten initialen Testdatensets durch exploratives Erweiterung durch den Entwickler den ermittelten Datensatz zu ergänzen.

7. 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[4] auch die Möglichkeit sie direkt mit einzubeziehen. Bei diesem Ansatz werden die Eingabewerte mit dem Zustand der Datenbank in Verbindung gebracht. Dabei gibt es verschiedene Vorgehensweisen.

7.1. Eingabewerte zum Testen von Datenbankanwendungen

Mit dem Erzeugen relevanter Eingabewerte für funktionale Anwendungstests haben 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 Tools für das funktionale Testen. Es nutzt Metainformationen aus der Datenbank in Kombination mit Voreinstellungen vom Entwickler um eine Testdatenbank synthetisch zu erzeugen, um dadurch die Testabdeckung zu erhöhen.

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 zusätzlich noch im Zusammenhang mit Bedingungen innerhalb der Anwendung betrachtet, wodurch gegebenen-

[4] http://www.mockobjects.com/

falls weitere Eingabewerte erzeugt werden.

TODO: Weitere Paper betrachten!

7.2. Liveanalyse von Datenbankanwendungen

Eine alternative Möglichkeit die Performance eine Datenbankanwendung zu ermitteln ist das Monitoring. Softwarelösungen wie New Relic[5] 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 können häufig genutzte Werte aus dem Betrieb für Vorschläge von Testdaten zur Weiterentwicklung der Anwendung genutzt.

8. Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurden Konzept und Implementierung der Vorschlagsgenerierung relevanter Testwerte für Performanceanalysen vorgestellt. Die entwickelten Ansätze sind Bestandteil der Werkzeugpalette der vom Bachelorprojekt erschaffenen Web-IDE und dienen als Grundlage sowohl für Vorhersagen von Laufzeit und Ergebnisgrößen als auch für die Visualisierung von SQL-Statements.

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

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

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 des SQL-Statements durch den SQL-Befehl `EXPLAIN PLAN` liefert dafür 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

Sollte ein bestimmtes System genutzt werden, z.B. SAP ERP-Software, 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 Werkzeuge den Entwicklern von Geschäftsanwendungen während der Entwicklungsphase Engpässe aufzudecken und zu beheben um so das kostenintensive 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.
Hier kommt der Titel.
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.
Hier kommt der Titel.
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.
Hier kommt der Titel.
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.

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

B. SQL-Statements zur Bestimmung von Testwerten

```
1  -- Häufigste Werte
2  SELECT <column>, COUNT(<column>) AS OCCURENCES
3  FROM <schema>.<table>
4  GROUP BY <column>
5  ORDER BY OCCURENCES DESC, <column> ASC
6  LIMIT 3;
7
8  -- Seltenste Werte
9  SELECT <column>, COUNT(<column>) AS OCCURENCES
10 FROM <schema>.<table>
11 GROUP BY <column>
12 ORDER BY OCCURENCES ASC, <column> ASC
13 LIMIT 3;
14
15 -- Bestimmung der Anzahl an distinkten Werten
16 SELECT COUNT(DISTINCT <column>) AS OCCURENCES
17 FROM <schema>.<table>;
18
19 -- OCCURENCES_HALF = OCCURENCES / 2
20
21 -- Werte um dem Median
22 SELECT <column>, COUNT(<column>) AS OCCURENCES
23 FROM <schema>.<table>
24 GROUP BY <column>
25 ORDER BY OCCURENCES ASC, <column> ASC
26 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

D. Tabellen zur Evaluation

		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 7: Ermittelte, häufigste Werte für die Eingabedaten

		Seltenste Werte					
Spalte	Distinkte Werte	Wert	Vorkommen	Wert	Vorkommen	Wert	Vorkommen
BUDAT	903	20031231	1	20050131	1	20050224	1
BLDAT	933	20031231	1	20041029	1	20050105	1
LIFNR	247	3021	1	3101	1	3491	1
BUKRS	9	3451	4426	8211	816	4101	259
ZLSCH	6	A	1	L	2	U	3

Tabelle 8: Ermittelte, seltenste Werte für die Eingabedaten

		Werte um den Median					
Spalte	Distinkte Werte	Wert	Vorkommen	Wert	Vorkommen	Wert	Vorkommen
BUDAT	903	20091127	1	20091202	1	20091203	1
BLDAT	933	20091216	1	20091218	1	20091221	1
LIFNR	247	7015822	1	7018820	1	7019738	1
BUKRS	9	3601	157	4101	259	8211	816
ZLSCH	6	R	10	T	181		5739

Tabelle 9: Ermittelte Werte um den Median für die Eingabedaten

E. Verteilungen der betrachteten Spalten der Evaluation

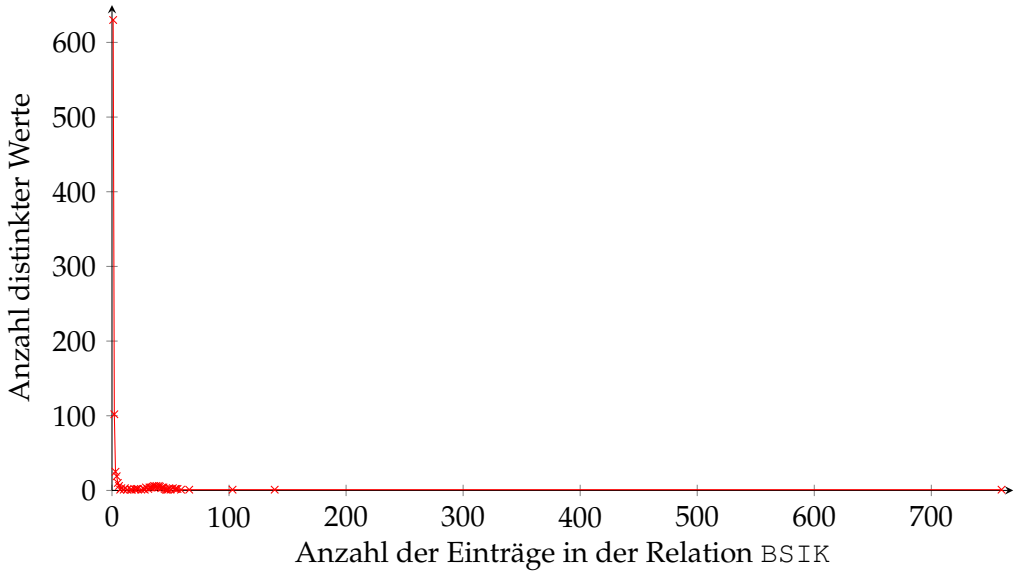

Abbildung 11: Verteilung der distinkten Werte in der Spalte BUDAT

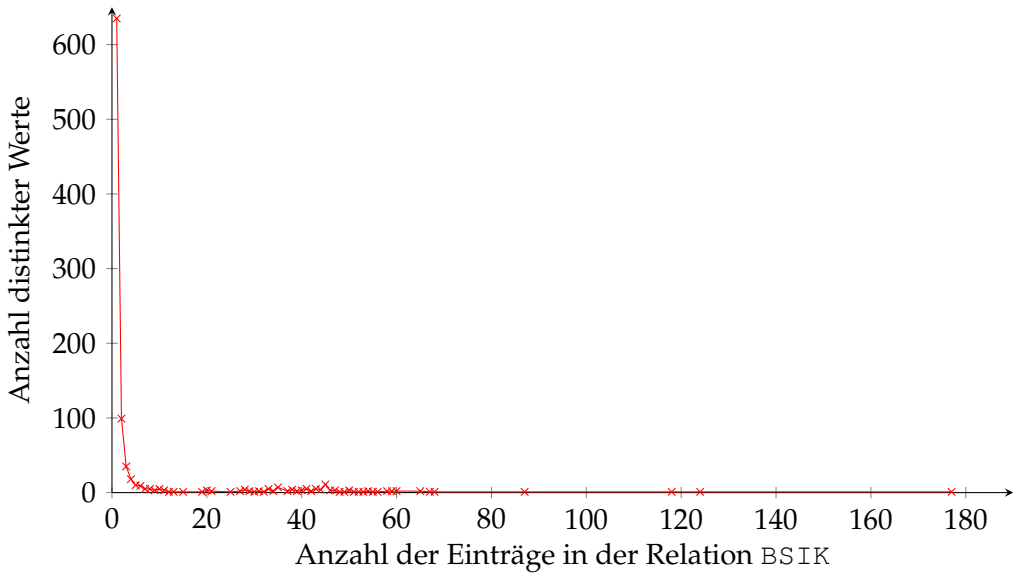

Abbildung 12: Verteilung der distinkten Werte in der Spalte BLDAT

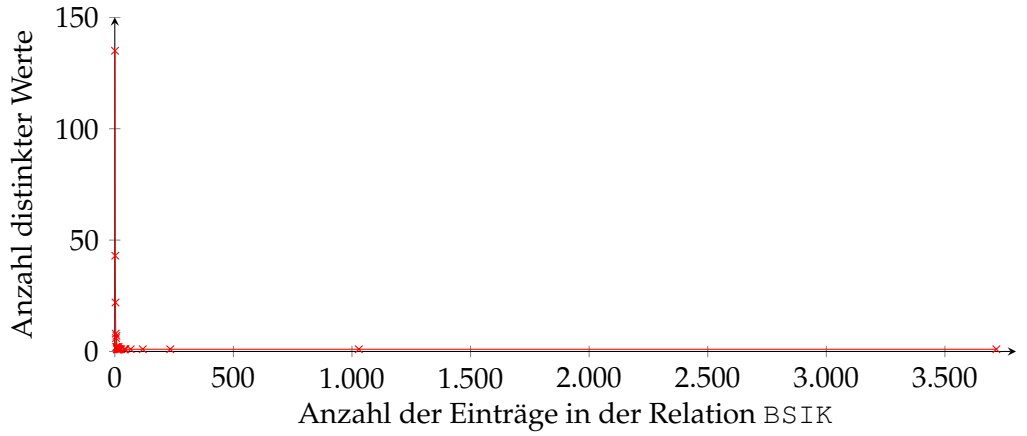

Abbildung 13: Verteilung der distinkten Werte in der Spalte LIFNR

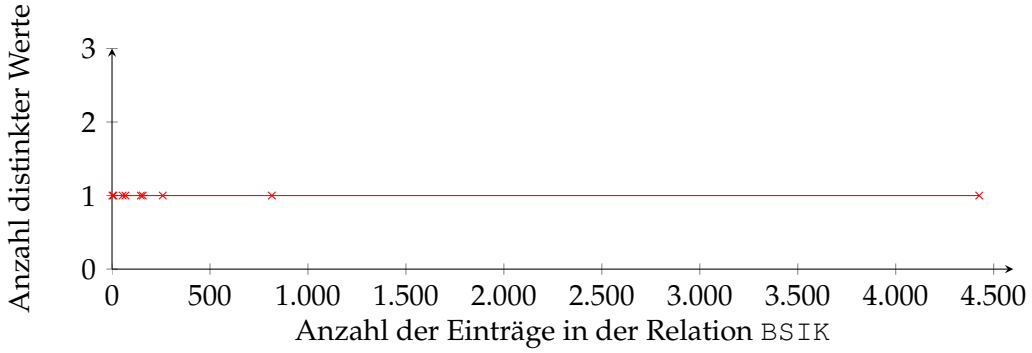

Abbildung 14: Verteilung der distinkten Werte in der Spalte BUKRS

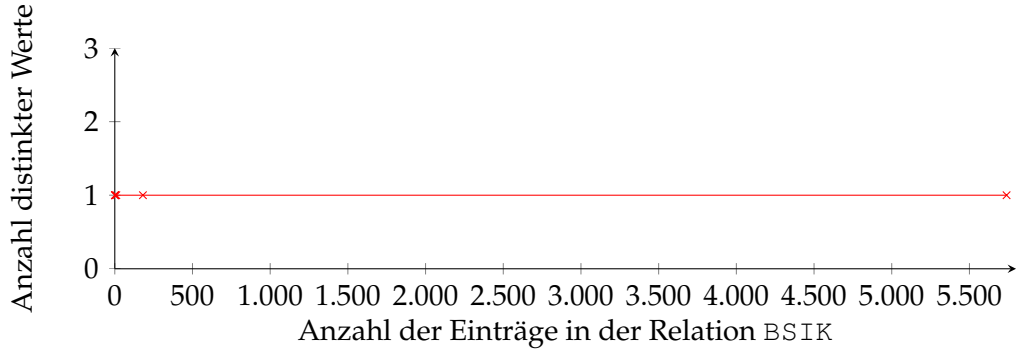

Abbildung 15: Verteilung der distinkten Werte in der Spalte ZLSCH