Synthetische Datengenerierung aus PostgreSQL für PostgreSQL

Studienarbeit

Timon Erhart
timon.erhart@ost.ch

Jari Elmer
jari.elmer@ost.ch

Ostschweizer Fachhochschule
Rapperswil

Experte:
Prof. Stefan Keller

Institutsleiter, Institut für Software OST

Betreuer:
Nicola Jordan

Laborleiter, Institut für Software OST

23. Dezember 2021

Abstract

In den Bereichen Software- und Data-Engineering sowie Machine-Learning besteht eine grosse Nachfrage nach umfangreichen Datensätzen. Dabei ist Datenschutz oftmals ein Grund, keine Realdaten zu verwenden. Die Generierung von synthetischen Daten mit ähnlichen statistischen Eigenschaften wie die Originaldaten ermöglichen es, einfach solche Datensets zu erstellen.

Das Kommandozeilen-Tool `pgsynthdata` erzeugt synthetische Daten ausgehend von einer PostgreSQL-Datenbank und füllt diese in eine generierte Datenbank mit gleicher Struktur ab. Grundlage für die Datengenerierung bilden statistische Werte, die PostgreSQL zur Verfügung stellt.

Ziel der Arbeit ist es, den bestehenden Prototyp in einen wartbaren, erweiterbaren und einfach zu nutzenden Zustand zu überführen. Das Programm soll auf beliebigen PostgreSQL-Datenbanken anwendbar sein und vollständig synthetisch generierte Datenbanken erstellen. Die schon unterstützten Datentypen sollen um weitere Datentypen ergänzt werden.

Unter Berücksichtigung gängiger Software Design Praktiken und Einbindung moderner Entwicklungswerkzeuge wurde ein umfassendes Refactoring vorgenommen und ein Plugin-System für die leichte Anbindung der Datengeneratoren geschaffen. Durch die Implementierung von neuen Generatoren für bisher noch nicht unterstützte Datentypen wie Arrays, Enums oder Spatial-Types (PostGIS) wurde das Tool einerseits erweitert und andererseits das Plugin-System validiert.

Entstanden ist ein wartbares und erweiterbares Programm mit einem flexiblen Plugin-System für Generatoren. Durch die klare Trennung nach Zuständigkeit der Module konnte die Qualität und Testbarkeit erhöht werden. Die neuen Generatoren haben demonstriert, dass sich das Plugin-System auch für komplexere oder benutzerdefinierte Datentypen eignet. Spezifische Generatoren ermöglichen über eine benutzerfreundliche Konfiguration die individuelle Anpassung an erweiterte Anwendungsfälle, wie die Generierung von realistische Personendaten oder Adressen.

Management Summary

Ausgangslage

In den Bereichen Software- und Data-Engineering sowie Machine-Learning besteht eine grosse Nachfrage nach umfangreichen Datensätzen. Dabei ist der Datenschutz oftmals ein Grund, keine Realdaten zu verwenden. Eine Pseudonymisierung bietet keinen vollständigen Schutz vor De-Anonymisierungs-Attacken. Eine komplette Anonymisierung wäre eine mögliche Lösung. Doch diese ist aufwändig und zeitintensiv.

Am vielversprechendsten sind rein synthetisch generierte Daten mit ähnlichen statistischen Eigenschaften wie die Originaldaten. Das hat den zusätzlichen Vorteil, dass beliebige Datenmengen erzeugt werden können. In Abbildung 1 wird der Prozess der Generierung synthetischer Daten visualisiert.

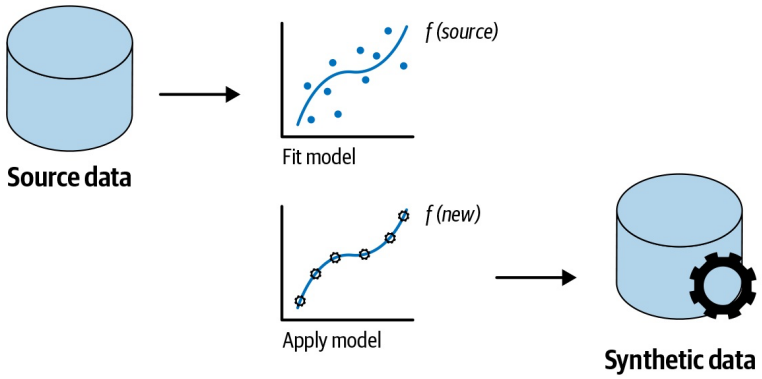


Abbildung 1: Generierung von synthetischen Daten aus Originaldaten
mit freundlicher Genehmigung (Emam, Mosquera & Hoptroff, 2020)

Ein an der Ostschweizer Fachhochschule (OST) entwickelter Softwareprototyp `pgsynthdata` setzt darum auf das Prinzip der rein synthetischen Datengeneratoren. Das Kommandozeilen-Tool ist in der Lage, fast beliebige PostgreSQL-Datenbanken zu synthetisieren und in eine generierte Datenbank mit gleicher Struktur und vergleichbaren statistischen Eigenschaften abzufüllen. Die Abbildung 2 veranschaulicht diesen Prozess. Als Grundlage dienen die Kataloge von PostgreSQL für funktionale Abhängigkeiten sowie Statistiken, die PostgreSQL zum Zwecke der Anfrageoptimierung erstellt. Ein Alleinstellungsmerkmal des Tools ist, dass es ohne aufwändige manuelle Konfiguration auskommt.

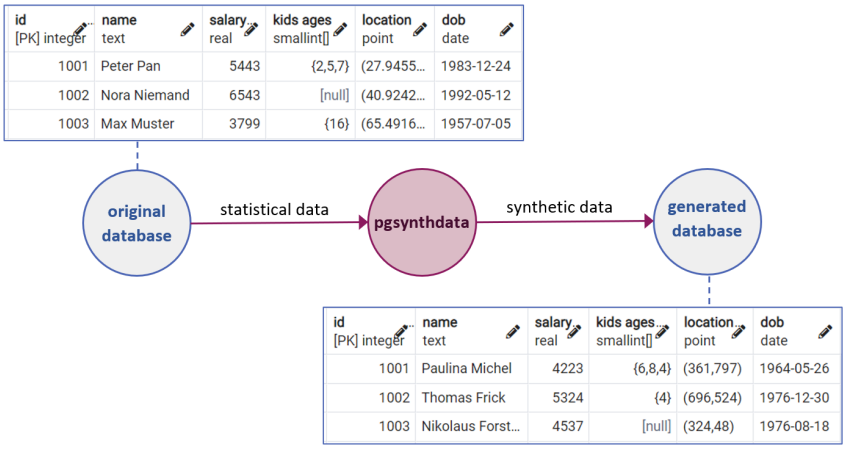


Abbildung 2: Anwendung von pgsynthdata

Das Ziel dieser Arbeit war es, die bestehende Software `pgsynthdata` in einen wartbaren, erweiterbaren und einfach zu nutzenden Zustand zu überführen. Zudem wurde die Funktionalität um Generatoren für Datentypen wie enumerated Types (Enum), Arrays und Geometrien erweitert.

Vorgehen und Resultate

Um das bestehende Tool in den durch die Aufgabenstellung definierten Zustand zu bringen, benötigte es zu Beginn ein umfassendes Refactoring des gesamten Tools. Der Fokus lag dabei auf einem Plugin-System, welches es ermöglicht, neue Generatoren einfacher zu integrieren. Um eine saubere Architektur mit klaren Verantwortlichkeiten zu erhalten, wurde der komplette Programmablauf neu geschrieben. Durch diese Entkopplung der Module konnte auch die Testabdeckung stark verbessert werden.

Nach der Übernahme der bestehenden Generatoren ins neue System, wurde ein Array-Generator für numerische Typen, ein Generator für PostGIS-Geometrie-Typen und ein Enum-Generator implementiert. Dies war in zweierlei Hinsicht wichtig: Zum einen wurde das Tool weiterentwickelt und unterstützt nun mehr Datentypen, zum anderen konnten die Funktionalität und Flexibilität des Plugin-Systems überprüft werden.

Aus dem Bedürfnis heraus, statt zufälliger Zeichenketten realistische Eigennamen generieren zu können, ist eine Möglichkeit entstanden, per Spalten-Kommentar im Schema spezifische Generatoren zu konfigurieren. Die implementierten Generatoren für Eigennamen und Postadressen verdeutlichen dieses Konzept. Das

Konfigurieren per Kommentar hat sich als benutzerfreundlich und unkompliziert bewährt, ausserdem benötigt es keine speziellen Fachkenntnisse.

Fazit und Ausblick

Entstanden ist ein wartbares und erweiterbares Programm mit einem flexiblen Plugin-System und mit Generatoren für eine Vielzahl von Datentypen. Constraints wie Unique- oder Foreign-Keys werden unterstützt. Bei nicht unterstützten Datentypen wird dem User eine entsprechende Warnung angezeigt.

Das Tool ist im Stande, aus verschiedenen PostgreSQL-Datenbanken synthetische Daten zu generieren. Weiterhin gibt es viele denkbare Erweiterungen, um die Funktionalität des Tools zu erhöhen sowie mehr Datentypen zu unterstützen. Als Beispiel wäre eine Unterstützung von zusammengesetzten Primärschlüssel denkbar.

Zudem hat es sich gezeigt, dass die konfigurierbare Auswahl der Generatoren interessante neue Möglichkeiten eröffnet. Dieses Konzept könnte noch weiterverfolgt werden. So wäre es denkbar, dass spezifische Generatoren nicht nur über die Konfiguration ausgewählt, sondern auch gleich noch parametrisiert werden. Zum Beispiel wäre eine sprachregionsspezifische Parametrisierung für Personen- und Strassennamen interessant.

Danksagung

Wir möchten allen Unterstützern dieser Arbeit unseren Dank aussprechen.

Zuerst möchten wir uns bei unserem Experten Prof. Stefan Keller für die spannende und herausfordernde Arbeit sowie seine wertvollen Beiträge und Ideen bedanken.

Ein spezieller Dank gilt unserem Betreuer Nicola Jordan, der uns über die gesamte Dauer der Arbeit begleitet und mit hilfreichen Ratschlägen unterstützt hat.

Zusätzlich bedanken wir uns bei Labian Gashi, Kevin Ammann und Etienne Baumgartner für ihre Vorarbeiten, auf denen wir aufbauen konnten.

Zum Schluss gilt unseren Dank auch unseren Familien, Partnern und Freunden für die positive Unterstützung und Geduld.

Inhaltsverzeichnis

Abstract **i**

Management Summary **iv**

Danksagung **v**

1 Einleitung **1**

1.1 Ausgangslage 2

1.1.1 Problemstellung 2

1.1.2 Vorarbeiten 2

1.1.3 Synthetische Daten 3

1.1.4 Stand der Technik 5

1.2 Ziel der Arbeit 5

1.3 Rahmenbedingungen 6

1.4 Aufbau der Arbeit 6

2 Analyse **8**

2.1 Systemkontext 9

2.1.1 Technologische Vorgaben 9

2.1.2 Das Tool pgsynthdata 10

2.1.3 Vorarbeit 11

2.1.4 PostgreSQL Statistiken und Informationsschemas 12

2.1.5 Vergleichbare Tools 13

2.2 Requirements 14

2.2.1 Nichtfunktionale Anforderungen 15

2.2.2 Funktionale Anforderungen 16

3 Design **18**

3.1 Architektur des Gesamtsystems 19
3.2 Prozess der Datengenerierung 20
3.2.1 Überblick und Ablauf 20
3.2.2 Erstellen der Zieldatenbank 21
3.2.3 Reihenfolge der Tabellengenerierung 22
3.3 Modellierung der Datenbankinformationen 23
3.3.1 Datenbank-Struktur 23
3.3.2 Statistische Informationen 24
3.4 Plugin-System für Generatoren 25
3.4.1 Überblick und Ablauf 25
3.4.2 Generatorspezifische Abfragen 26
3.5 Array Generator 27
3.5.1 Struktur 27
3.5.2 Ablauf 28
3.5.3 Statistiken und Typinformationen 29
3.6 Konfigurierbare Wahl von spezifischen Generatoren 30
3.6.1 Konfigurations-Konzept 30
3.6.2 Eigennamen-Generator 31
3.6.3 Adressen-Generator 32
3.6.4 Copy- und Ignore-Generator 32
3.7 Spatial Generator 32
3.7.1 Registierung im Plugin-System 33
3.7.2 Verwendete Statistiken 33
3.7.3 Unterstützte Datentypen 33
3.8 Enumerated Generator 33
3.9 Kommandozeilenschnittstelle 35

4 Implementation **37**

4.1 Plugin-System 38
4.1.1 Dynamischer Import der Generator-Module 38
4.1.2 Registrierungs-Dekorator 38

4.1.3 Auswahl des spezifischsten Generators 40
4.1.4 Generatoren-Signatur 42
4.2 Datenbankanbindung 42
4.2.1 Dynamische Insert Queries 42
4.2.2 Auslesen der Spaltenkommentare 44
4.3 Generatoren 44
4.3.1 Custom-Query für Nummeric Array Generatoren 44
4.3.2 Spezifische Generatoren mittels Faker 45
4.4 Verwendete Technologien 45
4.4.1 Python-Version 45
4.4.2 Entwicklungstools 46
4.4.3 Externe Libraries 47
4.5 Testing 48
4.5.1 Automatisierte Unit-Tests 48
4.5.2 Manuelle Systemtests 49

5 Resultate **50**
5.1 Zielerreichung 51
5.1.1 Architektur und Refactoring 51
5.1.2 Plugin-System 52
5.1.3 Array Generator 52
5.1.4 Spezifische Generatoren 53
5.1.5 Spatial Generator 54
5.1.6 Enumerated Generator 55
5.1.7 Usability 55
5.1.8 Performance und Skalierbarkeit 56
5.1.9 Testabdeckung 56
5.2 Einschränkungen 56
5.2.1 Voraussetzungen 56
5.2.2 Allgemein unterstützte Datentypen 56
5.2.3 Datentypen für Fremdschlüssel 57
5.3 Ausblick 57

5.3.1 Self-Referencing . . . 57
5.3.2 Zusammengesetzte Unique- und Primary-Constraints . . . 57
5.3.3 Datenbanken mit mehreren Schemata . . . 58
5.3.4 Parametrisierung für spezifische Generatoren . . . 58
5.3.5 Multiplication-Factor . . . 58
5.3.6 Performance-Verbesserungen . . . 58
5.3.7 Integration von Machine-Learning . . . 59

6 Schlussfolgerung 61

7 Projektmanagement 63
7.1 Entwicklungsprozess . . . 64
7.1.1 Methodik . . . 64
7.1.2 Dokumentation von Design-Entscheidungen . . . 64
7.1.3 Workflow . . . 65
7.2 Qualitätssicherung . . . 65
7.2.1 Code Reviews . . . 65
7.2.2 Automatisierte Unit-Tests . . . 66
7.2.3 Testabdeckung . . . 66
7.2.4 Code-Formatierungsrichtlinien . . . 66
7.2.5 Statische Code-Analysen . . . 66
7.2.6 Manuelle Systemtests . . . 67
7.3 Projektplanung und Meilensteine . . . 67
7.3.1 Einarbeitung (M0) . . . 67
7.3.2 Analyse & Design (M1) . . . 68
7.3.3 Einreichung Aufgabenstellung (M2) . . . 68
7.3.4 Zwischenpräsentation (M3) . . . 68
7.3.5 Implementation (M4) . . . 68
7.3.6 Testing & Release (M5) . . . 69
7.3.7 Finalisierung & Abgabe (M6) . . . 69
7.4 Risikoanalyse . . . 69
7.4.1 Risiken . . . 69
7.4.2 Mitigation . . . 70

7.4.3 Eingetretene Risiken 71
7.5 Zeiterfassung 71
7.5.1 Zeitauswertung pro Person 71
7.5.2 Zeitauswertung pro Kategorie 72
7.6 Sitzungen 72

A Aufgabenstellung **73**

B Installationshinweise und Benutzerhandbuch **75**

C Hinweise zu CI/CD **86**

D Codemetriken **88**

E Manuelle Systemtests **89**

F Ausdruck Analyze Kommando **93**

G Ausdruck Show Kommando **100**

H Testabdeckung **105**

I Performance Test **107**
I.1 Ausgangslage 107
I.2 Resultate 108
I.3 Raw-Output 109

Literaturverzeichnis **112**

Abbildungsverzeichnis **114**

Tabellenverzeichnis **116**

Listings **117**

Anmerkungen **118**

Glossar

Black-Box-Tests Black-Box-Testing bezeichnet eine Methode für Software-Tests, bei der die Tests ohne Kenntnisse über die innere Funktionsweise der Software durchgeführt werden. Das Ziel ist es, die Übereinstimmung einer Software mit ihrer Spezifikation zu überprüfen. Im Gegensatz dazu stehen White-Box-Tests, die mit Bezug auf die Implementierung entwickelt werden. x, 49

Dependency-Inversion-Principle Ein Software-Entwurfsprinzip, demzufolge Abhängigkeiten nur in einer Richtung bestehen sollten, und zwar von den höheren zu den niedrigeren Ebenen. Dies verhindert eine starke Kopplung und zyklische Abhängigkeiten. x, 15

DSGVO Die Datenschutz-Grundverordnung ist eine seit 2018 in Kraft stehende Verordnung der Europäischen Union, welche die Regeln zur Verarbeitung personenbezogener Daten vereinheitlicht. Ziel ist es, den Schutz personenbezogener Daten innerhalb der EU sicherzustellen, sowie der freie Datenverkehr innerhalb des europäischen Binnenmarktes zu gewährleisten. x, 5

GitLab GitLab ist eine Webanwendung zur Versionsverwaltung von Softwareprojekten. Es bietet unter anderen ein Issue-Tracking-System mit Kanban-Board und ein System für Continuous Integration (CI). x, 9, 64, 65

Low-Coupling/High-Cohesion Bei diesen Software Design Prinzip werden nur zusammenhängende Bereiche zu Modulen zusammengefasst (High Coehesion). Dadurch entstehen zwischen den Modulen sehr geringe Abhängigkeiten (Low Coupling). Dies erhöht Testbarkeit, Wartbarkeit und Austauschbarkeit der Module. x, 11, 15, 51

Open-Close-Principle Ein Prinzip beim Softwaredesign. Ein gutes Design sollte offen für Erweiterungen, aber geschlossen für Änderungen sein. Das bedeutet, dass die Funktionalität erweitert werden kann, ohne dass der bestehende Code angepasst werden muss.. x, 15, 52

OpenStreetMap Ein open-source Projekt, welches frei nutzbare Geodaten sammelt. Kern des Projekts ist eine offen zugängliche Datenbank mit Geoinformationen. Aus diesen Daten können sowohl freie als auch kommerzielle Landkarten erstellt werden. x, 32

pgAdmin pgAdmin ist eine Open-Source-Software zur Entwicklung und Administration von PostgreSQL-Datenbanken. Seine graphische Benutzeroberfläche erleichtert die Administration. x, 31

PIP requirement specifiers Ein Funktion von PIP (Package Installer for Python) welches die Requirements in einem File `requirements.txt` bereit stellt. x, 12

PostGIS PostGIS ist eine Datenbankerweiterung für PostgreSQL. Es fügt Unterstützung für geografische Objekte hinzu. i, iii, x, 16, 32, 33, 52, 54, 61

PostgreSQL PostgreSQL ist ein freies, objektrelationales Datenbankmanagementsystem. Wird oft kurz Postgres genannt. i, ii, iv, x, 2, 3, 5, 6, 9, 10, 12, 13, 16, 21, 24, 26, 29, 30, 32–34, 44, 47, 56, 86

Pseudonymisierung Bei der Pseudonymisierung werden Identifikationsmerkmal wie zum Beispiel der Name durch ein Pseudonym (zumeist ein Code, bestehend aus einer Buchstaben- oder Zahlenkombination) ersetzt, um die Feststellung der Identität des Betroffenen auszuschliessen oder wesentlich zu erschweren. x, 2

Python-Decorator Ein Python-Dekorator ist eine Funktion, die eine andere Funktion aufnimmt und das Verhalten der letzteren Funktion erweitert, ohne sie explizit zu verändern. x, 25, 38

pythonic Steht für *Idiomatic Python*. Beschreibt einen Ansatz, wie man in Python etwas programmiert. Es gibt jeweils viele Wege dasselbe zu erreichen, meistens gibt es einen bevorzugten Weg. Dieser wird pythonic genannt (Alexandru et al., 2018). x, 9, 15

Separation-of-Concerns Separation of Concerns ist eines der grundlegendsten Prinzipien in der Softwareentwicklung. Dabei solle der Code nach klaren Verantwortlichkeiten getrennt werden, sodass jeder Teil eine bestimmte Aufgabe erfüllt. x, 15, 51

Akronyme

AI Artificial Intelligence. x, 14
API Application Programming Interface. x, 14

CI Continuous Integration. x, 46, 48, 65
CLI Command Line Interface. x, 6, 16, 18–20, 25, 35, 36, 47, 49, 51, 55, 56, 58, 67, 117

FK Fremdschlüssel. x, 23, 59

GUI Graphical User Interface. x, 14, 31

IDE Integrated Development Environment. x, 23
IFS Institut für Software. x, 6, 69

ML Machine Learning. x, 3, 14, 17, 59

OST Ostschweizer Fachhochschule. ii, x, 6, 9

PEP Python Enhancement Proposals. x, 23, 45–47, 66

SDK Software Development Kit. x, 14
SQL Structured Query Language. x, 42, 44

UDT User-defined Type. x, 12, 16, 29, 34, 39, 52, 54, 55, 61

Kapitel 1

Einleitung

In diesem Kapitel sind die Ausgangslage, das Ziel der Arbeit und deren Rahmenbedingungen aufgezeigt. Der letzte Abschnitt gibt einen Überblick über die Struktur der folgenden Kapitel.

1.1 Ausgangslage

Die Ausgangslage beschreibt die angetroffene Problemstellung, die Vorarbeiten und der aktuelle Stand der Technik. Zudem gibt es eine Einführung in das Themenfeld der synthetischen Daten.

1.1.1 Problemstellung

In den Bereichen Software- und Data-Engineering sowie Machine-Learning besteht eine grosse Nachfrage nach umfangreichen Datensätzen. Die Verwendung von Realdaten ist oft aus Datenschutzgründen nicht möglich. Es muss eine Anonymisierung oder Pseudonymisierung durchgeführt werden, was aufwändig und zeitintensiv ist und je nach dem keinen vollständigen Schutz bietet oder die Eigenschaften der Daten zu stark verändert. Die Generierung von synthetischen Daten löst dieses Problem.

Das Kommandozeilen-Tool `pgsynthdata` soll in der Lage sein, fast beliebige PostgreSQL-Datenbanken zu synthetisieren und in eine generierte Datenbank mit gleicher Struktur und vergleichbaren statistischen Eigenschaften abzufüllen. In der Abbildung 1.1 ist die Anwendung des Programms zu sehen.

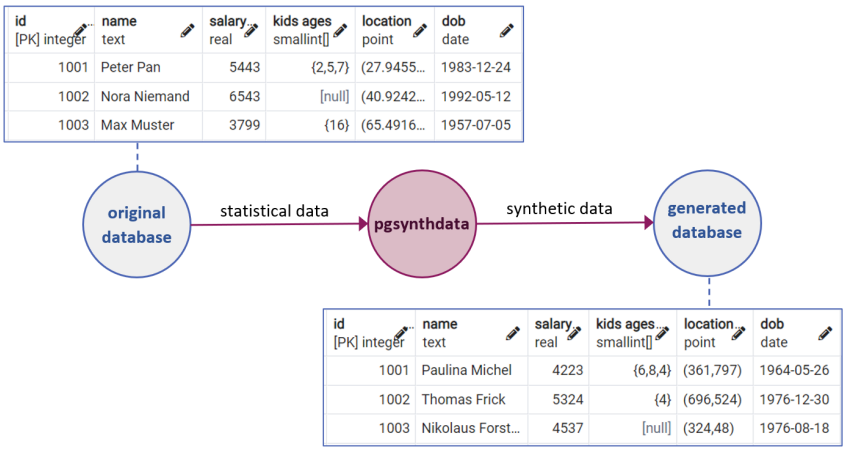


Abbildung 1.1: Anwendung von pgsynthdata

1.1.2 Vorarbeiten

Das bestehende Tool `pgsynthdata` wurde initial im Rahmen eines Datenbankseminars (Gashi, 2020) geschrieben und danach als Projektarbeit (Ammann, 2021) weiterentwickelt. Mit dem entwickelten Prototyp wurde das Konzept erarbeitet

und gezeigt, wie eine synthetische Datengenerierung mit statistisch ähnlichen Eigenschaften für PostgreSQL möglich ist. Dabei wurden gängige Basis-Datentypen wie Zahlen, Text oder Daten unterstützt. Eine detaillierte Analyse der Vorarbeiten ist im Unterabschnitt 2.1.3 zu finden.

1.1.3 Synthetische Daten

Um den Prozess der synthetischen Datengenerierung besser zu verstehen, folgen nach der Definition des Begriffs einige Anwendungsmöglichkeiten und Überlegungen zum Datenschutz bei dem Verwenden von synthetischen Daten.

Definition

Auf konzeptioneller Ebene handelt es sich bei synthetischen Daten nicht um echte Daten, sondern um Daten, die aus echten Daten generiert wurden und dieselben statistischen Eigenschaften wie die echten Daten aufweisen. Das bedeutet, dass ein Analytiker, der mit einem synthetischen Datensatz arbeitet, ähnliche Analyseergebnisse erhalten sollte, wie er sie mit echten Daten erhalten würde (Emam et al., 2020, p.8).

Die Abbildung 1.2 visualisiert den Prozess, wie aus den Originaldaten zuerst ein statistisches Modell abgeleitet wird, aus welchem die statistisch ähnlichen synthetischen Daten generiert werden können.

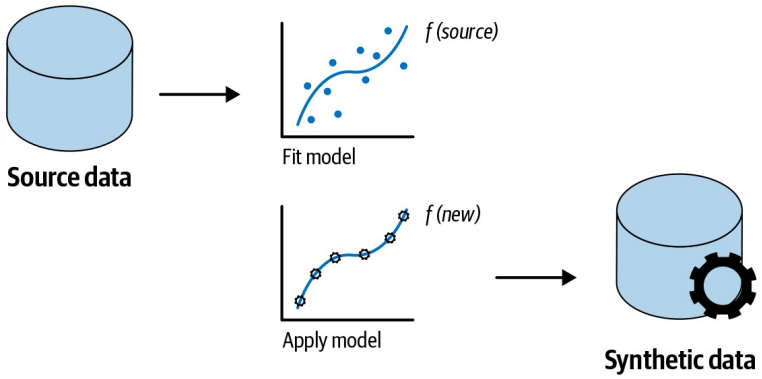


Abbildung 1.2: Synthetische Datengenerierung aus Originaldaten
mit freundlicher Genehmigung (Emam et al., 2020)

Anwendungsbereiche

Mögliche Anwendungsfälle von synthetischen Daten sind unter anderem das Generieren von Trainingsdaten für Machine Learning (ML), Testdaten für Stress- oder Robustheitstests (Fuzzing) im Software-Testing. Generell eignet sich jede

Anwendung, bei denen möglichst viele und originalgetreue Daten benötigt werden, die Originaldaten aber wegen Datenschutz-, Vertraulichkeits- oder sonstigen Gründen nicht verwendet werden können.

Datenschutzbedenken

Die Analyse der Risiken für die Privatsphäre bei synthetischen Daten ist ein wichtiges Thema. Allgemein wird angenommen, dass synthetische Daten ein vernachlässigbares Risiko für die Privatsphäre darstellen, da es keine eindeutige Zuordnung zwischen den Datensätzen in den synthetischen Daten und den Datensätzen in den Originaldaten gibt. In der Praxis ist es jedoch möglich, bei der Generierung synthetischer Daten das Synthesemodell zu stark an die realen Daten anzupassen. Das bedeutet, dass die generierten Daten den Originaldaten sehr ähnlich sind, was zu Datenschutzproblemen führt, da ein generierter Datensatz einer Person in der realen Welt zugeordnet werden könnte. (Emam et al., 2020, p.137)

Zum Beispiel kann auch aus einer synthetisch generierten Lohntabelle, welche jedoch ähnliche statistische Eigenschaften wie das Original ausweist, der maximale Lohn ausgelesenen werden. Kann nun aus den weiteren Tabellen die ursprüngliche Firma ermittelt werden, so hat man mit ziemlich grosser Sicherheit den Lohn des CEO's herausgefunden. Je nach Unternehmen stellt dies eine sensible Information dar.

Somit gibt es zwingend einen Interessenkonflikt zwischen möglichst exakten Daten (möglichst gleich wie Originaldaten) und möglichst randomisierte Daten, die keine Rückschlüsse ermöglichen. Man muss zwangsweise einen Kompromiss finden, wo die generierten Daten statistisch genügend gut am Original sind, jedoch genügend verändert wurden, um keine Rückschlüsse zu ermöglichen. Wie das Beispiel oben zeigt, ist diese Grenze individuell und je nach Kontext verschieden. Eine Beurteilung wo diese Grenze liegt ist daher zwingend vorzunehmen bevor man synthetische Daten generiert oder diese veröffentlicht. Die Abbildung 1.3 zeigt eine solche Kompromissfindung.

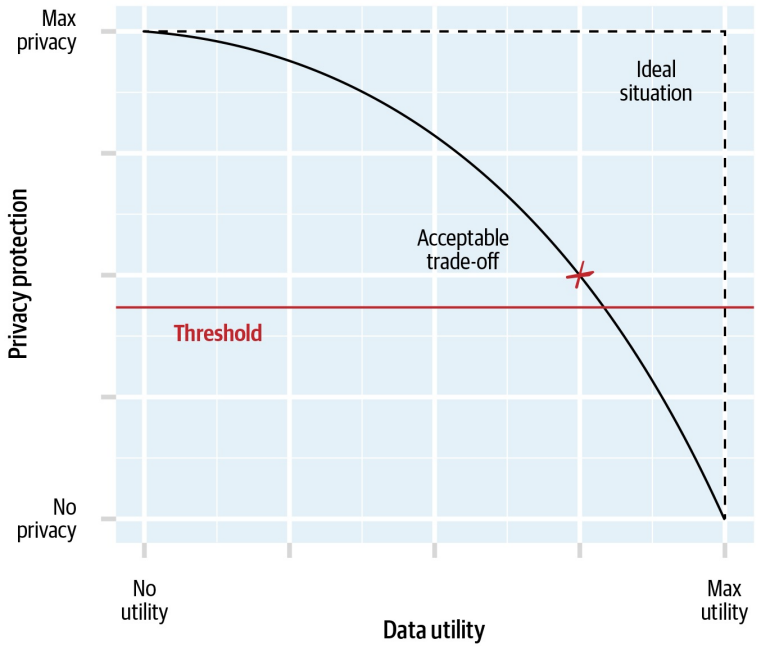


Abbildung 1.3: Kompromissfindung zwischen guten synthetischen Daten und Datenschutz

mit freundlicher Genehmigung (Emam et al., 2020)

1.1.4 Stand der Technik

Der steigende Bedarf an grossen Datensätzen im Bereich Machine-Learning und Data-Engineering führt auch zu einer verstärkten Forschung in diesem Bereich (Muppidi, 2020). Dazu beigetragen haben auch gesetzliche Regelungen im Bereich des Datenschutzes wie die europäische DSGVO. Dies hat zu einer breiten Palette verschiedener Werkzeuge zur Synthese oder Anonymisierung von Originaldaten geführt. Im Unterabschnitt 2.1.5 ist eine Auswahl solcher Werkzeuge zu finden.

1.2 Ziel der Arbeit

Ziel der Arbeit ist den bestehenden Prototyp (Unterabschnitt 1.1.2) in einen wartbaren, erweiterbaren und einfacher nutzbaren Zustand zu bringen. Alle bestehenden Generatoren sollen unverändert übernommen werden. Neue Generatoren für bisher noch nicht unterstützte Datentypen sollen die modulare Erweiterbarkeit demonstrieren und zur Erweiterung beitragen. Dabei sollen wie bisher möglichst die Statistiken von PostgreSQL für die Generierung verwendet werden.

Das Tool soll als Alleinstellungsmerkmal weiterhin nur ein Minimum an Konfiguration benötigen und über ein Command Line Interface (CLI) angesprochen werden. Die detaillierte Aufgabenstellung befindet sich im Anhang A.

1.3 Rahmenbedingungen

Die folgende Arbeit wurde im Rahmen einer Studienarbeit an der OST im Auftrag des Institut für Software (IFS) durchgeführt. Die Studienarbeit wird jedem Student mit 8 ECTS vergütet, was einen Zeitrahmen von je 240 Stunden entspricht. Die Arbeit fand im Zeitraum vom 20. September bis 24. Dezember 2021 statt.

1.4 Aufbau der Arbeit

Die nachfolgende Zusammenfassung gibt einen Überblick über den Zweck und Inhalt der kommenden Kapitel:

Kapitel 2 Analyse beinhaltet die Analyse der Problemstellung. Dabei wird ein Fokus auf den Systemkontext, Statistiken von PostgreSQL und der vorhandenen Vorarbeiten gelegt. Zudem sind die funktionalen und nicht funktionalen Anforderungen an die Applikation beschrieben.

Kapitel 3 Design beschreibt zum einen generell die Architektur und den Ablauf des Programms. Zum anderen wird auch genau beschrieben, wie die Anforderungen umgesetzt werden können, welche Statistischen Daten notwendig sind oder wie Informationen im Tool modelliert sind. Wichtige Entscheide sind jeweils speziell begründet.

Kapitel 4 Implementation demonstriert, wie wichtige Teile der Applikation implementiert sind. Es ist auch beschrieben, welche Anforderungen an Generatoren gestellt werden. Zudem werden verwendete Technologien und Tools dokumentiert. Zum Schluss wird auf die verschiedenen verwendeten Test-Arten eingegangen.

Kapitel 5 Resultate fasst alles zusammen, was in dieser Arbeit erreicht wurde. Darauf basieren wird ein Ausblick gegeben. Dieser beschreibt, welche Probleme noch ungelöst sind und wie das Programm erweitert werden könnte.

Kapitel 6 Schlussfolgerung reflektiert das Erreichte und gibt eine Interpretation der Resultate.

Kapitel 7 Projekt Management schildert das Vorgehen, die Verantwortlichkeiten und den Ablauf des Projektes. Der Zeitplan ist durch ein Gantt-Projektplan dargestellt und die Meilensteine werden genauer beschrieben. Zudem sind die Risikoanalysen und Zeitauswertungen hier untergebracht.

Kapitel 2

Analyse

In diesem Kapitel wird analysiert, in welchem Kontext sich die Applikation befindet. Aus den verschiedenen Betrachtungen in Kombination mit der Aufgabenstellung, ergeben sich die funktionalen und nicht-funktionalen Anforderungen für diese Arbeit.

2.1 Systemkontext

Der Systemkontext beschreibt die Umgebung, in der sich das System befindet und von dem es beeinflusst wird. Zuerst werden die vorgegebenen Technologien, das Tool und die Vorarbeiten analysiert. Danach folgen die Metainformation, welche PostgreSQL zur Verfügung stellt und die für die Generierung von synthetischen Daten verwendet werden. Eine Analyse des Marktumfeldes, zeigt welche vergleichbare Tools existieren.

2.1.1 Technologische Vorgaben

Ausgehend von den Vorarbeiten und der Aufgabenstellung sind folgende Technologien festgelegt.

PostgreSQL

PostgreSQL[1] ist ein freies, objektrelationales Datenbankmanagementsystem. Es unterstützt viele verschiedene Datentypen und läuft auf allen gängigen Betriebssystemen und den wichtigsten Linux Distributionen. Ein wichtiger Punkt für die Erstellung von synthetisierten Daten sind statistische Informationen. PostgreSQL macht es sehr einfach, viele Statistiken über die Daten aus dem Datenbanksystem auszulesen. PostgreSQL bewirtschaftet diese Statistiken, um einen optimalen Ausführungsplan von Abfragen erstellen zu können.

Python

Die interpretierte Programmiersprache Python[2] bietet viele Vorteile und ist auf Grund ihrer Flexibilität in verschiedenen Anwendungsgebieten weit verbreitet (Lutz, 2013). Sie ist mit den gängigen Betriebssystemen kompatibel und es existieren zahlreiche, frei verfügbaren, Libraries und Tools, welche die Entwicklung immens erleichtern und beschleunigen. Ein weiterer Pluspunkt ist die leicht zu erlernende Syntax, sowie die grosse und aktive Community. In der Python-Community wird darauf geachtet, dass der geschrieben Code pythonic ist, das bedeutet, er ist kompakt, aussagekräftig und leicht lesbar (Alexandru et al., 2018).

Infrastruktur

Für die gesamte Entwicklungszeit stand die GitLab-Instanz der OST[3] zur Verfügung. Der finale Stand ist auf einem öffentlichen GitLab-Repository[4] publiziert.

Lizenz

Das Tool steht unter einer open-source MIT-Lizenz und ist offen für Verbesserungen und Vorschläge für neue Funktionen.

2.1.2 Das Tool pgsynthdata

Das Kommandozeilen-Tool `pgsynthdata` erzeugt synthetische Daten ausgehend von einer PostgreSQL-Datenbank und füllt diese in eine generierte Datenbank mit gleicher Struktur ab. Als Grundlage dienen die Kataloge von PostgreSQL für funktionale Abhängigkeiten sowie Statistiken, die PostgreSQL zum Zwecke der Anfrageoptimierung erstellt (Unterabschnitt 2.1.4). Diese Informationen werden den Datengeneratoren übergeben, welche daraus statistische Modelle für die synthetische Datengenerierung erstellen können. Die Generatoren sollen einfach ausgewechselt und erweitert werden können. Ein Alleinstellungsmerkmal des Tools ist, dass es ohne aufwändige und komplizierte Konfiguration auskommt. Der Kontext dieses Prozesses ist in Abbildung 2.1 ersichtlich.

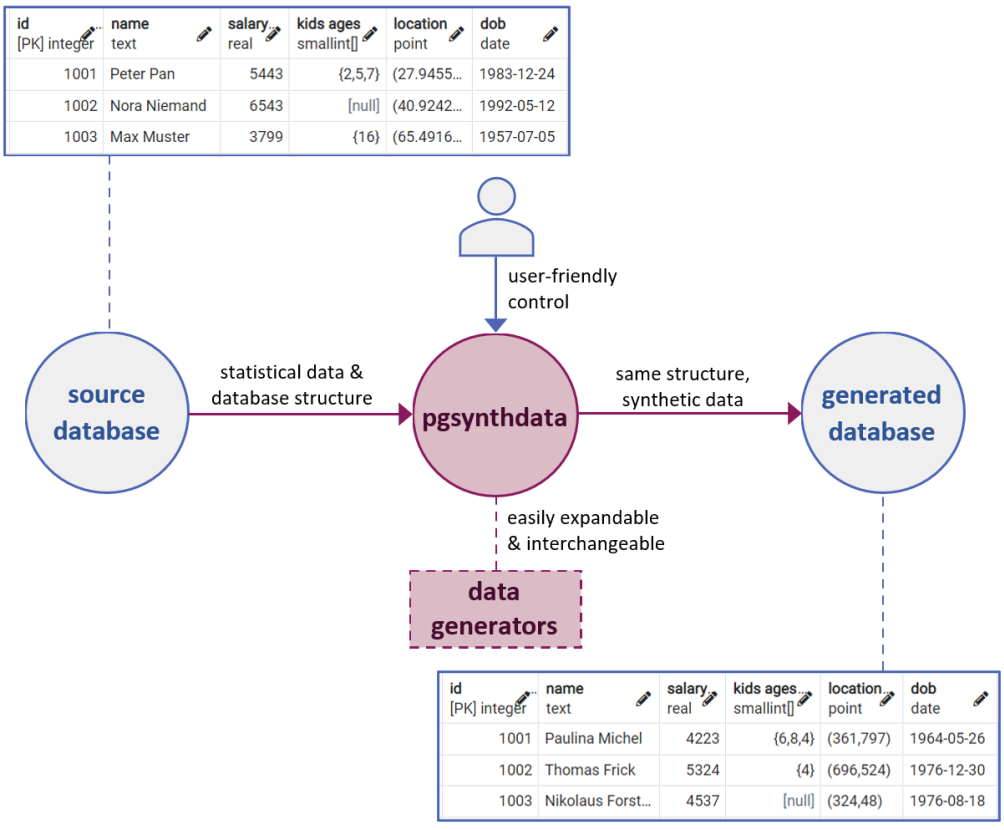


Abbildung 2.1: Kontext-Diagramm von pgsynthdata

2.1.3 Vorarbeit

Beim Analysieren des Prototyps aus der Vorarbeit sind uns einige Punkte bezüglich Software-Engineering aufgefallen, die verbessert werden könnten. Die gefunden Aspekte dienen unter anderem als Motivation für diese Arbeit und wiederspiegeln sich auch in den Anforderungen.

Wartbarkeit und Testbarkeit

Die bestehende Struktur weist hohe Abhängigkeiten unter den Modulen auf. Zudem sind die Verantwortlichkeiten nicht sauber geregelt und Codeteile zur Datengenerierung über mehrere Module verteilt. Dies verletzt das Designprinzip Low-Coupling/High-Cohesion und verhindert dadurch ein effektives Testing (Martin, 2009). Als Beispiel wird die Datenbank aus mehreren Modulen direkt angesprochen, sodass sich diese schwer simulieren lässt. Ausserdem erschweren die komplexen Verstrickungen unter den Modulen die Wartbarkeit und erhöhen die Einarbeitungszeit für neue Entwickler.

Erweiterbarkeit

Um einen Generator für einen neuen Datentyp zu implementieren, sind in mehreren Modulen an diversen Stellen im Code Anpassungen nötig. Das bedeutet, um den neuen Generator zu implementieren, muss der gesamte Code im Detail verstanden werden. Auch Teile die für den Generator eigentlich nicht relevant sind. Ausserdem erhöht es die Gefahr von Programmierfehler.

Performance

Die Laufzeit des Tools ist für die, in den Vorarbeiten verwendete, Testdatenbank (Tennis_ATP) relativ lange. Durch Optimierungen bei der Datengenerierung oder beim Abfüllen in die Datenbank, könnte die Laufzeit möglicherweise verbessert werden.

Ansteuerung der Generatoren

Die statistischen Daten und Spalteninformationen der Originaldatenbank sind gefiltert und stark auf die bisherigen Generatoren zugeschnitten. Für die Implementation von neuen Datentypen wäre es besser die Daten ungefiltert zu übergeben, da man nicht wissen kann, was ein neuer Generator benötigt. Zum Beispiel fehlen für die Implementierung eines Array-Generators zwingend benötigte Werte.

Dependency Management

Das Dependency Management wird aktuell auf Basis eines einfachen PIP requirement specifiers gemacht, zudem wird auf die ältere Python Version 3.7 gesetzt. Das Dependency Management soll durch ein state-of-the-art Tool mit mehr Funktionalität abgelöst werden. Es ist auch auf die bei Beginn der Arbeit aktuelle Python Version 3.9 zu setzten.

2.1.4 PostgreSQL Statistiken und Informationsschemas

In PostgreSQL gibt es mehrere interne Tabellen, die Metadaten über die Datenbanken enthalten. Die zwei für die Datengenerierung wichtigsten Tabellen werden hier analysiert und beschrieben.

Statistiken

Die Tabelle `pg_stats` enthält aufbereitete Statistiken für jede Spalte einer Tabelle. PostgreSQL benützt diese Daten unter anderen um Ausführungspläne für SQL-Queries zu optimieren (Schonig, 2017). Diese Statistiken dienen dem Generator als Ausgangslage für das statistische Modell. Um die Werte zu aktualisieren, muss vor der Abfrage der SQL-Befehl `vacuum analyze` ausgeführt werden. Ein Beispiel dieser Statistik ist in Abbildung 2.2 zu sehen. Weitere Informationen sind in der PostgreSQL Dokumentation[5] zu finden.

Data Output | Explain | Messages | Notifications

	attname name	null_frac real	n_distinct real	most_common_vals anyarray	most_common_freqs real[]	histogram_bounds anyarray
1	player_id	0	-1	[null]	[null]	{100001,100580,10...
2	last_name	0.0008	-0.5381339	{Kim,Lee,Smith,Rodri...	004,0.0004,0.0004,0.0004}	{"A Riches",Ahumad...
3	hand	0.104	3	{U,R,L}	34333,0.27746665,0.0251}	[null]
4	first_name	0.0034	-0.17479704	{David,Daniel,Michael,...	014666667,0.0014333334}	{"A Baisley",Ae,Alber...
5	country_...	0083333335	201	{USA,ESP,AUS,GER,GB...	666,0.0007,0.0007,0.0007}	{AFG,AHO,AHO,AND,...
6	birth_date	0.21013333	-0.24269176	{19830301,19910307,...	002,0.0002,0.0002,0.0002}	{186711,18900108,...

Abbildung 2.2: Wichtigste Werte der Tabelle pg_stats

Information-Schema Columns

Die Tabelle `information_schema.columns` enthält Informationen über die Spalten einer Tabelle. Dazu gehört der Datentyp und dessen Eigenschaften wie zum Beispiel die signifikanten Stellen einer Zahl oder der User-defined Type (UDT). Auf Grund dieser Informationen weiss der Datengenerator, welches Datenformat

er erzeugen muss. Ein Beispiel dieser Statistik ist in Abbildung 2.3 zu sehen. Weitere Informationen sind in der PostgreSQL Dokumentation[6] zu finden.

Data Output | Explain | Messages | Notifications

	column_name name	data_type character vary	is_nullable character vary	character_maximum_length integer	numeric_precision integer	numeric_precision_radix integer	datetime_precision integer	udt_name name
1	player_id	integer	NO	[null]	32	2	[null]	int4
2	first_name	character v...	YES	60	[null]	[null]	[null]	varchar
3	birth_date	integer	YES	[null]	32	2	[null]	int4

Abbildung 2.3: Wichtigste Werte der Tabelle information_schema.columns

2.1.5 Vergleichbare Tools

Folgende Tools verfolgen ein ähnliches Ziel wie `pgsynthdata`. Gemeinsam haben Alle, dass sie Originaldaten schützen wollen. Sie verfolgen dabei aber unterschiedliche Ansätze. Zum Beispiel gibt es Datenbank-Erweiterungen wie PostgreSQL Anonymizer und PostgreSQL Faker, welche direkt aus SQL angesprochen werden und open-source sind. Oder kommerzielle Tools wie MOSTLY AI und Replica Synthesis die in bestehende Umgebungen integriert werden können und stark mit Machine-Learning arbeiten. Der DataSyntesizer ist auch ein Python-Tool, welches jedoch nicht direkt mit der Datenbank kommuniziert, sondern einen manuellen Export und anschliessender Import seitens der Datenbank erfordert.

PostgreSQL Anonymizer

Dalibo Labs PostgreSQL Anonymizer[7] ist eine Datenbank Erweiterung, sodass der Anonymizer als Funktion innerhalb PostgreSQL zur Verfügung steht. Dies ermöglicht zum einen das permanente anonymisieren der Datenbank (destruktiv) und zum anderen eine dynamisches maskieren der Daten pro Abfrage. Zu diesem Zweck gibt es so genannte Masking-Funktions, welche komplett zufällige Daten erzeugen, Daten teilweise maskieren oder ihnen Rauschen hinzufügen. Konfiguriert wird das Tool grösstenteils über Security-Labels in der Datenbank.

Er verfolgt somit einen anderen, aber durchaus sehr spannenden Ansatz für die Anonymisierung der Daten. Der PostgreSQL Anonymizer ist aktuell in der Beta-Phase. Kann aber schon von jedem ausprobiert werden.

PostgreSQL Faker

PostgreSQL Faker[8] ist wie der PostgreSQL Anonymizer eine Datenbank-Erweiterung. Sie basiert auf der Python Faker Library[9] und erlaubt den Zugriff über SQL auf die Funktionen dieser Library. Auf diese Wiese ist es auch möglich, die Daten on-the-fly über Security-Labels und Maskierungsfunktionen zu verfremden.

Mostly AI

MOSTLY AI[10] ist nach eigenen Angaben die führende Plattform für Synthetische Daten. Es gibt eine gratis Version mit einem limitierten Funktionsumfang. Gesteuert wird das Tool über ein Webbasiertes Graphical User Interface (GUI). Es kann sich auch mit verschiedenen Datenbankmanagement Systemen verbinden. Vor der Generierung der Daten kann man überprüfen mit welcher Art die einzelnen Spalten der Tabellen generiert werden. Es ist auch möglich gleich einzugreifen und Änderungen an der Generierung vorzunehmen.

Die Daten werden mittels Artificial Intelligence (AI) analysiert. Aus dem resultierenden Modell werden die synthetischen Daten generiert. Nach der Generierung wertet das Tool die Genauigkeit der generierten Daten aus und prüft ob aus denen keine Rückschlüsse mehr auf die Originaldaten gemacht werden kann. Gemäss Platzer (2021) unterstützt MOSTLY AI seit kurzem auch geospatial Daten.

Replica Synthesis

Replica Syntesis[11] wird von einer Firma namens Replica Analytics entwickelt, welche sich zum Ziel gesetzt hat, Gesundheitsdaten weltweit für Sekundäranalysen zugänglich zu machen (Replica Analytics, 2021).

Die Software kann reale Daten mit Hilfe von ML analysieren und erstellt Modelle zur Erzeugung synthetischer Datensätze. Zudem gibt es Validierungs-Funktionen, die im Anschluss die Originaldaten mit den generierten Daten vergleichen. Die Software lässt sich parametrisieren und über eine Application Programming Interface (API) oder ein Software Development Kit (SDK) integrieren.

Data Synthesizer

DataSynthesizer[12] ist ein Python-Tool, um aus gegebenen Input Daten synthetische Daten zu generieren. Dafür werden im Tool statistische Informationen berechnet, um ein Modell für die synthetischen Daten zu generieren. Der DataSynthesizer liest die Daten über ein CSV-File ein und ist nicht in der Lage direkt mit einer Datenbank zu kommunizieren. (Ping, Stoyanovich & Howe, 2017)

2.2 Requirements

Ausgehend von der Aufgabenstellung und der Analyse der Vorarbeit wurden folgende Anforderungen an die Software definiert.

2.2.1 Nichtfunktionale Anforderungen

Die nicht-funktionalen Anforderungen stellen die Anforderungen an das Software-Design sicher und beheben die bei der Analyse der Vorarbeiten festgestellten Probleme.

Erweiterbarkeit

Datengeneratoren sollen über ein Plugin-System eingebunden werden. Neue Generatoren können so einfach und an einem Ort registriert werden. Dies sorgt dafür, dass das Programm erweiterte werden kann ohne das im Hauptprogramm Änderungen vorgenommen werden müssen. Es ist ein Open-Close-Principle anzustreben.

Um das Plugin-System aufzubauen ist ein Refactoring des gesamten Codes nötig. Bei diesem Refactoring soll darauf geachtet werden, dass Code möglichst pythonic geschrieben wird.

Wartbarkeit

Das Refactoring erlaubt es, die Programmstruktur von Grund auf neu zu definieren. Sie soll eine klare Aufteilung vermitteln und nach dem Design Prinzipen Separation-of-Concerns gestaltet sein. Dies wirkt sich positiv auf die Wartbarkeit und Testbarkeit der einzelnen Software-Teile aus (Martin, 2009).

Testbarkeit

Dank einer klaren Struktur (Dependency-Inversion-Principle) mit weniger Abhängikeiten (Low-Coupling/High-Cohesion) sollte es möglich sein, grosse Teile der Software mit Unit-Tests abzudecken und unabhängig von den anderen Teilen testen zu können. So können Fehler bei Änderungen und Erweiterungen schnell erkannt werden (Martin, 2009). Durch eine gute Testabdeckung wird auch die Erweiterbarkeit und Wartbarkeit vereinfacht.

Performance und Skalierbarkeit

Performance Messungen sollen Flaschenhälse aufdecken, um so begründete Optimierungen vorzuschlagen und wenn möglich gleich umzusetzen. Diese Überlegungen sollen die Skalierbarkeit sicherstellen. Die Performance Messungen werden nach der Implementation gemacht.

Usability

Die Usability soll durch eine Überarbeitung des CLI verbessert werden, um so die Optionen und Befehle einheitlicher zu gestalten. Warnungen und Fehlermeldungen sollen so aussagekräftig wie möglich sein, damit der Benutzer darauf reagieren kann.

Dem Benutzer soll vor dem Generieren der Datenbank die Möglichkeit gegeben werden, das Schema der Source-Datenbank anzuzeigen und die Auswahl der Generatoren zu überprüfen. Nach dem Generieren soll der Benutzer eine Analyse vornehmen können. Dabei werden die PostgreSQL Statistiken, der Original- und Synthetischen-Daten, sich gegenübergestellt.

2.2.2 Funktionale Anforderungen

Bei den funktionalen Anforderungen geht es hauptsächlich darum, mehr Datentypen zu unterstützen oder die Funktionalität anderweitig zu verbessern.

Array Datentypen unterstützen

Es soll mit einem Beispiel gezeigt werden, wie Array Datentypen im Plugin-System untergebracht werden können. Als Minimum müssen Numeric-Arrays erstellt werden. Je nach verfügbarer Zeit können auch weiter Datentypen implementiert werden. Um die Komplexität in Grenzen zu halten, soll der Array-Generator nur 1-dimensionale Arrays behandeln können.

Spatial Datentypen unterstützen

Die für PostgreSQL verfügbare Erweiterung PostGIS stellt erweiterte geometrische Datentypen und Funktionen zur Verfügung. Ein Prototyp soll zeigen, dass auch solche UDT's statistisch analysiert, generiert und abgefüllt werden können.

Konfigurierbare Wahl von spezifische Generatoren

Aus einem ursprünglichen Wunsch realistische Namen oder Postleitzahlen generieren zu können, ist die Anforderung für eine simple Konfigurationsmöglichkeit entstanden. Durch diese Konfiguration sollen für bestimmte Spalten explizit Generatoren ausgewählt werden können, welche spezifische Funktionen erfüllen, wie des Anfangs erwähnte Generierung von Personen- und Adressdaten. So soll es auch möglich sein, Spalten zu ignorieren oder eins-zu-eins zu kopieren. Die Konfiguration soll dabei möglichst einfach und nutzerfreundliche erfolgen.

Enumerated Datentypen unterstützen (optional)

Der Generator soll zufällig Einträge aus der Liste der möglichen Enum-Werte auslesen. So wird demonstriert, dass es über das Plugin-System möglich ist, den Wertebereich von Enum-Typen auszulesen.

Generatoren mit ML-Funktionalität (optional)

Sowohl bei der Selektion des am besten geeigneten Generator wie auch bei der Generierung von Daten bietet sich eine Integration von ML Unterstützung an. Es soll untersucht werden wie sich ML in die bestehende Struktur und das Plugin-System integrieren lässt und Vorschlage für dessen Umsetzung gemacht werden.

Kapitel 3

Design

In diesem Kapitel werden die Überlegungen und Entscheidungen zum Softwaredesign beschrieben. Die Grundlage dafür bilden die in der Analyse definierten Anforderungen. Zuerst kommt ein Überblick über das Gesamtsystem und das Refactoring, danach wird das Plugin-System beschrieben. Während in den letzten Abschnitten die Generatoren sowie das CLI behandelt werden.

Relevante Design-Entscheidungen sind gemäss der Y-Methode nach Zimmermann (2020) dokumentiert, welche im Kapitel Projektmanagement (Unterabschnitt 7.1.2) näher erläutert wird.

3.1 Architektur des Gesamtsystems

Ausgehend von dem in Abbildung 2.1 gezeigten Kontext-Diagramm, entstand das Komponenten-Diagramm in Abbildung 3.1. Das Ziel der neuen Architektur ist es, wenige Abhängigkeiten zwischen den Modulen zu erhalten und klare Verantwortlichkeiten den Modulen zuzuweisen. Beispielsweise kommuniziert nur ein Modul direkt mit der Datenbank. Die einzelnen Komponenten und deren Aufgabe werden im Anschluss näher beschrieben.

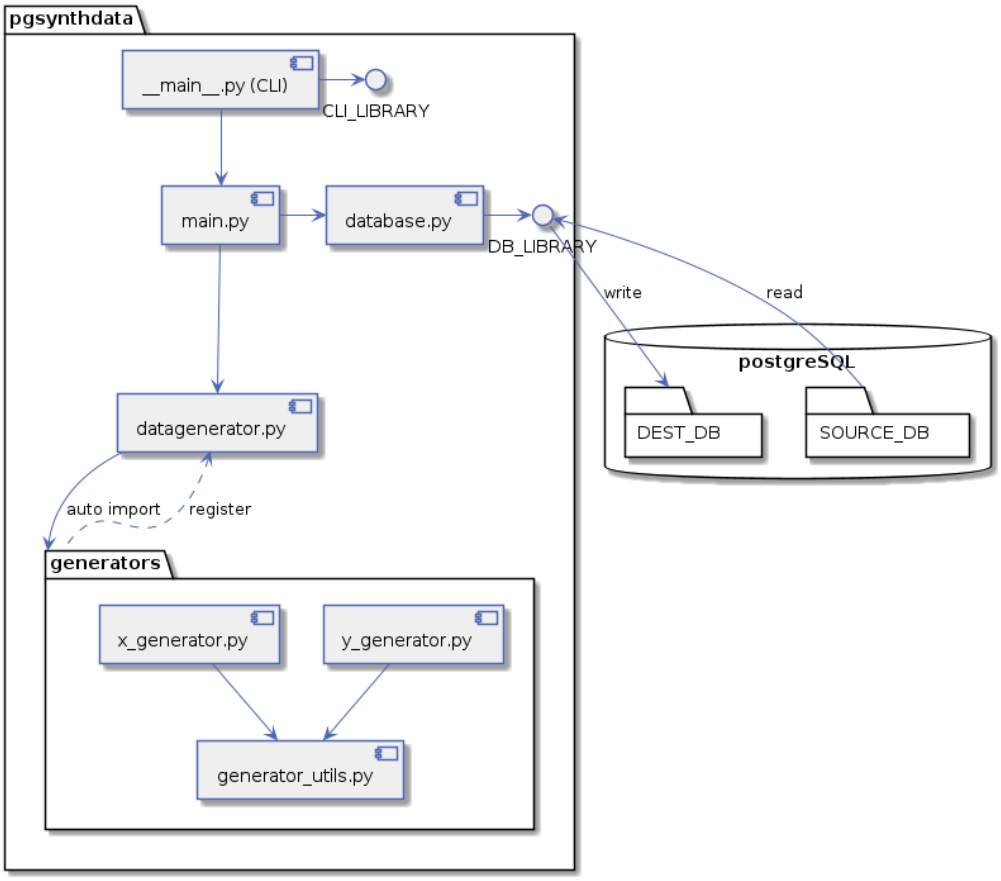


Abbildung 3.1: Komponentendiagramm

- **__main__**: Beinhaltet die gesamte CLI-Logik und somit die Interaktion mit dem Benutzer.
- **main**: In diesem Hauptmodul ist der Ablauf der Datengenerierung untergebracht. Es bietet die drei vorhandenen Befehle `generate`, `analyze` und `show` an und steuert den Datenfluss zwischen Datenbankmodul und den Generatoren.
- **database**: Alle Datenbank-Zugriffe werden über dieses Modul gemacht. Wie das Erstellen der neuen Datenbank, das Kopieren des Schemas oder

das abfüllen der generierten Daten.
- **datagenerator**: Dies Module beinhalte das Plugin-Systems. Es bietet den Generatoren die Möglichkeit sich zu registrieren. Zudem ist es in der Lage den passendsten Generator für eine Datenbank-Spalte zu finden.
- **..._generator**: Die einzelnen Generatoren beinhalten jeweils mindestens eine Funktion, welche als Generator verwendet werden kann sich beim `datagenerator` registriert.
- **generator_utils**: Dieses Modul beinhaltet wichtige Funktionen, welche von mehreren Generatoren verwendet werden können.

3.2 Prozess der Datengenerierung

Im ersten Abschnitt wird der Prozess der Datengenerierung erläutert und nachfolgend auf zwei spezielle Aspekte im Detail eingegangen.

3.2.1 Überblick und Ablauf

Der in Abbildung 3.2 ersichtliche Programmablauf zeigt den kompletten Prozess der Datengenerierung.

Zuerst wird die Ziel-Datenbank erstellt und das gesamte Schema übernommen (Unterabschnitt 3.2.2). Danach wird aus der Quelldatenbank die Struktur mit allen Tabellen- und den Spalteninformationen gelesen.

Aufgrund der Eigenschaften jeder Spalte kann nun der spezifischste Generator ermittelt werden. Wenn klar ist, welcher Generator verwendet werden soll, werden die zur Verfügung stehenden Statistiken geladen und allenfalls generatorspezifische Abfragen ausgeführt.

Wegen möglichen Fremdschlüssel-Abhängigkeiten müssen die Tabellen in der korrekten Reihenfolge generiert werden. Dafür werden diese Relationen aus der Datenbank gelesen und die Tabellen nach Abhängigkeiten topologisch sortiert (Unterabschnitt 3.2.3). Falls der Benutzer in den CLI-Argumenten gewisse Tabellen ausgeschlossen hat, werden diese im selben Schritt gefiltert. Falls eine zyklische Abhängigkeit oder Abhängigkeiten auf gefilterte Tabellen entdeckt werden, wird eine Fehlermeldung angezeigt und der Prozess abgebrochen.

Nach der topologischen Sortierung können die synthetischen Daten in der ermittelten Reihenfolge generiert und in die Zieldatenbank abgefüllt werden.

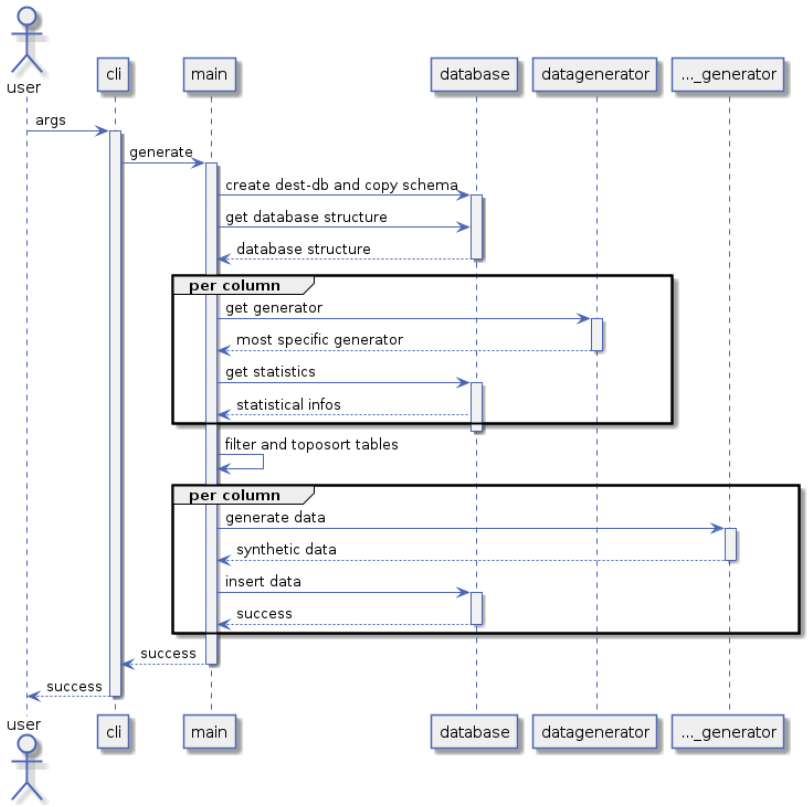


Abbildung 3.2: Ablauf der Generierung einer synthetischen Datenbank

3.2.2 Erstellen der Zieldatenbank

Das Erstellen der Zieldatenbank wird über ein SQL-Befehl erledigt. Für das Kopieren des Schemas kommen zwei PostgreSQL Client Applications[13] zum Einsatz. Diese sind `pg_dump`, um das Schema der Source-Datenbank auszulesen und `pg_restore`, um das ausgelesene Schema in die Destination-Datenbank einzulesen. Der grobe Ablauf des Erstellen der Datenbank ist in Abbildung 3.3 ersichtlich.

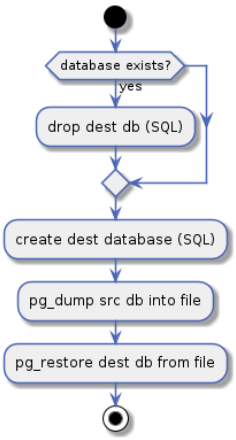


Abbildung 3.3: Ablauf beim Kopieren des Datenbankschema

3.2.3 Reihenfolge der Tabellengenerierung

Um die Schlüssel-Abhängigkeiten zwischen den Tabellen zu berücksichtigen, müssen diese vor dem Generieren und Einfügen der Daten topologisch sortiert werden. Das Ziel dieser Sortierung ist es, eine Reihenfolge der Tabellen in Bezug auf ihre gegenseitige Abhängigkeit zu erhalten. Alle Tabellen, welche sich auf derselben Ebene befinden haben keine Abhängigkeiten untereinander und könnten sogar parallel erstellt werden. Falls zyklische Abhängigkeiten entdeckt werden, wird ein Fehler angezeigt.

Abbildung 3.4 zeigt ein Beispiel einer solchen Topologie auf und Listing 3.1 beschreibt den Output, der erwartet wird.

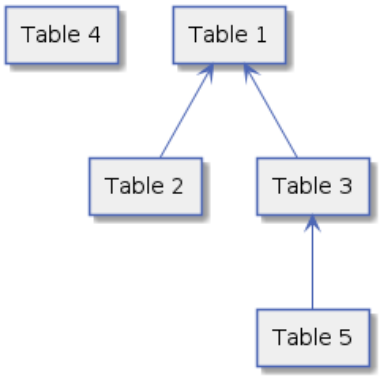


Abbildung 3.4: Beispiel Tabellen Topologie

```
1 [
2     ["Table 1", "Table 4"],
3     ["Table 2", "Table 3"],
4     ["Table 5"],
5 ]
```

Listing 3.1: Beispiel Tabellen-Topologie als Liste von Listen

3.3 Modellierung der Datenbankinformationen

Informationen, die aus der Datenbank gelesen werden, wie die Datenbankstruktur oder statistische Informationen müssen in eine Python-Struktur überführt werden. Nachfolgende Entscheidungen haben dazu geführt, dass dazu Python-Datenklassen verwendet werden. Die daraus entstandenen Strukturen werden in den nachfolgenden Abschnitten beschrieben.

Im Zusammenhang mit der Modellierung der Datenbank-Informationen, mit der Anforderung der Erweiterbarkeit, der Wartbarkeit und der Testbarkeit, entschieden wir uns für den Einsatz von Python Datenklassen und verwerfen den Einsatz von direkten SQL-Resultsets. So erreichen wir die Möglichkeit das Programm statisch auf Typenkorrektheit zu prüfen und ermöglichen die Unterstützung der Code-Vervollständigung durch das Integrated Development Environment (IDE). Dabei wird in Kauf genommen, dass mehr Code geschrieben werden muss und die Datenklassen bei neuen Informationen aus der Datenbank erweitert werden müssen.

Alle Statement-Resultate, welche nicht direkt in Datenklassen abgelegt werden können, sind vom Typ `Dict` auf Stufe Row.

Im Zusammenhang mit der Rückgabe von SQL-Resultsets, mit der Anforderung der einfachen Verwendbarkeit mit dem Zugriff auf die Spalten per Spaltenname, entschieden wir uns für den Einsatz von der `psycopg2.extras.DictCursor`-Factory für die Erstellung der Resultsets und verwerfen die Alternativen des positionellen Zugriffs oder dem Einsatz des `RealDictCursor`-Factory oder `NamedTupleCursor`-Factory. So erreichen wir einen Zugriff auf die Resultate per Spaltenname und die Möglichkeit eine Row direkt in ein Datenklassen-Konstruktor zu entpacken (`Class(**row)`). Dabei wird in Kauf genommen, dass eine `pgsycpg2` spezifische Funktionalität verwendet wird, welche nicht im Python Enhancement Proposals (PEP) 249 beschrieben ist.

3.3.1 Datenbank-Struktur

Es wird eine Datenstruktur benötigt, aus der pro Datenbank-Spalte ein Generator ausgewählt werden kann. Zudem muss man wissen welche Tabellen es gibt und welche Spalten zur Tabelle gehören. Diese Information ist zum einen dafür notwendig, um Fremdschlüssel (FK)-Beziehungen zu generieren, welche die Referenzielle Integrität beachten und zum andern wird sie verwendet um dynamische Insert-SQL-Queries zu generieren (Unterabschnitt 4.2.1). Abbildung 3.5 zeigt auf, wie diese Informationen in der Software abgebildet werden.

Damit die Tabellen in der korrekten Reihenfolge generiert werden, um Beziehungen korrekt herzustellen, werden zusätzlich zu den in der Abbildung 3.5 aufgeli-

steten Informationen noch spezifische Beziehungs-Informationen aus der Datenbank gelesen. Diese ermöglichen eine topologische Sortierung der Tabellen, um die Daten in der korrekten Reihenfolge zu generieren.

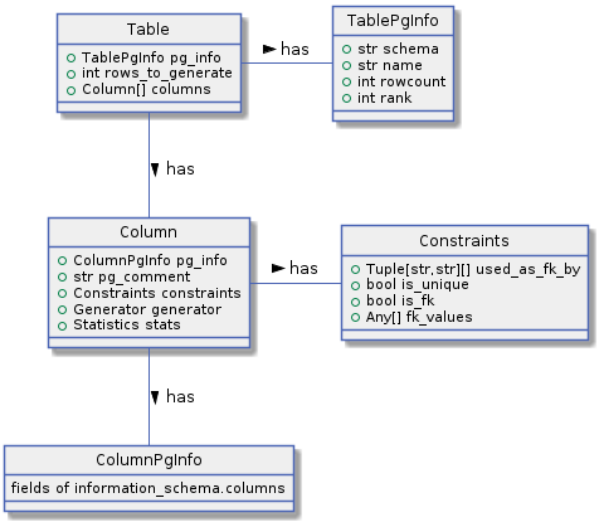


Abbildung 3.5: Datenklassen Datenbankinformationen

3.3.2 Statistische Informationen

Die Abbildung 3.6 zeigt, welche statistischen Informationen aus der Datenbank ausgelesen werden. Das Kernstück bilden die `pg_stats`, in denen alle Werte der Statistik-Tabelle von PostgreSQL unverändert enthalten sind. Der Aufbau dieser Tabelle wird in Abschnitt 2.1.4 näher beschrieben.

Ergänzt werden diese Statistiken durch generatorspezifische SQL-Queries in `custom_stats`, welche nur für die betreffende Spalte ausgeführt werden. Dank der Möglichkeit eigene SQL-Statements zu definieren, eröffnen sich viele neue Wege, um eigene statistische Abfragen zu machen. So können die Generatoren ein optimales Modell erstellen, um die Daten zu synthetisieren.

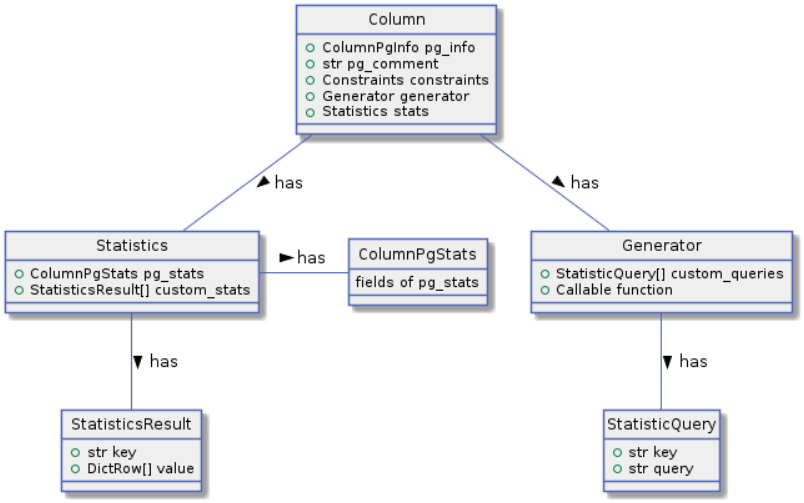


Abbildung 3.6: Datenklassen Statistikinformationen

3.4 Plugin-System für Generatoren

Das Plugin-System sorgt dafür, dass neue Generatoren einfach implementiert und bestehende Generatoren erweitert werden können, ohne in die übrige Programmstruktur eingreifen zu müssen. Zuerst folgt ein Überblick wie das System aufgebaut ist. Danach wird im Detail auf die generatorspezifischen Custom-Quieres eingegangen.

3.4.1 Überblick und Ablauf

Ein wichtiges Merkmal von Python ist, dass es eine sehr dynamische Sprache ist. So ist es möglich Importe von Modulen nicht nur statisch im Code zu schreiben, sondern zur Laufzeit Module programmatisch nachzuladen. Ein Beispiel für die Verwendung dynamischer Importe ist eine flexible Plugin-Architektur. Dies wird erreicht, in dem man Plugin-Module zum Programmstart dynamisch lädt und sich die enthaltenen Plugins per Python-Decorator beim System registrieren (RealPython, 2020). Andere Python-Libraries, wie die für das CLI verwendete Click-Library[14], benützen dieses Konzept, um Kommandos dynamisch nachzuladen.

Im Zusammenhang mit der Registrierung von Generatoren, mit der Anforderung der einfachen Erweiterung und Einbindung von neuen Generatoren, entschieden wir uns für die Registrierung über Python-Decorators und verwerfen eine statische Liste von vorhandenen Generatoren. So erreichen wir ein Maximum an Abstraktion und eine Registrierung, ohne in andere Teile des Programms eingreifen zu müssen.

> Dabei wird in Kauf genommen, dass der Entwicklungsaufwand, für die Erstellung des Decorators höher liegt, als bei einer statischen Liste.

Das Datagenerator-Modul kümmert sich um das korrekte Importieren der Plugins beim Start des Programms (Unterabschnitt 4.1.1). Dabei müssen die Plugins dynamisch erkannt und im Datagenerator registriert werden.

Nach der Registrierung kann das Hauptprogramm, basierend auf dem Datentyp sowie dessen Eigenschaften und Constraints, den geeignetsten Generator beim Datagenerator abfragen (Unterabschnitt 4.1.3). Anschliessend kann er den Generator mit definierten Parametern mit Details zum Datentyp und den statistischen Werten ansprechen und bekommt vom Generator die gewünschten synthetisch generierten Werte zurück. Die Abbildung 3.7 zeigt die Funktion und den Ablauf des Plugin-Systems.

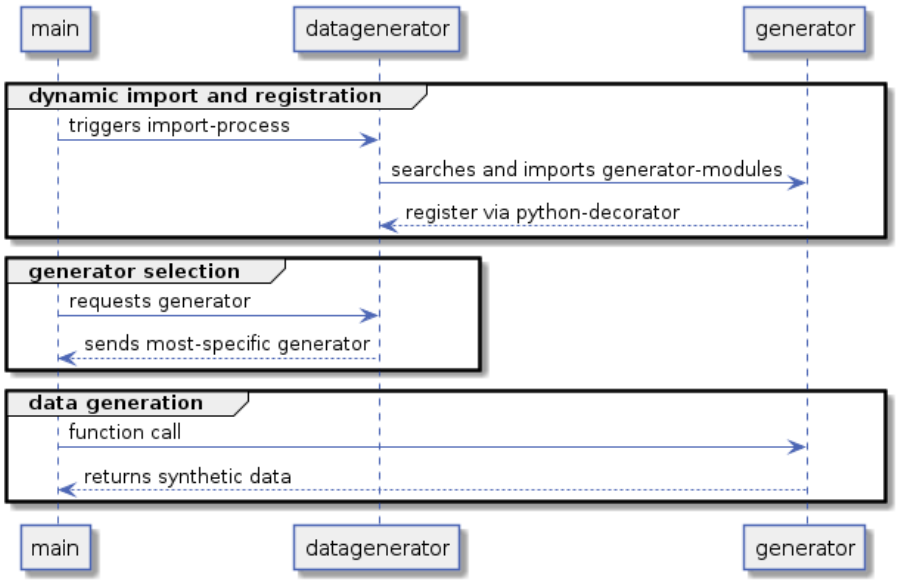


Abbildung 3.7: Funktionsweise des Plugin-Systems

Die Generatoren können bei der Registrierung alle wichtigen Aspekte definieren (Unterabschnitt 4.1.2) und bekommen, die für die Datengenerierung nötigen Informationen und Statistiken, per Parameter übergeben (Unterabschnitt 4.1.4). Eine Besonderheit besteht darin, dass die Generatoren bei der Registrierung eigene SQL-Queries mitgeben können (Unterabschnitt 3.4.2).

3.4.2 Generatorspezifische Abfragen

PostgreSQL kann nicht immer alle statistischen Informationen liefern, die für die Datengenerierung benötigt werden. Jeder Generator kann deshalb bei der

Registrierung spezifische Custom-Queries mitgeben. Die Queries werden nur für die entsprechende Spalte ausgeführt. Die Resultate werden dem Generator beim Aufruf mitgegeben.

Damit die Custom-Queries generisch für beliebige Spalten- und Tabellen-Namen funktionieren, werden spezielle Platzhalter verwendet. Diese sind im Abschnitt 4.1.2 genauer beschrieben. Diverse Generatoren machen von Custom-Quieries Gebrauch, wie zum Beispiel der Array-Generator (Unterabschnitt 4.3.1).

3.5 Array Generator

Das Erstellen von synthetischen Arrays ist komplexer als andere Datentypen. Folgend wir beschrieben, wie der Ablauf eines Array-Generator aufgebaut ist und welche Statistiken verwendet werden.

3.5.1 Struktur

Eine wichtige Design-Entscheidung für die Array Generatoren ist, ob für jeden Datentyp ein Generator erstellt wird oder ein einziger Generator, der für alle Datentypen Arrays generieren kann:

> Im Zusammenhang mit den Array Generatoren, mit der Anforderung der Erweiterbarkeit und Wartbarkeit, entschieden wir uns für das Erstellen eines Array Generator pro Datentyp (numeric, character, date ...) und verwerfen einen Array Generator für sämtliche Datentypen. So erreichen wir die Möglichkeit die Array-Elemente spezifischer zu erstellen und vermindern gleichzeitig die Komplexität der Array Generatoren. Dabei wird in Kauf genommen, dass für jeden Datentyp ein Array Generator implementiert werden muss, was einen Mehraufwand bedeutet.

Die Auswirkungen dieser Entscheidung schlagen sich auch auf die Abhängigkeiten der Array Generator Modulen nieder. In Abbildung 3.8 erkennt man, dass pro Array Generator ein neues Modul erstellt wird. Diese haben wiederum eine Abhängigkeit auf deren Array Element Generator und auf die Array Generator Utils. In den Utils werden Funktionen ausgelagert, welche von allen Array-Generatoren verwendet werden. So kann der in der Y-Entscheidung erwähnte Mehraufwand reduziert werden.

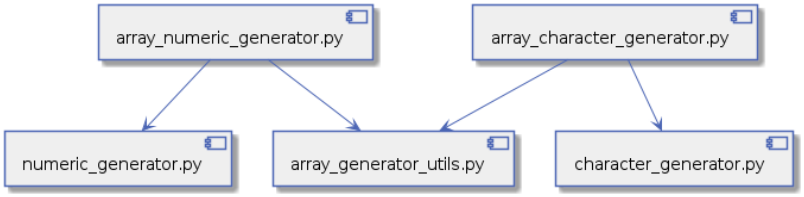


Abbildung 3.8: Modul-Abhängigkeiten der Array Generatoren

3.5.2 Ablauf

In Abbildung 3.9 ist der Ablauf der Array-Generierung zu sehen. Dabei wird zuerst eine Liste aller zu generierenden Array-Längen erstellt. Die Summe aller dieser Array-Längen ergibt die Anzahl benötigter Elemente. Diese können über die bestehenden Generatoren erstellt werden (Abschnitt 3.5.3). Danach können die Elemente in Arrays abgefüllt werden mit der zuvor generierten Länge. Falls noch `NULL`-Einträge gesetzt werden müssen, werden diese am Schluss unter die bestehenden Arrays gemischt.

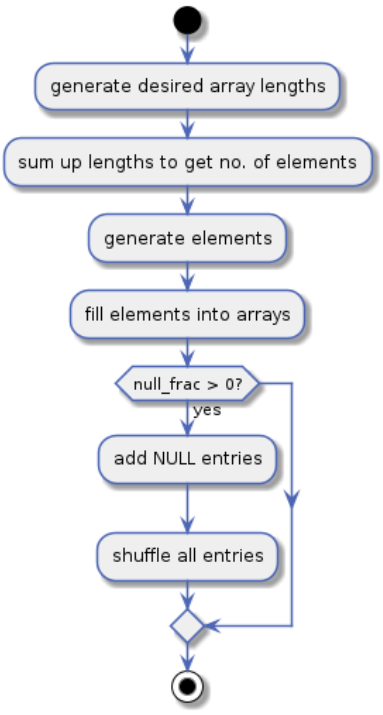


Abbildung 3.9: Ablauf Array Generator

3.5.3 Statistiken und Typinformationen

Die von PostgreSQL gelieferten Statistiken und Informationen für Arrays, müssen teilweise anders interpretiert werden als bei den bisherigen Generatoren. Für die Generierung werden sowohl Statistiken über die Spalte (wie zum Beispiel: Anzahl `NULL`-Werte, Histogramm der Arrays-Längen) als auch Statistiken und Informationen über die Elemente in den Arrays (wie zum Beispiel: Datentyp, Min-/Max-Werte) selbst benötigt.

Datentyp

Bei den bisherigen Generatoren konnte der Datentyp aus dem Attribut `data_type` erkannt werden. Bei Arrays ist dieser Eintrag jedoch immer `ARRAY` und der effektive Datentyp für die Elemente kann nur aus dem Attribut `udt_name` gelesen werden. Es muss deshalb ein Mapping zu den bisher verwendeten Typen gemacht werden. Die Tabelle 3.1 zeigt das Mapping des Nummeric-Array-Generator. Zu beachten ist, dass zur Unterscheidung von normalen UDT-Namen und Array-UDT-Namen ein Unterstrich vorangestellt wird.

Datentyp	**udt_name**
smallint[]	_int2
bigint[]	_int4
integer[]	_int8
numeric[]	_numeric
double precision[]	_float8

Tabelle 3.1: Mapping zwischen Nummeric Array-Datentypen und udt_name

Array-Längen und NULL-Einträge

Der Generator versucht ähnliche Array-Längen wie in der Originaltabelle zu erstellen. Dafür ist in der PostgreSQL-Statistik (Abschnitt 2.1.4) ein Attribut `elem_count_histogram` verfügbar, welches die Häufigkeit von Array-Längen wiedergibt. Dieses Histogramm wird für die Generierung der synthetischen Array-Längen verwendet.

Die Anzahl der `NULL`-Einträge in der Spalte (nicht zu verwechseln mit `NULL`-Elementen in den Arrays) können aus dem Attribut `null_frac` berechnet und generiert werden.

Array-Elemente und Häufigkeiten

Für die Generierung der Elemente in den Arrays stehen in den PostgreSQL-Statistiken die beiden Attribute `most_common_elems` sowie

`most_common_elem_freqs` zur Verfügung. Diese geben die häufigsten verwendeten Elemente an sowie deren Häufigkeit im Verhältnis auf die Anzahl nicht-`NULL` Zeilen. Anders als bei dem `most_common_freqs` welches bei den Generatoren für einfache Datentypen verwendet wird, gibt dieser Wert also nicht absolute Verhältnisse an. Der Array-Generator normalisiert deshalb die `most_common_elem_freqs_eq` nach der Formel (3.1) so, dass deren Summe 1.0 (100%) ergibt. So kann man es als Grundlage für das Generieren der Elemente verwenden.

$$\forall f_{mce} \quad f_{new_i} = \frac{f_{mce_i}}{\sum_{j=0}^{N} f_{mce_j}} \tag{3.1}$$

Sind diese Werte von PostgreSQL nicht gegeben, so werden zufällige Werte zwischen dem Minimum und Maximum der Elemente generiert. Diese Grenzwerte werden über eine Custom-Query (Abschnitt 4.1.2) ermittelt.

Für die `NULL`-Elemente gibt es ein dem `elem_count_histogram` angehängter Wert. Dieses gibt jedoch nicht das eigentliche Verhältnisse von `NULL`-Werten wieder, sondern lediglich das Verhältnisse von Arrays die einen oder mehrere `NULL`-Werte enthalten. Für die Datengenerierung ist dieser Wert unbrauchbar und muss ebenfalls über eine Custom-Query abgefragt werden.

Details zu diesen Custom-Queries befinden sich in der Implementation (Unterabschnitt 4.3.1).

3.6 Konfigurierbare Wahl von spezifischen Generatoren

Gemäss der Anforderung im Abschnitt 2.2.2 gibt es ein Bedürfnis zufällig realistische Daten (z.B. Echtnamen, existierende PLZ) generieren zu lassen. Bis anhin wäre für Namen lediglich ein String mit ähnlicher Länge aber völlig zufälligen Zeichen generiert worden. Neu sollen spezifische Generatoren realistische Daten generieren. Nachdem das Konzept erklärt wird, wird der vielfältige Einsatzbereich solcher Generatoren anhand von vier Beispiel demonstriert.

3.6.1 Konfigurations-Konzept

Die spezifischen Generatoren benötigt zwangsweise eine Konfiguration, was mit dem bisherigen Konzept der Non-Konfiguration bricht. Deshalb soll die Konfiguration möglichst einfach für den Benutzer sein.

Im Zusammenhang mit der Konfiguration spezifischer Generatoren, mit der Anforderung der Einfachheit, Klarheit und Erweiterbarkeit, entschieden wir uns für die Konfiguration über Spaltenkommentare und

> verwerfen die Erkennung über Spaltennamen oder Prädikate. So erreichen wir eine generische Lösung, um spezifische Generatoren zu verwenden. Dabei wird in Kauf genommen, dass eine minimale Konfiguration der Datenbank erforderlich ist.

Syntax

Die Konfiguration wird als Spaltenkommentar hinterlegt, es sollen aber weiterhin andere Kommentar für andere Zwecke möglich sein. Es ist deshalb notwendig eine Syntax für die Konfiguration zu definieren, damit diese im Kommentar gefunden und ausgewertet werden kann.

Als Erkennungsmerkmal ist `PGSYNTHDATA` definiert. Dahinter folgt der Comment-Identifier innerhalb von eckigen Klammern für die Wahl des spezifischen Generators. Davor und danach dürfen beliebige andere Kommentare platziert werden.

Beispiel eines Konfigurationkommentars: `"... PGSYNTHDATA[PERSON_NAME] ..."`.

Kommentar hinterlegen

Die Hinterlegung der Spaltenkommentare und damit die Konfiguration ist einfach und benutzerfreundlich. Die Kommentare können entweder als SQL-Befehl (Listing 3.2) oder in einem Tool mit pgAdmin über ein GUI gesetzt werden.

```
1 COMMENT ON COLUMN test_table.name IS 'PGSYNTHDATA[PERSON_NAME]';
```

Listing 3.2: Beispiel Spaltenkommentar hinterlegen

3.6.2 Eigennamen-Generator

Ein Beispiel eines spezifischen Generators ist der Eigennamen-Generator, der mit Hilfe einer externe Library realistische Eigennamen für Personen generiert. Unterabschnitt 4.3.2 im Kapitel Implementation zeigt, wie einfach und schnell ein solcher Generator geschrieben werden kann. Das Person Generator Modul bietet Generatoren mit folgenden Comment-Identifiers an:

- `PERSON_FIRST_NAME`: Generiert Vornamen (z.B. 'Paul')
- `PERSON_LAST_NAME`: Generiert Nachnamen (z.B. 'Muster')
- `PERSON_NAME`: Generiert Namen (z.B. 'Paul Muster')

3.6.3 Adressen-Generator

Ein weiteres Beispiel ist der Adressen-Generator. Folgende Generatoren bez. Comment-Identifiers werden angeboten:

- `ADDRESS_ZIP`: Generiert Postleitzahlen (z.B. 8640)
- `ADDRESS_STREET`: Generiert Strassennamen (z.B. 'Oberseestrasse')
- `ADDRESS_BUILDING_NR`: Generiert Hausnummern (z.B. 10)
- `ADDRESS_CITY`: Generiert Städte (z.B. 'Rapperswil')
- `ADDRESS_STREET_ADDRESS`: Generiert Strassenname mit Hausnummern (z.B. 'Oberseestrasse 10')
- `ADDRESS_COUNTRY`: Generiert Ländernamen (z.B. 'Schweiz')
- `ADDRESS_COUNTRY_CODE`: Generiert Ländercode (z.B. 'CH')

3.6.4 Copy- und Ignore-Generator

Diese beiden Generatoren erzeugen keine synthetischen Daten, ermöglichen es jedoch gezielt gewisse Spalten eins-zu-eins zu kopieren oder komplett zu ignorieren. Dadurch ist eine grosse Flexibilität und Anpassungsfähigkeit für anwendungsspezifische Datenbankstrukturen in der Praxis gegeben. Beim Ignore-Generator wird die Spalte mit dem Default-Wert des Datentypes (meistens `NULL`) gefüllt und darf dementsprechend keine `NOT NULL` Einschränkung haben.

3.7 Spatial Generator

Die für PostgreSQL erhältliche Erweiterung PostGIS[15] beinhaltet geometrische Datentypen und Funktionen und wird oft in Geoinformationssystemen wie beispielsweise OpenStreetMap verwendet. Dabei werden ähnliche Elemente mit Attributen und der Geoinformation in einer Tabelle zusammengefasst. Ein Beispiel solch einer Tabelle ist in Abbildung 3.10 zu sehen, in der neben den Attributen Stationsname, Bezirk und bediente Linien auch die Koordinaten als PostGIS Geometry-Type (ST_POINT) gespeichert sind.

objectid numeric	name character vary	borough character var	routes character	geom geometry
1	Cortlandt St	Manhattan	R,W	0101000020266900003717...
2	Rector St	Manhattan	1	0101000020266900000CBE3...
3	South Ferry	Manhattan	1	0101000020266900000C676...
4	138th St	Bronx	4,5	0101000020266900000F4CF...
5	149th St	Bronx	4	010100002026690000084DA...

Abbildung 3.10: Beispiel einer Tabelle mit PostGIS Geometry-Typ

Solche Geometry-Typen wie beispielsweise ST_POINT, ST_LINE oder ST_POLYGON werden in einem speziellen Binärformat gespeichert. Deshalb können diese Typen nur, mit den von der PostGIS-Erweiterung zur Verfügung gestellten Funktionen und Operationen, verwendet werden. In den folgenden Unterkapiteln wird beschrieben, wie mit dieser Herausforderung umgegangen wird.

3.7.1 Registierung im Plugin-System

Der Datentyp der PostGIS-Spalten ist in den Column-Informations mit `USER-DEFINED` angegeben. Um den Generator für alle PostGIS-Typen zu registrieren, muss in den `udt_names` Einschränkungen noch `geometry` angegeben werden.

3.7.2 Verwendete Statistiken

Da die PostGIS-Datentypen in einem speziellen Binärformat gespeichert werden, sind die Informationen aus `pg_stats` wenig hilfreich und oft gar nicht vorhanden. Deshalb benützt der PostGIS-Generator eigene Custom-Queries, welche PostGIS-spezifische Funktionen verwenden. Dank diesen kann der Generator die relevanten Informationen ermitteln.

3.7.3 Unterstützte Datentypen

Als Proof-of-Concept wird der ST_POINT-Datentyp für X/Y-Punkte unterstützt. Zuerst wird anhand der Resultate der Custom-Quieries geprüft, ob die Geometries in der Spalte alle vom Typ `ST_POINT` sind. Falls nicht, wird die betreffende Spalte leer gelassen. Um die Punkte in denselben Bereich zu generieren wie die Originalen, werden in den Custom-Queries die Minimal- und Maximalwerte der X und Y Koordinate ermittelt. Anhand dieser Wertebereiche können nun zufällige neue Koordinaten generiert und als Punkte zurückgegeben werden. Weiter Datentypen können nach dem gleichen Prinzip implementiert werden.

3.8 Enumerated Generator

In PostgreSQL kann der Datenbankbenutzer eigene Enum-Datentypen erstellen. Diese werden in den Column Informations als `USER-DEFINED` angegeben und im Feld UDT-Name wird, der vom Benutzer definierte, Namen angezeigt. Dieser unterscheidet sich somit von Anwendungsfall zu Anwendungsfall und ist dem Programm nicht im Voraus bekannt. So kann er nicht von anderen UDT-Typen wie beispielsweise PostGIS (Unterabschnitt 3.7.1) unterschieden werden. Die Abbildung 3.11 verdeutlicht diesen Konflikt.

	column_name name	data_type character varying	udt_name name
1	an_enum	USER-DEFINED	mood
2	not_an_enum	USER-DEFINED	geom

Abbildung 3.11: Konflikt bei der Erkennung von Enumerated-Typen

Es gibt zwar eine PostgreSQL-interne Tabelle für die Abfrage der Enum-Werte, diese müsste jedoch vom Hauptprogramm für alle Datenbanken abgefragt werden, unabhängig davon, ob die Datenbank Enums enthält oder nicht. Auch andere Codeteile, wie die Auswahl des passenden Generators, wären davon betroffen. Diese Module würden also generatorspezifische Element enthalten, was es, wenn möglich zu vermeiden gilt.

Um dieses Problem zu lösen, ohne einen Spezialfall einbauen zu müssen, wird der Enumerated-Generator für alle UDTs registriert, für die es keinen spezifischeren Generator gibt.

Y

Im Zusammenhang mit dem Erstellen eines Enumerated-Generators, mit der Anforderung der Erkennung von Enums, entschieden wir uns für eine Generator-Registration, welche auf alle `USER-DEFINED`-Datentypen zutrifft, sofern dafür kein anderer Generator registriert ist und verwerfen einen entsprechenden Sonderfall im Hauptprogramm. So erreichen wir eine Enum-Erkennung, ohne das Hauptprogramm umbauen zu müssen. Dabei wird in Kauf genommen, dass in gewissen Spezialfällen erst beim Generieren der Werte klar ist, dass für den vorliegenden Datentyp kein Generator existiert (falls `USER-DEFINED` ohne Generator und doch kein Enum).

Über die Custom-Query in Listing 3.3 fragt der Generator die interne Enum-Tabelle ab um alle gültigen Enum-Werte der Spalte auslesen. Gibt die Custom-Query keine Werte zurück, so handelt es sich nicht um einen gültigen Enum-Type und der Datengenengenerator meldet eine Warnung an den Benutzer und generiert keine Daten für die Spalte. Andernfalls pickt er die geforderte Anzahl an zufälligen Werten aus der Liste.

```
SELECT
  enu.enumlabel as label
FROM
  information_schema.columns col
  INNER JOIN pg_type typ ON col.udt_name = typ.typname
  INNER JOIN pg_enum enu ON typ.oid = enu.enumtypid
WHERE
  col.table_name = {table_name_str}
  AND col.column_name = {column_name_str}
```

Listing 3.3: Custom-Query um alle möglichen Enum-Werte auszulesen

3.9 Kommandozeilenschnittstelle

Das Tool kann über die CLI angesprochen werden. Der Fokus liegt auf einfacher Bedienung und wenig Konfiguration. Dies wird durch sinnvolle Default-Optionen erreicht, sodass eine Parametrisierung in Normalfall nicht notwendig ist. Es gibt drei wichtige Kommandos:

- **Generate**: Dies ist der zentrale Befehl und erzeugt aus der Originaldatenbank eine Datenbank mit der gleichen Struktur, aber rein synthetischen Daten.
- **Show**: Kann verwendet werden, um die Struktur einer Datenbank anzuzeigen und ist besonders vor der Generierung nützlich. Es zeigt relevante Informationen wie den Datentyp, ob es sich um einen Fremdschlüssel handelt, welcher Generator verwendet wird oder die Kommentare der Spalte an.
- **Analyze**: Analyze vergleicht die Originaldatenbank mit der generierten Datenbank und ist für eine Validierung nach der Generierung nützlich. Die wichtigsten statistischen Eigenschaften der ursprünglichen und der generierten Spalte werden nebeneinander angezeigt. Die Ausgabe ist eine Datei im Format html oder optional json.

Generelle und Kommando-spezifische Optionen ermöglichen eine erweiterte Konfiguration. Der syntaktische Aufbau der Kommandos inklusive Beispiele sind in Listing 3.4 ersichtlich. Die vollständige Beschreibung der Kommandos und deren Optionen befindet sich im Benutzerhandbuch (Anhang B).

```
# Syntaktischer Aufbau der Komamndos
pgsynthdata [GENERAL_OPTIONS] SRC_DATABASE COMMAND  [COMMAND_OPTIONS]

# Einfache Beispiele mit Default-Konfiguration
pgsynthdata "test_db" show
pgsynthdata "test_db" generate "test_db_gen"
pgsynthdata "test_db" analyze "test_db_gen"

# Beispiel einer komplexen Konfiguration
pgsynthdata --user "admin" --host "localhost" --port 5432 --tables
↪  "tabl1:table2:table3" --verbose "test_db" analyze "test_db_gen"
↪  --drop --owner "postgres" --m_factor 1.5
```

Listing 3.4: Aufbau eines `pgsynthdata` CLI Befehles, inkl. Beispiele

Kapitel 4

Implementation

Die folgenden Abschnitte geben Einblicke in relevante Aspekte der Implementation. Diese sind für das Software-Engineering und die Weiterentwicklung relevant. Der letzte Abschnitt geht auf Softwaretests für die Sicherung der Qualität ein.

4.1 Plugin-System

Beim Plugin-Sytem ist der dynamische Import der Module sowie die Registrierung der Generatoren von Bedeutung. Beide nutzen dazu python-spezifische Techniken.

4.1.1 Dynamischer Import der Generator-Module

Der dynamische Import der Generator-Modulen ist die Grundlage des Plugin-Systems. Es ermöglicht das Erstellen neuer Generatoren ohne eine Änderung im Modul `datagenerator`, welches für die Verwaltung der Generatoren zuständig ist.

Beim Laden des Datagenerator-Modules wird automatisch nach Generator-Modulen gesucht und diese dynamisch mit Hilfe der Bibliothek `importlib` importiert. Damit die Module erkannt werden, müssen diese im Ordner `./generators` als Module (Python-Files) mit der Endung `*_generator.py` vorliegen. Alle nach diesem Muster gefundenen Module werden importiert. Das Listing 4.1 zeigt eine vereinfachte Version dieses Prozesses.

```
1 from importlib import import_module
2 def _import_generators():
3     for filename in os.listdir(path):
4         if filename.endswith("_generator.py"):
5             importname = ".generators." + filename[:-3]
6             import_module(importname, package="pgsynthdata")
7 _import_generators() # execute import at startup
```

Listing 4.1: Vereinfachte Version des dynamisches Importes für Generatoren

4.1.2 Registrierungs-Dekorator

Beim Importieren eines Generator-Modules, werden die darin enthaltenen Generator-Funktionen automatisch mittels eines Python-Decorators beim Plugin-System registriert. Diesem Dekorator (`@generator`) können alle für die Registrierung relevanten Parameter mitgegeben werden. In Listing 4.2 ist ein Beispiel einer solchen Registrierung zu sehen.

```python
@generator( # registration decorator (example only)
    {"integer", "numeric", "real"},
    udt_names_only={'int8', ...},
    custom_queries=[StatisticQuery("myName", "SELECT ...")]),
    comment_identifier="SPECIAL_GENERATOR_IDENTIFIER"
)
def nummeric_generator(...):
```

Listing 4.2: Beispielanwendung des Registrierungs-Dekorators

Als Parameter wird angegeben, auf welchen *Datentyp* der Generator sich registrieren will, ob gewisse *UDT's* oder ein *Comment Identifier* vorausgesetzt ist und welche *Custom-Queries* ausgeführt werden sollen. Die einzelnen Punkte sind in den nachfolgenden Unterkapiteln beschrieben.

Falls für die angegebenen Registrierungsparameter (ausser Custom-Queries) schon ein Generator registriert ist, wird eine Warnung ausgegeben und der schon registrierte Generator überschrieben.

Data Type

Als erster Parameter ist der Datentyp anzugeben. Es können auch mehre Datentypen als Set registriert werden. So kann ein Generator für ähnliche Datentypen registriert werden, was oft der Fall ist.

UDT Names Only (optional)

Optional können UDT-Namen als Set angegeben, für die der Generator geeignet ist. Wird dieser Parameter weggelassen, so ist der Generator für alle UDT's gültig.

Custom-Queries (optional)

Optional kann eine Liste von Custom-Queries registriert werden. Eine Custom-Query besteht aus einem Namen sowie die eigentliche Datenbank-Abfrage. Über den Namen kann der Generator später das Resultat der Query auslesen. Um die Erfassung zu erleichtern, gibt es die Datenklasse `StatisticQuery`.

Die Query muss generisch geschrieben werden. Für den Spalten- und Tabellen-Name müssen deshalb die Platzhalter `{column_name}` sowie `{table_name}` verwendet werden. Listing 4.3 ist ein Beispiel einer solchen Custom-Query mit den angesprochenen Platzhalter.

```
StatisticQuery(
  name = "minimum_value",
  query = "SELECT min({column_name}) FROM {table_name}"
)
```

Listing 4.3: StatisticQuery mit Platzhalter Beispiel

Comment Identifier (optional)

Hier kann ein Identifier angegeben werden, dieser Generator wird dann nur verwendet, wenn im Kommentar der Spalte der Identifier entsprechend hinterlegt ist. Beim Ermitteln eines passenden Generators wird im Spaltenkommentar über einen Regex-Match explizit nach einer Konfiguration gesucht (`"PGSYNTHDATA\[[a-zA-Z_\-]+\]"`). Das Alphabet für die Identifier ist dementsprechend eingeschränkt auf folgende Zeichen: `a-z A-Z _ -`

4.1.3 Auswahl des spezifischsten Generators

Für die Datengenerierung soll der bestpassende Generator ausgewählt werden. Um dies zu ermögliche, werden die Generatoren bei der Registrierung in einer Baumstruktur, wie in Abbildung 4.1 sichtbar, aufgebaut. Der Baum besteht aus drei Ebenen, welche der Relevanz der Registrierungsfaktoren entsprechen:

1. Comment Identifiers
2. Data Type
3. UDT Name

Ist einer der Registrierungsfaktoren nicht angegeben (`None`), so gilt dieser Pfad als Fallback und wird genommen, falls keiner der anderen Faktoren derselben Ebene zutrifft. Eine rekursive Traversierung des Baumes ermöglicht es somit, den spezifischsten Generator zu finden.

Die entsprechende Funktion ist in Listing 4.4 inkl. eines Beispielaufrufs abgebildet. In Abbildung 4.1 ist der Weg der Traversierung dieses Beispielaufrufs mit den Parametern `comment_identifier = None`, `data_type = "bigint"` und `udt_name = "int8"` violett hervorgehoben.

Falls kein Generator gefunden wird, gibt es eine Warnung und es werden für diese Spalte keine Daten generiert.

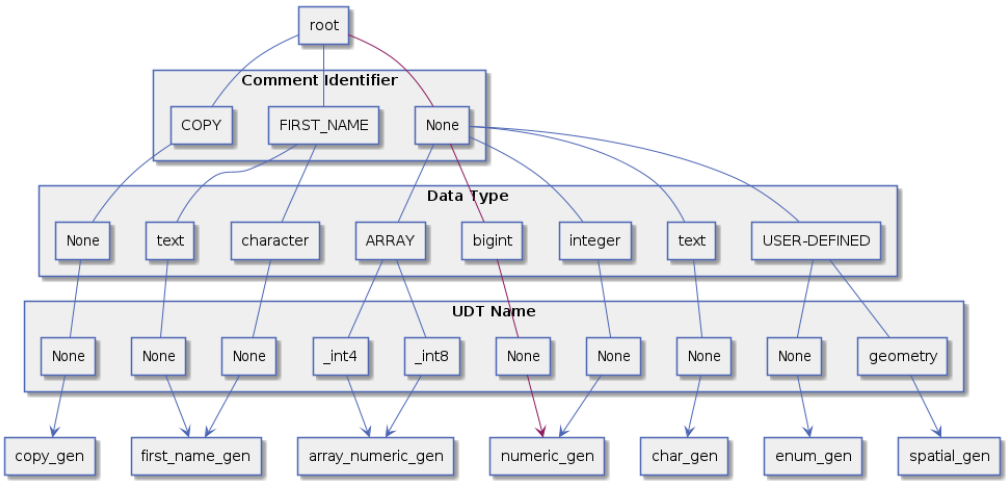


Abbildung 4.1: Baum der Generatoren (nicht vollständig)

```python
def _get_from_generators_tree(
    keys: List[Optional[str]],
    tree: Union[Dict, Generator, None] = _generators_tree,
) -> Optional[Generator]:
    # if tree is no dict, it is either None or a Generator
    if not isinstance(tree, dict):
        return tree

    key = keys[0]
    gen = _get_from_generators_tree(keys[1:], tree.get(key, None))

    # fallback to less specific key (= bracktrace)
    if gen is None and key is not None:
        gen = _get_from_generators_tree(keys[1:], tree.get(None,
        ↪  None))

    return gen

# example call with [Comment Identifier, Data Type, UDT Name]
numeric_gen = _get_from_generators_tree([None,"bigint","int8"])
```

Listing 4.4: Generatoren-Baum-Traversierung um den passender Generator zu finden

4.1.4 Generatoren-Signatur

Damit der generische Programmteil jeden Generator verwenden kann, müssen die Generatoren dieselbe Signatur aufweisen. Die in Listing 4.5 sichtbare Signatur ist deshalb für alle Generatoren vorausgesetzt.

So werden den Generatoren möglichst alle zur Verfügung stehenden Information übergeben, sodass eine höchstmögliche Flexibilität sichergestellt ist und der Generator selbst entscheiden kann welche Informationen für ihn relevant sind und welche nicht.

```
1 @generator(...) # (details omitted)
2 def nummeric_generator( # parameters
3     stats: Statistics,
4     column_info: ColumnPgInfo,
5     constraints: Constraints,
6     rows_to_gen: int
7 ):
8     return [...] # size of 'rows_to_gen'
```

Listing 4.5: Beispiel einer Generatoren-Signatur

Die Parameter `stats`, `column_info` und `constraints` sind als Datenklassen aufgebaut und in Abschnitt 3.3 beschrieben. Der letzte Parameter `rows_to_gen` gibt an wie viele Datenpunkte erstellt werden müssen. Vom Generator wird erwartet, dass er eine Liste der Grösse `rows_to_gen` des angefragten Datentyps zurückgibt.

Für Standard-Python Datentypen wird ein automatisches Mapping zu Standard-Structured Query Language (SQL) vorgenommen (z.B. `string` zu `TEXT`). Wird für den Insert eine Funktion benötigt, muss die Klasse `AsIs` verwendet werden (z.B. `AsIs('ST_Point(1.0,4.2)')`).

4.2 Datenbankanbindung

Hier werden zwei datenbankspezifische Implementationsdetails gezeigt. Relevant ist zum einen das Erstellen der dynamischen Insert-Queries für das Befüllen der Zieldatenbank und zum anderen die komplexe SQL-Query, um die Spaltenkommentare auszulesen.

4.2.1 Dynamische Insert Queries

Nach dem Generieren der Synthetischen Daten, müssen diese in die neue Datenbank eingetragen werden. Die Funktion `insert_data(...)` in der `Database`-Klasse

übernimmt diese Aufgabe. Die Funktion ist in Listing 4.6 als vereinfachte Version zu sehen. Als Parameter erhält sie den Namen der Tabelle und ein Dictionary mit dem Spaltenname als Key und einer List von Werten.

Der Spaltenname muss, mit dem in der Datenbank übereinstimmen. Dieser wird dazu verwendet, um dynamisch eine Insert-Query zu generieren. Das Ausführen der Inserts für alle Daten-Reihen wird an das für die Datenbankanbindung zuständige externe Modul übergeben (Abschnitt 4.4.3). Dieses stellt eine Funktion zur Verfügung, um grosse Datenmengen in die Datenbank zu schreiben (`psycopg2.extras.execute_values()`).

Die Daten müssen dem externen Modul als `Dict` pro Row übergeben werden. Die Generatoren erstellen aber pro Spalte eine Liste in einem `Dict`. Es ist deshalb nötig, die Daten entsprechend zu transformieren. In Listing 4.6 werden diese Transformierte Daten in die Variabel `data_for_insert` gespeichert. Der Unterschied der beiden Darstellungen ist in Listing 4.7 zu erkennen.

```
1  def insert_data(self, tablename: str, data: Dict[str,
   ↪  Optional[Iterable[Any]]]):
2      # filter Columns with no Values
3      data = {k: v for k, v in data.items() if v is not None}
4      # transform to dict of columns to list oft dict rows
5      data_for_insert = [dict(zip(data, t)) for t in
       ↪  zip(*data.values())]
6
7      sql_query, template = _create_insert_query(tablename,
       ↪  data.keys())
8      with self._conn.cursor() as cur:
9         psycopg2.extras.execute_values(
10            cur, sql_query, data_for_insert, template=template
11        )
```

Listing 4.6: Vereinfachte Insert Data Funktion der Database Klasse

```
1  dict_of_columns = {"id": [1, 2, 3], "name": ["Luke", "Lea", "Ben"]}
2  dict_per_row    = [
3                      {"id": 1, "name": "Luke"},
4                      {"id": 2, "name": "Leia"},
5                      {"id": 3, "name": "Ben"}
6                    ]
```

Listing 4.7: Unterschiedliche Darstellung der Insert-Daten

4.2.2 Auslesen der Spaltenkommentare

Um das Konzept der Comment-Identifiers zu unterstützen, müssen die Spalten-Kommentare aus der Datenbank gelesen werden. Die Spalten-Kommentare müssen bei PostgreSQL über eine Spezielle Tabelle abgefragt werden. Im Listing 4.8 ist die implementierte SQL-Query zu sehen, um den Spaltenkommentar einer Spalte auszulesen.

```
1  SELECT
2      (
3          SELECT
4              pg_catalog.col_description(c.oid,
               ↪  cols.ordinal_position::int)
5          FROM pg_catalog.pg_class c
6          WHERE
7              c.oid         = (SELECT cols.table_name::regclass::oid)
8              AND c.relname = cols.table_name
9      ) as column_comment
10 FROM information_schema.columns cols
11 WHERE
12     cols.table_catalog     = 'Datenbank Name' -- Platzhalter
13     AND cols.table_name    = 'Tabellen Name'  -- Platzhalter
14     AND cols.column_name   = 'Spalten Name'   -- Platzhalter
```

Listing 4.8: SQL Query um Spaltenkommentar auzulesen

4.3 Generatoren

In diesem Abschnitt wird beschrieben, wie der Array Generator die fehlenden Informationen kompensieren kann. Es wird zudem ein Beispiel eines spezifischen Generators gegeben welcher Personen-Namen generieren kann.

4.3.1 Custom-Query für Nummeric Array Generatoren

Dem Array Generator fehlen wichtige statistische Informationen, die nicht von `pg_stats` geliefert werden. Diese beziehen sich vor allem auf die Elemente in den Arrays.

Mit der in Listing 4.9 aufgezeigten SQL-Query können die, in Abschnitt 3.5.3 beschriebenen, fehlenden Informationen aus der Datenbank abgefragt werden. Dabei wird jedes Array durchgegangen (`unnest()`) und die Min-/Max-Werte sowie der Anteil `NULL`-Werte ermittelt. Diese stehen anschliessend dem Generator zur Verfügung.

```
SELECT
  min((SELECT min(v) FROM unnest({column_name}) AS t(v))),
  max((SELECT max(v) FROM unnest({column_name}) AS t(v))),
  (1-sum((SELECT count(v) FROM unnest({column_name}) AS t(v)))
    /sum((SELECT count(*) FROM unnest({column_name}) AS t(v))))
    AS null_frac
FROM {table_name}
```

Listing 4.9: Custom-Query für den Array Elemente Generator

4.3.2 Spezifische Generatoren mittels Faker

Spezifische Generatoren können für viele Anwendungsfälle sehr einfach und schnell implementiert werden. In Listing 4.10 ist ein Beispiel eines spezifischen Generators basierend auf der Faker-Library (Unterabschnitt 4.4.3) zu sehen. Über eine List-Comprehension wird die genau geforderte Anzahl an Werten über den Faker generiert. Der Generator ist zudem in der Lage eindeutige Namen zu generieren, falls es eine solche Einschränkung auf der Spalte gibt.

```
@generator("text", comment_identifier="PERSON_NAME")
def person_name_generator(
    stats: Statistics, column_info: ColumnPgInfo, constraints:
    ↪  Constraints, rows_to_gen: int
):
    fake = Faker()
    if constraints.is_unique:
        return [fake.unique.name() for _ in range(rows_to_gen)]
    return [fake.name() for _ in range(rows_to_gen)]
```

Listing 4.10: Eigennamen Generator basierend auf der Faker-Library

4.4 Verwendete Technologien

Diese Kapitel gibt einen Überblick über die Verwendeten Tools und Libraries. Diese vereinfachen die Entwicklung und führen zu besserer Codequalität.

4.4.1 Python-Version

Es wird mit der zu Beginn der Arbeit aktuellen CPython Version 3.9 gearbeitet. Diese ist nach Langa (2019) im PEP 596[16] bis im Oktober 2025 unterstützt.

Grundsätzlich das Tool mit alle Python-Versionen von 3.7 bis 3.10 kompatibel.

4.4.2 Entwicklungstools

Die folgenden Python Tools wurden für die Entwicklung von `pgsynthdata` verwendet.

Dependency Management

Um das Entwicklungsprojekt zu managen, bietet es sich an, einen Paketierung- und Dependency-Manager zu verwenden. `pgsynthdata` verwendet dafür Poetry[17]. Damit können die Abhängigkeiten zu den verwendeten Libraries (Unterabschnitt 4.4.3) einfach verwaltet werden. Zudem arbeitet jeder Entwickler und die Continuous Integration (CI)-Pipeline mit denselben Versionen der Libraries.

In der `pyproject.toml` Datei speichert Poetry alle Dependencies. Zudem sind in dieser Datei Angaben zum Projekt an sich zu finden (z.B. Version, Name und Autoren). Alle Dependencies lassen sich mit dem Konsolenbefehl `poetry install` installieren.

Code Formatierung

Ein einheitlicher Code-Style ist bei Entwicklungsprojekten sehr wichtig. Wird mit Python gearbeitet ist es sogar noch viel wichtiger, da dort die Einrückung eine syntaktische Relevanz hat. Um sicherzustellen, dass jeglicher Code mit denselben Regeln formatiert ist, wird Black[18] verwendet. Die CI Pipeline überprüft, ob alle Regeln eingehalten werden. Nach Langa (2018), ist Black PEP 8[19] konform. Dieses PEP beschreibt den Style Guide, der für die Standard-Python-Bibliotheken vorausgesetzt wird (van Rossum, Warsaw & Coghlan, 2001).

Die Code-Formatierung kann am einfachsten über Poetry gestartet werden: `poetry run black .` dabei werden gleich alle Files im Projekt überprüft und ggf. angepasst.

Imports Sortierung

Um alle die `import` Zeilen einheitlich und automatisiert zu sortieren wird das Tool isort[20] verwendet. Auch hier überprüft die CI-Pipeline, ob die korrekte Sortierung angewendet wird.

Auch dieses Tool kann über Poetry gestartet werden: `poetry run isort .`

Linter und Static Type Checker

Um weitere grobe Code Qualitätsfehler zu finden und den Code zu verbessern, werden die Tools pylint[21] (Linter) und mypy[22] (Static Type Checks) eingesetzt. Die Tools liefern Verbesserungsvorschläge welche beurteilt und gegebenenfalls behoben werden müssen.

Beide Tools sind wiederum über Poetry aufrufbar:

```
poetry run pylint **/*.py
poetry run mypy -p pgsynthdata
```

Unittest und Testabdeckung

Für automatische Unit-Tests wird pytest[23] verwendet. Die Testabdeckung wird mit coverage[24] berechnet. Der Einsatz der beiden Tools wird in Unterabschnitt 4.5.1 genauer beschrieben.

4.4.3 Externe Libraries

Auf folgende 3rd-Party-Libraries ist `pgsynthdata` aufgebaut. Verwendete Python Standardbibliotheken werden nicht explizit aufgelistet.

Datenbankanbindung

Um mit der Datenbank zu kommunizieren wird Adapter Psycopg[25] verwendet. Es ist der etablierteste PostgreSQL Adapter für Python. Dadurch ist eine gute Dokumentation vorhanden und es steht eine grosse Community dahinter.

Psycopg richtet sich nach dem PEP 249[26] von Lemburg (2001). Dadurch ist die Datenbankanbindung nicht extrem eng mit dem `database` Modul gekoppelt und könnte ggf. ausgetauscht werden.

Bei einem Austausch müssten aber trotzdem einige Funktionen, welche auf spezifischen pgsycopg Funktionen aufbauen, im `database` Modul angepasst werden. Dazu gehören Funktionen wie das Ausführen der Insert-Statements (Unterabschnitt 4.2.1) oder das Auslesen der Resultsets als `Dict` (Abschnitt 3.3).

Command Line Interface

Da `pgsynthdata` nur über die Command-Line verwendet werden kann, ist das CLI ein sehr wichtiger Aspekt. In Python gibt es verschiedene Varianten ein solches CLI zu erstellen. Zum einen kann direkt auf die Argumente des Scripts zugegriffen werden. Da muss aber ein manelles Parsing gemacht werden. Zudem

muss man auch von Hand eine "Help Funktion" schreiben. Um diesen Prozess zu automatisieren und die Wartbarkeit zu erhöhen wird Click[27] eingesetzt.

Fake Daten Generator

Um realistische Daten zu generieren, wird die Library Faker[28] verwendet. Die Library unterstützt die Generierung einer Vielzahl an spezifischen Daten wie beispielsweise Personendaten oder Adressdaten.

4.5 Testing

Beim Testing wird primär auf automatisierte Tests gesetzt. Diese können bei Code-Änderungen sehr schnell einen Überblick geben, ob die Software noch so funktioniert wie erwartet. Die automatisierten Tests können lokal ausgeführt werden und sind in der CI-Pipeline eingebaut.

4.5.1 Automatisierte Unit-Tests

Für die Unit-Tests wird pytest verwendet. Pro Modul existiert ein Test-File mit den Unit-Tests. Die Unit-Test werden mit den Konsolenbefehl `poetry run pytest` ausgeführt.

Gewisse Test setzten eine lokale Installation der PostgreSQL-Application-Tools sowie eine aktive Datenbankverbindung mit definierten Testdatenbank voraus. Dafür müssen die Environment Variablen in Listing 4.11 gesetzt sein.

```
1 TEST_DBNAME=pgsynthdata_testdb
2 TEST_USER=postgres
3 TEST_PW=postgres
4 TEST_HOST=localhost
5 TEST_PORT=5432
```

Listing 4.11: Environment Variabeln für Datenbank Unit-Test

Testabdeckung

Die Testabdeckung kann mit dem Tool `coverage` berechnet werden. Nach dem Ausführen kann ein Bericht auf der Konsole ausgegeben werden. Um eine genauere Analyse zu erhalten, welche Codezeilen nicht durch Tests abgedeckt werden,

kann coverage auch ein HTML-Bericht erstellen. Für all diese Usecases sind in Listing 4.12 Beispiele aufgeführt. Ein aktueller coverage Report kann im Anhang H eingesehen werden.

```
# run tests and calculate coverage
poetry run coverage run -m --source=pgsynthdata pytest tests

# report on console
poetry run coverage report

# report as website
poetry run coverage html
open htmlcov/index.html
```

Listing 4.12: Verwendete coverage Befehle

Testdatenbank und Mocking

Die Unit-Tests prüfen, ob eine Verbindung zur Testdatenbank möglich ist. Ist keine Verbindung möglich, werden gewisse Tests mit einer Warnung übersprungen. Um die Datengeneratoren trotzdem testen zu können, gibt es eine serialisierte Struktur der Testdatenbank. Dieses Mocking ermöglicht rasche und trotzdem realitätsnahe Unit-Tests.

Eine detaillierte Beschreibungen und Installationsanleitung dieser beiden Komponenten ist im Benutzerhandbuch (Anhang B) zu finden.

4.5.2 Manuelle Systemtests

Als Ergänzung zu den automatisierten Unit-Tests, gibt es manuell durchzuführende Systemtests, die das korrekte Zusammenspiel aller Komponenten, sowie die Interaktion mit dem Benutzer über das CLI, überprüfen. Sie sind als Black-Box-Tests konzipiert und werden anhand eines Protokolls durchgeführt und beurteilt. Der Anhang E zeigt ein solches Testprotokoll.

Kapitel 5

Resultate

Dieses Kapitel zeigt die erreichten Ziele und unter welchen Bedingungen das Tool eingesetzt werden kann. Abschliessend werden Vorschläge für mögliche Weiterentwicklungen des Programms gemacht.

5.1 Zielerreichung

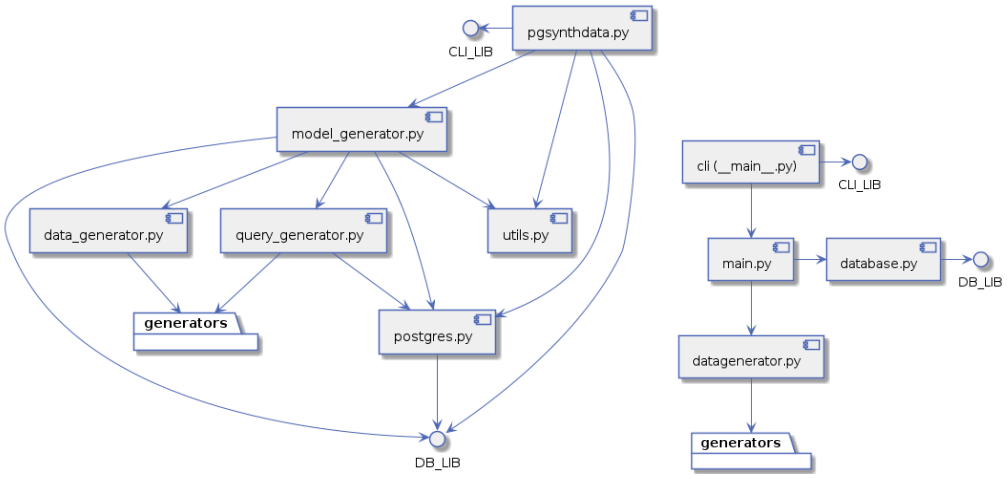
Dieser Abschnitt beinhaltet die Bewertung der Ziele, welche durch die Aufgabenstellung sowie in den funktionalen und nicht-funktionalen Anforderungen definiert sind.

5.1.1 Architektur und Refactoring

Durch das Refactoring ist ein erweiterbares und wartbares Tool entstanden. Die Aufteilung der Module nach deren Zuständigkeitsbereichen hat sich dahingehend bewährt, dass es eine klare Hierarchie und weniger Abhängigkeiten zwischen den Modulen gibt. Dadurch wird der Programmablauf nicht nur übersichtlicher und fehlerresistenter, sondern ermöglicht auch ein besseres Testing. Die Abbildung 5.1 zeigt einen Vorher/Nachher Vergleich der beiden Architektur.

Die Anzahl Module des Hauptprogramm (ohne Generatoren) konnte Dank der Konzentration auf Low-Coupling/High-Cohesion von sechs auf vier Module verringert werden. Die CLI-Logik ist nun vollständig vom Rest des Programms separiert, so dass das Tool auch als Modul in einem anderen Python-Projekt Verwendung finden könnte oder es einfacher wäre, das Tool mit einer grafischen Benutzeroberfläche zu erweitern.

Die Separation-of-Concerns macht sich vor allem bei den Generatoren bemerkbar. War die Logik zum Ansprechen der Generatoren auf mehrere Module verteilt, ist diese kompakt in einem Ort konzentriert. Ein weiteres Beispiel ist die Verbindung zur Datenbank, die jetzt über ein einzelnes Modul abstrahiert wird.



(a) Vor Refactoring (b) Nach Refactoring

Abbildung 5.1: Abhängigkeiten der Python-Module

5.1.2 Plugin-System

Die Hauptmotivation für ein Plugin-System war, dass beim Einbauen von neuen Generatoren an diversen Codestellen und über mehrere Modulen verteilt Änderungen nötig waren. Dieses Problem wird durch das Open-Close-Principle des Plugin-Systems gelöst. Die gesamte Logik eines Generators kann in einem Modul, mit definierter Schnittstelle zum Hauptprogramm, untergebracht werden.

Die Implementation der diversen Generatoren für bisher noch nicht unterstützte Datentypen hat bewiesen, dass diese Konzept funktioniert. Insbesondere die ungefilterte Übergabe aller zur Verfügung stehenden Information und Statistiken, die parametrisierbare Registrierung mit der Möglichkeit Custom-Queries auszuführen, sowie die durchdachte Auswahl der Generatoren haben sich sehr bewährt. Beispielsweise ist es nun möglich Generatoren für Arrays zu bauen, obwohl diese bislang noch nicht verwendete Informationen und Statistiken benötigen. Die spezialisierten Generatoren für Adress- oder Personendaten konnten mit wenigen Zeilen Code implementiert (Unterabschnitt 4.3.2) und komfortabel über Kommentare selektiert werden. Auch Generatoren für unbekannte UDT wie Enums oder PostGIS-Geometrien werden vom System unterstützt.

5.1.3 Array Generator

Mit der Implementierung des Array Generators für Numeric-Datentypen wurde gezeigt, dass das System auch komplexere Datentypen generieren kann. Dabei war es nötig, gewisse statistische Daten als Custom-Query abzufragen. So konnte ein Generator gebaut werden, der sehr nahe an die statistischen Eigenschaften der Originaldaten kommt. Dabei wurde einerseits darauf geachtet, dass die generierten Array-Längen (Abbildung 5.2) sowie die Häufigkeiten der generierten Elemente (Abbildung 5.3) ähnlich verteilt sind.

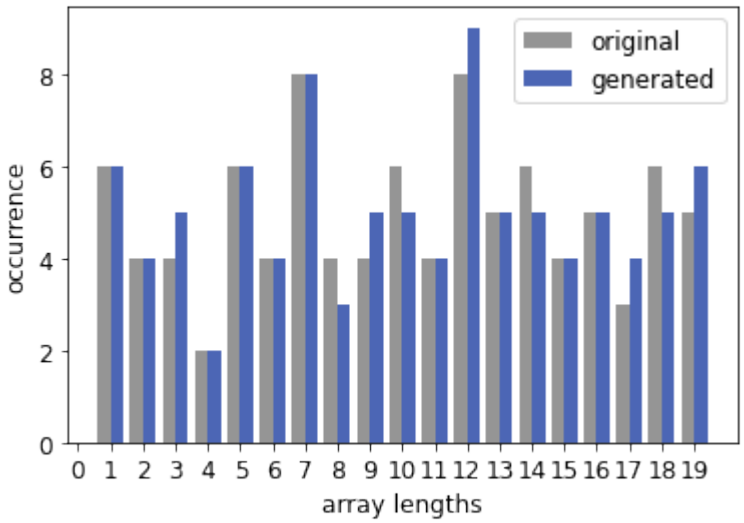


Abbildung 5.2: Histogramm der Array-Längen

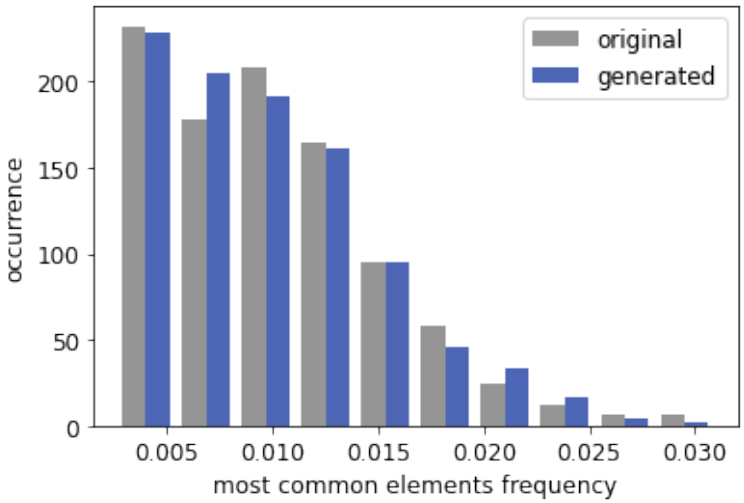


Abbildung 5.3: Histogramm der Element-Häufigkeiten

5.1.4 Spezifische Generatoren

Die Konfiguration der spezifischen Generatoren ist über Spalten-Kommentare gelöst. Dieser Ansatz ist einfach und benutzerfreundlich.

Die implementierten Generatoren für Personen- und Adressdaten haben bewiesen, dass eine einfache Integration von spezifischen Generatoren möglich ist. Das Zusammenspiel mit der Faker-Library ermöglicht mit geringem Aufwand eine

Vielzahl von solchen spezifischen Typen zu generieren.

Die implementierten Prototypen berücksichtigen UNIQUE-Beschränkungen. Eine Erweiterung auf statistische Werte wie der Anteil an `NULL`-Werten, Verteilungsdaten wie Most-Common-Values oder das Histogramm wäre denkbar.

Im Anhang F ist in der Tabelle `person_generator` die Anwendung des Generators für Eigennamen zu sehen.

5.1.5 Spatial Generator

Die Implementation eines Generators für PostGIS-Geometrietypen beweist, dass das Plugin-System auch mit UDT-Typen umgehen kann. Im Gegensatz zu den bisherigen SQL-Standardtypen war dies in zweierlei Hinsicht speziell: Zum einen kann PostgreSQL solche benutzerspezifischen und binärcodierten Typen nicht auswerten und liefert daher auch fast keine statistischen Daten. Andererseits müssen die generierten Daten beim Einfügen spezielle behandelt werden. Für die fehlenden Statistiken mussten eigene statistische Berechnungen, mit Unterstützung von PostGIS-Funktionen, per Custom-Query erstellt werden. Eine spezielle, von der Datenbankanbindung zur Verfügung gestellte, Wrapper-Klasse hilft, die Daten unverändert in die Datenbank einzufügen.

Als Proof-of-Concept werden `ST_POINT`-Typen unterstützt. Der Generator erzeugt zufällige Punkte, deren Koordinaten die gleichen Minimal- und Maximalwerte wie die Originaldaten aufweisen und diese somit im gleichen geografischen Gebiet liegen. Die Abbildung 5.4 zeigt die synthetische Datengenerierung anhand von New Yorker U-Bahnstationen. Weitere Geometrietypen können nach dem gleichen Konzept implementiert werden.

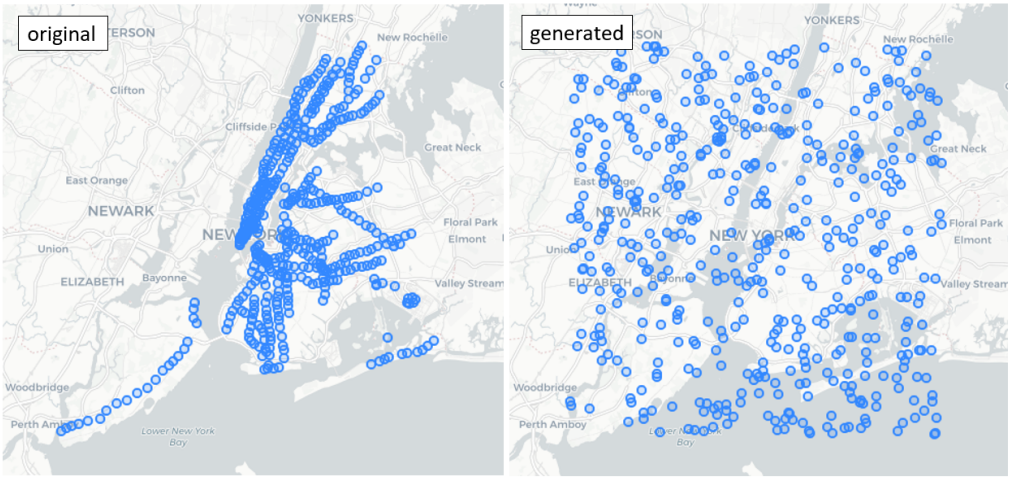


Abbildung 5.4: Originale und generierte Punktkoorinaten des Spatial Generators

5.1.6 Enumerated Generator

PostgreSQL unterstützt von sich aus Enum-Typen. Diese sind jedoch in den Colum-Informations nicht speziell als Enums gekennzeichnet, sondern werden wie andere UDT's angegeben. Trotzdem kann das System durch ausgeklügelte Registrierungsparameter und Wahl der Generatoren solche Typen unterstützen. Leidglich der Wertebereich der Enums wird per Custom-Query aus einer internen PostgreSQL-Tabelle geladen.

Im Anhang F ist in der Tabelle `enum_generator` die Anwendung dieses Generators zu sehen.

5.1.7 Usability

Die CLI Befehle wurden vom bestehenden Code weitgehend übernommen. Einzelne Anpassungen waren nötig, um die Verwendung klarer und einheitlicher zu machen. Die Usability konnte vor allem durch das umfassende Benutzerhandbuch erhöht werden. Dieses erklärt zum einen wie das Tool installiert und verwendet wird. Zum anderen gibt es eine Einsicht in die interne Struktur, um so eigene Generatoren hinzuzufügen. Ein Ausdruck des Benutzerhandbuchs ist im Anhang B zu finden.

Durch den neu definierten `show` Befehl kann das Schema und relevante Informationen einer Datenbank vor dem Generieren angeschaut werden. Darin ist auch erkennbar, welche Generatoren für die Erzeugung der synthetischen Daten gewählt werden. Mit `analyze` können nach der Generierung die Statistiken der Original- und Synthetischen-Daten verglichen werden. Man kann beim Report-Format zwischen HTML und JSON wählen. So kann man entweder den Analyse-Report direkt anschauen oder weiterverarbeiten. Die Ausgaben dieser beiden Befehle befinden sich im Anhang G und F. Die Abbildung 5.5 zeigt den nun möglichen Workflow.

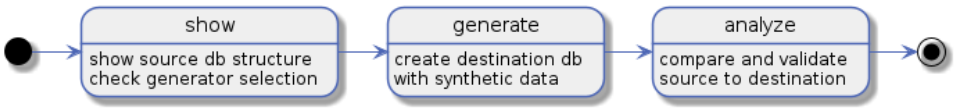


Abbildung 5.5: Möglicher Workflow für die Erzeugung synthetischer Daten mit pgsynthdata

Ausserdem wurden sämtliche Meldungen an den Benutzer überarbeitet und darauf geachtet, dass aussagekräftige Warnungen oder Fehlermeldungen an den Benutzer ausgegeben werden. Mit einem Verbose-Flag ist es möglich, erweiterte Informationen während der Ausführung der Befehle anzuzeigen.

5.1.8 Performance und Skalierbarkeit

Durch die Performance Tests (Anhang I) konnte die Skalierbarkeit der Applikation überprüft werden. Durch die gewonnenen Erkenntnisse, konnten verschiedene Verbesserungsvorschläge ausgearbeitet werden welche sich Unterabschnitt 5.3.6 befinden.

5.1.9 Testabdeckung

Durch die Entkopplung der Module konnte die Testbarkeit verbessert werden. Die totale Abdeckung durch Unit-Tests hat sich von 58% auf 75% erhöht. In den Hauptmodulen `main`, `database` und `datagenerator` wird sogar eine Testabdeckung von 94% erreicht. Niedrigere Werte gibt es für den CLI-Teil (0%), welcher aber durch die manuellen Systemtests abgedeckt wird. Sowie für Teile der Legacy-Generatoren (67%) welche nicht im Fokus dieser Arbeit lagen. Der gesamte Coverage-Report ist im Anhang H zu finden.

5.2 Einschränkungen

Damit das Tools verwendet werden kann, müssen folgenden Einschränkungen beachtet werden.

5.2.1 Voraussetzungen

Für die Originaldatenbank und den Client, auf dem die Applikation läuft, gelten die folgende Voraussetzungen:

- Die PostgreSQL Client Applications[29] müssen installiert sein
- Kein Self-Referencing von Tabellen
- Keine zusammengesetzte Unique- oder Primary-Constraints
- Alle Tabellen müssen im Schema `public` liegen

5.2.2 Allgemein unterstützte Datentypen

Für die folgenden Datentypen können synthetische Daten generiert werden:

- **Nummeric**: `integer`, `smallint`, `bigint`, `numeric`, `double precision`, `real`
- **Character**: `text`, `character varying`, `character`
- **Date/Time**: `date`, `timestamp with time zone`, `timestamp without time zone`
- **Binary**: `bytea`
- **Boolean**: `boolean`
- **Enumerated**

- **Geometric**: `point`, `line`, `polygon`, `circle`, `box`, `lseg`, `path`
- **Nummeric Arrays**: `integer[]`, `smallint[]`, `bigint[]`, `numeric[]`, `double precision[]`, `real[]`
- **Spatial (PostGIS-Geometries)**: `ST_Point`

5.2.3 Datentypen für Fremdschlüssel

Folgende Datentypen können als Fremdschlüssel verwendet werden.

- **Nummeric**: `integer`, `smallint`, `bigint`, `numeric`, `double precision`, `real`
- **Character**: `text`, `character varying`, `character`
- **Date/Time**: `date`, `timestamp with time zone`, `timestamp without time zone`

5.3 Ausblick

Das Tool ist im Stande aus verschiedenen Datenbanken synthetische Daten zu generieren. Weiterhin gibt es viele denkbare Erweiterungen, um die Funktionalität des Tools zu erhöhen sowie mehr Datentypen zu unterstützen.

5.3.1 Self-Referencing

Self-Referencing wird in Datenbanken oft verwendet. Um diese Funktionalität einzubauen, benötigt es eine zusätzliche topologische Sortierung der einzelnen Spalten einer Tabelle. Auf diese Weise kann die richtige Reihenfolge bei der Generierung sichergestellt werden. Der bestehende FK-Generator sollte danach in der Lage sein, die Fremdschlüssel auch für Self-Referencing zu generieren.

5.3.2 Zusammengesetzte Unique- und Primary-Constraints

Das Zusammensetzten von mehreren Spalten zu einem Unique- oder Primary-Constraint ist ein weit verbreitetes Vorgehen bei Datenbanken.

Um dies zu unterstützten, müssten die Ausgaben mehrere Generatoren zusammengefasst werden. Jeder Generator könnte nur ein Teil der insgesamt geforderten Anzahl an Werten generieren. Dies würde durch Permutation die benötigte Anzahl eindeutiger Werte ergeben. Ein Teil dieser Logik würde im Hauptprogramm liegen, was jedoch mit dem Konzept der Trennung der Generatorlogik und Hauptprogramm brechen würde.

Alternativ könnte die Schnittstelle zu den Generatoren entsprechend umdefiniert werden, damit Generatoren Information zu mehreren Spalten erhalten und auch

zurückgeben können. Auch in diesem Fall wären immer noch Anpassungen am Hauptprogramm-Ablauf nötig.

Der grosse Mehrwert würde aber beide Varianten rechtfertigen.

5.3.3 Datenbanken mit mehreren Schemata

Aktuell können nur Tabellen generiert werden, welche sich im `public`-Schema befinden. Um auch Tabellen aus anderen Schemas zu unterstützen, muss bei allen SQL-Statements im Modul `database` und den bestehenden Custom-Queries der Filter auf die Tabelle mit einem Filter auf das Schema ergänzt werden.

Es wäre sicherlich eine gute Idee, diese Erweiterung über die CLI parametrisierbar zu machen. Auf diese Weise könnten Einschränkungen auf bestimmte Schemata vorgenommen werden, ähnlich wie es jetzt schon für Tabellen möglich ist.

5.3.4 Parametrisierung für spezifische Generatoren

Die Syntax der Konfiguration ist so gewählt, dass zu einem späteren Zeitpunkt eine erweiterte Konfiguration möglich wäre. Als mögliche Erweiterung könnte man Parameter für die Generatoren erlauben. Beispielsweise wäre es so möglich beim Erstellen von Vornamen die Sprachregion und das Geschlecht zu berücksichtigt: `"PGSYNTHDATA[FIRST_NAME(locale=de_CH, gender=M)]"`.

5.3.5 Multiplication-Factor

Das Multiplizieren der bestehenden Daten wird zwar unterstützt, muss aber als experimentelle Funktion angeschaut werden. Es wurde aus dem bestehenden Code übernommen. Verschiedenen Test zeigten, dass die Funktion einige Mängel aufweist. So kommt es vor, dass das Generieren aussergewöhnlich lange dauert oder das Programm in eine Endlosschleife fällt. Zum Beispiel wenn der Algorithmus 1000 eindeutige Werte aus einen Wertebereich von 1 bis 100 generieren müsste.

Es benötigt ein komplettes Redesign der bestehenden Generatoren mit dem Fokus auf die Unterstützung des Multiplication-Factor. Dabei muss ein Konzept gefunden werden, wie mit solchen Problemen umgegangen werden kann, ohne dass die Statistik zu stark abweicht.

5.3.6 Performance-Verbesserungen

Die Performance Tests (Anhang I) haben gezeigt, dass in diesem Feld noch Verbesserungen möglich sind. Die folgenden drei Verbesserungen sind am erfolgver-

sprechendsten.

FK-Generator

Der auf viel Legacy-Code beruhenden FK-Generator basiert auf einem speziellen Konstrukt von verschachtelten Schlaufen. In bestimmten Situationen, kann es zu nicht deterministischen Ausführungen kommen, welche unnötig lange dauern oder sogar nie terminieren. Dieses Verhalten hat sich auch in den Performancetests (Anhang I) gezeigt. Die Logik des FK-Generators könnte aber einfacher, sicherer und effizienter geschrieben werden. Ausserdem könnte die Logik so abstrahiert werden, dass sämtliche Datentypen mit Generator für eindeutige Werte unterstützt werden.

Unterschiede innerhalb Generatoren

Wie in den Performance Tests festgestellt wurde, weist der `character_generator` extreme Laufzeitunterschiede für dieselbe Anzahl von Datensätzen auf. Bei anderen Generatoren können ähnliche Probleme auftreten.

Dieses Phänomen müsste tiefer analysiert werden. Es könnte sein, dass unterschiedliche statistische Grundlagen zu verschiedenen Durchlaufzeiten führen. Die Unterschiede sind teilweise so markant, dass ein grosses Verbesserungspotential besteht.

Multithreading

Die Datengenerierung auf Stufe Spalte und/oder Tabelle könnte parallelisiert werden. Durch die topologischen Sortierung (Unterabschnitt 3.2.3) ist bekannt, welche Tabellen gleichzeitig generiert werden dürfen. Beim parallelen Generieren von Spalten müsste allenfalls der Vorschlag zur Unterstützung von Self-Referencing (Unterabschnitt 5.3.1) beachtet werden.

5.3.7 Integration von Machine-Learning

Sowohl bei der Selektion des spezifischsten Generators wie auch bei der Generierung von Daten ist eine Integration von ML denkbar.

Durch das Plugin-System und Custom-Queries wäre das Erzeugen von Daten mit Hilfe von ML sehr einfach zu integrieren, ohne dass Anpassung am Hauptprogramm nötig sind. Durch ML-Unterstützung könnten noch genauere Modelle erzeugt oder Korrelationen zwischen den Spalten berücksichtigt werden.

Für Selektion von Generatoren mittels ML wäre eine Anpassung bei der Generatorwahl notwendig. Dafür könnten aber zum Beispiel spezifische Genera-

toren auch ohne Comment-Identifiers selektiert oder die Parameter (Unterabschnitt 5.3.4) ermittelt werden. Dies würde die konfigurationsarme Benutzung des Tools fördern.

Kapitel 6

Schlussfolgerung

Die gesetzten Ziele wurden alle erreicht und sämtliche Anforderungen inklusive einer optionalen konnten erfüllt werden.

Entstanden ist ein wartbares und erweiterbares Programm mit einem flexiblen Plugin-System und Generatoren für eine Vielzahl von Datentypen. Dies unter Berücksichtigung moderner Software-Engineering Techniken. Durch die Entkopplung der Module konnte ausserdem die Testabdeckung stark verbessert werden.

Bei der Implementation des Generators für Nummeric-Arrays wurde das Plugin-System einer ersten Bewährungsprobe unterzogen. Es hat sich gezeigt, dass diese Implementation aufgrund der ungefilterten Übergabe sämtlicher statistischen Werte problemlos möglich war. Zusätzliche statistische Werte konnten über die Custom-Queries ermittelt werden.

Die Generatoren für UDT-Typen, wie Enum und PostGIS-Geometrien konnten dank des Plugin-System in kurzer Zeit und ohne Anpassungen am Hauptprogramm implementiert werden. Sie bewiesen, dass sich das Konzept auch für solche Datentypen eignet.

Die Konfiguration der spezifischen Generatoren per Kommentar hat sich als benutzerfreundlich und unkompliziert bewährt. Die implementierten Generatoren für Echtnamen und Adressen zeigen jedoch lediglich einen möglichen Anwendungsfall auf. Diese Konfigurationsmöglichkeit bildet die Grundlage für eine nahezu unbegrenzte Zahl weiterer Anwendungsfälle. Zum Beispiel ist die Integration von Machine-Learning basierten Generatoren denkbar. Durch die Erweiterung der Konfigurations-Syntax mit Parametrisierungsmöglichkeiten, könnten noch spezifischere Daten erzeugt werden.

In Bezug auf die Benutzerfreundlichkeit wurde das Tool wesentlich verbessert. Die neuen Kommandos ermöglichen nun auch einen Workflow, für die Erzeugung synthetischer Daten mit anschliessender Analyse. Darüber hinaus sorgen das umfassende Benutzerhandbuch und die aussagekräftige Benutzermeldungen für eine bessere Usability des Programms.

Weiterhin gibt es viele denkbare Erweiterungen, um die Funktionalität des Tools zu erhöhen sowie mehr Datentypen zu unterstützen. Eine Verbesserung der bestehenden Generatoren, sowohl in Bezug auf deren Performance, als auch auf die Qualität der synthetischen Daten und ihre statistischen Eigenschaften, ist ebenfalls denkbar und sinnvoll.

Kapitel 7

Projektmanagement

Dieses Kapitel enthält relevante Information zum Projektmanagement. Nach der Beschreibung des Entwicklungsprozesses zeigt der Projektplan den Ablauf und die einzelnen Phasen. Risiken werden analysiert und Wege zur Mitigation aufgezeigt. Zum Schluss kommt die Zeitauswertung des Projektes.

7.1 Entwicklungsprozess

Der Entwicklungsprozess beschreibt, nach welcher Methodik die Entwicklung geplant und durchgeführt wurde. Dazu gehört das Dokumentieren von Entscheidungen und der Workflow bezüglich Issues und GIT-Branching-Modell.

7.1.1 Methodik

Bei der Planung wird vom Groben ins Kleine vorgegangen. Als Grobplanung dient der Projektplan mit den Meilensteinen, der die Themen und den Zeitrahmen einer Phase vorgibt. Gearbeitet wird in wöchentlichen Iterationen. Zu Beginn jeder Iteration werden die einzelnen Tätigkeiten genauer analysiert, falls nötig in Arbeitspakete aufgeteilt und mit einer Zeitschätzung versehen. So kann kontrolliert werden, ob die Arbeitslast in den Zeitrahmen passt.

Für jedes Arbeitspaket existiert ein GitLab-Issue. Dies ermöglicht eine einfache Überwachung der aufgewendeten Zeit pro Task. Zudem kann der Entwicklungsprozess nach GitFlow organisiert werden (Unterabschnitt 7.1.3).

Um den Projektstand und Fortschritt zu messen, wird das Kanban-Board von GitLab verwendet. Auf dem sind alle Arbeistpakete mit ihrem Status ersichtlich. Da für das Projekt ein definiertes Zeit-Budget einzuhalten ist, wird die aufgewendete Zeit gemäss der Beschreibung in Abschnitt 7.5 erfasst und periodisch ausgewertet.

7.1.2 Dokumentation von Design-Entscheidungen

Nach Dasanayake, Markkula, Aaramaa und Oivo (2015) sind Software-Design-Entscheidungen ein wichtiger Punkt in der Software-Entwicklung. Durch eine Umfrage wurde herausgefunden, dass die mangelnde Dokumentation eine der grössten Herausforderungen darstellt. Da solche Entscheide oft zu Beginn gemacht werden und zudem eine grosse Auswirkung auf die gesamte Lebensdauer der Applikation haben, ist es wichtig diese Entscheidungen gut zu dokumentieren.

Wichtige Entscheidungen werden deshalb mit der Y-Methode nach Zimmermann (2020) erfasst. Dadurch können Entscheidungen zu einem späteren Zeitpunkt und ohne Informations-Overhead nachvollzogen werden. Die folgende Vorlage zeigt den Aufbau einer Entscheidung nach der Y-Methode:

> Y
>
> Im Zusammenhang mit *(use case or component)*, mit der Anforderung *(non-functional concern)*, entschieden wir uns für *(option 1)* und verwerfen *(option 2)*. So erreichen wir *(quality)*. Dabei wird in Kauf genommen, dass *(consequence)*.

7.1.3 Workflow

Der Workflow ist durch gute Erfahrungen mit den GitFlow-Branching-Modell inspiriert (Pidoux, 2014).

Für jedes Feature wird ein Branch über das jeweilige GitLab-Issue erstellt. Für die Integration eines Features in den Develop-Branch muss ein Merge-Request erstellt werden. Die Merge-Requests setzten das erfolgreiche Durchlaufen der CI-Pipeline und ein Code-Review voraus. Auf dem Develop-Branch existiert also immer eine lauffähige Version der Software. Ein Merge in den Master-Branch erfolgt nur, wenn die Systemtests bestanden sind. Abbildung 7.1 zeigt ein vereinfachtes Beispiel des Workflows.

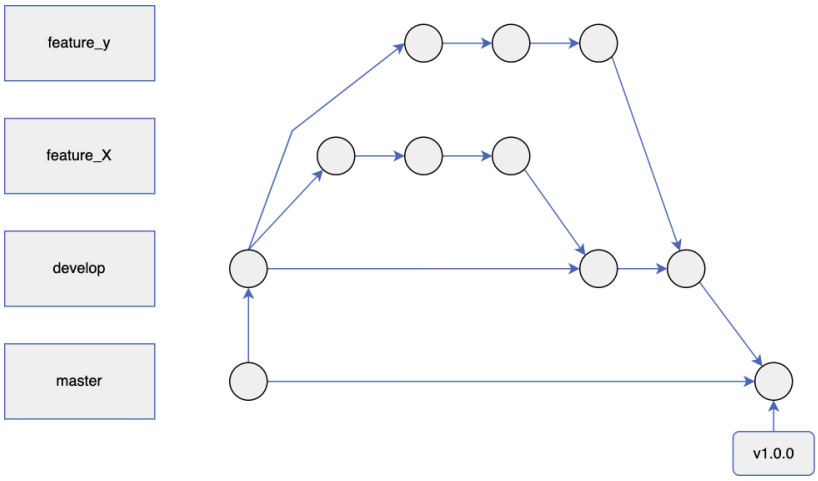


Abbildung 7.1: Git Workflow

7.2 Qualitätssicherung

Die getroffenen Massnahmen sind ein wichtige Instrumente zur Verbesserung und Sicherung der Qualität des Produkts. Die Kontrollen sollten nach Möglichkeit automatisiert werden. Um dies zu erreichen, wird die automatisierte CI-Pipeline von GitLab eingesetzt. Der Aufbau der Pipeline ist im Anhang C genauer beschrieben.

7.2.1 Code Reviews

Jedes Feature wird in einem eigenen Branch entwickelt. Beim Merge-Request in den Develop-Branch muss die andere Person ein Code-Review durchfuhren. Allfällige Fragen oder Bedenken müssen im Team abgesprochen werden, bevor der Merge bestätigt werden darf.

7.2.2 Automatisierte Unit-Tests

In der CI-Pipeline werden die Unit-Test ausgeführt. Die Pipeline fährt dabei ein PostgreSQL-Dockerimage hoch, um so auch die Datenbankschnittstelle testen zu können. Dadurch können Fehler früh erkannt und korrigiert werden. Ausserdem stellt es sicher, dass auf dem Develop-Branch immer eine lauffähige Version des Programms existiert.

7.2.3 Testabdeckung

In den selbst geschriebenen Modulen wird eine Testabdeckung der Uni-Test von 80% oder höher angestrebt. Davon ausgenommen sind Legacy-Codeteile der Generatoren deren Verbesserung nicht im Fokus dieser Arbeit stehen.

Die Testabdeckung darf nicht als Indikator für Test-Qualität missinterpretiert werden. Gerade bei interpretierten und dynamisch typisierten Sprachen wie Python ist jedoch es sinnvoll möglichst sämtliche Codeteile zu durchlaufen um Typenfehler oder ungültig Funktionsaufrufe frühzeitig und automatisiert zu erkennen.

7.2.4 Code-Formatierungsrichtlinien

Als Python-Styleguide wird PEP 8[30] definiert. Das Tool Black sorgt für deren Einhaltung. In der CI-Pipeline ist Black als Check eingebaut, sodass ein Merge-Request fehlschlägt, wenn die Formatierung korrekt wird.

Zusätzlich wird eine Sortierung der Imports von der Pipeline überprüft. Dafür ist das Tool Isort zuständig. Es sortiert und überprüft die Imports-Sortierung.

7.2.5 Statische Code-Analysen

Die beiden Tools pylint (Linter) und mypy (Static Type Check) werden für statische Code-Analysen eingesetzt um möglichst stabilen Code zu gewährleisten. Beide Werkzeuge sind im Unterabschnitt 4.4.2 genauer beschrieben.

Die Ausgaben dieser Tools liefern wichtige Verbesserungsvorschläge die regelmässig überprüft und wenn nötig behoben werden. Diese Überprüfung ist zwingend vor einem Release auf dem Master-Branch durchzuführen. Die beiden Tools werden jedoch wegen einigen False-Positives und Issues im Legacy-Code nicht in der Pipeline forciert.

7.2.6 Manuelle Systemtests

Durch die Systemtests (Unterabschnitt 4.5.2) wird das Zusammenspiel aller Komponenten und Funktionen überprüft sowie das CLI getestet. Die Tests werden manuell vor einem Release ausgeführt. Der Ablauf und die Ergebnisse sind in einem Rapport festgehalten.

7.3 Projektplanung und Meilensteine

Das Projekt ist in mehre Phasen gegliedert. Das Ziel ist eine Grobplanung welche in den einzelnen Phasen verfeinert wird. Die Phasenabschlüsse und die extern vorgegebenen Termine ergeben Meilensteine, welche in den folgenden Unterabschnitten erläutert werden. Tätigkeiten wie Dokumentation und Projektmanagement ziehen sich über das gesamte Projekt. Abbildung 7.2 zeigt den Projektplan als Gantt-Diagramm.

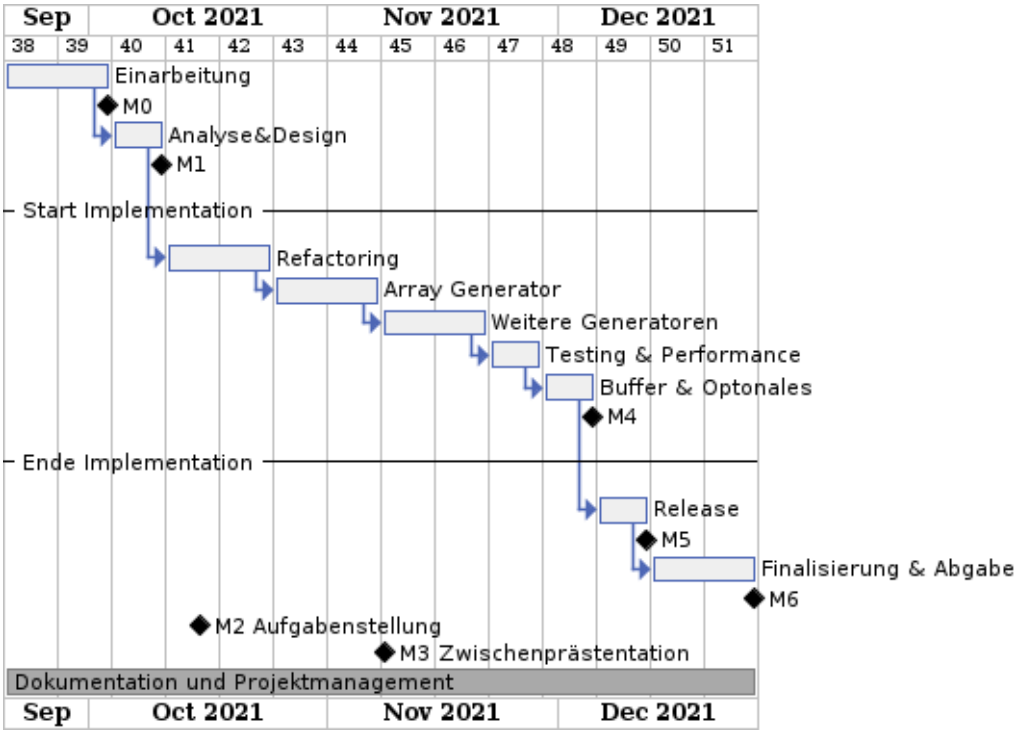


Abbildung 7.2: Gantt Projektplan

7.3.1 Einarbeitung (M0)

Zeitraum: 20.09. - 03.10.2021

Das Ziel dieser Phase ist es, sich in das Themengebiet der synthetischen Datengenerierung einzuarbeiten sowie sich mit den Technologien Python und PostgreSQL inklusive der verwendeten Entwicklungswerkzeuge und Libraries vertraut zu machen. Auch die Vorarbeiten werden betrachtet inklusive ersten Tests und Code-Inspektionen des Prototyps.

7.3.2 Analyse & Design (M1)

Zeitraum: 04.10. - 10.10.2021

Der Kontext und die Vorarbeiten werden genauer analysiert und daraus Requirements abgeleitet. Die Entwicklungsumgebung und Pipeline wird aufgesetzt und aus dem Systemkontext die Architektur designt. Ziel ist es ein technologischer Durchstich zu machen, um dadurch die Architektur zu validieren. Die einzelnen Requirements werden in der Implementierungsphase umgesetzt.

7.3.3 Einreichung Aufgabenstellung (M2)

Datum: 10.10.2021

Zu diesem Zeitpunkt wird die Aufgabenstellung von allen Parteien abgesegnet.

7.3.4 Zwischenpräsentation (M3)

Datum: 19.10.2021

Der bisherige Stand sowie Resultate werden dem Experten sowie dem Betreuer präsentiert. Das Feedback und daraus resultierende Beschlüsse werden in einem Sitzungsprotokoll festgehalten.

7.3.5 Implementation (M4)

Datum: 11.10. - 10.10.2021

In dieser Phase werden die Requirements in wöchentlichen Zyklen analysiert, wenn nötig in Subtasks aufgeteilt, designend und implementiert. In den wöchentlichen Sitzungen wird definiert, was in der nächsten Iteration zu erledigen ist und die Aufgaben in den Backlog des Issue-Boards gestellt. Zur Definition-of-Done gehörten sinnvolle Unit-Tests sowie die Dokumentation. Am Ende ist ein Puffer von einer Woche vorgesehenes. Wird dieser nicht benötigt könne in diesem Zeitraum optionale Requirements implementiert werden.

7.3.6 Testing & Release (M5)

Datum: 11.10. - 10.10.2021

In dieser Phase werden keine neuen Features mehr implementiert, sondern nur noch Systemtests und Bugfixes gemacht, um ein stabiles Produkt zu erhalten. In diese Phase gehört auch die Fertigstellung der Installationsanleitung und des Benutzerhandbuches im Repository. Der resultierende Release wird versioniert und im Repository des IFS veröffentlicht.

7.3.7 Finalisierung & Abgabe (M6)

Datum: 13.12. - 24.12.21

Die Dokumentation wird abgeschlossen und alle weiteren für die Abgabe verlangten Dokumente bereitgestellt. Da am Produkt nicht mehr gearbeitet wird, kann die Zielerreichung gemessen und Schlussfolgerungen gezogen werden. Die Metriken des Projektes und des Projektmanagements werden ausgewertet und dokumentiert. Am Ende dieser Phase erfolgt die offizielle Einreichung der Arbeit was gleichzeitig den Abschluss des Projektes darstellt.

7.4 Risikoanalyse

Die Risikoanalyse identifiziert Risiken und zeigt Massnahmen zu deren Mitigation auf.

7.4.1 Risiken

Die Tabelle 7.1 listet die Risiken auf. Diese werden im Anschluss erläutert.

Nr.	Risiko	Schaden	Eintrittsw.
1	Fehlendes Knowhow / Erfahrung in Python	Mittel	Mittel
2	Refactoring dauert zu lange	Mittel	Klein
3	Plugin-System erfüllt Anforderungen nicht	Hoch	Klein
4	Fehlendes Knowhow in Statistik & Data Science	Tief	Hoch

Tabelle 7.1: Risiken für das Projekt

Legende Schaden:

- Hoch: Führt zum Scheitern der Arbeit
- Mittel: Verringert die Funktionalität oder den Umfang
- Tief: Wenig bis keine signifikanten Auswirkungen

Legende Eintrittswahrscheinlichkeit:

- Hoch: $> 70\%$
- Mittel: $30 - 70\%$
- Tief: $< 30\%$

#1 Fehlendes Knowhow / Erfahrung in Python

Beide Teammitglieder haben wenig Erfahrung mit Python und den verwendeten Libraries. Die fehlende Erfahrung könnte zum Scheitern des Projekts führen.

#2 Refactoring dauert zu lange

Das Refactoring des bestehenden Codes dauert zu lange. Es kann kein fertiges Produkt gezeigt werden.

#3 Plugin-System erfüllt Anforderungen nicht

Das Plugin-System genügt den Anforderungen der Generatoren nicht. Es muss trotzdem in den Core-Code eingegriffen werden.

#4 Fehlendes Knowhow in Statistik & Data Science

Beide Teammitglieder geringe Erfahrung mit Statistiken und Data Science. Die generierten Daten könnten statistisch nicht komplett einwandfrei sein oder die Vertiefung in diese Themen zu viel Zeit in Anspruch nehmen.

7.4.2 Mitigation

#1 Fehlendes Knowhow / Erfahrung in Python

Genügend Zeit für Einarbeitung in die Technologien einplanen. Bei Fragen Unklarheiten so früh wie möglich Hilfe anfordern und in den wöchentlichen Sitzungen wichtige Designentscheide und Zwischenstände präsentieren. So können grobe Designfehler vermieden werden.

#2 Refactoring dauert zu lange

Refactoring und Ziel Design/Architektur gut planen. Technologischer Durchstich so früh wie möglich.

#3 Plugin-System erfüllt Anforderungen nicht

Gutes Design in der Analysephase und Prototyp bauen. Sich an Literatur und Reference-Designs orientieren. Darüber hinaus kann das Risiko durch einen frühzeitigen technologischen Durchstich erheblich verringert werden.

#4 Fehlendes Knowhow in Statistik & Data Science

Die Qualität der in den Vorarbeiten entwickelten Generatoren als ausreichend betrachten, da deren Verbesserung nicht im Fokus der Arbeit steht. Verbesserungen vorschlagen und nur umsetzten, wenn diese Zwingend nötig sind und das entsprechende Knowhow vorhanden ist.

7.4.3 Eingetretene Risiken

Es sind keine Risiken eingetreten und das Projekt konnte erfolgreich abgeschlossen werden. Dazu beigetragen haben eine realistische Projektplanung und die Massnahmen zur Risikominimierung.

7.5 Zeiterfassung

Die Zeiten werden per GitLab Timetracking[31] erfasst. Für die Auswertung wird das Tool gtt[32] verwendet und mit Hilfe eines separaten Projektes einem Rapport für das Team und die Betreuer generiert. Dieser Rapport wurde mindestens einmal pro Woche generiert und in der Sitzung besprochen, um bei Abweichungen schnell reagieren zu können.

7.5.1 Zeitauswertung pro Person

Die Tabelle 7.2 enthält die total für das Projekt aufgewendeten Stunden pro Person, sowie die prozentuale Abweichung zu der geforderten Zeit.

Name	Soll	Ist	Abweichung
Jari Elmer	240h	255h	6%
Timon Erhart	240h	257h	7%

Tabelle 7.2: Total aufgewendete Zeit pro Teilnehmer

7.5.2 Zeitauswertung pro Kategorie

Alle Arbeitspakete sind mit einer Kategorie versehen. So kann analysiert werden, für welche Tätigkeiten wie viel Zeit aufgewendet wird. In der Tabelle 7.3 können die dazugehörigen absoluten und relativen Werte nachgeschlagen werden.

Kategorie	**Zeit**	**Anteil**
Dokumentation	244h	48%
Implementation	159h	31%
Infrastruktur CI/CD	7h	1%
Projektmanagement & Meetings	70h	14%
Technologische Einarbeitung	32h	6%

Tabelle 7.3: Zeit und Anteil pro Kategorie

7.6 Sitzungen

Es werden wöchentliche Sitzungen abgehalten. Dabei wurden die folgenden Punkte besprochen:

- Was wurde in der aktuellen Woche erreicht
- Gab es Fragen/Unklarheiten
- Welche Tätigkeiten sind in der nächsten Woche zu erledigen

Die Sitzungsprotokolle werden den Betreuern in der Abgabe zur Verfügung gestellt.

Anhang A

Aufgabenstellung

IFS INSTITUTE FOR SOFTWARE

OST Ostschweizer Fachhochschule

Synthetische Datengenerierung aus PostgreSQL für PostgreSQL

Studienarbeit im Herbstsemester 2021, Bachelor-Studiengang Informatik

Einleitung

Die zunehmende Durchdringung von Machine-Learning und Data-Engineering des heutigen Alltags führt zu einer erhöhten Nachfrage nach grossen Datensätzen. In vielen Bereichen ist es kaum möglich, rasch und unkompliziert tagesaktuelle Datensätze zu beziehen, da diese oft mühsam von Hand anonymisiert werden müssen, um Bedenken und Gesetzen bzgl. Datenschutz gerecht zu werden. Insbesondere Datensätze mit personenbezogenen Daten, seien sie finanzieller Natur oder Gesundheits-Informationen, unterliegen höheren Schutz-Anforderungen.
Hier bieten sich synthetische Daten an, welche basierend auf Real-Daten so generiert werden, dass sie bestimmte Aspekte wie statistisches Verhalten oder Personennamen der Originaldaten beibehalten. Solche synthetische Daten bieten den Vorteil, dass sie keine Rückschlüsse zur Realität mehr erlauben und sich dennoch für realitätsnahe Auswertung sowie für Modell-Training eignen.

Aufgabe

Diese Arbeit führt das bestehende Programm in einen wartbaren, erweiterbaren und einfachen nutzbaren Zustand über. Die Weiterentwicklung hat den Fokus auf Demonstration des modularen Aufbaus, durch das Hinzufügen von bisher noch nicht unterstützten Datentypen. Das Programm soll auf beliebigen PostgreSQL-Datenbanken angewendet werden können (mit Ausnahme von nicht unterstützen Datentypen sowie definierten Einschränkungen). Wenn möglich sollen bestehende Libraries und Tools verwendet werden, um einen grösseren Umfang zu demonstrieren.
Wünschenswert ist es, bestehende Literatur hinzuzuziehen und allfällige darauf basierte Funktionen hinzuzufügen.

Vorgaben – Technologien – Rahmenbedingungen

- PostgreSQL (mit Test-Tabellen)
- Python mit pytest

Vorgehensskizze

- Aufgaben: Einarbeiten. Umgebung aufsetzen. Modularer Aufbau erstellen. Bestehende Funktionalität einbauen.
- Theorie: Verschiedene Ansätze und Probleme.
- Neue Typen und Arten unterstützen. Verbessern der Nutzbarkeit. Anwendbarkeit auf andere PostgreSQL-Datenbanken als den Beispiel-Datenbanken.

Termine und Bewertungsschema

Es gelten die üblichen Termine und Regelungen zum Ablauf und zur Bewertung der Arbeit (fünf Aspekte) des Bachelor-Studiengangs Informatik der OST. Zusätzlich wird Gewicht gelegt auf moderne Softwareentwicklung (Versioning, Continuous Testing & Integration, etc.) sowie auf eine lauffähige, getestete Software.

Beteiligte

Diplomanden: Jari Elmer und Timon Erhart
Betreuer: Nicola Jordan, Laborleiter und Prof. Stefan Keller, Institut für Software OST

Anhang B

Installationshinweise und Benutzerhandbuch

Dies ist ein Auszug des README.md File aus dem Code-Repository. Die aktuelle Version kann auf dem öffentlichen GitLab-Repository[33] eingesehen werden.

Pure Synthetic Data Generation for PostgreSQL

Description

`pgsynthdata` is a lightweight tool for PostgreSQL written in Python. It can generate purely synthetic data from existing databases that have the same statistical properties as the original data. The result is a generated database with the same structure as the original, but with synthetic data.

The synthetic data is generated using the information provided by the pg_stats view. More explicitly by using statistical information such as, most common values, their frequencies, histogram, fraction of nulls, the number of distinct values etc... From this information, the generators create a statistical model that produces fully synthetic data that do not contain actual values or fragments of the "real" data but are very similar in terms of "shape" and their properties.

All this is done without any cumbersome configuration. However, several specialized generators can be used with very little configuration effort to generate even more realistic data, e.g. person names, street names or postal codes by using column comments

Table of Contents

- Pure Synthetic Data Generation for PostgreSQL
  - Description
  - Table of Contents
  - Installation
    - Prerequisite
    - Stable version from PyPi
    - Clone repository
  - Constraints
  - Usage
    - General Options
    - Commands
      - Generate
      - Show
      - Analyze
    - Configuration for specific Data Generators
  - Customization "build your own generators"
    - Overview and plugin-system
    - Generator registration decorator
      - Data type
      - UDT names only (optional)
      - Custom queries (optional)
      - Comment identifier (optional)
    - Generator function signature
    - Testing
      - Test database
      - Test pickle
      - Run tests
      - Systemtests
    - Contributions
  - Authors

Installation

Prerequisite

Make sure you have these two components ready for this tool to work:

1. You must have the PostgreSQL Client Applications **localy** installed

   > You can use the console command `pg_dump --version` and `pg_restore --version` to check if the needed utilities are installed

2. You should have a database for which you want to generate data. The database can exist locally or be accessible via a PostgreSQL docker container.

   > Don't have a database available? Look at the Testing section. There you will find an example of how to set up a test database with Docker.

Stable version from PyPi

```
pip install pgsynthdata
```

After installation, you can run the program from your console. See chapter usage for more information

Clone repository

If you decide to clone this repo instead of installing it from PyPi, all necessary dependencies can be installed with Poetry:

```
poetry install
```

Use the following command to run the application:

```
poetry run python -m pgsynthdata CLI_ARGUMENTS
```

Constraints

There are some limitations regarding the database structure and supported data types.

1. Currently these PostgreSQL types are supported for columns which do not have any external constraints

- Nummeric: `integer`, `smallint`, `bigint`, `numeric`, `double precision`, `real`
- Character: `text`, `character varying`, `character`
- Date/Time: `date`, `timestamp with time zone`, `timestamp without time zone`
- Binary: `bytea`
- Boolean: `boolean`
- Enumerated:
- Geometric: `point`, `line`, `polygon`, `circle`, `box`, `lseg`, `path`
- (Nummeric:) Arrays `integer[]`, `smallint[]`, `bigint[]`, `numeric[]`, `double precision[]`, `real[]`
- Spatial (PostGIS): `ST_Point`

2. Data with **foreign key** constraints can only be of *Numeric*, *Character* or *Datetime* type (and its subtypes and variations)

3. Currently, **composite unique keys** are not supported. Also, no tables with **self-references** are supported

Usage

```
pgsynthdata [GENERAL_OPTIONS] SOURCE_DATABASE COMMAND [COMMAND_OPTIONS]
```

The first thing the tool does after each command is to ask you for the database password. This will never be stored or logged on the console or anywhere else.

General Options

These options are valid for all commands

General Option	Description
`-h/--host`	Hostname of the Database (default: localhost)
`-p/--port`	Port of the Database (default: 5432)
`-u/--user`	User of the Database (default: postgres)
`-t/--tables`	Name(s) of table(s) to be filled with data, separated by `:` (default: all tables)
`-v/--verbose`	Turn on verbose mode

Commands

There are three commands, which are explained below

Generate

Generate is the main command. It generates a database with the same structure but purely synthetic data from an original database.

```
pgsynthdata [GENERAL_OPTIONS] SOURCE_DATABASE generate DEST_DATABASE [COMMAND_OPTIONS]
```

Command Option	Description
`-m/--m_factor` *	Multiplication factor for the number of lines to be generated (default: 1.0)
`-o/--owner`	Owner of the created destination database
`-d/--drop`	Drop the existing destination databases first (default: false)

> *Remarks for `-m/--m_factor`: This feature is experimental and there are many reasons this can fail, e.g., when unique data must be generated but the range is smaller than the number of values to generate

Here are some examples:

```
pgsynthdata mydb generate mydb_gen
pgsynthdata -t "table_a:table_b:table_c" mydb generate mydb_gen --drop
pgsynthdata -u testuser -p 1234 --verbose mydb generate mydb_gen -d -m 2.0 --owner testuser2
```

Show

Show can be used to display the structure of a database. It shows relevant information for the database, such as the data type, whether it is a foreign key/nullable/unique, which generator is used, and column comments. This is especially useful before generation. There are no 'COMMAND_OPTIONS'.

```
pgsynthdata [GENERAL_OPTIONS] SOURCE_DATABASE show
```

Here are some examples:

```
pgsynthdata mydb show
pgsynthdata -u testuser mydb show
```

Analyze

Analyze compares the original database with the generated database in terms of statistical key properties. It retrieves the relevant statistical information for the databases from `pg_stats` and saves it as either HTML or JSON. The file will be saved in the current working directory as `Analyze ORIGINAL_DATABASE GENERATED_DATABASE.html/json`. The original and generated column will be adjacent to allow easier comparison of the values. This is useful for validation after generation.

```
pgsynthdata [GENERAL_OPTIONS] ORIGINAL_DATABASE analyze GENERATED_DATABASE [COMMAND_OPTIONS]
```

Command Option	Description
`-f/--format {html/json}`	Output format (default: html)

Here are some examples:

```
pgsynthdata mydb analyze mydb_gen
pgsynthdata mydb analyze mydb_gen --format=json
```

Configuration for specific Data Generators

There are several specific data generators. For the tool to use these generators, a SQL comment must be present at this column.

For example, there is one that can generate real first names. The generator has the identifier `PERSON_FIRST_NAME`. To use it, the comment in that column must contain `PGSYNTHDATA[PERSON_FIRST_NAME]` (including the square brackets). Any other characters may be placed before and after `blah blah PGSYNTHDATA[PERSON_FIRST_NAME] blah blah`.

You can easily enter comments via pgAdmin, or as SQL command. More information about PostgreSQL comments can be found here.

```
COMMENT ON COLUMN test_table.name IS 'PGSYNTHDATA[PERSON_FIRST_NAME]';
```

Until now, these specific generators are available:

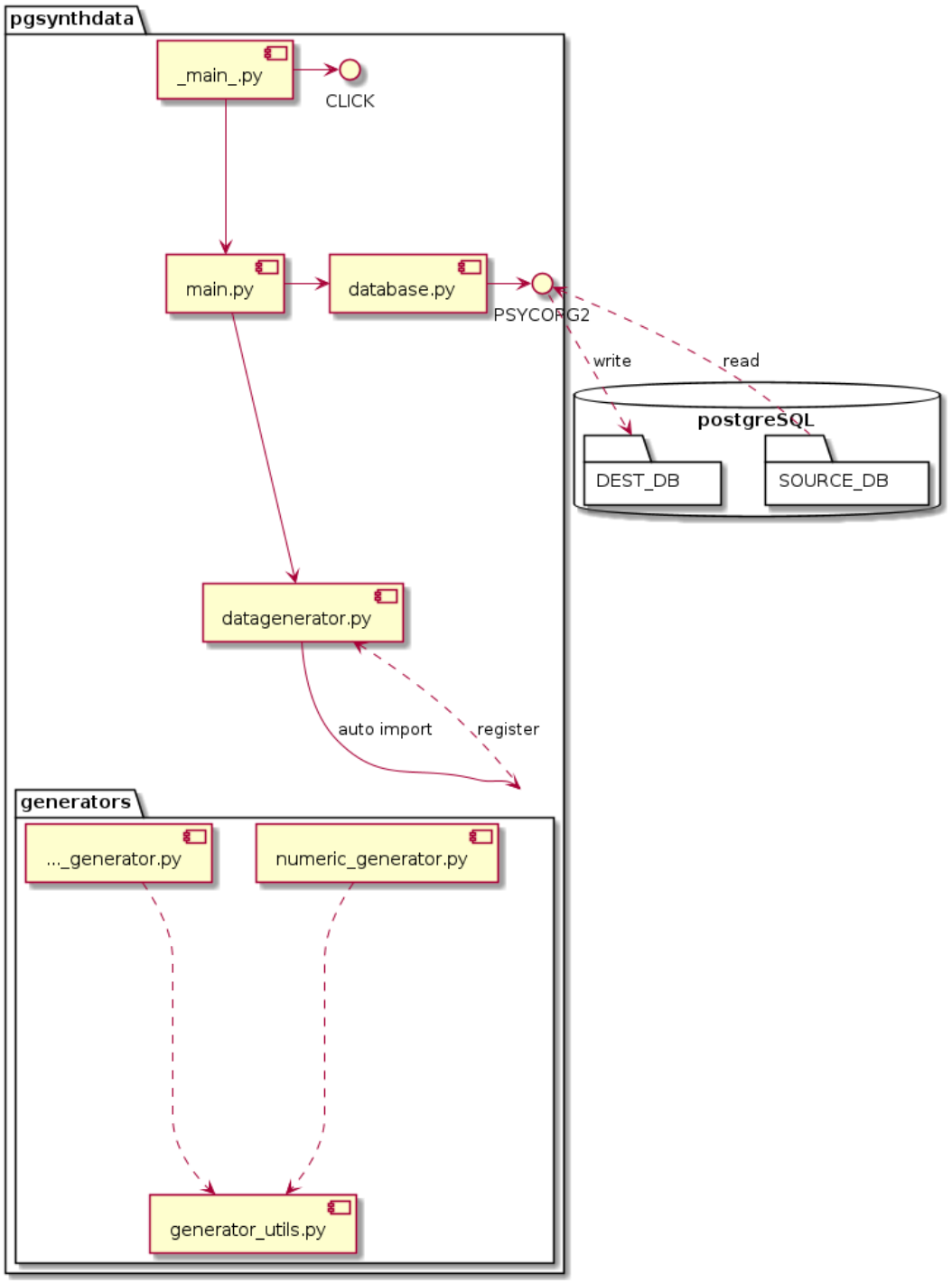
Generator identifier	Description
PERSON_FIRST_NAME	Generates real first names, e.g. "Rick"
PERSON_LAST_NAME	Generates real last names, e.g. "Sanchez"
PERSON_NAME	Generates real full names, e.g. "Rick Sanchez"
PERSON_PREFIX	Generates real name prefixes
PERSON_SUFFIX	Generates real full suffixes
ADDRESS_STREET	Generates real street names, e.g. "Oberseestrasse"
ADDRESS_ZIP	Generates real street zips, e.g. "8640"
ADDRESS_BUILDING_NR	Generates real building numbers, e.g. "10"
ADDRESS_CITY	Generates real city names, e.g. "Rapperswil"
ADDRESS_STREET_ADDRESS	Generates real addresses, e.g. "Oberseestrasse 10"
ADDRESS_COUNTRY	Generates real country names, e.g. "Schweiz"
ADDRESS_COUNTRY_CODE	Generates real country codes, e.g. "CH"
COPY	The original data is taken one-to-one (but shuffled)
IGNORE	No data generated (filled with the default type, most likely `NUll`)

Customization "build your own generators"

Overview and plugin-system

The system allows easy integration of new generators. To add a generator, you only have to place a Python-file which name ends with `..._generator.py` it in the `./pgsynthdata/pgsynthdata/generators` subfolder. The file must contain at least one decorated generator-function. The plugin-system will then automatically recognise and import it. Helper functions that are useful for all generators are placed in the `generator_utils.py` module.

System overview:



Generator registration decorator

The registration an configuration of your generator is done via a python decorator. This `@generator` registration decorator can be imported from the `datagenerator` module. You can then specify the data type, restrict it to specific `udt_names`, define your own `custom_queries` and specify your own `comment_identifier` for specialized generators. Only the first data type parameter is mandatory, all others are optional and can be used in any combination.

For every column, the most specific generator is chosen by this logic:

1. Generators which have a matching `comment_identifer` (and matching `data_type` and `udt_names`)
2. If no comment identifier is given or matches, the `data_type` and `udt_names` are..
3. If no `data_type` is matching, it looks for a `None` type and matching `udt_names`

Data type

The first parameter of the `@generator` registration decorator defines the data type for which the generator is selected. Provide a single type e.g. `"integer"` or a set of types `{"integer", "bigint"}`. The data type corresponds to the `data_type` field of the PostgreSQL column infos. This is the only mandatory parameter.

Example: The following generator will only be used for certain date/time types

```
@generator({"date", "timestamp without time zone"})
def datetime_generator(...):
```

You can provide `None` for this parameter, which means that it **matches every type** (only useful in combination with other parameters). An example that uses this is the `copy_generator.py`.

UDT names only (optional)

Optionally, provide a set of UDT names for which the generator is valid. The UDT name corresponds to the `udt_name` field of the PostgreSQL column infos.

Example: The following generator is used only for Arrays with one of the given user defined types.

```
@generator(
    "ARRAY",
    udt_names_only={"_int2", "_int4", "_int8", "_numeric"},
)
def numeric_array_generator(...):
```

If no `udt_names_only` is provided, the generator matches all UDT's.

Custom queries (optional)

Sometimes, the information and statistics provided by PostgreSQL are not sufficient. Therefore, it is possible to provide a list of custom queries that will be executed for each column that uses this generator. Custom queries are wrapped in the `StatisticQuery` class provide by the datagenerator module.

The results of the custom queries are provided via the `stats` parameter of the generator function. There you will find a dict named `custom_stats` containing the results of all custom queries. The dict-key is equal to given `name`. The Result itself is a `List[DictRow]`. Further Information about the `DictRow` class can be found on the documentation of psycopg2.

```
from ..datagenerator import generator, StatisticQuery, Statistics
@generator( "sometype",
            custom_queries=[
               StatisticQuery(name="absolut_min", query="SELECT min({column_name}) as abs_min FROM {table_n
               StatisticQuery(name="absolut_max", query="SELECT max({column_name}) as abs_max FROM {table_n
```

```
        ])
def sometype_generator(stats: Statistics, ...):
  # access custom query result


  absolut_min = stats.custom_stats.get("absolut_min")[0]["abs_min"]
  absolut_max = stats.custom_stats.get("absolut_max")[0]["abs_max"]
```

There are 4 placeholders that you can use in your query:

Placeholder	Description
`{column_name}`	Column Name as SQL Identifier. Use it for example like this: `SELECT {column_name} ...`
`{table_name}`	Table Name as SQL Identifier. Use it for example like this: `SELECT ... FROM {table_name}`
`{column_name_str}`	Column Name as SQL String. Use it for example like this: `SELECT ... WHERE column_name = {column_name_str}`
`{table_name_str}`	Table Name as SQL String. Use it for example like this: `SELECT ... WHERE table_name = {column_name_str}`

Comment identifier (optional)

If you want to be able to select your generator by the comments, set the `comment_identifier` parameter. This generator will then be selected for all columns that have this comment identifier (and that match the other specified parameters sus as `data_types` and `udt_names_only`).

Example: The following generator is selected for all columns with the comment `PGSYNTHDATA[MY_IDENTIFIER]` and data type `integer`

```
@generator( "integer",
            comment_identifier="MY_IDENTIFIER"
          )
def my_identifier_generator(...):
```

Generator function signature

Four arguments are passed to a generator. These arguments should provide you with all the necessary information and statistics to create synthetic data from them:

```
from ..datagenerator import generator, Constraints, Statistics, ColumnPgInfo

@generator(...)
def mytype_generator(
    stats: Statistics,
    column_info: ColumnPgInfo,
    constraints: Constraints,
    rows_to_gen
) -> List[Any]:
```

```
    # generate data ...
    return generated_data # return a list with the size of 'rows_to_gen'
```

- `stats`: Contains all the statistical information gathered for this column. In `stats.pg_stats` everything from the pg_stats table is accessible. In `stats.custom_queries` you can find the results of you custom queries.
- `column_info`: Contains the information provided by information_schema.columns table about the column for which to generate data.
- `constraints`: Aggregated constraint information. Now only the `is_unique` Field of this object is interesting for the normal generators. All the other fields are used especially by the `fk_generator.py` for foreign key generation.
- `rows_to_gen`: Exact number of rows which the generator must deliver.

The `Statistics`, `ColumnPgInfo` and `Constraints` data classes provide easy access to the information and allow auto-completion from your IDE. Examples:

```
col_name = column_info.column_name
null_frac = stats.pg_stats.null_frac
```

Extended information:

- If you for some reason can't return any values you can return a single `None`. If the column is nullable, the datagenerator module will then log a warning and fill the column with the default value of the type (most likely `NULL`). If it isn't nullable, the process will be terminated with an error message.
- If you must return a special or custom type that is code like a string, you can use the `AsIs` wrapper class. The string gets then placed in the INSERT query "as it is" and not as `'text'`. Example:

```
col_name = column_info.column_name
null_frac = stats.pg_stats.null_frac
```

- Use the `logging` module to log things (only shown with the `--vergbose` flag) or warnings (always shown). Example:

```
import logging
#...
logging.info("hello there:)") # only with '--verbose' flag
logging.warning("hey, important!") # allways shown
```

Testing

Test database

The tests attempt to connect to a test database.

A SQL dump of the `pgsynthdata_testdb` database can be found in `./pgsynthdata/test_resources` folder. This database should be used to run the unit tests. You can import the database via pgAdmin or by using these PostgreSQL CLI commands:

```
psql -U postgres -h localhost -p 5432 -c "create database pgsynthdata_testdb"
psql -U postgres -h localhost -p 5432 -v ON_ERROR_STOP=on -d "pgsynthdata_testdb" -f "mydb/pgsynthdata_test
```

To spin up the test database, you can use the following docker command:

```
docker run --rm -e POSTGRES_DB=pgsynthdata_testdb -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -
```

Test pickle

A Python pickle of the test database is used to test the generators even without connection to the test database. It contains the serialized structure of the test database.

If a change is made to the test database, the `./test/testdb_pickler.py` file should be executed to update the test pickle.

Run tests

You can run the unit tests with `poetry run pytest`. The unit tests attempt to connect to the test database described above. For that it is also required to have a `.env` file in the root folder (`./pgsynthdata`) with the following environment variables set. However, if the database is not reachable, these tests are skipped.

```
TEST_DBNAME=pgsynthdata_testdb
TEST_USER=postgres
TEST_PW=postgres
TEST_HOST=localhost
TEST_PORT=5432
```

Systemtests

The folder `./pgsynthdata/test_resources` contains a protocol and necessary files and for manual system tests

Contributions

Improvements and enhancements are welcome! Please write sensible unit tests for your code and run `poetry run black .` , `poetry run isort .` before you create a Pull Request.

Authors

- Timon Erhart
- Jari Elmer

Based on preliminary work by Etienne Baumgartner, Kevin Ammann and Labian Gashi.

Anhang C

Hinweise zu CI/CD

Es wird gitlab-ci verwendet, Listing C.1 zeigt die komplette Konfiguration. Es gibt zwei verschiedene Stages. Die erste behandelt alle Unit-Test (`test`) und die zweite übernimmt die Code-Qualitätsprüfung (`lint`).

Für die Unit-Tests wird zusätzlich ein PostgreSQL-Service gestartet. So können auch Verbindungstests zur Datenbank ausgeführt werden. Die Unit-Tests werden in Form eines JUnit Reports GitLab als Artefakt zur Verfügung gestellt (`report.xml`). So können bei einem Merge-Request gleich die Testresultate angeschaut werden. Bei der Ausführung der Unit-Tests wird gleichzeitig die Code-Coverage berechnet. Das Total der Coverage wird von GitLab aus dem Job-Log geparsed[34]. Für die Test-Coverage-Visualization[35] wird ein `coverage.xml` Artefakt erstellt.

Die Code-Qualität überprüft `black` und `isort`. Sie müssen ohne Fehler ihren Check abschliessen, damit die Pipeline den Status `success` erhält.

```yaml
image: python:3.9

stages:
  - test
  - lint

before_script:
  - pip install poetry
  - poetry config virtualenvs.create false
  - poetry install

Unit Tests:
  stage: test
  services:
    - postgres:13-alpine
  variables:
    # Variables for postgres service
    POSTGRES_DB: synthdata_test
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_HOST_AUTH_METHOD: trust
    # Variables for our script
    TEST_DBNAME: synthdata_test
    TEST_USER: postgres
    TEST_PW: postgres
    TEST_HOST: postgres
    TEST_PORT: 5432
  script:
    - poetry run coverage run -m --source=pgsynthdata pytest
    ↪  --junitxml=report.xml tests
    - poetry run coverage xml
    - poetry run coverage report
  artifacts:
    when: always
    reports:
      junit: report.xml
      cobertura: coverage.xml

Python Code Lint:
  stage: lint
  script:
    - poetry run black --check .

Import Sort Check:
  stage: lint
  script:
    - poetry run isort -c .
```

Listing C.1: GitLab CI Pipeline

Anhang D

Codemetriken

Codezeilen

Die selbst geschriebene Codebasis für Hauptablauf und implementierte Generatoren umfasst **2'345** Codezeilen. Dazu kommen die Unit-Tests welche weitere **1'078** Codezeilen umfassen. Die übernommenen Legacy-Generatoren betragen **461** Codezeilen.

Unit-Tests

Es existieren **71** Unit-Tests wobei **140** `asserts` ausgeführt werden und bei **10** Tests werden Exceptions erwartet.

Pushed Commits auf den develop-Branch

Es wurden **459** Commits auf den develop-Branch gepushed. Da teilweise bei den Merge-Requests die Funktion "Squash commits" verwendet wurde, kann davon ausgegangen werden, dass die wirkliche Anzahl an getätigten Commits noch um einiges höher liegt.

Merge Requests

Über die gesamte Projektdauer wurden **72** Merge Requests eröffnet und **67** davon gemergt.

Anhang E

Manuelle Systemtests

Dies ist die aktuelle Version des Testprotokolls des Systemtests für den Release vom 20.12.2021. Das gesamte Testprotokoll mir früheren Systemtests kann auf dem öffentlichen GitLab-Repository[36] heruntergeladen werden.

Es gibt noch zwei Tests, welche den Status `error` erhalten haben. Dabei handelt es sich um Tests mit dem Multiplication-Factor. Diese Thematik wurde in den Resultaten im Unterabschnitt 5.3.5 behandelt.

Testprotokoll Systemtest ***Testprotokoll-Version: 1.0***

Projekt pgsynthdata
Version develop 3ee0426aee5e2468b648ef7c632013a5f7783cec
Testdatum 20.12.21
Tester Jari Elmer

Legende 'Befund':
Test bestanden
Test bestanden, Verbesserungen sind möglich
Test nicht bestanden

Allgemeine Vorraussetzungen:
- PostrgreSQL mit CLI-Tools lokal installiert
- pgsynthdata_testdb importiert
- pgsynthdata installiert

Nr	Kategorie	Vorraussetzung	Testschritte	erwartetes Resultat	Befund
1	Stabilität	Leere DB (ohne Tabellen)	Generate darauf aufrufen: *poetry run python -m pgsynthdata pgsynthdata_systest_emptydb generate pgsynthdata_systest_emptydb_gen*	Leere DB wird erstellt (keine Exception oder Errormeldung)	Ok
2	Usability	Leere DB (ohne Tabellen)	Show darauf aufrufen:	Leere DB wird mit sinnvoller Meldung ausgegeben (keine Exception oder Errormeldung)	Ok
3	Usability	Leere DB (ohne Tabellen)	Analyze darauf aufrufen:	Leere DB-Analyse wird mit sinnvoller Meldung generiert (keine Exception oder Errormeldung)	Ok
4	Stabilität	Leere Tabelle (ohne Rows)	Generate darauf aufrufen: *poetry run python -m pgsynthdata pgsynthdata_systest_emptytable generate pgsynthdata_systest_emptytable_gen*	Leere Tabelle wird erstellt (keine Exception oder Errormeldung)	Ok
5	Usability	Leere Tabelle (ohne Rows)	Show darauf aufrufen	Leere Tabelle wird mit sinnvoller Meldung ausgegeben (keine Exception oder Errormeldung)	Ok
6	Usability	Leere Tabelle (ohne Rows)	Analyze darauf aufrufen	Leere DB-Analyse wird mit sinnvoller Meldung generiert (keine Exception oder Errormeldung)	Ok
7	Stabilität	Leere Tabelle (mit leeren Rows)	Generate darauf aufrufen: *poetry run python -m pgsynthdata pgsynthdata_systest_emptyrow generate pgsynthdata_systest_emptyrow_gen*	Leere Tabelle wird erstellt (keine Exception oder Errormeldung)	Ok
8	Usability	Leere Tabelle (mit leeren Rows)	Show darauf aufrufen	Leere Tabelle wird mit sinnvoller Meldung ausgegeben (keine Exception oder Errormeldung)	Ok
9	Usability	Leere Tabelle (mit leeren Rows)	Analyze darauf aufrufen	Leere DB-Analyse wird mit sinnvoller Meldung generiert (keine Exception oder Errormeldung)	Ok
10	Funktionalität	Tabelle mit Daten jedoch ohne FK	Generate darauf aufrufen: *poetry run python -m pgsynthdata pgsynthdata_systest_nofk generate pgsynthdata_systest_nofk_gen*	Tabelle wird erzeugt und enthält korrekte synth. Daten	Ok
11	Stabilität	Tabelle mit leeren Arrays (ARRAY[]::int[] erzeugen)	Generate darauf aufrufen:	Leeres Array wird generiert	Ok
12	Stabilität	Bei einem Generator fehlerhafte CustomQuery einfügen (PosgreSQL Exception): *StatisticQuery(name="systemtest_sql_error", query="SELECT x FROM {table_name}")*	Generate darauf aufrufen: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systest_querryerror*	Warning/Errormeldung. Row wird nicht generiert (enthält NULL). Keine Exception (ausser Row ist UNIQUE und kann desswegen nicht generiert werden)	Ok
13	Usability		Generate darauf aufrufen mit falschem DB-Name: *poetry run python -m pgsynthdata x generate pgsynthdata_systest_wrongdbname*	Saubere Errormeldung. Keine Exception	Ok

14	Usability	Offene Verbindung zu pgsynthdata_testdb_gen (zB über pgAdmin)	DEST_DB von einer anderen Verbindung (zB pgAdmin) besetzt: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_testdb_gen* --drop	Saubere Errormeldung. Keine Exception	Ok
15	CLI allg. Options		Passwort korrekt eingegeben: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systests_pw*		Ok
16	CLI allg. Options		Passwort falsch eingegeben: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systests_wrongpw*		Ok
17	CLI allg. Options		Username-Option mit gültigen User: *poetry run python -m pgsynthdata -u postgres pgsynthdata_testdb generate pgsynthdata_systests_user*	Normale Funktion. DB wird generiert	Ok
18	CLI allg. Options		Username-Option mit ungültigem User: *poetry run python -m pgsynthdata -u x pgsynthdata_testdb generate pgsynthdata_systests_wronguser*	Saubere Erromeldung. Keine Exception	Ok
19	CLI allg. Options		Port-Option mit gültigen Port: *poetry run python -m pgsynthdata -P 5432 pgsynthdata_testdb generate pgsynthdata_systests_port*	Normale Funktion. DB wird generiert	Ok
20	CLI allg. Options		Port-Option mit ungültigem Port: poetry run python -m pgsynthdata -P 9999 pgsynthdata_testdb generate pgsynthdata_systests_wrongport	Saubere Erromeldung. Keine Exception	Ok
21	CLI allg. Options		Host-Option mit gültigen Host: *poetry run python -m pgsynthdata -H localhost pgsynthdata_testdb generate pgsynthdata_systests_host*	Normale Funktion. DB wird generiert	Ok
22	CLI allg. Options		Host-Option mit ungültigem Host: *poetry run python -m pgsynthdata -H x pgsynthdata_testdb generate pgsynthdata_systests_wronghost*	Saubere Erromeldung. Keine Exception	Ok
23	CLI allg. Options		Tables-Option mit einer gültigen Table: poetry run python -m pgsynthdata -t numeric_generator pgsynthdata_testdb generate pgsynthdata_systests_1tableonly	Normale Funktion. DB wird mit dieser Tabelle generiert	Ok
24	CLI allg. Options		Tables-Option mit einer ungültigem Table: *poetry run python -m pgsynthdata -t x pgsynthdata_testdb generate pgsynthdata_systests_wrong1tableonly*	Saubere Erromeldung. Keine Exception	Ok
25	CLI allg. Options		Tables-Option mit drei gültigen Tables: *poetry run python -m pgsynthdata -t numeric_generator:datetime_generator:character_generator pgsynthdata_testdb generate pgsynthdata_systests_3tableonly*	Normale Funktion. DB wird mit den 3 Tabellen generiert	Ok
26	CLI allg. Options		Tables-Option mit drei Tables, eine ungültig: *poetry run python -m pgsynthdata -t numeric_generator:datetime_generator:x pgsynthdata_testdb generate pgsynthdata_systests_wrong3tableonly*	Saubere Erromeldung. Keine Exception	Ok
27	CLI allg. Options		Tables-Option mit einer Table mit FK-Abhängikeit: *poetry run python -m pgsynthdata -t fk_generator pgsynthdata_testdb generate pgsynthdata_systests_fktableonly*	Saubere Erromeldung. Keine Exception	Ok
28	CLI allg. Options		Verbose-Option: *poetry run python -m pgsynthdata --verbose pgsynthdata_testdb generate pgsynthdata_systests_verbose*	Verbose-Meldungen werden ausgegeben. Vergleich mit Ausgabe ohne verbose	Ok
29	CLI allg. Options		Ohne Verbose-Option	Start und Ende wird ausgegeben	Ok

30	CLI generate Options	Owner "chris" muss existieren	Owner-Option gesetzt: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systests_owner --owner chris*	Generierte DB hat Owner "Chris"	Ok
31	CLI generate Options	DB "pgsynthdata_testdb_gen" existiert schon	Drop-Option nicht gesetzt, DEST_DB exsistiert schon: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_testdb_gen*	Saubere Erromeldung. Keine Exception	Ok
32	CLI generate Options		Drop-Option nicht gesetzt, DEST_DB existiert noch nicht	Normale Funktion. DB wird generiert	Ok
33	CLI generate Options	DB "pgsynthdata_testdb_gen" existiert schon	Drop-Option gesetzt, DEST_DB existiert schon: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_testdb_gen --drop*	Normale Funktion. DB wird generiert	Ok
34	CLI generate Options	DB "pgsynthdata_systest_noexist" existiert noch nicht	Drop-Option gesetzt, DEST_DB existiert noch nicht: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systest_noexist --drop*	Normale Funktion. DB wird generiert	Ok
35	CLI generate Options		Multiplicator-Option auf 1.0 gesetzt (explizit): *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systest_m10 -m 1.0*	Normale Funktion. DB wird mit 100% Daten generiert	Ok
36	CLI generate Options		Multiplicator-Option auf 0.5 gesetzt: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systest_m05 -m 0.5*	Normale Funktion. DB wird mit 50% Daten generiert	error
37	CLI generate Options		Multiplicator-Option auf 2.0 gesetzt: *poetry run python -m pgsynthdata pgsynthdata_testdb generate pgsynthdata_systest_m20 -m 2.0*	Normale Funktion. DB wird mit 200% Daten generiert	error
38	CLI show		Show ausführen: *poetry run python -m pgsynthdata pgsynthdata_testdb show*	Alle Tabellen und Spalten werden ausgegeben Attribute werden korrekt angegeben (Pro möglichen Wert eine Stichprobe) Generatorwahl ist korrekt (Alle)	Ok
39	CLI analyze Format-Options		Analyze für HTML *poetry run python -m pgsynthdata pgsynthdata_testdb analyze pgsynthdata_testdb_gen*	HTML-File wird erstellt Alle Tabellen und Spalten sind vorhanden Werte stimmen mit pg_stats überein (10% Stichproben)	Ok
40	CLI analyze Format-Options		Analyze für JSON *poetry run python -m pgsynthdata pgsynthdata_testdb analyze pgsynthdata_testdb_gen --format=json*	JSON-File wird erstellt JSON-File ist valide (https://jsonlint.com/) Alle Tabellen und Spalten sind vorhanden Werte stimmen mit pg_stats überein (10% Stichproben)	Ok

Anhang F

Ausdruck Analyze Kommando

Dies ist ein Ausdruck der HTML-Seite, welche das Analyze-Kommando, angewendet auf die originale und generierte Testdatenbank, erzeugt. Das vollständige Kommando lautet:

```
pgsynthdata "pgsynthdata_testdb" analyze "pgsynthdata_testdb_gen"
```

Analye database 'pgsynthdata_testdb' (source) and 'pgsynthdata_testdb_gen' (generated)

array_numeric_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
float8_array	0.2	-0.8	None	None	['{0.016243983…	['0.001799058…	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…
float8_array (g…	0.2	-0.8	None	None	['{6.299126096…	['94883289706…	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…
id	0.0	-1.0	None	None	['1', '2', '3', '4', '…	None	None	None
id (gen)	0.0	-1.0	None	None	['1', '2', '3', '4', '…	None	None	None
int2_array	0.2	-0.78	['{36}', '{50}']	[0.02, 0.02]	['{0,60,54,NUL…	['0', '1', '2', '3', '…	[0.0375, 0.025,…	[1.0, 1.0, 1.0, 1…
int2_array (gen)	0.2	-0.76	['{31}', '{NULL}']	[0.03, 0.03]	['{3,38,53,28}', '…	['3', '8', '10', '12'…	[0.0125, 0.025,…	[0.0, 0.0, 0.0, 0…
int4_array	0.2	-0.8	None	None	['{7}', '{9}', '{26,…	['1', '7', '9', '26', …	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…
int4_array (gen)	0.2	-0.8	None	None	['{23,244,601}',…	['8', '12', '23', '2…	[0.025, 0.0125,…	[1.0, 1.0, 1.0, 1…
int8_array	0.2	-0.8	None	None	['{15,8402,600…	['15', '87', '98', '…	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…
int8_array (gen)	0.2	-0.8	None	None	['{288,832}', '{3…	['18', '76', '134',…	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…
numeric_array	0.2	-0.8	None	None	['{0.005353303…	['0.005353303…	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…
numeric_array …	0.2	-0.8	None	None	['{6523659995…	['96895174788…	[0.0125, 0.012…	[1.0, 1.0, 1.0, 1…

binary_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
binary_data	0.0	-0.2	['\\x7f7f']	[1.0]	None	None	None	None
binary_data (g…	0.0	-0.2	['\\x48656c6c6f…	[1.0]	None	None	None	None
id	0.0	-1.0	None	None	['1', '2', '3', '4', '5']	None	None	None
id (gen)	0.0	-1.0	None	None	['1', '2', '3', '4', '5']	None	None	None

boolean_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
boolean	0.19	2.0	['t', 'f']	[0.5, 0.31]	None	None	None	None
boolean (gen)	0.0	2.0	['t', 'f']	[0.53, 0.47]	None	None	None	None
id	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
id (gen)	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
unique	0.98	-0.01999998	None	None	['f', 't']	None	None	None
unique (gen)	1.0	0.0	None	None	None	None	None	None

character_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
character	0.19	-0.45	['D', 't', 'l', 'j', 'p',…	[0.04, 0.04, 0.0…	['d', 'e', 'f', 'G', '…	None	None	None
character (gen)	0.16	-0.21	['p', 'b', 'h', 'd', '…	[0.1, 0.09, 0.09…	['i', 'r', 's', 'y']	None	None	None
character varying	0.19	-0.81	None	None	['aE', 'AGcDSS…	None	None	None
character varyi…	0.0	-1.0	None	None	['arnvdzvaxas',…	None	None	None
id	0.0	-1.0	None	None	['loremipsumat…	None	None	None
id (gen)	0.0	-1.0	None	None	['aariawecpygv…	None	None	None
text	0.19	-0.81	None	None	['aEBgDpl', 'ag…	None	None	None
text (gen)	0.0	-1.0	None	None	['aapeqvfeczna…	None	None	None
unique	0.0	-1.0	None	None	['blaloremipsu…	None	None	None
unique (gen)	0.0	-1.0	None	None	['acnpytboqwb…	None	None	None

copy_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
copy_int	0.0	-1.0	None	None	['1', '2', '3', '4', '5']	None	None	None
copy_int (gen)	0.0	-1.0	None	None	['1', '2', '3', '4', '5']	None	None	None
copy_text	0.0	-1.0	None	None	['aaa', 'bbb', 'cc…	None	None	None
copy_text (gen)	0.0	-1.0	None	None	['aaa', 'bbb', 'cc…	None	None	None
copy_udt_asis	0.0	-1.0	None	None	['01010000000…	None	None	None
copy_udt_asis …	0.0	-1.0	None	None	['01010000000…	None	None	None

datetime_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
date	0.19	-0.81	None	None	['2008-01-31', '…	None	None	None
date (gen)	0.0	-0.99	['2011-03-11']	[0.02]	['2008-01-31', '…	None	None	None
id	0.0	-1.0	None	None	['2021-11-25', '…	None	None	None
id (gen)	0.0	-0.62	['2021-12-07', '…	[0.03, 0.03, 0.0…	['2021-11-25', '…	None	None	None

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
timestamp with…	0.19	-0.81	None	None	['2008-01-02 1…	None	None	None
timestamp with…	1.0	0.0	None	None	None	None	None	None
timestamp with…	0.19	-0.81	None	None	['2008-02-22 0…	None	None	None
timestamp with…	0.0	-1.0	None	None	['2008-03-06 0…	None	None	None

enumerated_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
id	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
id (gen)	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
mood	0.2	5.0	['jubilant', 'sad',…	[0.2, 0.16, 0.15…	None	None	None	None
mood (gen)	0.0	5.0	['solala', 'jubila…	[0.25, 0.22, 0.2…	None	None	None	None
unique	0.95	-0.050000012	None	None	['sad', 'solala', '…	None	None	None
unique (gen)	1.0	0.0	None	None	None	None	None	None

fk_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
fk	0.0	-1.0	None	None	['0', '1000', '200…	None	None	None
fk (gen)	0.0	-1.0	None	None	['1297', '3513', '…	None	None	None

fk_generator_ref

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
id	0.0	-1.0	None	None	['0', '1000', '200…	None	None	None
id (gen)	0.0	-1.0	None	None	['1297', '3513', '…	None	None	None

geometric_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
box	0.2	0.0	None	None	None	None	None	None
box (gen)	0.0	0.0	None	None	None	None	None	None
circle	0.2	0.0	None	None	None	None	None	None
circle (gen)	0.0	0.0	None	None	None	None	None	None
id	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
id (gen)	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
line	0.2	0.0	None	None	None	None	None	None
line (gen)	0.0	0.0	None	None	None	None	None	None
lseg	0.2	0.0	None	None	None	None	None	None
lseg (gen)	0.0	0.0	None	None	None	None	None	None
path	0.2	0.0	None	None	None	None	None	None
path (gen)	0.0	0.0	None	None	None	None	None	None
point	0.2	0.0	None	None	None	None	None	None
point (gen)	0.0	0.0	None	None	None	None	None	None
polygon	0.2	0.0	None	None	None	None	None	None
polygon (gen)	0.0	0.0	None	None	None	None	None	None

numeric_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
bigint	0.19	-0.81	None	None	['62', '101', '201…	None	None	None
bigint (gen)	0.0	-0.99	['712']	[0.02]	['73', '76', '109',…	None	None	None
double precision	0.19	-0.81	None	None	['157.5825424…	None	None	None
double precisio…	0.0	-1.0	None	None	['218', '245', '54…	None	None	None
id	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
id (gen)	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
integer	0.19	-0.81	None	None	['30', '171', '223…	None	None	None
integer (gen)	0.0	-1.0	None	None	['31', '252', '716…	None	None	None
numeric	0.19	-0.81	None	None	['29.61444056…	None	None	None
numeric (gen)	0.0	-1.0	None	None	['1220', '1257', '…	None	None	None
real	0.19	-0.81	None	None	['393.52185', '4…	None	None	None
real (gen)	0.0	-0.99	['3154']	[0.02]	['422', '492', '52…	None	None	None
smallint	0.19	-0.81	None	None	['358', '410', '44…	None	None	None
smallint (gen)	0.0	-0.99	['2653']	[0.02]	['358', '360', '41…	None	None	None
unique	0.19	-0.81	None	None	['215', '262', '45…	None	None	None
unique (gen)	0.0	-1.0	None	None	['228', '246', '55…	None	None	None

person_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
PERSON_FIR…	0.2	-0.79	['Jeanine']	[0.02]	['Agnès', 'Alain'…	None	None	None
PERSON_FIR…	0.0	-0.99	['Aurélie']	[0.02]	['Agnieszka', 'A…	None	None	None
PERSON_LAS…	0.2	-0.67	['Schaub', 'Stei…	[0.03, 0.03, 0.0…	['Ackermann', '…	None	None	None
PERSON_LAS…	0.0	-0.81	['Bucher', 'Stad…	[0.03, 0.03, 0.0…	['Albrecht', 'Arn…	None	None	None
PERSON_NAME	0.2	-0.8	None	None	['Albert Wehrli',…	None	None	None
PERSON_NA…	0.0	-1.0	None	None	['Agathe Bossh…	None	None	None
PERSON_PR…	0.2	2.0	['Dr.', 'Prof.']	[0.4, 0.4]	None	None	None	None
PERSON_PR…	0.0	2.0	['Dr.', 'Prof.']	[0.51, 0.49]	None	None	None	None
PERSON_SU…	0.2	1.0	['']	[0.8]	None	None	None	None
PERSON_SU…	0.0	1.0	['']	[1.0]	None	None	None	None
id	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
id (gen)	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
unique	0.2	-0.8	None	None	['Abel', 'Alban', …	None	None	None
unique (gen)	0.0	-1.0	None	None	['Aida', 'Aisha', …	None	None	None

spatial_generator

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
id	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
id (gen)	0.0	-1.0	None	None	['0', '1', '2', '3', '…	None	None	None
point	0.2	-1.0	None	None	['01010000005…	None	None	None
point (gen)	None	None	None	None	None	None	None	None
unique	0.2	-1.0	None	None	['0101000000E…	None	None	None
unique (gen)	None	None	None	None	None	None	None	None

spatial_ref_sys

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	most_common_elems	most_common_elem_freqs	elem_count_histo
auth_name	0.0	3.0	['EPSG', 'ESRI']	[0.7275294, 0.…	None	None	None	None
auth_name (gen)	0.0	3.0	['EPSG', 'ESRI']	[0.7275294, 0.…	None	None	None	None
auth_srid	0.0	-1.0	None	None	['2000', '2084', '…	None	None	None
auth_srid (gen)	0.0	-1.0	None	None	['2000', '2084', '…	None	None	None
proj4text	0.0	-0.746353	['+proj=geocen…	[0.016235294, …	['', '+proj=bonn…	None	None	None
proj4text (gen)	0.0	-0.746353	['+proj=geocen…	[0.016235294, …	['', '+proj=bonn…	None	None	None
srid	0.0	-1.0	None	None	['2000', '2084', '…	None	None	None

column	null_frac	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	least_common_vals	least_common_freqs	num_histo
srid (gen)	0.0	-1.0	None	None	['2000', '2084', ...	None	None	None
srtext	0.0	-0.998	['', 'GEOGCRS[...	[0.0016470589...	['COMPD_CS[" ...	None	None	None
srtext (gen)	0.0	-0.998	['', 'GEOGCRS[...	[0.0016470589...	['COMPD_CS[" ...	None	None	None

Anhang G

Ausdruck Show Kommando

Dies ist ein Ausdruck der Konsolenausgabe welche das Show-Kommando, angewendet auf die originale Testdatenbank, erzeugt. Das vollständige Kommando lautet:

```
pgsynthdata "pgsynthdata_testdb" show
```

```
Password:
****************************************
array_numeric_generator (public): 100 rows
+---------------+-------+--------+----------+-----------+----------+-------------------------+---------+
| attname       | is_fk | unique | nullable | data_type | udt_name | generator               | comment |
+---------------+-------+--------+----------+-----------+----------+-------------------------+---------+
| id            | False | True   | NO       | integer   | int4     | numeric_generator       |         |
| int2_array    | False | False  | YES      | ARRAY     | _int2    | array_numeric_generator |         |
| int4_array    | False | False  | YES      | ARRAY     | _int4    | array_numeric_generator |         |
| int8_array    | False | False  | YES      | ARRAY     | _int8    | array_numeric_generator |         |
| numeric_array | False | False  | YES      | ARRAY     | _numeric | array_numeric_generator |         |
| float8_array  | False | False  | YES      | ARRAY     | _float8  | array_numeric_generator |         |
+---------------+-------+--------+----------+-----------+----------+-------------------------+---------+

****************************************
binary_generator (public): 5 rows
+-------------+-------+--------+----------+-----------+----------+-------------------+---------+
| attname     | is_fk | unique | nullable | data_type | udt_name | generator         | comment |
+-------------+-------+--------+----------+-----------+----------+-------------------+---------+
| id          | False | True   | NO       | integer   | int4     | numeric_generator |         |
| binary_data | False | False  | YES      | bytea     | bytea    | binary_generator  |         |
+-------------+-------+--------+----------+-----------+----------+-------------------+---------+

****************************************
boolean_generator (public): 100 rows
+---------+-------+--------+----------+-----------+----------+-------------------+---------+
| attname | is_fk | unique | nullable | data_type | udt_name | generator         | comment |
+---------+-------+--------+----------+-----------+----------+-------------------+---------+
| id      | False | True   | NO       | integer   | int4     | numeric_generator |         |
| boolean | False | False  | YES      | boolean   | bool     | boolean_generator |         |
| unique  | False | True   | YES      | boolean   | bool     | boolean_generator |         |
+---------+-------+--------+----------+-----------+----------+-------------------+---------+

****************************************
character_generator (public): 100 rows
+--------------------+-------+--------+----------+--------------------+----------+----------------------+---------+
| attname            | is_fk | unique | nullable | data_type          | udt_name | generator            | comment |
+--------------------+-------+--------+----------+--------------------+----------+----------------------+---------+
| id                 | False | True   | NO       | text               | text     | character_generator |         |
| text               | False | False  | YES      | text               | text     | character_generator |         |
| character varying  | False | False  | YES      | character varying  | varchar  | character_generator |         |
| character          | False | False  | YES      | character          | bpchar   | character_generator |         |
| unique             | False | True   | YES      | text               | text     | character_generator |         |
+--------------------+-------+--------+----------+--------------------+----------+----------------------+---------+

****************************************
copy_generator (public): 5 rows
+---------------+-------+--------+----------+--------------+----------+----------------+--------------------+
| attname       | is_fk | unique | nullable | data_type    | udt_name | generator      | comment            |
+---------------+-------+--------+----------+--------------+----------+----------------+--------------------+
| copy_int      | False | False  | YES      | integer      | int4     | copy_generator | PGSYNTHDATA[COPY] |
```

```
| copy_text     | False | False  | YES      | text         | text     | copy_generator | PGSYNTHDATA[COPY] |
| copy_udt_asis | False | False  | YES      | USER-DEFINED | geometry | copy_generator | PGSYNTHDATA[COPY] |
+---------------+-------+--------+----------+--------------+----------+----------------+-------------------+

********************************************
datetime_generator (public): 100 rows
+-----------------------------+-------+--------+----------+-----------------------------+-------------+--------------------+---
------+
| attname                     | is_fk | unique | nullable | data_type                   | udt_name    | generator          |
comment |
+-----------------------------+-------+--------+----------+-----------------------------+-------------+--------------------+---
------+
| id                          | False | False  | NO       | date                        | date        | datetime_generator
|         |
| date                        | False | False  | YES      | date                        | date        | datetime_generator
|         |
| timestamp with time zone    | False | False  | YES      | timestamp with time zone    | timestamptz | NO GENERATOR FOUND
|         |
| timestamp without time zone | False | False  | YES      | timestamp without time zone | timestamp   | datetime_generator
|         |
+-----------------------------+-------+--------+----------+-----------------------------+-------------+--------------------+---
------+

********************************************
enumerated_generator (public): 100 rows
+---------+-------+--------+----------+--------------+----------+----------------------+---------+
| attname | is_fk | unique | nullable | data_type    | udt_name | generator            | comment |
+---------+-------+--------+----------+--------------+----------+----------------------+---------+
| id      | False | True   | NO       | integer      | int4     | numeric_generator    |         |
| mood    | False | False  | YES      | USER-DEFINED | mood     | enumerated_generator |         |
| unique  | False | True   | YES      | USER-DEFINED | mood     | enumerated_generator |         |
+---------+-------+--------+----------+--------------+----------+----------------------+---------+

********************************************
fk_generator (public): 100 rows
+---------+-------+--------+----------+-----------+----------+--------------+---------+
| attname | is_fk | unique | nullable | data_type | udt_name | generator    | comment |
+---------+-------+--------+----------+-----------+----------+--------------+---------+
| fk      | True  | False  | YES      | integer   | int4     | fk_generator |         |
+---------+-------+--------+----------+-----------+----------+--------------+---------+

********************************************
fk_generator_ref (public): 100 rows
+---------+-------+--------+----------+-----------+----------+-------------------+---------+
| attname | is_fk | unique | nullable | data_type | udt_name | generator         | comment |
+---------+-------+--------+----------+-----------+----------+-------------------+---------+
| id      | False | True   | NO       | integer   | int4     | numeric_generator |         |
+---------+-------+--------+----------+-----------+----------+-------------------+---------+

********************************************
geometric_generator (public): 100 rows
```

```
+----------+--------+----------+-----------+-----------+-----------------------+---------+
| attname  | is_fk  | unique   | nullable  | data_type | udt_name  | generator         | comment |
+----------+--------+----------+-----------+-----------+-----------+-------------------+---------+
| id       | False  | True     | NO        | integer   | int4      | numeric_generator |         |
| point    | False  | False    | YES       | point     | point     | geometric_generator |         |
| line     | False  | False    | YES       | line      | line      | geometric_generator |         |
| polygon  | False  | False    | YES       | polygon   | polygon   | geometric_generator |         |
| circle   | False  | False    | YES       | circle    | circle    | geometric_generator |         |
| box      | False  | False    | YES       | box       | box       | geometric_generator |         |
| lseg     | False  | False    | YES       | lseg      | lseg      | geometric_generator |         |
| path     | False  | False    | YES       | path      | path      | geometric_generator |         |
+----------+--------+----------+-----------+-----------+-----------+-------------------+---------+

**********
numeric_generator (public): 100 rows
+-----------------------+--------+----------+-----------+-----------------------+-----------+-------------------+---------+
| attname               | is_fk  | unique   | nullable  | data_type             | udt_name  | generator         | comment |
+-----------------------+--------+----------+-----------+-----------------------+-----------+-------------------+---------+
| id                    | False  | True     | NO        | integer               | int4      | numeric_generator |         |
| integer               | False  | False    | YES       | integer               | int4      | numeric_generator |         |
| smallint              | False  | False    | YES       | smallint              | int2      | numeric_generator |         |
| bigint                | False  | False    | YES       | bigint                | int8      | numeric_generator |         |
| numeric               | False  | False    | YES       | numeric               | numeric   | numeric_generator |         |
| double precision      | False  | False    | YES       | double precision      | float8    | numeric_generator |         |
| real                  | False  | False    | YES       | real                  | float4    | numeric_generator |         |
| unique                | False  | True     | YES       | integer               | int4      | numeric_generator |         |
+-----------------------+--------+----------+-----------+-----------------------+-----------+-------------------+---------+

**********
person_generator (public): 100 rows
+----------+--------+----------+-----------+-----------+-----------+-----------+---------+
| attname  | is_fk  | unique   | nullable  | data_type | udt_name  | generator |         |
| comment  |        |          |           |           |           |           |         |
+----------+--------+----------+-----------+-----------+-----------+-----------+---------+
| id       | False  | True     | NO        | integer   | int4      | numeric_generator |
|          |        |          |           |           |           |           |
| PERSON_FIRST_NAME | False  | YES       | text      | text      | person_firstname_generator |
PGSYNTHDATA[PERSON_FIRST_..
| PERSON_LAST_NAME  | False  | YES       | text      | text      | person_lastname_generator |
PGSYNTHDATA[PERSON_LAST_N..
| PERSON_NAME       | False  | YES       | text      | text      | person_name_generator     |
PGSYNTHDATA[PERSON_NAME]
| PERSON_PREFIX     | False  | YES       | text      | text      | person_prefix_generator   |
PGSYNTHDATA[PERSON_PREFIX..
| PERSON_SUFFIX     | False  | YES       | text      | text      | person_suffix_generator   |
PGSYNTHDATA[PERSON_SUFFIX..
| unique            | False  | True      | YES       | text      | text      | person_firstname_generator |
PGSYNTHDATA[PERSON_FIRST_..
+-------------------+--------+-----------+-----------+-----------+-----------+-----------+
```

```
    ---+
136
137 ****************************************
138 spatial_generator (public): 100 rows
139 +---------+-------+--------+----------+--------------+----------+------------------+---------+
140 | attname | is_fk | unique | nullable | data_type    | udt_name | generator        | comment |
141 +---------+-------+--------+----------+--------------+----------+------------------+---------+
142 | id      | False | True   | NO       | integer      | int4     | numeric_generator |         |
143 | point   | False | False  | YES      | USER-DEFINED | geometry | spatial_generator |         |
144 | unique  | False | False  | YES      | USER-DEFINED | geometry | spatial_generator |         |
145 +---------+-------+--------+----------+--------------+----------+------------------+---------+
146
147 ****************************************
148 spatial_ref_sys (public): 8500 rows
149 +-----------+-------+--------+----------+-------------------+----------+-----------------+----------------------+
150 | attname   | is_fk | unique | nullable | data_type         | udt_name | generator       | comment              |
151 +-----------+-------+--------+----------+-------------------+----------+-----------------+----------------------+
152 | srid      | False | True   | NO       | integer           | int4     | ignore_generator | PGSYNTHDATA[IGNORE] |
153 | auth_name | False | False  | YES      | character varying | varchar  | ignore_generator | PGSYNTHDATA[IGNORE] |
154 | auth_srid | False | False  | YES      | integer           | int4     | ignore_generator | PGSYNTHDATA[IGNORE] |
155 | srtext    | False | False  | YES      | character varying | varchar  | ignore_generator | PGSYNTHDATA[IGNORE] |
156 | proj4text | False | False  | YES      | character varying | varchar  | ignore_generator | PGSYNTHDATA[IGNORE] |
157 +-----------+-------+--------+----------+-------------------+----------+-----------------+----------------------+
158
159
160 (pgsynthdata-In5XDOTz-py3.9) PS C:\Users\tturbo\Documents\HSR\UNTERRICHT\S5\SA\Code\gitlab.ost.ch\pgsynthdata>
```

Anhang H

Testabdeckung

Der Coverage-Report in Tabelle H.1 kann mit den Befehlen in Listing H.1 erstellt werden. Er wurde lokal und mit funktionierender Testdatenbank ausgeführt

```
1 poetry run coverage run -m --source=pgsynthdata pytest tests
2 poetry run coverage report
```

Listing H.1: Coverage Report generieren

Module	**statements**	**missing**	**excluded**	**coverage**
__main__.py	119	119	0	0%
database.py	242	7	0	97%
datagenerator.py	114	7	0	94%
generators\address_generator.py	50	28	0	44%
generators\array_generator_utils.py	19	0	0	100%
generators\array_numeric_generator.py	39	1	0	97%
generators\binary_generator.py	18	4	0	78%
generators\boolean_generator.py	11	1	0	91%
generators\character_generator.py	21	5	0	76%
generators\copy_generator.py	16	11	0	31%
generators\datetime_generator.py	28	8	0	71%
generators\enumerated_generator.py	18	3	0	83%
generators\fk_generator.py	17	3	0	82%
generators\generator_utils.py	40	3	0	92%
generators\geometric_generator.py	20	4	0	80%
generators\ignore_generator.py	4	0	0	100%
generators\legacy\bytea_generator.py	5	0	0	100%
generators\legacy\date_generator.py	41	29	0	29%
generators\legacy\generator_utils.py	96	20	0	79%
generators\legacy\geometry_generator.py	33	1	0	97%
generators\legacy\numeric_generator.py	75	42	0	44%
generators\legacy\varchar_generator.py	25	0	0	100%
generators\numeric_generator.py	26	5	0	81%
generators\person_generator.py	37	4	0	89%
generators\spatial_generator.py	11	4	0	64%
main.py	146	14	0	90%
Total	1271	323	0	75%

Tabelle H.1: Testabdeckung aus coverage.py v6.1.2, vom 2021-12-06 14:03

Anhang I

Performance Test

Durch den hier beschriebenen Performance Tests werden, wie in den nicht funktionalen Anforderungen gefordert, Flaschenhälse im System gefunden. Es wird der Aufbau des Tests gezeigt und eine Interpretation der Resultate beschrieben.

I.1 Ausgangslage

Für die Messung wurde die grosse Datenbank `dvdstore_500mb` aus dem Modul Data Engineering eingesetzt.

Der Programmcode wurde mit Zeitmessungen ergänzt. Diese werden jeweils auf einzelne Funktionen mittels eines Decorators angewendet. Der Decorator in Listing I.1 kann eine Funktion von einem Generator unterscheiden um beim Generator noch den Namen der Spalte auszugeben.

```python
def timer_func(func, is_generator=False):
  def decorator(*args, **kwargs):
    t1 = time()
    result = func(*args, **kwargs)
    t2 = time()
    from pgsynthdata.database import ColumnPgInfo

    time_str = f"{(t2-t1):.4f}s"
    if is_generator and args[1] and isinstance(args[1],
    ↪  ColumnPgInfo):
      logging.info(
        f"TIME: Generator   {func.__name__:<27} executed in
        ↪  {time_str:>10}    ({args[1].column_name})"
        )
    else:
      logging.info(
        f"TIME: Function    {func.__name__:<27} executed in
        ↪  {time_str:>10}"
        )
    return result
  return decorator
```

Listing I.1: Zeitmessungsdecorator

Da die `dvdstore_500mb` Datenbank extrem gross ist, wird über den Multiplication-Factor nur 20% der Originalgrösse generiert (`-m 0.2`).

```
poetry run python -m pgsynthdata --verbose ds2 generate ds2_gen -m
↪  0.2 -d
```

Listing I.2: Kommandozeilenbefehl für den Start des Performance Tests

I.2 Resultate

Generell erkennt man, dass die Abfrage der statistischen Informationen und das Selektieren der Generatoren schnell ist und sicherlich kein Flaschenhals darstellt.

Bei der Generierung der Daten ist das nicht mehr so einfach zu sagen. Es gibt grosse Unterschiede wie lange es dauert, bis ein Generator fertig ist. Auch derselbe Generator kann unterschiedlich lange für dieselbe Menge an Daten haben.

Als Beispiel kann hier von der Tabelle `customers` die Spalte `zip` (150.1354s),

`firstname` (1.0657s) und `gender` (0.4750s) genommen werden. Dies sind alles Spalten, welche vom `character_generator` generiert werden und alle müssen dieselbe Anzahl an Werten generieren (200'000). Nur gibt es eine extreme Spannweite in Bezug auf die benötigte Zeit für die Generierung.

Der `fk_generator` benötigt auffallend viel Zeit. Beispiel: Tabelle `orders` Spalte `customerid` 570.0582s.

Das Einfügen der Daten in die Datenbank hat viel mit der Art der Daten und den Constraints auf der Datenbank zu tun. In Tabelle I.1 sind einige Beispiele zu sehen.

Tabelle	Spalten	Zeilen	Zeit
customer	20	200'000	26.5803s
products	7	100'000	6.0364s
inventory	3	100'000	2.8560s
categories	2	3	0.0042s

Tabelle I.1: Beispiel Zeiten für Insert der Performance Test Daten

I.3 Raw-Output

Auf Zeile 29 wurde der sich wiederholende Output gekürzt.

```
1  INFO:root:Generate database ds2 in ds2_gen
2  INFO:root:load generator plugins
3  Start generating database 'ds2_gen'
4  INFO:root:connect to source database
5  INFO:root:connection to database ds2 on host localhost:5432 with user postgres
6  INFO:root:create destination database
7  INFO:root:Check if destination database exists
8  INFO:root:Create destination database
9  INFO:root:connect to destination database
10 INFO:root:connection to database ds2_gen on host localhost:5432 with user postgres
11 INFO:root:copy schema to destination database
12 INFO:root:Copy database Structure
13 INFO:root:Create Database Dump
14 INFO:root:TIME: Function    dump_database               executed in    0.2926s
15 INFO:root:Import Database Dump
16 INFO:root:TIME: Function    import_database             executed in    0.1317s
17 INFO:root:TIME: Function    copy_database               executed in    0.4255s
18 INFO:root:Analyze Database with 'VACUUM ANALYZE'
19 INFO:root:Finished with 'VACUUM ANALYZE'
20 INFO:root:TIME: Function    analyze                     executed in    1.4600s
21 INFO:root:TIME: Function    get_constraints             executed in    0.0125s
22 INFO:root:TIME: Function    get_tables                  executed in    0.2313s
23 INFO:root:TIME: Function    get_column_comment          executed in    0.0045s
24 INFO:root:TIME: Function    get_generator               executed in    0.0000s
25 INFO:root:TIME: Function    get_stats                   executed in    0.0024s
26 INFO:root:TIME: Function    get_column_comment          executed in    0.0046s
27 INFO:root:TIME: Function    get_generator               executed in    0.0000s
28 INFO:root:TIME: Function    get_stats                   executed in    0.0023s
29 ...
```

```
30 INFO:root:TIME: Function    _get_database_structure     executed in    3.9981s
31 INFO:root:TIME: Function    get_relations               executed in    0.0042s
32 INFO:root:generate data for table 'inventory'
33 INFO:root:TIME: Generator   numeric_generator           executed in    0.0867s  (prod_id)
34 INFO:root:TIME: Generator   numeric_generator           executed in    1.7279s
   ↪  (quan_in_stock)
35 INFO:root:TIME: Generator   numeric_generator           executed in    0.2834s  (sales)
36 INFO:root:TIME: Function    _generate_data              executed in    2.0984s
37 INFO:root:insert data in table 'inventory'
38 INFO:root:TIME: Function    insert_data                 executed in    2.8560s
39 INFO:root:generate data for table 'customers'
40 INFO:root:TIME: Generator   numeric_generator           executed in    0.1825s
   ↪  (customerid)
41 INFO:root:TIME: Generator   character_generator         executed in    1.0657s
   ↪  (firstname)
42 INFO:root:TIME: Generator   character_generator         executed in    1.5068s
   ↪  (lastname)
43 INFO:root:TIME: Generator   character_generator         executed in    2.5172s
   ↪  (address1)
44 INFO:root:TIME: Generator   character_generator         executed in    0.4000s
   ↪  (address2)
45 INFO:root:TIME: Generator   character_generator         executed in    1.1688s  (city)
46 INFO:root:TIME: Generator   character_generator         executed in    0.6439s  (state)
47 INFO:root:TIME: Generator   character_generator         executed in  150.1354s  (zip)
48 INFO:root:TIME: Generator   character_generator         executed in    0.5363s  (country)
49 INFO:root:TIME: Generator   numeric_generator           executed in    0.4727s  (region)
50 INFO:root:TIME: Generator   character_generator         executed in    3.1277s  (email)
51 INFO:root:TIME: Generator   character_generator         executed in    1.7703s  (phone)
52 INFO:root:TIME: Generator   numeric_generator           executed in    0.5020s
   ↪  (creditcardtype)
53 INFO:root:TIME: Generator   character_generator         executed in    2.4921s
   ↪  (creditcard)
54 INFO:root:TIME: Generator   character_generator         executed in    0.8027s
   ↪  (creditcardexpiration)
55 INFO:root:TIME: Generator   character_generator         executed in    1.6659s
   ↪  (username)
56 INFO:root:TIME: Generator   character_generator         executed in    0.4654s
   ↪  (password)
57 INFO:root:TIME: Generator   numeric_generator           executed in    0.9192s  (age)
58 INFO:root:TIME: Generator   numeric_generator           executed in    0.5013s  (income)
59 INFO:root:TIME: Generator   character_generator         executed in    0.4750s  (gender)
60 INFO:root:TIME: Function    _generate_data              executed in  171.3573s
61 INFO:root:insert data in table 'customers'
62 INFO:root:TIME: Function    insert_data                 executed in   26.5803s
63 INFO:root:generate data for table 'categories'
64 INFO:root:TIME: Generator   numeric_generator           executed in    0.0006s
   ↪  (category)
65 INFO:root:TIME: Generator   character_generator         executed in    0.0004s
   ↪  (categoryname)
66 INFO:root:TIME: Function    _generate_data              executed in    0.0014s
67 INFO:root:insert data in table 'categories'
68 INFO:root:TIME: Function    insert_data                 executed in    0.0042s
69 INFO:root:generate data for table 'products'
70 INFO:root:TIME: Generator   numeric_generator           executed in    0.1013s  (prod_id)
71 INFO:root:TIME: Generator   numeric_generator           executed in    0.2812s
   ↪  (category)
72 INFO:root:TIME: Generator   character_generator         executed in    1.1513s  (title)
73 INFO:root:TIME: Generator   character_generator         executed in    1.0315s  (actor)
74 INFO:root:TIME: Generator   numeric_generator           executed in    0.3054s  (price)
75 INFO:root:TIME: Generator   numeric_generator           executed in    0.2423s  (special)
76 INFO:root:TIME: Generator   numeric_generator           executed in    0.1992s
   ↪  (common_prod_id)
```

```
77  INFO:root:TIME: Function    _generate_data             executed in    3.3132s
78  INFO:root:insert data in table 'products'
79  INFO:root:TIME: Function    insert_data                executed in    6.0364s
80  INFO:root:generate data for table 'cust_hist'
81  INFO:root:TIME: Generator   fk_generator               executed in    0.0438s
    ↪  (customerid)
82  INFO:root:TIME: Generator   numeric_generator          executed in    1.2466s  (orderid)
83  INFO:root:TIME: Generator   numeric_generator          executed in    3.9370s  (prod_id)
84  INFO:root:TIME: Function    _generate_data             executed in    5.2279s
85  INFO:root:insert data in table 'cust_hist'
86  INFO:root:TIME: Function    insert_data                executed in    0.0001s
87  INFO:root:generate data for table 'orders'
88  INFO:root:TIME: Generator   numeric_generator          executed in    0.1428s  (orderid)
89  INFO:root:TIME: Generator   datetime_generator         executed in    2.2105s
    ↪  (orderdate)
90  INFO:root:TIME: Generator   fk_generator               executed in  570.0582s
    ↪  (customerid)
91  INFO:root:TIME: Generator   numeric_generator          executed in    0.6165s
    ↪  (netamount)
92  INFO:root:TIME: Generator   numeric_generator          executed in   10.8042s  (tax)
93  INFO:root:TIME: Generator   numeric_generator          executed in    0.5478s
    ↪  (totalamount)
94  INFO:root:TIME: Function    _generate_data             executed in  584.4471s
95  INFO:root:insert data in table 'orders'
96  INFO:root:TIME: Function    insert_data                executed in    7.1772s
97  INFO:root:generate data for table 'orderlines'
98  INFO:root:TIME: Generator   numeric_generator          executed in    1.2268s
    ↪  (orderlineid)
99  INFO:root:TIME: Generator   fk_generator               executed in    0.0206s  (orderid)
100 INFO:root:TIME: Generator   numeric_generator          executed in    3.3463s  (prod_id)
101 INFO:root:TIME: Generator   numeric_generator          executed in    1.2662s
    ↪  (quantity)
102 INFO:root:TIME: Generator   datetime_generator         executed in    9.2072s
    ↪  (orderdate)
103 INFO:root:TIME: Function    _generate_data             executed in   15.0683s
104 INFO:root:insert data in table 'orderlines'
105 INFO:root:TIME: Function    insert_data                executed in    0.0016s
106 INFO:root:finish generate Database
107 INFO:root:close connection to database ds2 on host localhost:5432 with user postgres
108 INFO:root:close connection to database ds2_gen on host localhost:5432 with user postgres
109 INFO:root:TIME: Function    generate                   executed in  829.0395s
110 Finish generating database 'ds2_gen'
```

Literaturverzeichnis

Alexandru, C. V., Merchante, J. J., Panichella, S., Proksch, S., Gall, H. C. & Robles, G. (2018). On the usage of pythonic idioms. In *Proceedings of the 2018 acm sigplan international symposium on new ideas, new paradigms, and reflections on programming and software* (S. 1–11).

Ammann, K. (2021). *Erweiterung eines generators für rein synthetische daten* (Bericht). Rapperswil, CH: Institute for Software.

Dasanayake, S., Markkula, J., Aaramaa, S. & Oivo, M. (2015). Software architecture decision-making practices and challenges: An industrial case study. In *2015 24th australasian software engineering conference* (S. 88-97). doi: 10.1109/ASWEC.2015.20

Emam, K., Mosquera, L. & Hoptroff, R. (2020). *Practical synthetic data generation: Balancing privacy and the broad availability of data.* O'Reilly Media, Incorporated.

Gashi, L. (2020). *Pure synthetic data generation by the example of a postgresql and python-based tool* (Bericht). Rapperswil, CH: Hochschule für Technik Rapperswil.

Langa, L. (2018). The black code style [Software-Handbuch]. Zugriff auf `https://black.readthedocs.io/en/stable/the_black_code_style/index.html`

Langa, L. (2019, Jun). *Pep 596, python 3.9 release schedule.* Zugriff auf `https://www.python.org/dev/peps/pep-0596/#lifespan`

Lemburg, M.-A. (2001, Mar). *Pep 249, python database api specification v2.0.* Zugriff auf `https://www.python.org/dev/peps/pep-0249`

Lutz, M. (2013). *Learning python: Powerful object-oriented programming.* O'Reilly Media, Incorporated.

Martin, R. (2009). *Clean code: A handbook of agile software craftsmanship.* Prentice Hall. Zugriff auf `https://books.google.ch/books?id=dwSfGQAACAAJ`

Muppidi, S. (2020, September). *Growing applications of synthetic data.* Sierra Ventures. Zugriff auf `https://www.sierraventures.com/blog/growing-applications-of-synthetic-data/`

Pidoux, E. (2014). *Git best practices guide.* Packt Publishing Ltd.

Ping, H., Stoyanovich, J. & Howe, B. (2017). Datasynthesizer: Privacy-preserving synthetic datasets. In *Proceedings of the 29th international conference on scientific and statistical database management* (S. 1–5).

Platzer, M. (2021, May). *Synthetic geo data with mostly ai 1.5.* MOSTLY AI. Zugriff auf `https://mostly.ai/blog/synthetic-geo-data/`

RealPython. (2020, July). *Dynamic imports.* Zugriff auf `https://realpython.com/python-import/#dynamic-imports`

Replica Analytics, M. (2021, December). *About replica analytics.* Zugriff auf `https://replica-analytics.com/company`

Schonig, H.-J. (2017). *Mastering postgresql 9.6.* Packt Publishing Ltd.

van Rossum, G., Warsaw, B. & Coghlan, N. (2001, Jul). *Pep 8, style guide for python code.* Zugriff auf `https://www.python.org/dev/peps/pep-0008/`

Zimmermann, O. (2020, Apr). *Architectural decisions - the making of.* Zugriff auf `https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html`

Abbildungsverzeichnis

1 Generierung von synthetischen Daten aus Originaldaten ii
2 Anwendung von pgsynthdata iii

1.1 Anwendung von pgsynthdata 2
1.2 Synthetische Datengenerierung aus Originaldaten 3
1.3 Kompromissfindung zwischen guten synthetischen Daten und Datenschutz 5

2.1 Kontext-Diagramm von pgsynthdata 10
2.2 Wichtigste Werte der Tabelle pg_stats 12
2.3 Wichtigste Werte der Tabelle information_schema.columns 13

3.1 Komponentendiagramm 19
3.2 Ablauf der Generierung einer synthetischen Datenbank 21
3.3 Ablauf beim Kopieren des Datenbankschema 22
3.4 Beispiel Tabellen Topologie 22
3.5 Datenklassen Datenbankinformationen 24
3.6 Datenklassen Statistikinformationen 25
3.7 Funktionsweise des Plugin-Systems 26
3.8 Modul-Abhängigkeiten der Array Generatoren 28
3.9 Ablauf Array Generator 28
3.10 Beispiel einer Tabelle mit PostGIS Geometry-Typ 32
3.11 Konflikt bei der Erkennung von Enumerated-Typen 34

4.1 Baum der Generatoren (nicht vollständig) 41

5.1 Abhängigkeiten der Python-Module 51
5.2 Histogramm der Array-Längen 53
5.3 Histogramm der Element-Häufigkeiten 53
5.4 Originale und generierte Punktkoorinaten des Spatial Generators 54

5.5 Möglicher Workflow für die Erzeugung synthetischer Daten mit pgsynthdata 55

7.1 Git Workflow 65

7.2 Gantt Projektplan 67

Tabellenverzeichnis

3.1 Mapping zwischen Nummeric Array-Datentypen und udt_name . 29

7.1 Risiken für das Projekt 69
7.2 Total aufgewendete Zeit pro Teilnehmer 71
7.3 Zeit und Anteil pro Kategorie 72

H.1 Testabdeckung aus coverage.py v6.1.2, vom 2021-12-06 14:03 . . 106

I.1 Beispiel Zeiten für Insert der Performance Test Daten 109

Listings

3.1 Beispiel Tabellen-Topologie als Liste von Listen 22
3.2 Beispiel Spaltenkommentar hinterlegen 31
3.3 Custom-Query um alle möglichen Enum-Werte auszulesen 35
3.4 Aufbau eines `pgsynthdata` CLI Befehles, inkl. Beispiele 36
4.1 Vereinfachte Version des dynamisches Importes für Generatoren . 38
4.2 Beispielanwendung des Registrierungs-Dekorators 39
4.3 StatisticQuery mit Platzhalter Beispiel 40
4.4 Generatoren-Baum-Traversierung um den passender Generator zu finden 41
4.5 Beispiel einer Generatoren-Signatur 42
4.6 Vereinfachte Insert Data Funktion der Database Klasse 43
4.7 Unterschiedliche Darstellung der Insert-Daten 43
4.8 SQL Query um Spaltenkommentar auzulesen 44
4.9 Custom-Query für den Array Elemente Generator 45
4.10 Eigennamen Generator basierend auf der Faker-Library 45
4.11 Environment Variabeln für Datenbank Unit-Test 48
4.12 Verwendete coverage Befehle 49
C.1 GitLab CI Pipeline 87
H.1 Coverage Report generieren 105
I.1 Zeitmessungsdecorator 108
I.2 Kommandozeilenbefehl für den Start des Performance Tests . . . 108

Anmerkungen

1. https://www.postgresql.org/
2. https://www.python.org/
3. https://gitlab.ost.ch/ts-sa-synthdata/pgsynthdata
4. https://gitlab.com/geometalab/pgsynthdata
5. https://www.postgresql.org/docs/9.5/view-pg-stats.html
6. https://www.postgresql.org/docs/8.0/infoschema-columns.html
7. https://labs.dalibo.com/postgresql_anonymizer
8. https://gitlab.com/dalibo/postgresql_faker
9. https://faker.readthedocs.io/en/master/
10. https://mostly.ai/
11. https://replica-analytics.com/Replica-Synthesis-Software
12. https://github.com/DataResponsibly/DataSynthesizer
13. https://www.postgresql.org/docs/9.3/reference-client.html
14. https://github.com/pallets/click/tree/main/examples/complex
15. https://postgis.net/
16. https://www.python.org/dev/peps/pep-0596/#lifespan
17. https://python-poetry.org
18. https://github.com/psf/black
19. https://www.python.org/dev/peps/pep-0008/
20. https://pycqa.github.io/isort/
21. https://pylint.org/
22. http://mypy-lang.org/
23. https://docs.pytest.org/
24. https://coverage.readthedocs.io/
25. https://www.psycopg.org/
26. https://www.python.org/dev/peps/pep-0249
27. https://click.palletsprojects.com/
28. https://faker.readthedocs.io
29. https://www.postgresql.org/docs/current/reference-client.html
30. https://www.python.org/dev/peps/pep-0008/
31. https://docs.gitlab.com/ee/user/project/time_tracking.html
32. https://github.com/kriskbx/gitlab-time-tracker
33. https://gitlab.com/geometalab/pgsynthdata
34. https://docs.gitlab.com/ee/ci/pipelines/settings.html#add-test-coverage-results-to-a-merge-request
35. https://docs.gitlab.com/ee/user/project/merge_requests/test_coverage_visualization.html
36. https://gitlab.com/geometalab/pgsynthdata/-/blob/master/test_resources/System_tests_protocol.xlsx