accenture

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Duale Hochschule Baden-Württemberg Mannheim

Projektarbeitarbeit

Vorbereitungen für automatisierte Softwaretests für Berechnungen von KPIs.

Studiengang Wirtschaftsinformatik

Studienrichtung Data Science

Verfasser(in):	Silvan Biewald
Matrikelnummer:	1942347
Firma:	Accenture
Abteilung:	Technology Solutions
Kurs:	WWI18DSB
Studiengangsleiter:	Prof. Dr. Bernhard Drabant
Wissenschaftliche(r) Betreuer(in):	Prof. Dr. Bernhard Drabant
Firmenbetreuer(in):	Janina Rustaee
Bearbeitungszeitraum:	24.07.2020 – 16.11.2020

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema: *Vorbereitungen für automatisierte Softwaretests für Berechnungen von KPIs.* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Silvan Biewald

Abstract

Ziel dieser Arbeit ist es für einen konkreten Anwendungsfall zwei verschiedene Automatisierungsmöglichkeiten eines Datenbankentests zu vergleichen, um einem Projektteam eine dieser Möglichkeiten zu empfehlen. Dabei handelt es sich bei der einen Variante um ein komplexes, bereits vorhandenes Test-Framework, das nicht genau auf diesen Anwendungsfall zugeschnitten ist, und bei der anderen um ein simples, vom Projektteam für diesen konkreten Anwendungsfall speziell entwickeltes. Da keine Möglichkeit eines Testlaufs der beiden bestanden, werden Vor- und Nachteile der zwei Varianten aufgezeigt und in Hinblick auf die Bedürfnisse des Anwendungsfalls gegenübergestellt.

Die komplexere Variante übersteigt die simple nur im Punkt Zuverlässigkeit. Der Vergleich zeigt, dass für diesen Anwendungsfall die simplere Version dem Projekt durch das Zugeschnittensein auf denselben große Vorteile bietet und eine bessere Übersichtlichkeit über die Ergebnisse des Tests gewährt. Außerdem legt er nahe, dass die Zeitersparnis bei dieser Variante größer ist. Aus diesem Grund wird dem Projektteam empfohlen die einfachere Variante zu wählen.

Inhaltsverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

DBMS	Database Management System
SQL	Structured Query Language
KPI	Key Performance Indicator
FDD	Framework für Deequ basierte Datenbankentests
DBFALT	Datenbanken Falsch Alt
DBFNEU	Datenbank Falsch Neu
DBR	Datenbanken Richtig
JSON	Java Script Object Notation
SRS	Software Requirements Specification
TRS	Test Requirements Specification
UI	User Interface
IDE	Integrated Development Environment
RC	Remote Control
HTML	Hypertext Markup Language
API	Application Programming Interface
RDD	Resilient Distributed Data Set
Mllib	Machine Learning library
AWS	Amazon Web Services
S3	Simple Storage Service
ETL	Extract, Transform, Load
SES	Simple Email Service
SDK	Software Development Kit

1 Einleitung

Das Testen ist ein wichtiger Teil eines jeden Softwareprojekts. Manuelles Testen kann dabei einen großen Zeit- und Arbeitsaufwand für ein Projekt bedeuten. Unter anderem aus diesem Grund werden Softwaretests automatisiert. Ziel dieser Arbeit ist es für den Testprozess eines Projektes zur Erstellung einer Anwendung, die Key Performance Indicator (KPI) berechnen und nutzerfreundlich anzeigen soll, Automatisierungsmöglichkeiten zu erkunden. Zu diesem Zweck werden für einen bestimmten Datenbankentest zwei Möglichkeiten zu dessen Automatisierung verglichen, ohne dass diese im vorliegenden Anwendungsfall eingesetzt werden müssen. Daraus soll eine Empfehlung resultieren, die das Projektteam bei der Entscheidung unterstützt, welche der beiden Varianten implementiert werden soll.

Zunächst werden detailliertere Informationen zu dem vorliegenden Projekt, sowie den in dieser Arbeit zu betrachtenden Tests gegeben. Darauf folgen grundlegende Themen zum Verständnis von Softwaretests, wie zum Beispiel der Testprozess oder einige Testarten. Im Anschluss werden Vorteile und Grenzen des automatisierten Testens beschrieben und zwei verschiedene Tools, die zur Automatisierung von Tests verwendet werden können, vorgestellt. Ein kurzer Überblick über einige während der Arbeit am vorliegenden Projekt verwendete Software schließen die theoretischen Grundlagen ab. Anschließend werden detaillierte Informationen zum vorliegenden Projekt und dessen Testvorgängen näher beschrieben. Diese sind aufgeteilt in das Testen der Datenbank und das Testen der Anwendung. Für beide Fälle werden die bestehenden manuellen Tests, sowie die Konzepte für deren Automatisierung beleuchtet. Dabei ist eines davon ein bereits existierendes und das zweite ein vom Projektteam für diesen Anwendungsfall erstelltes System. Die Vor- und Nachteile der beiden Automatisierungsmöglichkeiten des Datenbankentests werden einzeln aufgezeigt und später miteinander verglichen. Zum Schluss erfolgt eine Empfehlung an das Projektteam, welche der beiden Varianten Anwendung finden sollte. Es wird erwartet, dass die vom Projektteam erstellte Variante in diesem Fall mehr Vorteile bietet.

2 Grundlegendes zum Projekt

Die in dieser Arbeit zu vergleichenden Automatisierungsvarianten von Softwaretests stehen im Zusammenhang mit einem Projekt des IT-Beratungsunternehmens Accenture, an dem der Autor im Rahmen der Praxisphase seines dualen Studiums teilnahm. Gegenstand des Projekts ist es für einen international agierenden Kunden wichtige KPI's für dessen Unternehmen durch eine Applikation darzustellen. Bei diesen Kennzahlen handelt es sich um wichtige Leistungsindikatoren des Unternehmens, wie zum Beispiel die monatlich verkauften Einheiten eines Produkts, oder Veränderung des Umsatzes zwischen zwei Perioden. Zum Start des Projektes bestanden bereits mehrere dieser Berichterstattungs-Anwendungen auf Seite des Kunden. Ziel des Projektes ist es nun ein System zu erstellen, das alle bestehenden Anwendungen zusammenfasst. Für jede Anwendung existiert dabei eine eigene Datenbasis. Unter Datenbasis wird im Folgenden die Menge an Daten verstanden, die benötigt wird, um alle in den Anforderungen definierten KPI's einer dieser Anwendungen korrekt berechnen zu können. Eine Datenbasis setzt sich wiederum aus ein oder mehreren Datenbanken zusammen. Die gesamte Datenbasis aller bestehenden Anwendungen kann Fehler enthalten und wird im Folgenden auch als Datenbanken Falsch Alt (DBFALT) bezeichnet. Es müssen die einzelnen Datenbasen zu einer Datenbank zusammengefasst und eine vollkommen neue Anwendung geschrieben werden. Die Anwendung wurde im Zeitraum, in dem diese Arbeit erstellt wurde, fast vollständig fertiggestellt. Die Datenbank dieser Anwendung wird auch als Datenbank Falsch Neu (DBFNEU) bezeichnet.

Die größten Schwierigkeiten ergaben sich dabei durch die Vielzahl an Quellen für die Datenbasis. Diese Quellen waren meist nicht homogen, sondern basierten auf unterschiedlichen Datenmodellen, so bestanden zum Beispiel unterschiedliche Bezeichnungen für ein und dieselbe Information. Deshalb wurde zunächst ein neues Datenmodell erstellt, dass im laufenden Prozess sukzessive implementiert wird. Dazu kam die Fehlerhaftigkeit der DBFALT. So lagen unter anderem fehlende, oder falsche Einträge in den Tabellen der Datenbanken vor. Das war ein Problem für die zu erstellende Applikation, da fehlerhafte Werte in den Quellen zu falschen KPI's in der Berichterstattungs-Anwendung führen. Zudem wurden im Zuge der Implementierung des neuen Datenmodells fast täglich Änderungen an der Datenbasis vorgenommen. Aus diesen Gründen machten sich Testvorgänge notwendig.

3 Grundlagen zu Softwaretests und zur verwendeten Software

Dieses Kapitel enthält alle theoretischen Grundlagen, die für das Verständnis dieser Arbeit notwendig sind.

3.1 Einführung in Softwaretests

Zunächst werden die Grundlagen von Softwaretests in Bezug auf Definition, wichtige Terminologien, Gründe für das Testen, den Testprozess und einigen Testarten beschrieben.

3.1.1 Testen - eine Begriffsdefinition

Als eine allgemeine Definition für Testen bietet Hambling et al. folgende an:

> Testen ist die systematische und methodische Untersuchung eines Arbeitsprodukts unter Verwendung einer Vielzahl von Techniken mit der ausdrücklichen Absicht zu zeigen, dass es seinen gewünschten oder beabsichtigten Zweck nicht erfüllt. Dies geschieht in einer Umgebung, die der Umgebung, die im Echtbetrieb eingesetzt werden soll, möglichst nah kommt. [1, section WHAT TESTING IS AND WHAT TESTING DOES paragraph 2]

Mithilfe von Tests wird also überprüft, ob ein Produkt allen Anforderungen entspricht, die an dieses gestellt werden, indem mögliche Fehler gefunden und aufgezeigt werden. Bei Softwaretest unterscheiden Hambling et al. unter drei verschiedenen Arten von Fehlern. Zum einen dem Error, der einen menschlichen Fehler bei der Erstellung des Codes meint, zum anderen dem Defect, der sich aus einem Error ergibt, wenn dieser zum Beispiel in einem Code niedergeschrieben wird, und dem Failure, der aus einem Defect entsteht und das sichtbare falsche Verhalten eines Systems nach Ausführung des Codes meint. Dabei muss ein Defect nicht notwendigerweise immer einen Failure hervorrufen [1]. Testen ist außerdem immer ein geplanter, nachvollziehbarer und kontrollierbarer Prozess, bei dem

das Produkt unter Bedingungen zu prüfen ist, die denen in der Liveumgebung möglichst nahe kommen.

3.1.2 Klärung wichtiger Begriffe zum Testen

Um das Thema Testen zu behandeln, müssen vorab einige wichtige Begriffe geklärt werden. Bevor es zum Testen kommen kann, ist ein Verständnis für die Anforderungen notwendig, die das zu testende Produkt erfüllen muss. Dieses Wissen wird als Testbasis, oft zum Beispiel in der Software Requirements Specification (SRS) festgehalten. Die Testbasis enthält also Beschreibungen von Reaktionen, die das Produkt in bestimmten Situationen oder Zuständen zeigen soll. Diese Beschreibungen werden als Testbedingungen bezeichnet. Wenn für eine Testbedingung konkrete In- und erwartete Outputs festgelegt werden, spricht man von sogenannten Testfällen. Der Testvorgang dagegen bezeichnet die Durchführung von Testfällen, zu dem auch die Schaffung der Voraussetzungen für jeden einzelnen Testfall gehören. [1] Im Folgenden wird unter Tests eine Menge von Testfällen verstanden.

3.1.3 Gründe für das Testen

Das Testen kann unterschiedliche Gründe haben, wobei das Testen an sich keinen Wert hat oder Vorteil bringt. Erst infolge der Auswertung der Testergebnisse können Maßnahmen eingeleitet werden, die zum Beispiel für ein Unternehmen einen Wert generieren. Das Auftretens von Defects selbst führt unter Umständen zu Failures im System, die bei der Arbeit mit oder an dem Produkt stören können oder diese unmöglich machen. Sowohl im Entwicklungsprozess des Produktes als auch bei seiner Verwendung durch Endnuzter ist es dabei besonders wichtig, Defects möglichst früh zu finden, da dadurch Kosten, die für die Behebung aufgewendet werden müssen, reduziert werden können [2].

Andere Gründe für das Testen sind eher qualitativer Natur. Kunden erwarten, dass ein Produkt auf eine bestimmte Art und Weise funktioniert. Wenn aber ständig Defects auftreten, dann sind die Funktionen des Produktes eingeschränkt und die Erfahrung des Kunden mit dem Produkt verschlechtert sich. Daher ist es für Unternehmen sinnvoll eine Minimierung der Defects anzustreben. In anderen Situationen kann die Erhöhung der Verlässlichkeit einer Software sogar zur Minimierung von Risiken beitragen. [2] Ein Beispiel hierfür ist ein

Atomkraftwerk, da Failures in einem solchen Umfeld das Potenzial haben Menschenleben in großer Zahl in Gefahr zu bringen und deshalb unbedingt vermieden werden müssen.

3.1.4 Der Testprozess

Testen ist ein Prozess. Allerdings handelt es sich um keinen universellen Prozess, der einmal definiert und in jedem Fall genau so ausgeführt werden kann. Vielmehr sollte der Prozess auf den jeweiligen Anwendungsfall angepasst werden. An dieser Stelle wird also nur ein allgemeiner Testprozess vorgestellt. Eine mögliche Beschreibung eines Testprozesses beschreibt Hambling et al. folgendermaßen:
Während der Testplanung wird zunächst umrisshaft entschieden, was getestet werden soll, auf welche Weise gearbeitet wird und wie die Arbeitsverteilung sein soll. Dazu gehört auch die Erstellung der Abschlusskriterien, durch die genau definiert wird, ab wann das Testen als abgeschlossen anzusehen ist. Die Planung wird in Form eines Testplans schriftlich festgehalten.

Hambling et al. beschreiben als nächstes die Testüberwachung und -steuerung, weil sie von diesem Punkt an kontinuierlich bis zum Ende des Testprozesses durchgeführt werden muss. Die Testüberwachung meint vor allem den Vergleich vom Ist-Zustand und dem im Testplan definierten Soll-Zustand. Die Teststeuerung geht mit der Testüberwachung einher, da sie die Einleitung von Maßnahmen zur Einhaltung des Testplans beinhaltet.

In der Testanalyse wird die Testbasis daraufhin analysiert, was getestet werden soll. In diesem Schritt muss auch entschieden werden, was getestet werden kann, um daraus die Testbedingungen zu entwickeln. In diesem Schritt werden auch mögliche Errors in der Testbasis identifiziert, um zu verhindern, dass diese später im Code implementiert zu Defects werden.

Im Testdesign werden aus den gefundenen Bedingungen die Testfälle erstellt. Wichtig ist hierbei, wie schon im vorherigen Schritt bei den Testbedingungen, dass die Testfälle sehr klar und genau definiert werden, damit es bei der späteren Durchführung der Tests nicht zu falschem Testen kommt.

Die Testimplementierung kann bereits mit dem Testdesign einhergehen. Darunter versteht man alle Maßnahmen, die dazu dienen, die spätere Durchführung der einzelnen Testfälle zu ermöglichen. Dieser Schritt hängt maßgeblich von dem vorliegenden Anwendungsfall ab, so muss zum Beispiel für das automatisierte Testen zunächst ein entsprechendes Skript erstellt werden.

Im Schritt der Testausführung werden die einzelnen Testfälle manuell oder automatisiert durchgeführt und die Ergebnisse aufgezeichnet, sodass gefundene Defects später bearbeitet werden können.

Wenn alle Testfälle durchgeführt wurden und alle Abschlusskriterien erfüllt sind, kann der Testprozess abgeschlossen werden. Es wird empfohlen alle für das Testen erstellten Dokumente mit den gewonnenen Erfahrungen für zukünftige Projekte aufzubewahren [1].

3.1.5 Statisches und dynamisches Testen

Da jeder Softwaretest entweder statischer oder dynamischer Natur ist, soll auf diese beiden Konzepte hier näher eingegangen werden. Das Testen von Software kann nicht nur während der Ausführung eines Programms geschehen, sondern auch davor. Dabei nennt sich das Testen davor, also ohne Ausführung eines Codes, statisches Testen. Da diese Art zu Testen keinen fertigen Code benötigt, können solche Tests teilweise bereits zu einem frühen Zeitpunkt im Lebenszyklus eines Produkts durchgeführt werden. Dies kann sehr vorteilhaft sein, weil möglichst frühzeitig gefundene Defects in der Regel weniger aufwendig zu beheben sind und so geringere Kosten verursachen. Unter dynamischen Testen versteht man alle Tests, die während der Ausführung eines Codes erfolgen. Im Gegensatz zum statischen Testen setzt das dynamische Testen die Ausführbarkeit der Software voraus [1].

3.1.6 Einige Testarten

In diesem Kapitel werden einige Möglichkeiten vorgestellt, wie getestet werden kann.

White Box Tests

Beim White Box-Testen wird das zu testende Programm nicht aus der Position des Anwenders betrachtet, sondern aus der des Entwicklers. Der Fokus liegt daher auf dem Testen des Programmcodes, um die Funktionalitäten des Programms sicherzustellen. Da hierbei der Code nicht ausgeführt wird, handelt es sich um statisches Testen.

Es gibt verschiedene Prinzipien, die angewendet werden können, um die Fehlerlosigkeit von Code möglichst gut sicherzustellen [3]. Da im Zuge dieser Arbeit jedoch das Testen von Codes keine wesentliche Rolle spielt, werden die Prinzipien und Konventionen des Testens von Codes an dieser Stelle nicht weiter ausgeführt. Falls trotz alledem Interesse daran besteht sei auf folgende grundlegenden Konzepte verwiesen: "Desk Checking", "Code Walkthrough", "Formal Inspection", die bei Gopalaswamy [3] beschrieben sind.

Black Box Tests

Im Gegensatz zum White Box Testen wird beim Black Box Testen der Code selbst nicht betrachtet. Die Software wird quasi von einer außenstehenden Position, also aus der Blickrichtung des Nutzers, getestet. Dabei werden neben der Funktionalität der Software auch deren Reaktionen auf falsche Eingaben durch den Nutzer überprüft. Diese Vorgehensweise setzt kein Wissen um die genaue innere Funktionsweise der Software voraus. Der Wissensstand des Testers kann daher mit dem eines potenziellen Kunden übereinstimmen. Da die Perspektive eines Nutzers eingenommen wird, beschränken sich Überprüfungen nicht nur auf die Funktionsweise bzw. das Funktionieren der Software, sondern schließen auch deren Reaktion auf falsche Eingaben durch den Nutzer ein [3].

Diese Tests basieren sowohl auf den im Vorfeld definierten Anforderungen an das fertige Produkt, als auch auf impliziten Anforderungen, die von einem Endnutzer erwartet werden könnten, aber in der Testbasis nicht explizit festgestellt wurden. Um sicherzustellen, dass auch alle wichtigen impliziten Anforderungen eingehalten werden, können diese neben den im SRS festgehaltenen expliziten Anforderungen in der Test Requirements Specification (TRS) definiert werden [3].

Regressionstests

In manchen Fällen ist es notwendig dieselben Tests sehr häufig oder regelmäßig durchzuführen. Solche Tests werden als Regressionstests bezeichnet. Vor allem im Softwarebereich ist diese Art von Testen oft aufzufinden. Das liegt daran, dass an Software in vielen Fällen selbst nach Veröffentlichung des eigentlichen Produkts immer noch weiter gearbeitet wird. Jede neue Version der Software, die veröffentlicht wird, muss getestet werden. Gerade bei komplexen Systemen mit vielen Abhängigkeiten kann es vorkommen, dass das Hinzufügen oder Ändern einer Funktion zum einen Defects in dieser Funktion, zum anderen Defects an weiteren Stellen im System verursacht. Diese Problematik besteht bei jeder Änderung des Codes, also zum Beispiel auch bei der Behebung von Defects. Deshalb fokussieren sich Regressionstests darauf, bereits absolvierte Testfälle erneut durchzuführen. Das kann zum Beispiel nach jeder Behebung eines Defects geschehen [3].

3.2 Einführung in die Automatisierung von Softwaretests

Bei der Automatisierung von Softwaretests handelt es sich um die

> Entwicklung von Software zum Testen von Software. [3, section 16.1 paragraph 2]

Dieses Kapitel behandelt die Automatisierung von Softwaretests. Es wird aufgezeigt, warum Tests automatisiert werden, welche Herangehensweisen dabei verwendet werden können und welche Grenzen es gibt.

3.2.1 Gründe für die Automatisierung

Die manuelle Ausführung von Tests hat ihre Vorteile, auf die im nachfolgenden Kapitel noch eingegangen werden, erreicht aber auch schnell Grenzen. So hängt die Geschwindigkeit der Durchführung von Testfällen hier maßgeblich von menschlichen Fähigkeiten ab und erfordert oft eine große Anzahl an Testern. So kann ein bestimmter Testfall zwar wichtig für einen Test sein, jedoch einen so großen Aufwand erfordern, dass er von den Projektmitgliedern nicht geleistet werden kann. Um dem entgegenzuwirken, können Tests mithilfe

von Software automatisiert werden.

Einer der wichtigsten Gründe für Testautomatisierung ist daher die Geschwindigkeitssteigerung. Die meisten Tests, die ein Mensch durchführen kann, können automatisiert viel schneller abgeschlossen werden. Dies bietet zum einen die Möglichkeit mehr Testfälle durchzuführen, zum anderen können auch komplexe oder sehr zeitintensive Testfälle effektiver ausgeführt werden. So können Testfälle, die manuell nicht durchführbar wären, in den Testprozess integriert werden. Für manche Tests genügt eine einmalige Ausführung nicht, zum Beispiel bei Regressionstests. Diese Art von Tests müssen häufig wiederholt werden. Auch das kann in kleineren Projekten für zu viel Aufwand sorgen.
Doch Automatisierung senkt nicht nur die Zeit, die pro Testfall benötigt wird, sondern gibt auch Arbeitskräfte frei, die vorher manuell Tests durchführen mussten. Diese Personen können sich nun zum Beispiel darauf konzentrieren neue Testfälle zu erstellen.

Darüber hinaus werden menschliche Tester unweigerlich selbst Fehler machen, die das Testergebnis verfälschen. Dieses Problem tritt nicht auf, wenn das Testen von Software übernommen wird, sofern diese nicht fehlerhaft ist. Zudem ist keine Grenzen gesetzt, wie lange am Stück Testfälle ausgeführt werden können, da keine Ermüdungserscheinungen auftreten wie bei menschlichen Testern [4].

3.2.2 Grenzen der Automatisierung

Bei Betrachtung der positiven Aspekte von Automatisierung kann man leicht zum Schluss kommen, dass es immer von Vorteil ist Tests zu automatisieren. Das ist allerdings nicht der Fall. Auch der Testautomatisierung sind Grenzen geboten [4]. Eine davon geht bereits aus der im Kapitel 3.2 (Einführung in die Automatisierung von Softwaretests) gegebenen Definition für Softwaretestautomatisierung hervor. Die zu testende Software wird bei einem automatisierten Test mithilfe von Software überprüft. Daraus folgt, dass Tests nur automatisierbar sind, wenn sie durch Software ausgeführt werden können. Software kann in vielen Fällen die Schritte eines Tests nacheinander durchführen, doch die kreativen, intellektuellen und intuitiven Eigenschaften eines menschlichen Testers fehlen hierbei [5].

Es gibt Testfälle, die nicht automatisierbar sind. Gerade wenn das User Interface (UI) zu testen ist, wird die manuelle Variante oft vorgezogen. Beispielhaft kann man sich hier

eine Anwendung vorstellen, die auf verschiedenen Arten von Geräten für einen Nutzer einsetzbar sein soll. Dafür ist es wichtig zu überprüfen, ob das Interface auf allen Geräten gut erkennbar ist. Einem Menschen fällt in so einem Fall schnell auf, ob ein Text zu klein ist, oder ob bestimmte Hintergrundfarben die Lesbarkeit beeinträchtigen. Diese Art von Betrachtung ist für ein Programm nicht möglich [5].

3.2.3 Tools zur Automatisierung

Es existiert eine große Menge an Tools, die genutzt werden kann, um Softwaretests zu automatisieren. Welches davon zu wählen ist, hängt immer von dem vorliegenden Anwendungsfall ab. An dieser Stelle sollen nur die Tools näher beschrieben werden, die in dem dieser Arbeit zugrunde liegenden Projekt auch tatsächlich Anwendung fanden.

Selenium

Selenium ist ein Open Source Tool zur Automatisierung von Web Browsern. Das bedeutet, es bietet die Möglichkeit in einem Browser Maus- und Tastaturinputs zu simulieren. Der Einsatz beschränkt sich somit nicht nur auf das Testen, jedoch ist die Testautomatisierung in Browsern in den meisten Fällen das Ziel. Selenium ist ein weit verbreitetes Tool. Das liegt unter anderem an einfacher Nutzbarkeit und der Kompatibilität mit den meisten Browsern und einer Vielzahl von Programmiersprachen. Selenium ist ein übergreifendes Projekt für mehrere Tools zur Browser-Automatisierung [6].

- Selenium WebDriver
- Selenium Integrated Development Environment (IDE)
- Selenium Grid
- Selenium Remote Control (RC)

Wenn von Selenium gesprochen wird, ist in den meisten Fällen Selenium WebDriver gemeint. WebDriver, auch Selenium 2 genannt, besteht zum einen aus einer Code Bibliothek, die in verschiedenen Programmiersprachen verwendbar ist, und zum anderen aus browserspezifischen Treibern.
Mithilfe von Selenium WebDriver kann in Programmiersprachen, wie Java, Python und

C#, Code geschrieben werden, der Tastatur- und Mausinputs im Browser simuliert. Dafür muss auf die Hypertext Markup Language (HTML) Elemente der Website zugegriffen werden. Der entsprechende Treiber führt dann beim Ausführen des geschriebenen Programms die im Code definierten Schritte durch [7].

Ein weiteres zu Selenium gehöriges Tool ist Selenium IDE. Dies ist als ein Firefox Add-On und eine Chrome Extension umgesetzt. Eine der wichtigsten Funktionen von Selenium IDE ist das Aufnehmen und Wiedergeben von Interaktionen mit dem Browser. Somit können Arbeitsschritte einfach und schnell automatisiert werden. Für umfangreiche Testprozesse mit einer Vielzahl von Testfällen wird es jedoch nicht empfohlen. In solchen Fällen sollte Selenium WebDriver oder Grid verwendet werden [8].

Um die gleichzeitige Ausführung von Testfällen auf mehreren Maschinen mit unterschiedlichen Betriebssystemen und Browsern zu koordinieren kann Selenium Grid eingesetzt werden. Einer der größten Vorteile von Grid liegt in der Zentralisierung des Testprozesses. Der Nutzer hat die Möglichkeit in seinem System unterschiedliche Konfigurationen von zum Beispiel Browserversionen für eine Reihe von Tests festzulegen. Selenium Grid sendet dann Befehle an die entfernten Browser. So können die Tests in unterschiedlichen Umgebungen parallel durchgeführt werden [9].

Selenium RC, auch bekannt als Selenium 1, war das Hauptprojekt von Selenium, bevor es durch Selenium 2 ersetzt, beziehungsweise in Selenium 2 integriert wurde. In Selenium RC wurden erstellte Skripte über den Selenium Server im Browser ausgeführt. Heutzutage wird Selenium RC nicht mehr unterstützt [10].

Apache Spark

Apache Spark ist nicht wie Selenium ein Tool mit dem primären Nutzen der Testautomatisierung, sondern ein Open Source Framework für parralele Datenverarbeitung. Spark ist darauf ausgelegt große Datenmengen effizient zu verarbeiten. Die schnellen Ausführungszeiten entstehen vor allem durch die Speicherung der für das jeweilige Programm benötigten Datenmengen im Arbeitsspeicher, sodass diese für das Programm jederzeit schnell abrufbar sind. Spark wurde ursprünglich in Scala implementiert, bietet aber eine Application Programming Interface (API) für die Programmiersprachen Java, R, Scala und

Python an [11]. Grundsätzlich bestehen Spark Anwendungen immer aus einem Treiber und einer Reihe an Arbeitern [12].

Spark Komponenten Spark setzt sich aus mehreren Komponenten zusammen.
Spark Core ist meist bekannt unter Resilient Distributed Data Set (RDD) (deutsch: belastbarer verteilter Datensatz). RDD's sind eine Datenstruktur in Spark, die eine verteilte Menge an Objekten darstellt. Wichtig zu beachten ist, dass diese RDD's immutable - also nicht veränderbar sind. Das bedeutet, sollte eine Änderung an einem RDD vorgenommen werden, dann wird dieser nicht tatsächlich abgeändert, sondern ein neuer RDD erstellt, in dem die gewünschten Änderungen zur Zeit seiner Anfertigung angewendet werden. Außerdem ermöglichen RDD's parallelisierte Berechnungen auf den in ihnen enthaltenen Objekten [12].
Mithilfe von Spark Structured Query Language (SQL) ist es möglich Datensätze in einer ähnlichen Weise zu manipulieren, wie es mit SQL getan werden kann. Dafür werden als Datenstruktur hauptsächlich so genannte Dataframes verwendet. Diese besitzen im Vergleich zu RDD's nähere Informationen bezüglich der Struktur der Daten [11].
Eine Erweiterung von Spark Core stellt Spark Streaming dar. Durch diese Komponente ist in Spark nicht nur die sequentielle Verarbeitung großer Datenmengen möglich, sondern auch die Echtzeitverarbeitung von Streaming-Daten.
Die Spark **MLib!** (**MLib!**) Bibliothek besteht aus einer Reihe von Funktionen zur Unterstützung von Machine Learning für Apache Spark [12].
GraphX eröffnet die Möglichkeit die parallele Datenverarbeitung auf Graphen auszudehnen. Es besteht aus einer Reihe von Algorithmen zur Manipulation der Graphen [11].

Deequ Eine Erweiterung der Funktionalitäten von Apache Spark stellt Deequ dar. Dies ist eine von Amazon entwickelte Bibliothek zur Durchführung von Modultests auf Datenbanken [13]. Mit Modultests ist das separate Testen der einzelnen Komponenten eines Systems gegen deren komponentenspezifische Anforderungen gemeint [14].

Ein Nutzer kann mithilfe der Funktionen von Deequ Einschränkungen und Regeln für die zu überprüfenden Tabellen festlegen. Diese werden bei der Ausführung in SQL-Anfragen umgewandelt und durch die Unterstützung von Spark SQL auf die Tabellen angewendet.

Informationen darüber, ob die einzelnen Tests erfolgreich waren oder nicht können hinterher ausgegeben werden [13].

Eine Besonderheit von Deequ ist die Fähigkeit zu testende Einschränkungen für eine gegebene Tabelle automatisch zu generieren. Dabei handelt es sich um Empfehlungen, die von Deequ nach der Inspektion einer Tabelle an einen Nutzer in Form von Deequ Funktionen, gemacht werden können [13].

3.3 Grundlegende Konzepte von Datenbanken

Weil in dieser Projektarbeit das Testen von Datenbanken im Fokus steht, sollen in diesem Kapitel kurz einige grundlegende Konzepte von Datenbanken erläutert werden.

Unter einer Datenbank wird ein strukturierter Datenbestand verstanden, der durch ein Database Management System (DBMS) kontrolliert wird. Ein DBMS ist eine Software, die zur Verwaltung von Datenbanken eingesetzt werden kann. Die Kombination einer Datenbank und eines DBMS nennt man Datenbanksystem. Über das DBMS werden Datenbanken, Datenbankmodelle zugeordnet. Diese beschreiben die Daten. Eines der meistverwendeten Datenbankmodelle ist das Relationenmodell [15].

Relationale Datenbanken definieren sich darüber, dass die Inhalte einer Datenbank in Tabellen eingeteilt sind. Diese Tabellen werden auch Relationen genannt. Die Spalten werden als Attribute, und die Zeilen als Tupel bezeichnet. In dieser Arbeit werden die Begriffe Relation und Tabelle, Spalte und Attribut, sowie Zeile, Reihe und Tupel jeweils synonym verwendet. Alle in dieser Arbeit betrachteten Datenbanken folgen einem relationalen Datenbankenmodell. Um auf Datenbanken zuzugreifen wird in den meisten Fällen eine Version der Sprache SQL verwendet [15].

3.4 Verwendete Software

In dem dieser Projektarbeit zugrundeliegenden Projekt wurde Software von Drittanbietern unter anderem zur Unterstützung des Testprozesses verwendet. Dieses Kapitel soll diese

Software und deren für diese Arbeit relevanten Funktionalitäten beschreiben.

3.4.1 Jira und Xray

Jira ist eine Webanwendung die Teams beim Projektmanagement unterstützen kann. Zum Beispiel ist es möglich Aufgaben zu definieren, zu kategorisieren und Personen zuzuweisen. Diese werden auch als "issues" bezeichnet. Ein großer Fokus liegt auf agilen Arbeitsweisen, so wird unter anderem ein Kanbanboard zur Verfügung gestellt [16].
Gleichzeitig bietet Jira auch eine Erleichterung im Bereich des Testmanagements. Xray ist ein Add-on für Jira und ermöglicht es Tests zu definieren und zu verfolgen. Innerhalb eines Tests können eine Beschreibung des Tests, die zu schaffenden Vorherbedingungen des Tests und die einzelnen Testschritte spezifiziert werden. Für jeden Testschritt ist es zusätzlich möglich ein erwartetes Ergebnis und weitere Informationen oder Dokumente hinzuzufügen. Außerdem ist es möglich Tests zu sogenannten Testsets und auch Testplänen hinzuzufügen, was vor allem der Strukturierung dient.
Handelt es sich um einen manuellen Test, so kann dieser vom Tester gestartet und die Ergebnisse des Tests eingetragen werden. Dabei können den einzelnen Schritten Status zugeordnet werden. Alle zurückliegenden Ausführungen eines Tests bleiben gespeichert [17].

3.4.2 Kurzer Exkurs zu AWS

AWS ist ein Anbieter von über 175 cloudbasierten Services. Weltweit ist es die meistgenutzte Cloud Platform [18]. Im Folgenden werden nur die während des Testprozesses im vorliegenden Projekt eingesetzten Services und deren Möglichkeiten beschrieben. Hierfür wird nur ein grober Überblick über einige bestehende Funktionen dieser Services, aber keine genaueren Erklärungen zu deren Funktionsweise gegeben. Einige dieser Angebote können kostenlos sein, doch für dieses Projekt wurden nur kostenpflichtige Versionen verwendet.

Simple Storage Service (S3)

Der AWS Service S3 ist eine der meist genutzten Funktionen von AWS und bietet die Möglichkeiten Daten in der Cloud zu speichern. Daten können nur in so genannten Buckets

gespeichert werden. Um etwas in die Cloud hochzuladen muss ein Nutzer also zuerst mindestens einen Bucket erstellen. In diesen Buckets können wiederum Ordner oder Dokumente gesichert werden [19].

Athena

Athena ist ein Abfragedienst für Datenbanken, der es ermöglicht leicht auf in Amazon S3 gespeicherte Datenbanken zuzugreifen. Die Abfragen sind hierfür über standard SQL zu formulieren. Nutzern ist es somit möglich die gewünschten SQL-Abfragen einzugeben und innerhalb von Sekunden die resultierende Tabelle zu erhalten [20].

Glue

Mithilfe von Amazon Glue können Extract, Transform, Load (ETL) Prozesse ausgeführt werden. In diesem Fall ist die wichtigste Funktion der so genannte Glue Job. Ein solcher Job stellt ein Skript dar, das cloudbasiert ausgeführt werden kann. Zum Beispiel ist es hiermit möglich in S3 gespeicherte Tabellen zu transformieren. Für das schnelle Arbeiten mit großen Datenmengen wird Apache Spark eingesetzt. Glue Jobs müssen in den Programmiersprachen Python oder Scala verfasst werden.

Aufgrund der Art und Weise, wie Glue mit Daten arbeitet ist es nach einem ausgeführten Glue Job, der eine Tabelle verändert oder erstellt hat, notwendig diese zu crawlen. Crawler sind eine Funktion von Amazon Glue, die Metadaten(zum Beispiel das Schema der Daten) von Datenquellen sammelt und diese speichert. Erst wenn diese Metadaten gesammelt wurden ist es wieder möglich die entsprechende Tabelle zum Beispiel über Amazon Athena anzeigen zu lassen [21].

Simple Email Service (SES)

Amazon SES erlaubt es dem Nutzer über eine Plattform von Amazon Emails zu versenden und zu empfangen. Zu diesem Zweck ist es möglich zum Beispiel mithilfe eines entsprechenden AWS Software Development Kit (SDK) Skripte zu erstellen um das Versenden von Emails zu automatisieren [22].

4 Die Testvorgänge im Projekt

Im Kapitel 2 (Grundlegendes zum Projekt) wurde beschrieben, welche Problematik das Testen erforderlich macht. In diesem Kapitel soll genauer darauf eingegangen werden, was und auf welche Weise getestet wurde.

4.1 Ein kurzer Überblick

Grundsätzlich stand die Datenbasis im Zentrum des Testens. Zum einen wurde an der neu erstellten Datenbank zum anderen an der zu entwickelnden Berichterstattungssoftware getestet. Ausschließlich die Anwendung zu testen wäre nicht ausreichend gewesen, da damit Fehler in der DBFNEU von Fehlern in der Anwendung, oder an der Schnittstelle zwischen Datenbank und Anwendung nicht hätten unterschieden werden können. In beiden Fällen existieren sowohl manuelle Tests, als auch Konzepte für automatisierte Tests.

Um einen Überblick über die durchgeführten Tests zu behalten wurde die Jira Funktion Xray verwendet. Alle definierten Testfälle wurden in kleineren Gruppen von ungefähr 40 Testfällen pro Gruppe zusammengefasst. In Jira stellte jede dieser Gruppen einen Xray Test dar. Dafür wurden innerhalb eines Xray Tests pro Schrittfeld ein Testfall eingetragen. Die Ergebnisse der Testdurchführungen waren somit für alle zum Projekt gehörenden Personen in Jira abrufbar.

Alle Testfälle folgten dabei dem gleichen Muster. Gegenstand der Überprüfung war die Anzahl an Tabelleneinträgen, also Tupel, für eine bestimmte Kombination von Filtern. In einer fiktiven Relation “Schüler“ mit den Attributen “Fächer“, “Name“ und “Alter“ könnte auf diese Weise zum Beispiel überprüft werden, wie viele Schüler 18 Jahre alt sind und das Unterrichtsfach Spanisch besuchen. Da die Menge der zu testenden Filterkombinationen sehr hoch gewesen wäre, hätte man alle relevanten Kombinationen getestet, waren nur stichprobenartig Testfälle erstellt worden. Die erwarteten Ergebnisse für die Testfälle wurden aus einer anderen Menge an Datenbanken gewonnen. Es ist davon auszugehen, dass dort keine Fehler auftreten. Daher wird diese Menge an Datenbanken im Weiteren

Datenbanken Richtig (DBR) genannt. Um die erwarteten Ergebnisse zu erheben, galt es mithilfe von SQL-Anfragen manuell auf diese Datenbanken zuzugreifen und die erfassten Ergebnisse in Jira bereitzustellen. Eine weitere Schwierigkeit bestand darin, dass sowohl in die DBR als auch die DBFNEU regelmäßig neue, aktuellere Daten eingespeist wurden. Daher mussten auch die erwarteten Ergebnisse der Testfälle periodisch angepasst und neu erfasst werden.

4.2 Testen der Datenbank

Zum Zeitpunkt der Erstellung dieser Arbeit wurden Testdurchführungen von drei der definierten Testfall-Gruppen vorgenommen. Jede dieser Gruppen verlangte die Erhebung von erwarteten Werten aus jeweils einer anderen Datenbank aus der Menge DBR. In den folgenden Kapiteln sollen zuerst die Tests für die DBFNEU und danach diejenigen für die Anwendung näher beschrieben werden.

4.2.1 Manuelle Tests

Die Tests für die DBFNEU waren bezogen auf die zu erstellende Anwendungssoftware statischer Natur, da die Anwendung für die Durchführung dieser Tests nicht ausgeführt werden musste.
Alle zu DBR, DBFNEU und DBFALT gehörenden Datenbanken sind in S3 Buckets der Cloud Plattform AWS gespeichert. Die manuellen Tests der DBFNEU wurden mithilfe des Athena Services von AWS durchgeführt. Die dafür benötigten SQL-Anfragen wurden in den jeweiligen Testfällen in Xray eingetragen. Ein Tester musste die SQL-Anfrage in Athena eingeben, ausführen und das Ergebnis mit dem erwarteten Ergebnis abgleichen.
Die Anfragen waren alle sehr ähnlicher Natur, da sie einfach nur eine Reihe von Filtern auf eine Tabelle anwendeten. Das zu überprüfende Ergebnis bestand dann immer in der Anzahl der übrigen Reihen der gefilterten Tabelle. Diese Zahl musste vom Tester mit dem erwarteten Ergebnis abgeglichen werden. Die Erhebung der erwarteten Werte wurde auf ähnliche Art und Weise durchgeführt. Das bedeutete einen hohen zeitlichen und personellen Aufwand für den Testprozess.

Die in Jira sichtbaren Ergebnisse der Testfälle wurden von Entwicklern direkt verwendet, um an der Behebung der Defects in der DBFNEU zu arbeiten. Problematisch dabei war, dass die Datenbank eine sehr große Menge an Einträgen enthielt, weswegen fehlerhafte Einträge nicht einfach einzeln, manuell geändert werden konnten. Dies hätte einen zu hohen Zeitaufwand bedeutet. Deshalb mussten grundsätzliche Konzepte und Regeln der Datenbank angepasst werden. Nachdem eine solche Anpassung erfolgt war, musst die Datenbank daraufhin überprüft werden, ob die gefundenen Defects behoben, und ob neue Defects hinzugekommen waren. Auch immer dann, wenn aktuellere Daten in die Datenbank eingespeist wurden, waren alle Testfälle erneut zu überprüfen. Aus diesen Gründen wurden Regressionstests angewendet. Da es in der Natur von Regressionstests liegt, Testfälle häufig auszuführen, kam die Idee auf die Durchführung der Tests zu automatisieren. Dafür gab es verschiedene Möglichkeiten, die im nächsten Kapitel genauer betrachtet werden.

4.2.2 Automatisierte Tests

Eine Möglichkeit war es ein vom Kunden entwickeltes Tool zu verwenden, das bereits in anderen Projekten zum Testen von Datenbanken eingesetzt wurde. Im Folgenden auch als FDD bezeichnet. Diese Variante hatte den Vorteil, dass das System bereits komplett einsatzfähig zur Verfügung stand. Allerdings war es nicht genau für den vorliegenden Anwendungsfall entwickelt worden, sondern sollte hauptsächlich Formfehler in Datenbanken aufdecken und sicherstellen, dass gewisse Prinzipien eingehalten werden.

Eine andere Möglichkeit war ein neues Tool zu erstellen, das die Automatisierung übernehmen konnte. Die zweite Variante erforderte auf der einen Seite einen höheren Initialaufwand, bot aber auf der anderen auch viel mehr Freiheit, das Tool an die speziellen Anforderungen des Anwendungsfalles anzupassen.

Das Projektteam musste also eine Entscheidung treffen, welche der beiden Varianten eingesetzt werden sollte. Gleichzeitig war es im Rahmen des Projekts jedoch nicht tragbar beide Varianten zu implementieren und hinterher zu entscheiden, welche Variante als besser empfunden wurde. Diese Gegebenheiten führten zur Erstellung dieser Arbeit. Als relevante Kriterien für eine solche Entscheidung wurden hauptsächlich mögliche Zeiteinsparungen, gute Übersichtlichkeit der Testergebnisse, hohe Zuverlässigkeit und Einfachheit des Sys-

tems festgelegt.

Der Autor wurde damit beauftragt ein Konzept für ein neues Tool für die automatisierten Tests zu entwickeln und dieses wenn möglich bereits in Code umzusetzen. Um den initialen Mehraufwand möglichst gering zu halten, der bei dem bereits vorhandenen Tool nicht aufgewendet werden musste, entschied sich der Autor für die Erstellung eines einfachen Skripts, das nur die für die vorliegenden Testfälle benötigten Funktionalitäten enthalten sollte. Das Skript verfasste der Autor in der Programmiersprache Python und zur einfacheren Arbeit mit den großen Datenmengen wurde Spark verwendet.

Funktionsweise des Skripts

Ein Problem, das beim manuellen Testen aufgetreten war, bestand darin, dass die erwarteten Werte für die einzelnen Testfälle aus unterschiedlichen Datenbanken manuell gewonnen werden mussten. Um dem entgegenzuwirken wurde bei der Erstellung des Skriptes dafür gesorgt, dass gleichzeitig auf die jeweils benötigten Datenbanken aus DBR und DBFNEU, zugegriffen und die erhaltenen Werte verglichen wurden.

Die Definition der Testfälle geschieht in diesem Skript nicht über SQL-Anfragen, sondern über die Definition einer Reihe von Filtern, die auf entsprechende Tabellen angewandt werden. Dafür besteht pro in Xray - definierter Testfallgruppe eine Liste. Diese Einteilung wird vorgenommen, da jede dieser Gruppen einer anderen Datenbank aus DBR gegenübersteht. Jeder Testfall wird wiederum über eine Python Liste, in der zwei Python Dictionarys vorhanden sind, abgebildet. Die zwei Dictionarys enthalten eine Reihe von Spalten und korrespondierenden Werten, auf die gefiltert werden soll. Das erste Dictionary repräsentiert hierbei alle Filter für die Tabelle einer Datenbank aus DBR, und das zweite Dictionary alle Filter für die Tabelle aus DBFNEU. Ein Beispiel ist in Quelltext 4.1 sichtbar.

```
1 testcases_group_1 = [
2         #Diese Liste stellt einen Testfall dar:
3         [{"Spalte aus DBR Tabelle":"Filterwert", "eine weitere Spalte":"Filterwert"},
4         {"Spalte aus DBFNEU Tabelle":"Filterwert", "eine weitere Spalte":"Filterwert"}],
5         #Diese Liste stellt einen weiteren Testfall dar:
6         [{"":""}, {"":""}]
7 ]
```

Quelltext 4.1: `Beispieldefinition von Testfällen`

Somit müssen pro Testfall nicht mehr SQL-Anfragen erstellt werden, sondern nur aus Spalte und Wert bestehende Filterpaare. Die mit Testfällen gefüllten Listen werden dann in der Hauptfunktion des Skriptes verwendet. Diese Funktion ist in Quelltext 4.2 zu sehen. Das vollständige Dokument ist in Anhang 1 einzusehen.

```
1  def test_testcases(DBR_df, DBFNEU_df, testcases):
2          """filters DBR and DBFNEU dataframes by conditions specified in testcases and then
3          compares number of rows for the filtered dfs"""
4          for case in testcases:
5                  DBR_df2 = DBR_df
6                  DBFNEU_df2 = DBFNEU_df
7                  case_parameters = ""
8                  #filter DBR_df
9                  for column, value in case[0].items():
10                         if type(value) == list:
11                                 DBR_df2 = DBR_df2[col(column).isin(value)]
12                         else:
13                                 DBR_df2 = DBR_df2[col(column) == value]
14                 #filter DBFNEU_df
15                 for column, value in case[1].items():
16                         if type(value) == list:
17                                 DBFNEU_df2 = DBFNEU_df2[col(column).isin(value)]
18                                 case_parameters += f"{column}:{value} "
19                         else:
20                                 DBFNEU_df2 = DBFNEU_df2[col(column) == value]
21                                 value = str(value)
22                                 case_parameters += f"{column}:{value} "
23
24                 #compare dbs and save result
25                 if DBR_df2.count() == DBFNEU_df2.count():
26                         result_rows.append([case_parameters, "Success"])
27                 else:
28                         result_rows.append([case_parameters,"Failure"])
```

Quelltext 4.2: `Auszug aus Automation.py`

Die Hauptfunktion iteriert über alle eingetragenen Testfälle und filtert zuerst die aus DBR und danach die aus DBFNEU, respektiv in den Schleifen beginnend in Zeile 9 und 15. Um zu gewährleisten, dass die Werte, nach denen gefiltert werden soll, auch aus Listen von Werten bestehen können, werden Listen separat behandelt (siehe Zeilen 10 und 11 sowie 16 und 17). Nachdem beide Tabellen gefiltert wurden, wird die Anzahl ihrer Zeilen verglichen. Sollte die Zeilenanzahl gleich sein, wird der Testfall als erfolgreich gewertet und falls nicht, als fehlgeschlagen.
Das Resultat jedes Testfalls wird in einer separaten Tabelle gespeichert. Dort werden die Filteroptionen des Testfalls sowie der Status “Success“ oder “Failure“ eingetragen.

In der Praxis soll das Skript als ein Job des AWS Services Glue implementiert werden. Dort ist es nach einem selbst erstellten Zeitplan automatisch ausführbar. Die Tabelle, die das Ergebnis dieses Jobs darstellt, ist dann zum Beispiel über Athena einzusehen. Die Teile des Codes, in denen auf konkrete Datenbanken zugegriffen wird, sind aus datenschutzrechtlichen Gründen nicht im Anhang 1 enthalten.

Funktionsweise des FDD

Das vom Kunden verwendete Tool zur Durchführung von Datenbankentests ist um einiges komplexer als das eben beschriebene Skript. Das Ziel bei der Entwicklung dieses Frameworks war es, Qualitätschecks von Datenbanken zu ermöglichen.

Deshalb wurde die Spark Erweiterung Deequ verwendet. Die auszuführenden Testfälle werden in einer Java Script Object Notation (JSON) Datei festgelegt.JSON ist ein für Datenaustausch weit verbreitetes Dateiformat, das auch leicht für Menschen lesbar ist [23]. Pro Testfall müssen die betroffene Tabelle und Spalte, sowie die Art der Beschränkung angegeben werden. Die Definition der Beschränkungsarten wird hierbei mithilfe von Deequ Funktionen getätigt. Somit kann zum Beispiel getestet werden, ob alle Werte in einer Spalte dem gleichen Datentyp angehören oder ob jeder Wert in einer Spalte einzigartig ist. Alle existierenden Deequ - Funktionen sind unter [24] zu finden. Alternativ gibt es auch die Möglichkeit statt Deequ - Funktionen, SQL-Anfragen zu verwenden. In diesem Fall müssen Tabelle und Spalte nicht separat angegeben werden, da dies über die SQL-Anfrage selbst geschieht. Für die Form der SQL-Anfragen gibt es einzuhaltende Vorgaben. Das Ergebnis der SQL-Anfrage muss immer aus einer Tabelle mit einer Spalte bestehen. Im Fall, dass ein Test erfolgreich ist, muss diese Spalte den Text “Success“ enthalten. Andernfalls ist der eingetragene Text nicht relevant. Eine solche SQL-Anfrage könnte beispielhaft wie in Quelltext 4.3 dargestellt aussehen.

```
1 Select Case When
2 ( ( Select Count(*)
3 From table
4 Where column_1 = 500 and column_2 = 1)
5 = 123 )
6 Then 'Success'
7 Else 'Test Failed'
8 End As Result
```

Quelltext 4.3: Beispiel einer SQL-Anfrage im FDD

In dieser SQL-Anfrage wird die Tabelle "table" auf die Einträge gefiltert, in denen der Wert der Spalten "column_1" und "column_2" respektiv 500 und 1 beträgt. Sollte die Anzahl der Zeilen 123 entsprechen, so wird als Ergebnis "Success" widergegeben, andernfalls ist das Ergebnis "Test Failed". Alle für die Datenbankentests bestehenden Testfälle können unter Verwendung des FDD nur mithilfe von SQL-Anfragen, nicht aber über Deequ Funktionen, ausgedrückt werden.
Auf diese Weise mit SQL-Anfragen zu testen bringt die Begrenzung, dass die SQL-Anfragen in einer Zeile definiert werden müssen. Dies schränkt vor allem die Lesbarkeit ein. Aus diesem Grund bietet das FDD die Möglichkeit solche Testfälle in SQL-Dateien, anstatt in der eben beschriebenen JSON Datei, zu definieren. Dafür ist für jeden Testfall eine eigene SQL-Datei zu erstellen, in der zuerst einige Metadaten einzutragen sind, gefolgt von der SQL-Anfrage. Das Ergebnis muss auch hier der eben beschriebenen Form folgen. Auf diese Weise ist es möglich die Lesbarkeit der einzelnen SQL-Anfragen zu steigern, dies ist vor allem bei langen und komplexen Anfragen sinnvoll. Gleichzeitig ist pro Testfall eine Datei notwendig, was wiederum, gerade bei einer hohen Anzahl von Testfällen, die Übersichtlichkeit über die Tests negativ beeinträchtigt.
Unabhängig davon, ob das JSON oder SQL-Format gewählt wurde, gibt es die Möglichkeit den Schweregrad des Fehlschlags eines jeden Testfalles zu definieren. Dieser Schweregrad kann als hoch, mittel, niedrig oder nichtig angegeben werden.

Damit Tests ausgeführt werden können, müssen die präparierten Dokumente (JSON und/oder SQL-Dateien) in einem Ordner mit einem vorgegebenen Namen in einem AWS S3 Bucket bereitgestellt werden. Die Durchführung der Tests wird dann von einem AWS Glue Job vorgenommen. Dieser Job greift auf die betroffene Datenbank, deren Tabellen, die Deequ Tests und die SQL-basierten Tests zu. Daraufhin werden alle definierten Testfälle mit der Hilfe von Spark SQL überprüft. Die Ergebnisse werden wiederum in einer DynamoDB Tabelle gespeichert. Durch den AWS Service SES werden die Resultate der Testfälle nun per E-Mail verschickt. Dabei entspricht eine E-Mail einem Testfall. Nur für Testfälle, die fehlschlugen, werden, E-Mails versendet. Diese enthalten Informationen über Schweregrad und Art des Fehlers, betroffene Datenbank und Tabelle, sowie eine Beschreibung des Testfalls.

Eine weitere Funktion, die das Framework bietet, sind die Einschränkungsvorschläge von Deequ. Dabei handelt es sich um automatisch generierte Empfehlungen, welche Einschrän-

kungen bei einer gegebenen Datenbank überprüft werden sollten. Um diese Funktion zu nutzen muss in dem Glue Job, der andernfalls die definierten Tests ausführt, der Empfehlungsmodus explizit ausgewählt werden. Es ist jedoch nicht möglich Tests durchzuführen, während der Empfehlungsmodus aktiviert ist. Die computergenerierten Empfehlungen haben immer die Form von Deequ Funktionen, und nicht von SQL-Anfragen, . Als Ergebnis dieses Vorgangs wird eine JSON Datei gespeichert, in der die vorgeschlagenen Einschränkungen/Testfälle im korrekten Format definiert sind. Diese Datei kann also falls nötig angepasst und dann direkt für das Testen der Datenbank angewendet werden. Wie bereits erwähnt können die für die Datenbankentests festgelegten Testfälle nicht als Deequ Funktion ausgedrückt werden. Aus diesem Grund werden die Einschränkungsvorschläge von Deequ nicht direkt benötigt. Sie könnten jedoch trotzdem Anregung für eine andere Art von Testfällen liefern.

An welche E-Mail-Adressen die Ergebnisse der Testfälle gesendet werden, ist in den Job Parametern des Glue Jobs festzulegen. Zusätzlich zu E-Mails gibt es auch die Möglichkeit die Testergebnisse als Nachrichten in Microsoft Teams Kanälen anzuzeigen. In den Parametern des Glue Jobs wird auch die zu testende Datenbank spezifiziert. Daraus folgt, dass bei einer Ausführung dieses Glue Jobs nur Tabellen einer einzigen Datenbank überprüft werden können. Aus diesem Grund wird für gewöhnlich der Glue Job jeweils für alle zu überprüfenden Datenbanken dupliziert. Das bietet den Vorteil mehrere Glue Jobs gleichzeitig ausführen und somit zwei oder mehr Datenbanken parallel überprüfen zu können.

4.3 Testen der Anwendung

Wie im Kapitel 4.1 (Ein kurzer Überblick) erwähnt, werden auch Tests in der Berichterstattungs-Anwendung durchgeführt. Auch hier werden die Testfälle mithilfe von Xray überwacht. Die Testfälle entsprechen dabei genau denen für die Datenbanktests. In Xray wurden dafür trotzdem separate Tests erstellt, um Datenbanken- und Anwendungstests einfacher auseinanderhalten zu können.

4.3.1 Manuelle Tests

Um die Ausführung der Tests in der Anwendung zu erleichtern, haben die Tester Zugriff auf eine Tabelle, die im Endprodukt nicht enthalten sein wird, jedoch das standardmäßige Interface der Anwendung besitzt. Diese Tabelle umfasste die gesamte Datenbasis der Anwendung. Über das Interface der Applikation ist es möglich alle für einen Testfall relevanten Filter per Klick anzuwählen. Das angezeigte Ergebnis wird dann vom Tester mit dem erwarteten Ergebnis in Jira verglichen.

4.3.2 Automatisierte Tests

Um die gerade beschriebenen manuellen Tests automatisieren zu können sollte eine Software von Drittanbietern verwendet werden. Die Entscheidung hierfür fiel auf Selenium WebDriver. Dieser Beschluss wurde getroffen, da der Zugriff auf die Berichterstattungs-Anwendung nur über einen Browser geschehen kann und Selenium die Möglichkeit bietet, Arbeitsschritte in Browsern zu automatisieren. Dafür wurden die zur Durchführung der einzelnen Testfälle nötigen Schritte in Selenium definiert. Die so erzeugten Skripte filtern je nach Testfall die Tabelle und gleichen das Ergebnis mit dem erwarteten Ergebnis ab.

5 Vergleich der Möglichkeiten zur Testautomatisierung der Datenbankentests

In den zurückliegenden Kapiteln wurden alle Arten beschrieben, auf die im Projekt manuell getestet, sowie alle Möglichkeiten, die zur Automatisierung dieser Tests erkundet wurden. Im Folgenden werden die beiden Varianten zur Automatisierung der Datenbankentests verglichen. Zu diesem Zweck werden erst Vor- und Nachteile beider Tests erläutert und anschließend gegenübergestellt.

5.1 Variante 1 - Verwendung eines bestehenden Frameworks

Der größte Vorteil des Einsatzes des FDD für die Durchführung der Datenbankentests liegt in seiner Zuverlässigkeit. Das Framework wurde in einem separaten Projekt von dedizierten Softwareentwicklern erstellt und getestet. Zudem ist es bereits in mehreren Testprozessen zum Einsatz gekommen. Aus diesen Gründen kann von einer hohen Fehlertoleranz ausgegangen werden. Aktuell sind keine nicht behobenen Fehler im System bekannt. Auf der anderen Seite bietet es jedoch eine gewisse Abhängigkeit von den Urhebern. Entscheidungen über das Hinzufügen neuer Funktionen oder Änderungen des Prozesses werden vom zum Framework gehörigen Projektteam getroffen. Es ist natürlich möglich auch von außerhalb Vorschläge zu unterbreiten, deren Umsetzung ist jedoch nicht garantiert.

Das FDD besitzt eine Reihe verschiedener Funktionen, die den Testprozess unterstützen können. So zum Beispiel die Empfehlung von Einschränkungen, die während des Testdesigns eine Grundlage bieten kann. Die Möglichkeit Testfällen eine Bewertung ihrer Wichtigkeit zuzuordnen erleichtert bei der Auswertung der aufgetretenen Defects die Priorisierung. Informationen über fehlgeschlagene Tests sind per E-Mail erhältlich. Zudem können Testfälle sowohl über Deequ Funktionen als auch über SQL-Anfragen definiert werden, was mehr

Freiraum während der Testimplementierung liefert. Dafür macht es die Komplexität des FDD aufwendiger neue Funktionen hinzuzufügen. So ist nicht nur die Entscheidung über das Hinzufügen neuer Funktionen abhängig von einer Vielzahl an Personen, sondern auch deren Implementierung zeit- und kostenaufwendig.

Außerdem fordert das Verwenden dieses Frameworks an einigen Stellen einen Extra-Zeitaufwand. Die bestehenden SQL-Anfragen müssen in das vorgegebene Format umgewandelt, und dann in ein JSON - oder SQL - Dokument eingetragen werden. Bei der Erstellung neuer Testfälle müssen auch neue SQL-Anfragen kreiert werden. Die erstellten Dokumente sind in den entsprechenden S3 - Buckets hochzuladen. Erst dann kann der Glue Job ausgeführt werden. Außerdem müssen die erwarteten Ergebnisse der Testfälle weiterhin manuell erhoben werden, da das Framework pro Glue Job den Zugriff auf nur eine Datenbank ermöglicht. Der zeitliche Aufwand wird im Vergleich zur manuellen Variante lediglich in der Testdurchführung verringert, nicht jedoch in der Testimplementierung.

Ein weiterer Nachteil liegt in der Einarbeitungszeit, die benötigt wird, um mit diesem Framework testen zu können. Um das FDD zu verwenden ist es mindestens nötig die Dokumentation zu lesen, um Informationen über das einzuhaltende Format der Testfälle, den Speicherort für die Testfälle und die anzupassenden Parameter in den Eigenschaften des Glue Jobs zur Verfügung zu haben. Außerdem werden SQL - Kenntnisse vorausgesetzt.

5.2 Variante 2 - Erstellung eines neuen Frameworks

Die Stärken des vom Autor erstellten Skripts liegen hauptsächlich in seiner Simplizität. Es ist einzig und allein darauf ausgelegt zwei Tabellen aus einer oder mehreren Datenbanken zu Filtern und die Zeilenanzahl der gefilterten Tabellen zu vergleichen. Dadurch ist das Skript auch für Außenstehende leicht zu verstehen, sofern sie die benötigten Programmierkenntnisse in Python und Spark besitzen. Die Einfachheit des Skripts ruft aber gleichzeitig den Nachteil hervor, dass Testfälle, die nicht nach dem bestehenden Prinzip aufgebaut sind, nicht auf diese Weise automatisiert werden können. Zum aktuellen Zeitpunkt sind keine Testfälle geplant, die von dem bisher verwendeten Schema abweichen, jedoch ist es möglich, dass sich dies in der Zukunft ändert.

Ein anderer Vorteil besteht in der Art und Weise, wie auszuführende Testfälle implementiert werden. Es müssen hierbei nur Paare von Spalte und zu filterndem Werten in das Skript eingetragen werden. Das bedeutet es sind keine SQL-Kenntnisse nötig, um einen neuen Testfall einzutragen. Jedoch ist hier die Übersichtlichkeit über alle eingetragenen Testfälle eingeschränkt.

Dass das Skript extra für den vorliegenden Anwendungsfall entwickelt wurde, bedeutet zunächst einen Mehraufwand an Arbeitszeit und -kosten. Die Kosten hierfür sind jedoch relativ gering, da in das Skript nur die nötigsten Funktionen eingebaut wurden und somit keine umfangreiche Programmierung nötig war. Außerdem wurde das Programm von einem dualen Studenten und nicht einem voll ausgebildeten Softwareentwickler konzeptioniert und entwickelt, was Kosten spart. Es ist möglich, dass nicht bekannte Defects bestehen, da der duale Student weniger Erfahrung mitbringt, der Code von keiner zweiten Person überprüft und auch ein strukturiertes Testen des Skriptes nicht durchgeführt wurde.

Ein weiterer Punkt sind die Freiheiten, die das Verwenden dieses Skriptes eröffnet. Alle Entscheidungen über das Skript können vom Projektteam direkt getroffen werden. Ob eine neue Funktionalität hinzugefügt werden soll, hängt allein von den Bedürfnissen des Projektes ab. Das macht es möglich immer nur genau die Funktionen implementiert zu haben, die direkt gebraucht werden. Das spart Aufwand und senkt die Komplexität des Programms. Zudem ist das Hinzufügen neuer Funktionen auch vom Programmieraufwand her geringer, da nur wenige Abhängigkeiten zu beachten sind. Auch das ist nur dank der Einfachheit des Skriptes möglich.

Für den Testprozess bietet das Skript an mehreren Stellen Zeiteinsparungen. Die Testfälle in den Code einzutragen benötigt pro Testfall einen nur geringen Aufwand, da die einzutragenden Informationen bereits in den bestehenden SQL-Anfragen und Testfalldefinitionen in Xray vorhanden sind. Für zukünftige Testfälle müssen außerdem keine SQL-Anfragen mehr erstellt werden. Eine weitere zeitliche Einsparung passiert in der Erhebung der erwarteten Werte. Auch dafür ist bei jedem Testfall ein zeitlicher Aufwand nötig, da die hierfür benötigten Spalte - Werte - Kombinationen zunächst in das Skript einzutragen sind. Dies hat jedoch den Vorteil, dass die erwarteten Werte danach immer automatisiert erhoben werden und gleichzeitig immer aktuell sind.

Die Ergebnisse des Testens sind manuell über zum Beispiel Amazon Athena abrufbar. Dafür muss nach jeder Ausführung des Glue Jobs zusätzlich ein Crawler ausgeführt werden, der Metainformationen der erstellten Ergebnistabelle bereitstellt. Dieser Crawler kann immer automatisch nach der Ausführung des Glue Jobs arbeiten. Die Ausführungszeit eines solchen Crawlers beträgt wenige Sekunden. Die Einsicht in die Ergebnisse bildet somit zusätzlichen einmaligen manuellen Aufwand um einen Crawler zu konfigurieren, jedoch gleichzeitig eine gute Übersichtlichkeit über das Resultat eines jeden Testfalls. Zudem können die Ergebnisse leicht gefiltert werden.

Ein weiterer positiver Aspekt des vom Autor erstellten Skriptes ist, dass Definition und Ausführung aller Tests und Testfälle auf zwei oder mehreren Datenbanken innerhalb eines Glue Jobs zentralisiert sind. Gleichzeitig ist es mit sehr geringem Aufwand möglich zum Beispiel die Tests der einzelnen Testfallgruppen in unterschiedliche Jobs auszulagern, um die Übersichtlichkeit zu erhöhen.
Da die Testfälle direkt im Skript definiert werden müssen, werden alle bei einer Ausführung des Jobs nicht benötigten Testfälle herausgelöscht. Wenn diese jedoch bei der nächsten Ausführung wieder zu verwenden sind, müssen sie neu erstellt werden. Um dies zu umgehen kann auf das Löschen nicht gebrauchter Testfälle verzichtet werden, wobei dann immer alle Testfälle durchgeführt werden müssten. Das würde dazu führen, dass die Ergebnistabelle durch eine Vielzahl unnötiger Informationen unübersichtlicher wird. Eine Möglichkeit dieses Problem zu umgehen ist zum Beispiel die Testfälle im entsprechenden Format in Skript - unabhängigen Dokumenten zu speichern und nur in das Skript einzufügen, sollten sie auch benötigt werden.

5.3 Vergleich der beiden Varianten

Zusammenfassend ist zu sehen, dass die Variante mit dem FDD vor allem durch Zuverlässigkeit und Menge an Funktionalitäten überzeugt. Dafür ist sie jedoch unflexibler, da die Entscheidungsgewalt über das Tool außerhalb des eigenen Projektteams liegt. Das vom Autor erstellt Skript bekommt seine Vorteile vor allem durch Simplizität und Kontrolle des Projektteams über das Skript. Jedoch ist die Fehleranfälligkeit bei dieser Variante höher. Ausgeglichen werden kann dies über seine einfache Natur, denn so gibt es weniger Bereiche, in denen Defects auftreten können, gleichzeitig sind sie schneller auffind- und behebbar. In dem komplexen und langen Skript des anderen Frameworks hingegen kann das Finden von

Defects einen größeren Aufwand bedeuten.

Die Ergebnisse der Tests werden in der 1. Variante automatisch per E-Mail an festgelegte Personen oder Teams - Kanäle versendet. Hierbei sind auch weitere Informationen zu den fehlgeschlagenen Testfällen vorhanden. Für die zweite Variante muss nach der Ausführung des Glue Jobs manuell auf die erstellte Tabelle zugegriffen werden, um die Ergebnisse zu erlangen. Dafür wird hier eine bessere Übersichtlichkeit über alle durchgeführten Testfälle gewährt. Das ist im vorliegenden Anwendungsfall von Vorteil, da hier nicht jeder gefundene Defect einzeln behoben werden muss, sondern Strukturen in den aufgetretenen Defects herauszufinden und als Ganzes zu beheben sind. Das FDD liefert zwar schnell alle fehlgeschlagenen Testfälle per E-Mail jedoch gibt es keinen Überblick über dieselben.

Zwei der wichtigsten Ziele der Automatisierung sind die Zeiteinsparung und das Freisetzen von Arbeitskräften, um sie an anderen Stellen einsetzen zu können. Stellt man die beiden zu Grunde liegenden Varianten gegenüber so ist zu sehen, dass in Variante 1 die Implementierung sowohl bestehender als auch neuer Testfälle einen größeren Zeitaufwand verursacht als die gleiche Implementierung mit Variante 2. Die Ausführungszeit der beiden Glue Jobs ist nicht bekannt und wird daher als gleichwertig eingeschätzt. Die zweite Variante bietet zusätzlich Zeiteinsparungen bei der Erhebung von erwarteten Werten pro Testfall. Daraus folgt, dass Variante 2 für das Projekt zeitlich einen größeren Vorteil bietet als Variante 1.

5.4 Empfehlung an das Projektteam

Dem Projektteam wird empfohlen, das vom Autor für den konkreten Anwendungsfall erstellte Skript zur Automatisierung des Datenbankentests zu verwenden. Da das FDD in anderen Fällen schon zum Einsatz kam, bietet es zwar mehr Funktionen und könnte zuverlässiger arbeiten, unterliegt aber dennoch der anderen Variante. Diese bietet in den vom Projektteam als wichtig befundenen Kategorien, also durch ihren einfachen Charakter, die Übersichtlichkeit der Ergebnisse und die zu erwartende größere Zeitersparnis, eindeutige Vorteile.

6 Fazit und Ausblick

Der im Rahmen dieser Arbeit durchgeführte Vergleich zweier Automatisierungsmöglichkeiten hat ein eindeutiges Ergebnis erbracht und führte zu einer begründeten Empfehlung an das Projektteam. Im Ergebnis wird in diesem Anwendungsfall die Verwendung der vom Autor entwickelten Variante empfohlen. Obwohl davon auszugehen ist, dass die 1. Variante durch Zuverlässigkeit punkten kann, sind für dieses Projekt die Zeiteinsparungen, die Einfachheit des Systems sowie die Übersichtlichkeit der Ergebnisse von zentraler Wichtigkeit und in allen drei Kategorien ist die zweite Variante der ersten überlegen. Gleichzeitig ist jedoch anzuraten den Mehraufwand anzunehmen, das für diesen Anwendungsfall empfohlene Skript entsprechenden Softwaretests zu unterziehen, um mögliche Fehleranfälligkeiten zu reduzieren.

Die in dieser Arbeit verwendete Methode ist insofern kritisch zu betrachten, dass keine der beiden Varianten für den vorliegenden Anwendungsfall eingesetzt werden konnte. Daher liegen keine empirischen Daten zu Zeitersparnis und Zuverlässigkeit vor. Dennoch lassen die Ergebnisse des Vergleichs eindeutige Rückschlüsse darauf zu, welche der beiden Varianten für diesen Anwendungsfall größere Vorteile bringt.

Falls den Projektmitgliedern die Ergebnisse dieser Arbeit nicht ausreichen, um eine Entscheidung zu treffen, welches System implementiert werden soll, wäre eine empirische Untersuchung der beiden Möglichkeiten in der Live-Umgebung ein nächster Schritt.

A Anhang 1

```
#imports
from pyspark.context import SparkContext
from pyspark.sql.functions import *
from pyspark.sql import SQLContext
from pyspark.sql import SparkSession

sc = SparkContext()
sqlContext = SQLContext(sc)

spark = SparkSession \
    .builder \
    .config("spark.sql.broadcastTimeout", 36000) \
    .getOrCreate()

#get DBR_df's
DBR_ddf_group_1 = glueContext.create_dynamic_frame.from_catalog()
DBR_df_group_1 = orders_ddf.toDF()

DBR_ddf_group_2 = glueContext.create_dynamic_frame.from_catalog()
DBR_df_group_2 = orders_ddf.toDF()

DBR_ddf_group_3 = glueContext.create_dynamic_frame.from_catalog()
DBR_df_group_3 = orders_ddf.toDF()

#get DBFNEU_df
DBFNEU_ddf = glueContext.create_dynamic_frame.from_catalog()
DBFNEU_df = orders_ddf.toDF()

#set variables
result_rows = []

#define testcases
testcases_group_1 = [
```

```
#this list represents 1 testcase
[{"column␣from␣DBR␣df":"filter␣value",\
"colour":"green", "Age":"44"},\
{"column␣from␣DBFNEU␣df":"filter␣value", "amount":"320", \
"types":["type_1", "type_2", "type_3"]}],
#this list represents another test case
[{"":""}, {"":""}]
]

testcases_group_2 = [
    [{"":""}, {"":""}]
]

testcases_group_3 = [
    [{"":""}, {"":""}]
]

def test_testcases(DBR_df, DBFNEU_df, testcases):
    """filters DBR and DBFNEU dataframes by conditions
    specified in testcases and then
    compares number of rows for the filtered dfs"""
    for case in testcases:
        DBR_df2 = DBR_df
        DBFNEU_df2 = DBFNEU_df
        case_parameters = ""
        #filter DBR_df
        for column, value in case[0].items():
            if type(value) == list:
                DBR_df2 = DBR_df2[col(column).isin(value)]
            else:
                DBR_df2 = DBR_df2[col(column) == value]
        #filter DBFNEU_df
        for column, value in case[1].items():
            if type(value) == list:
                DBFNEU_df2 = DBFNEU_df2[col(column).isin(value)]
                case_parameters += f"{column}:{value}␣"
            else:
```

```
71                  DBFNEU_df2 = DBFNEU_df2[col(column) == value]
72                  value = str(value)
73                  case_parameters += f"{column}:{value}␣"
74
75          #compare dbs and save result
76          if DBR_df2.count() == DBFNEU_df2.count():
77              result_rows.append([case_parameters, "Success"])
78          else:
79              result_rows.append([case_parameters,"Failure"])
80
81  #execute function
82  test_testcases(DBR_df_group_1, DBFNEU_df, testcases_group_1)
83
84  test_testcases(DBR_df_group_2, DBFNEU_df, testcases_group_2)
85
86  test_testcases(DBR_df_group_3, DBFNEU_df, testcases_group_3)
87
88  #save result df
89  df.repartition(1).write.mode("overwrite").format("csv")\
90      .option("header", "true").save()
```

Literaturverzeichnis

[1] B. Hambling, Hrsg., *Software testing: An ISTQB-BCS certified tester foundation guide*, Fourth edition. Swindon, UK: BCS, The Chartered Institute for IT, 2019, ISBN: 1780174926.

[2] R. Black, *Advanced software testing / Rex Black*, 2nd ed. Rock Nook, 2014. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1680270788&lang=de&site=eds-live.

[3] R. Gopalaswamy und S. Desikan, *Software Testing: Principles and Practice*. Don Mills und Toronto: Pearson Education Canada und Pearson Canada [distributor], 2009, ISBN: 817758121X.

[4] R. Patton, *Software testing / Ron Patton*, 2nd ed. Sams Pub, 2005. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1680505769&lang=de&site=eds-live.

[5] T. Bucsics, *Basiswissen Testautomatisierung : Konzepte, Methoden und Techniken / Thomas Bucsics ... [and three others]*, 2., aktualisierte und überarbeitete Auflage. dpunkt.verlag, 2015. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1680241133&lang=de&site=eds-live.

[6] A. Axelrod, *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. Berkeley, CA: Apress, 2018, ISBN: 148423832X.

[7] Selenium, *Selenium Documentation*. Adresse: https://www.selenium.dev/documentation/en/.

[8] ——, *Selenium IDE*. Adresse: https://www.selenium.dev/selenium-ide/.

[9] ——, *Selenium Grid Documentation*. Adresse: https://www.selenium.dev/documentation/en/grid/.

[10] ——, *Selenium Legacy Documentation*. Adresse: https://www.selenium.dev/documentation/en/legacy_docs/selenium_rc/.

[11] B. Bengfort und J. Kim, *Interactive Spark using PySpark / Bengfort, Benjamin*, 1st edition. O'Reilly Media, Inc, 2016. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1684812895&lang=de&site=eds-live.

[12] S. Chellappan und D. Ganesan, *Practical Apache Spark : using the Scala API / Subhashini Chellappan, Dharanitharan Ganesan*. Apress, 2018, ISBN: 1484236521. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1680138332&lang=de&site=eds-live.

[13] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann und A. Grafberger, „Automating large-scale data quality verification," *Proceedings of the VLDB Endowment*, Jg. 11, Nr. 12, S. 1781–1794, 2018, ISSN: 2150-8097. DOI: 10.14778/3229863.3229867.

[14] A. Mili und F. Tchier, *Software Testing: Concepts and Operations*, 1. Auflage, Ser. Quantitative Software Engineering Series. New York, NY: John Wiley & Sons, 2015, ISBN: 1118662873.

[15] G. Saake, K.-U. Sattler und A. Heuer, *Datenbanken : Konzepte und Sprachen / Gunter Saake, Kai-Uwe Sattler, Andreas Heuer*, 6. Auflage. mitp Verlags, 2018, ISBN: 3958457789. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1680112104&lang=de&site=eds-live.

[16] Atlassian, *Jira Software*. Adresse: https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for#jira-for-agile-teams.

[17] Xray, *Xray Documentation*. Adresse: https://docs.getxray.app/display/XRAYCLOUD/About+Xray.

[18] Amazon, *AWS: What ist AWS*. Adresse: https://aws.amazon.com/de/what-is-aws/.

[19] ——, *AWS: S3*. Adresse: https://docs.aws.amazon.com/AmazonS3/latest/dev/.

[20] ——, *AWS: Athena*. Adresse: https://docs.aws.amazon.com/athena/latest/ug/.

[21] ——, *AWS: Glue*. Adresse: https://docs.aws.amazon.com/glue/latest/dg/.

[22] ——, *AWS: SES*. Adresse: https://docs.aws.amazon.com/ses/latest/DeveloperGuide/Welcome.html.

[23] B. Smith, *Beginning JSON / Ben Smith*, Ser. The expert's voice in web development. Apress, 2015. Adresse: https://login.ezproxy-dhma.redi-bw.de/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat07667a&AN=bkm.1680239597&lang=de&site=eds-live.

[24] Deequ, *Deequ Documentation*. Adresse: https://github.com/awslabs/deequ.

[25] Cucumber, *Cucumber Documentation*. Adresse: https://cucumber.io/docs/guides/overview/.