Analyse und Optimierung verschiedener Algorithmen zur Synchronisation von SQL-Datenbanken

Masterarbeit

Sandra Lederer

Betreuer: Prof. Dr. Dietmar Seipel
M. Sc. Falco Nogatz

Lehrstuhl: Lehrstuhl für Informatik I
(Effiziente Algorithmen und wissensbasierte Systeme)

Institut: Institut für Informatik

Abgabedatum: 12.6.2018

Zusammenfassung

Datenbanken bilden heutzutage das Fundament nahezu jeder Datenverwaltung von Firmen oder Behörden. Häufig wird für diese Datenbanken die Datenbanksprache SQL verwendet. SQL erlaubt neben Manipulationen der Datenbank auch den gleichzeitigen Zugriff mehrerer Benutzer. Letzteres funktioniert jedoch nur, wenn diese gleichzeitig Zugriff auf den Server haben, auf dem sich die Datenbank befindet. Andernfalls müssen die Datenbanken nach der Modifikation synchronisiert werden. Leider sieht jedoch nicht jedes SQL-System, z.B. das weit verbreitete Datenbanksystem MySQL, eine automatische Möglichkeit hierfür vor.

In dieser Arbeit wird ein Ansatz zur Lösung dieses Problems aufgezeigt. Dies geschieht unter Verwendung von Prolog und Shell-Skripten. Hierbei werden zunächst die Unterschiede zwischen den beiden Versionen der Datenbank, der Ursprungsdatenbank und der manipulierten Datenbank, mittels der Shell-Skriptesammlung tablediff ermittelt. Anschließend werden die Unterschiede mittels Prolog näher analysiert und basierend darauf die entsprechenden MySQL-Statements generiert. Um das Ziel einer möglichst zeit-optimierten Synchronisation zu gewährleisten, werden verschiedene Algorithmen realisiert. Schließlich werden tablediff und Prolog in einem Shell-Skript zusammengefasst, das durch eine Konfigurationsdatei gesteuert werden kann. Dieses Skript wendet zudem die generierten Statements auf die Ursprungsdatenbank an und synchronisiert somit die beiden Datenbanken.

Abschließend findet eine Evaluation der verschiedenen Algorithmen statt. In dieser werden die Algorithmen unter verschiedenen Ausgangsbedingen analysiert und experimentell der beste bestimmt. Für diese Evaluation wurde im Rahmen dieser Arbeit das Java-basierte Tool CSV-Generator entwickelt. Dieses Tool erlaubt es, zwei CSV-Dateien zu generieren, eine Originaldatei und eine manipulierte. Hierbei ermöglicht das Tool, jeden Parameter der Datenbank zu beeinflussen, wie z.B. die Anzahl der Datensätze oder die Anzahl der Felder, ebenso wie den Grad der Veränderungen.

Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mich während meiner Studienzeit und insbesondere der Zeit meiner Masterarbeit so tatkräftig unterstützt haben.

Mein Dank gilt Herrn Prof. Dr. Seipel für das Bereitstellen dieses interessanten Themas der Masterarbeit und die freundliche Hilfsbereitschaft, die er mir entgegenbrachte. Ebenso gilt mein Dank meinem Betreuer Herrn M.Sc. Falco Nogatz für die Bereitschaft, die Betreuung meiner Arbeit zu übernehmen und mir durch kritisches Hinterfragen wertvolle Hinweise zu geben. Auch haben mich die beiden stets moralisch unterstützt und motiviert.

Ferner möchte ich mich bei meinen Korrekturlesern Stephan Lederer, Sonja Lederer, Stefan Bodenlos, Andrea Tüchert, Matthias Mayer und Andreas Miller bedanken, die sich die Zeit genommen haben, konstruktive Kritik an meiner Masterarbeit zu üben und mich auf Schwächen hinzuweisen.

Außerdem gilt mein Dank meinem Vater Stephan Lederer und meiner Mutter Rosalinde Lederer für den motivierenden Beistand und die finanzielle Unterstützung während meines Studiums.

Vielen Dank allen für die Geduld und Mühen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einführung

Das erste Kapitel gibt einen groben Überblick über das Thema sowie die Struktur der Arbeit. Neben den Gründen, die für eine Implementierung in Prolog sprechen, werden ähnliche, in der Literatur zu findende Arbeiten vorgestellt und die Implementierungsziele definiert.

1.1 Motivation

Datenbanken bilden heutzutage das Fundament nahezu jeder Datenverwaltung von Firmen und Behörden. Sie finden ihren Einsatz in der Personal- und Kundenverwaltung, dem Einkauf und dem Vertrieb, etc., d.h. auf diese Datenbanken erfolgen täglich zahlreiche Zugriffe. Aufgrund des umfangreichen Einsatzes ist daher eine strukturierte, zeiteffiziente Datenbank von großer Bedeutung.

Im Bereich der Datenbanken hat sich neben Oracle die Anfragesprache „Structured Query Language“ durchgesetzt. Bei dieser handelt es sich um eine in den 70ern entwickelte relationale Datenbanksprache, die unter der Abkürzung SQL bekannt wurde. Die Grundidee hinter SQL war es, dem Endnutzer eine Sprache zur Verfügung zu stellen, mit der er auf möglichst „natürlichsprachliche“ Weise Abfragen und Manipulationen an einer Datenbank vornehmen kann. Gerade durch diese Eigenschaft tritt jedoch leicht das Problem auf, dass Anfragen sehr schnell komplex werden. [onl18, BJK11]

Dies liegt maßgeblich darin begründet, dass SQL zum Verwalten von Daten sowie für einfachere Anfragen konzipiert wurde. Komplexere Anfragen, wie beispielsweise die Ermittlung der unterschiedlichen Datensätze zweier Datenbanken, überschreiten jedoch die Möglichkeiten von Standard-SQL-Anfragen. Dies ist jedoch ein notwendiger Schritt zur effizienten Synchronisation von Datenbanken.

Ausgangspunkt dieser Arbeit ist folgender Vorgang: Eine Firma sendet die zu ändernden Tabellen an den Kunden. Dieser nimmt seine Änderungen vor und sendet die komplette Tabelle wieder zurück an die Firma. Dort werden die alten Tabellen gelöscht und die kompletten Tabellen, die vom Kunden zurück kamen, neu eingefügt.

Eine solche Tabelle enthält jedoch mehrere Millionen Datensätze. Dies führt zu einer Verarbeitungszeit von mehreren Stunden.

Bestünde jedoch die Möglichkeit, vor dem Löschen und Wiedereinfügen die Unterschiede der geänderten Datensätze zu ermitteln, wäre es möglich, nur diese Datensätze zu ändern. Dies würde zu einer erheblichen Zeiteinsparung führen.

Da sich dies mit reinen SQL-Statements schwierig gestaltet, könnte der Einsatz der logischen Programmiersprache PROLOG Abhilfe schaffen. Diese ist ein fester Bestandteil im Bereich der deduktiven Datenbanken, da sie mehr Möglichkeiten bietet als das in den relationalen Datenbanken angesiedelte SQL. Dazu zählen u.a. die einfache Bestimmung von Differenzen zwischen zwei Datensätzen sowie die Mustererkennung.

1.2 Literatur

Eine Literaturrecherche liefert eine Vielzahl von Arbeiten, die sich mit dem effizienten Synchronisieren zweier Datenmengen beschäftigen. Von diesen befassen sich einige sogar ganz konkret mit dem Synchronisieren von Tabellen. Andere Arbeiten wiederum greifen das Thema u.a. von Webseiten-Aktualisierungen, XML-Vergleichen, etc. auf. Keine von diesen Arbeiten untersucht jedoch eine konkrete Lösung mittels PROLOG, auf welcher in dieser Arbeit der Schwerpunkt liegt.

1.2.1 Synchronisation von Tabellen

Bei Datenbanken ist es üblich, dass mehrere Nutzer auf diese zugreifen und Änderungen durchführen. Daher tritt häufig das Problem auf, dass lokal verschiedene Versionen einer Datenbank existieren. Um diese wieder auf den selben Stand zu bringen ist eine Synchronisation nötig. Hierzu finden sich bereits zahlreiche Tools und Algorithmen.

In [Sch07] beispielsweise befasst sich mit dem Synchronisieren zweier MySQL-Tabellen. Der Algorithmus beruht auf der Idee eines Masters- und eines Slave-Servers, sowie der Gruppierung und Rekursion. Er setzt jedoch im Gegensatz zu dem in der Masterarbeit geplanten Algorithmus einen einspaltigen Primärschlüssel voraus. *mysqldiff* [Spi07] wiederum synchronisiert die Daten mittels temporärer Datenbanken, ist jedoch laut Autor nicht für größere Datenmengen geeignet.

Darüber hinaus existieren zahlreiche Bücher und Arbeiten, die sich mit der reinen Datenbankoptimierung mittels SQL-Tuning beschäftigen. Dazu zählt [Win12], in

dem der Autor aufzeigt, wie durch gezieltes Setzen der Indexierung bei Datenbanken Performanzverbesserungen erzielt werden können. Die meisten Arbeiten befassen sich jedoch mit der Optimierung von Anfragen und weniger mit der Synchronisation von Daten. Dazu zählen die Arbeiten von Oracle [Cen18] sowie IBM [Mul02]. Diese tragen zwar nicht konkret zur Lösung des gegebenen Problems bei, jedoch können die Ergebnisse zum Erstellen der `INSERT`-, `UPDATE`- und `DELETE`-Statements genutzt werden. Auch das MySQL-Manual [MyS18e] zeigt einige Optionen zur Effizienzsteigerung einzelner Statements auf.

Da bei MySQL-Dateien die Möglichkeit besteht, diese in CSV-Dateien umzuwandeln, können Unterschiede auch mittels Text-Differenztools ermittelt werden. Zu den bekanntesten hierbei zählen *Unix diff* [HM76] sowie *Kdiff* [Eib14]. Diese weisen allerdings den Nachteil auf, dass sie zeilenweise vergleichen, d.h. Verschiebungen oder Vertauschungen von Zeilen werden auch als Änderungen markiert.

Ein Tool, das diesen Aspekt berücksichtigt, ist die freie Shell-Skript-Sammlung *tablediff* [Nog18] von Falco Nogatz. Dieses vergleicht zwei CSV-Dateien und ermittelt, welche Datensätze hinzugefügt, gelöscht oder geändert wurden und erstellt auf dieser Basis die zur Synchronisation benötigten SQL-Statements. Dieses wird im Rahmen dieser Arbeit eingesetzt und in Kapitel 4 genauer vorgestellt.

1.2.2 Weitere Einsatzgebiete von Differenzalgorithmen

Neben der Anwendung im Bereich der Datenbanken finden derartige Algorithmen auch im Bereich des `UPDATE`'s von Webseiten ihre Anwendung. In [DBCK98] stellen die Autoren die Internet Difference Engine (AIDE) vor. Hierbei handelt es sich um ein Tool zum Verfolgen und Anzeigen von Änderungen von Internetseiten. Das Tool beinhaltet neben zwei weiteren Komponenten die Komponente *htmldiff* zum Identifizieren der Änderungen auf der Webseite mittels der *Hirshbergschen Lösung* für das LCS-Problem (longest common subsequence problem). Den selben Algorithmus verwendet das bereits erwähnte Tool *Unix diff* [HM76]. *Unix diff* ermittelt die Unterschiede zweier Textdateien und gibt diese als minimale Liste der geänderten Zeilen aus.

Andere Algorithmen nutzen die Baumstruktur von HTML-Dokumenten zur Ermittelung der Differenzen. Auch XML-Dokumente verfügen über eine derartige Baumstruktur. Bei *X-Diff* [WDC03] handelt es sich um ein Tool, dem ein derartiger Algorithmus zu Grunde liegt. Hierbei wird zwar auf eine hierarchische Struktur verzichtet, jedoch nicht auf das Wissen um die Vorfahren.

Auch beim Vergleich von verschiedenen Programmcode-Versionen finden Differenzalgorithmen ihren Einsatz. In [Yan91] stellt Wuu Yang einen Algorithmus vor, der die syntaktischen Unterschiede zwischen zwei Versionen eines Programms anhand von Bäumen ermittelt. Hier wird allerdings nur ein sehr grober Algorithmus umrissen, der in der Praxis nie implementiert wurde. Die Idee hinter diesem Algorithmus beruht wiederum auf dem bereits vorgestellten *mysqldiff* [Spi07]. Da weder bei CSV- noch bei SQL-Dateien eine Baumstruktur zu Grunde liegt, kann keiner dieser Algorithmen für das vorliegende Problem eingesetzt werden.

Ferner treten bei Data Warehouses Aktualisierungsprobleme auf. Da diese Daten aus verschiedenen, sich permanent ändernden Datenbanken zusammen führen, muss stets ein Vergleich erfolgen, welche Daten gelöscht, hinzugefügt oder aktualisiert werden müssen. In [CLS00] setzten die Autoren zur Lösung dieses Problems auf den Aufbau des Netzwerkes. Da hier jedoch keine Softwarelösung vorliegt, wird es nur der Vollständigkeit halber erwähnt.

1.3 Implementierungsziele

Diese Arbeit verfolgt das Ziel, zwei große, im CSV-Format vorliegende Datenmengen zu synchronisieren. Hierbei werden – neben der Korrektheit der Synchronisation – folgende Implementierungsziele fokussiert:

- **Zeiteffizienz:** Das wichtigste Implementierungskriterium stellt die Zeiteffizienz dar. Zwar existieren weitaus schnellere Sprachen als Prolog, wie beispielsweise C oder Shell-Skripte, jedoch soll im Rahmen des in Prolog Möglichen eine möglichst zeiteffiziente Lösung gefunden werden.
- **Universell:** Ferner soll keine optimierte Lösung für eine konkrete Datenbank implementiert werden, sondern das entwickelte Programm soll auf Datenbankdumps mit unterschiedlichen Graden an Veränderungen anwendbar sein.
- **Wartbarkeit und Lesbarkeit:** Einen weiteren wichtigen Punkt stellt eine leichte Wartbarkeit dar. Diese wird durch die leichte Lesbarkeit von Prolog-Code unterstützt.
- **Komfortabel:** Zudem soll eine für den Endnutzer möglichst komfortable Lösung geschaffen werden, bei dem dieser nur einen Vorgang zur Synchronisation starten muss und nicht viele Einzelvorgänge.

1.4 Problemstellung

Große Firmen haben meist sehr große Datenmengen in ihren Datenbanken zu verwalten. Dies führt zu dem Problem, dass Anfragen an die Datenbank sehr lange dauern können. Gleiches gilt für das Einfügen von neuen Daten in die existierende Datenbank bzw. das Aktualisieren von Datensätzen in der existierenden Datenbank. Gerade bei sehr großen Datenmengen, die oftmals mehrere Millionen Datensätze umfassen, stellt dies ein elementares Problem dar, da das Einfügen einzelner Tabellen durchaus mehrere Stunden in Anspruch nehmen kann. Zudem muss beim Einfügen auch die Datensicherheit sowie die Korrektheit und Konsistenz der Daten sichergestellt werden.

Ziel dieser Arbeit ist die Entwicklung und Implementierung verschiedener praktischer Ansätze zur effizienten Synchronisation von Datenbanken mit einer hohen Anzahl an Datensätzen. Hierbei soll eine möglichst zeitoptimierte Synchronisation erfolgen. Die Änderungen der zu synchronisierten Datenbanken umfasst sowohl `INSERT`-, `UPDATE`-, als auch `DELETE`-Statements.

Hierzu soll im ersten Schritt ein Snapshot im CSV- oder SQL-Format der aktuellen Datenbank geladen werden. Dabei ist darauf zu achten, dass eine Vielzahl von Attributen unterstützt werden muss, wie beispielsweise Sonderzeichen oder Textbegrenzungszeichen. Im nächsten Schritt sollen die Änderungen zwischen dem aktuellen Abbild der Datenbank und den zu importierenden Daten ermittelt werden. Aus diesen Unterschieden wird dann mittels SWI-Prolog ein optimaler Satz von `INSERT`-, `UPDATE`- und `DELETE`-Statements gebildet, die zur Synchronisation notwendig sind. Anschließend sollen diese Daten auf die bereits existierende Datenbank angewendet werden. Hierbei wird in dieser Arbeit das Datenbanksystem MySQL verwendet.

Abschließend soll eine Evaluation Aufschluss über die Effizienz der Algorithmen geben und untersuchen, unter welchen Umständen welcher Algorithmus der effizienteste ist.

1.5 Überblick

Die Arbeit ist in folgende Kapitel untergliedert: Einleitend erfolgt in Kapitel 2 eine kurze Einführung in die Datenbankanfragesprache SQL sowie die logische Programmiersprache Prolog. Hierbei werden anhand eines Beispiels einer Lagerverwaltung die wichtigsten SQL-Statements sowie einige hilfreiche Prolog-Prädikate erklärt. Ferner wird das Dateiformat CSV vorgestellt.

In Kapitel 3 erfolgt anschließend eine genauere Betrachtung über die Funktionsweise einiger SQL-Statements sowie deren möglichst effizienten Einsatz.

Die daraus resultierenden Ergebnisse werden in Kapitel 4 eingesetzt, um ein Prolog-Programm diff2sql zu konzipieren, das ausgehend von einer Patch-Datei die zur Synchronisation nötigen SQL-Statements berechnet und generiert. Hierbei wird auch die Shell-Skriptesammlung tablediff vorgestellt, die die dazu nötige Patch-Datei erzeugt. Hierbei wird zunächst der Grundalgorithmus vorgestellt und darauf aufbauend verschiedenen Variationen des Algorithmus.

In Kapitel 5 wird das Shell-Skript csv2sql vorgestellt. Dieses bindet tablediff sowie das Prolog-Programm diff2sql ein und übernimmt alle Zwischenschritte, die für einen reibungslosen Ablauf nötigt sind. Zudem wendet es die generierten Statements auf die Datenbank an.

Bevor die Evaluation stattfindet, wird in Kapitel 6 der Java-basierte CSV-Generator vorgestellt, der für die Generierung der Testdatenbanken entwickelt wurde.

Mittels dieser Datenbanken erfolgt in Kapitel 7 eine Evaluation der Laufzeit, die die verschiedenen Algorithmen von diff2sql gegenüberstellt und ermittelt, unter welchen Voraussetzungen welcher Algorithmus von Vorteil ist.

Abschließend werden die Ergebnisse in Kapitel 8 zusammengefasst und basierend darauf ein Fazit gezogen. Zudem wird ein Ausblick auf weitere Entwicklungsmöglichkeiten gegeben.

2 Grundlagen

Das nachfolgende Kapitel soll eine kurze Einführung in die Datenbanksprache SQL, das Dateiformat CSV, den Linux Befehl sed sowie die logische Programmiersprache PROLOG geben, die zum weiteren Verständnis dieser Arbeit benötigt werden.

Zur Veranschaulichung wird durchgängig das Beispiel einer Lagerverwaltung verwendet. Jeder Artikel hat eine ID `produkt_id`, einen Standort `standort`, eine Lagerhalle `lagerhalle`, eine kodierte Position innerhalb dieser Lagerhalle `position`, die sich zusammensetzt aus Regal\Regalposition und eine Menge `menge`. Der Lagerort eines Teils wird in diesem Beispiel eindeutig über `standort`, `lagerhalle` sowie `position` definiert. Diese drei Felder bilden somit zusammen den Schlüssel. Tabelle 2.1 zeigt die Tabelle `lagerorte` mit einigen Tabelleneinträgen.

ProduktID	Standort	Lagerhalle	Position	Menge
0457	Dortmund	D8E5	47\54	0.30
8459	Berlin	Z8W0	02\87	77.00
3458	München	S7T3	02\87	2.50
2934	Dortmund	D8E5	78\54	6.10
8459	Dortmund	R7T5	56\76	1000.40

Tabelle 2.1: Beispieltabelle einer Lagerverwaltung

2.1 SQL-Grundlagen

Im Folgenden wird die Datenbanksprache SQL kurz vorgestellt. Hierbei werden neben den wichtigsten Statements auch die beschrieben, die zum Arbeiten mit CSV-Dateien hilfreich sind. Dies geschieht auf Grundlage des relationalen Datenbankverwaltungssystemes MySQL. Dabei wird auf Grund des großen Befehlsumfangs der Sprache nur auf die wichtigsten Befehle und Parameter eingegangen, die zum Verständnis dieser Arbeit beitragen.

SQL bietet eine Vielzahl von Befehlen zum Erstellen und Manipulieren von Datenbanken. Eine Aufstellung der möglichen Befehle sowie die ausführliche Syntax ist unter [MyS18e] zu finden. Dies bildet zusammen mit [Sei11, EN09] die Grundlage dieses Kapitels.

2.1.1 Einführung

Bei der *Structured Query Language*, kurz SQL, handelt es sich um eine Datenbanksprache für relationale Datenbanken. Sie ist der Nachfolger der in den 1970er Jahren von Donald D. Chamberlin und Raymond F. Boyce entworfen Datenbanksprache SEQUEL [CB74] und wurde 1986 vom American National Standards Institute (ANSI) standardisiert. Sie ermöglicht neben der Definition der Struktur der Datenbank auch das Lesen, Löschen, Einfügen und Ändern von Einträgen sowie die Rechteverwaltung und Transaktionskontrolle. [BJK11, onl18, HV01]

Relationale Datenbanken basieren auf dem relationalen Datenbankmodell, das erstmals 1970 von Edgar F. Codd vorgestellt wurde und bis heute Standard für die meisten Datenbanken ist. Der Name relationales Datenbankmodell leitet sich von der mathematischen Relation ab. Diese stellt eine mathematische Beschreibung einer Tabelle dar. [BJK11]

„Operationen auf diesen Relationen werden durch die relationale Algebra bestimmt. Diese sind somit die theoretische Grundlage von SQL.“ [BJK11]

Eine relationale *Datenbank* ist eine Ansammlung von einer oder mehreren Tabellen. Ein Beispiel für eine Tabelle ist in Abbildung 2.1 dargestellt. Eine *Tabelle* wiederum besteht aus einzelnen Feldern. In einem *Feld* steht genau eine Information, wie z.B. die Artikelnummer oder die Position. Mehrere Felder, die die selbe Art von Information enthalten, werden untereinander geschrieben und bilden eine *Spalte*. So bilden z.B. mehrere Felder, die alle IDs von Produkten beinhalten, eine Spalte mit dem Spaltennamen `produkt_id`.

Meist wird jedoch nicht nur eine Information zu einem Objekt gespeichert, sondern mehrere. Die Informationen zum selben Objekt werden horizontal in eine *Zeile* eingetragen. Somit bildet eine Zeile einen *Datensatz* zu einem Objekt. Um zu vermeiden, dass das selbe Objekt mehrmals in der Tabelle vorkommt, kann dieses mit einem Schlüssel versehen werden. Ein *Schlüssel* beschreibt ein Objekt eindeutig. Dieser kann aus einem oder mehreren Feldern bestehen. So kann beispielsweise die Produkt-ID ein Schlüssel für einen Artikel sein oder Standort, Lagerhalle und Position ein kombinierter Schlüssel für den Lagerort. Bei Letzterem ist keines der drei Felder allein eindeutig. Ein Standort hat mehrere Lagerhallen und diese wiederum mehrere Positionen, aber zusammen identifizieren sie den Lagerort eindeutig.

Anfragen oder Manipulationen, wie beispielsweise das Löschen oder Ändern von Datensätzen, werden als Statements oder Befehle bezeichnet.

Abbildung 2.1 veranschaulicht noch einmal die wichtigsten Begriffe.

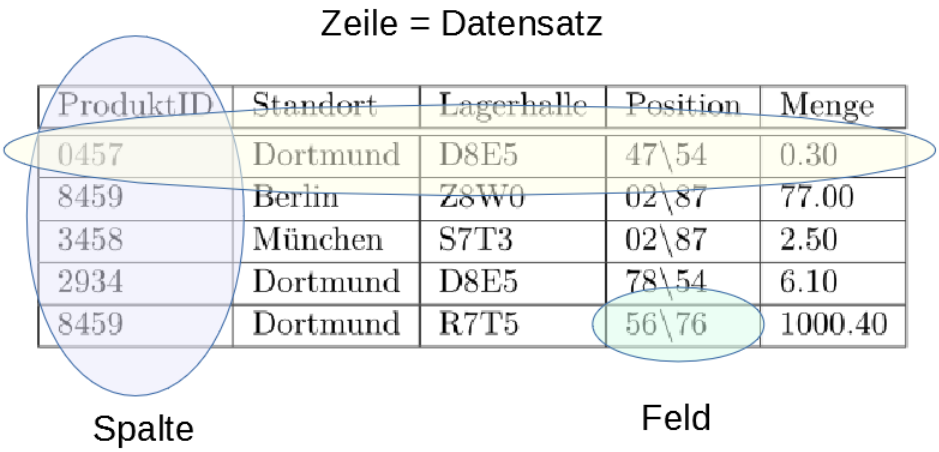

Abbildung 2.1: Begriffserklärung von Zeilen, Spalten und Feldern anhand einer Tabelle

2.1.2 Grundlegende SQL-Anweisungen

In diesem Unterkapitel soll auf die grundlegenden SQL-Statements eingegangen werden. Hierzu zählt das Erstellen von Datenbanken und Tabellen, sowie das Einfügen, Löschen und Aktualisieren von Datensätzen.

Erstellen von Datenbanken und Tabellen

Bevor einer Datenbank Tabellen oder Datensätze hinzugefügt werden können, muss diese zunächst erstellt werden. Dies geschieht mittels eines `CREATE`-Statements. Bei diesem Statement muss neben dem Typen, in diesem Fall `DATABASE`, noch der gewünschte Name `database_name` der Datenbank angegeben werden. Somit ergibt sich für das Erstellen einer Datenbank der folgende Befehl:

```
CREATE DATABASE database_name;
```

Befehle werden mit einem Semikolon abgeschlossen. Für die weitere Benutzung muss die Datenbank zunächst ausgewählt werden. Dies geschieht mittels des Befehls:

```
USE database_name;
```

Somit ergibt sich für das Anlegen und Selektieren einer neuen Datenbank mit der Bezeichnung `logistik` der folgende Code:

```
1 CREATE DATABASE logistik;
2 USE logistik;
```

Anlegen einer Tabelle

Nach dem Anlegen einer Datenbank können dieser nun Tabellen hinzugefügt werden. Die vereinfachte Syntax hierfür lautet:

```
CREATE TABLE table_name (
   column_1 datatype,
   column_2 datatype,
   ...
   PRIMARY KEY key_name
    ( column_k1,
      column_k2,
         ... )
);
```

Hierbei werden neben dem Tabellennamen `table_name` noch die jeweiligen Spaltennamen `column_n` mit dazugehörigem Datentyp `datatype` angegeben. Diese werden durch Kommas voneinander getrennt. Ferner wird der Name `key_name` des Primärschlüssels sowie die Spalten `column_kn`, aus denen sich dieser zusammensetzt, ergänzt. Die einzelnen Spalten stehen in einer Klammer und werden durch Kommas voneinander separiert. Der Befehl wird durch ein Semikolon abgeschlossen. Ein `datatype` setzt sich zusammen aus dem Datentypen sowie dessen maximale Länge. Die Länge wird hierbei als Ganzzahl in Klammern hinter den Datentypen geschrieben. Eine Auflistung der verfügbaren Datentypen ist unter [MyS18b] zu finden.

Um die Beispieltabelle 2.1 mit dem Namen `lagerorte` zu erzeugen, ergibt sich folgender SQL-Code:

```
CREATE TABLE lagerorte (
   produkt_id int(4),
   standort varchar(20),
   lagerhalle varchar(4),
   position varchar(5),
   menge decimal(8,2),
   PRIMARY KEY key_ort (
      standort, lagerhalle, position)
);
```

Als Tabellenname wird `lagerorte` gewählt. Die Produkt-ID `produkt_id` ist stets eine vierstellige Ganzzahl. Daher wird diese auf `int(4)` gesetzt. Die `4` gibt hierbei die Anzahl der Stellen an. Der Standort hingegen besteht nur aus Buchstaben.

Daher wird hierfür der Typ `varchar` gewählt. Die Länge wird auf 20 Zeichen begrenzt. Hierbei ist `varchar` dem Datentyp `char` vorzuziehen, da bei `char` die überschüssigen Zeichen mit Leerzeichen aufgefüllt werden, was zu einem erhöhten Speicherplatzbedarf führt. Das Feld `lagerhalle`, das eine Mischung aus Buchstaben und Zahlen darstellt, wird ebenfalls auf `varchar` und einer Länge von 4 Zeichen gesetzt. Analog dazu hat Position `position` 5 Zeichen und setzt sich aus Zahlen und einem `\` zusammen. Bei der Menge `menge` hingegen handelt es sich um eine Dezimalzahl. Daher wird für diese der Typ `decimal` gewählt. Die Anzahl der Vorkommastellen wird auf 8 gesetzt, die Anzahl der Nachkommastellen auf 2. Die beiden Angaben werden durch ein Komma getrennt. Der Schlüssel wiederum setzt sich aus den drei Spalten `standort`, `lagerhalle` und `position` zusammen. Der Name dieses kombinierten Schlüssels wird auf `key_ort` gesetzt.

Einfügen von Daten

Nun können der bestehenden Tabelle Datensätze hinzugefügt werden. Hierfür bietet SQL die `INSERT`-Anweisung. Die `INSERT`-Anweisung erlaubt es, neue Datensätze einzufügen. Hierbei existieren zwei Möglichkeiten. Bei der ersten wird in eine gegebene Tabelle `table_name` exakt eine Zeile mit n Werten `value_1` bis `value_n` eingefügt. Dabei ist n die Anzahl der Spalten. Listing 2.1 zeigt eine derartige `INSERT`-Anweisung.

Listing 2.1: Syntax eines `INSERT`-Statements zum Einfügen eines Datensatzes

```
1 INSERT INTO table_name
2 VALUES (value_1, value_2, value_3,...);
```

Der Befehl für das Einfügen mehrerer Zeilen auf einmal ist in Listing 2.2 zu finden.

Listing 2.2: Syntax eines `INSERT`-Statements zum Einfügen mehrerer Datensätze

```
1 INSERT INTO table_name
2 VALUES (value1_1, value1_2, value1_3, ...),
3        (value2_1, value2_2, value2_3, ...),
4          ... ;
```

Bei beiden Varianten dient das Komma als Separator und das Semikolon zum Befehlsabschluss. Ferner besteht noch die Möglichkeit Werte nur in bestimmte Spalten einzutragen. Da dies für diese Arbeit jedoch nicht benötigt wird, wird es nur der Vollständigkeit halber erwähnt.

Um der zuvor erzeugten Tabelle `lagerorte` nun die ersten beiden Spalten der Beispieltabelle 2.1 hinzuzufügen lautet der Code wie folgt:

Listing 2.3: Beispiel eines kombinierten `INSERT`-Statements

```
1 INSERT INTO lagerorte
2 VALUES
3     (0457, "Dortmund", "D8E5", "47\\54", 0.30),
4     (8459, "Berlin", "Z8W0", "02\\87", 77.00 );
```

Hierbei fällt auf, dass in der vorletzten Spalte statt eines einfachen ein doppelter Backslashes verwendet wurde. Dies liegt daran, dass einige Zeichen, wie u.a. der Backslash, in SQL eine Sonderfunktion haben. Sollen diese als normaler Text dargestellt werden, ist eine Maskierung nötig. Dies wird in Kapitel 2.1.5 genauer erklärt.

Löschen von Daten

Beim Bearbeiten von Tabellen müssen nicht nur neue Datensätze hinzugefügt, sondern auch alte Datensätze entfernt werden. Hierfür bietet SQL `DELETE`-Statements. Analog zum Einfügen von Datensätzen existiert auch für das Löschen von Datensätzen die Möglichkeit einen oder mehrere zu löschen. Die Grundstruktur für das Löschen eines Datensatzes sieht hierbei wie folgt aus:

```
1 DELETE FROM table_name
2 WHERE condition;
```

Neben dem Namen der Tabelle `table_name`, aus der die Werte gelöscht werden sollen, muss zudem die Bedingung `condition` angegeben werden, nach der diese ausgewählt werden. Die einzelnen Parameter dieser Bedingung werden mittels `AND` bzw. `OR` verknüpft. Soll hierbei lediglich eine Zeile gelöscht werden, so muss diese eindeutig angegeben werden. Dies geschieht beispielsweise über Angabe des kompletten Primärschlüssels. Sollen hingegen mehrere Datensätze gelöscht werden, so müssen die entsprechenden Schlüssel der Zeilen bzw. die gemeinsamen Werte angegeben werden

Um beispielsweise alle Vorkommen des Produkts mit der ID `8459` aus der Tabelle zu entfernen, bietet es sich an, gezielt nach dem Feld `produkt_id` mit dem Wert `8459` zu suchen. Der Code hierfür lautet:

```
1 DELETE FROM lagerorte
2 WHERE produkt_id = 8459;
```

Möchte man hingegen gezielt dieses Produkt nur aus dem Bestand in Dortmund entfernen und dabei sicherstellen, dass auch wirklich der Artikel in den anderen

Standorten erhalten bleibt, so empfiehlt es sich im `WHERE`-Teil die exakten Schlüssel anzugeben. Eine derartige Anfrage hat die Form:

```
1 DELETE FROM lagerorte
2 WHERE produkt_id = 8459 AND standort = "Dortmund";
```

Aktualisieren von Daten

Soll ein Datensatz nicht gleich gelöscht sondern nur geändert werden, so wird dies mit Hilfe eines `UPDATE`-Statements realisiert. Ein `UPDATE`-Statement erlaubt es ein oder mehrere Felder zu ändern, die eine bestimmte Bedingung erfüllen. Diese Bedingung bzw. Bedingungen werden im `WHERE`-Teil formuliert, wohingegen die neu gesetzten Werte im `SET`-Teil aufgeführt werden. Ein `UPDATE`-Statement hat somit die Form:

```
1 UPDATE table_name
2 SET column1 = value1, column2 = value2, ...
3 WHERE condition;
```

Um in Beispieltabelle 2.1 die Menge des Produktes mit der ID `8459` an allen Standorten auf den Wert `0` zu setzen, würde das SQL-Statement wie folgt heißen:

```
1 UPDATE lagerorte
2 SET menge = 0
3 WHERE produkt_id = 8459;
```

Mit diesem Statement wird die Menge in Dortmund und Berlin aktualisiert. Soll hingegen nur die Menge des Produktes in Dortmund geändert werden, so muss die `WHERE`-Bedingung um den konkreten Standort Dortmund erweitert werden.

```
1 UPDATE lagerorte
2 SET menge = 0
3 WHERE produkt_id = 8459 AND standort = "Dortmund";
```

Analog zu `DELETE` können auch hier Bedingungen mittels `AND` und `OR` verknüpft werden. Diese müssen entsprechend geklammert werden. Hierbei gelten die üblichen Regeln für logische Verknüpfungen.

2.1.3 Import und Export von Dateien

SQL wurde für die Ausführung über die Konsole konzipiert. Bei der Konsolenausführung kann stets ein Befehl eingegeben und dann ausgeführt werden. Möchte man mehrere Befehle nacheinander ausführen, z.B. zum Einfügen von neuen Datensätzen, gestaltet sich dies jedoch fehleranfällig. Daher bietet SQL die Möglichkeit Befehle aus Dateien heraus auszuführen. Ferner ist es möglich ganze Datensätze direkt aus CSV-Dateien zu importieren, ebenso wie zu exportieren.

Aufruf von Befehlen aus Dateien

Soll eine Reihe von Anweisungen hintereinander oder mehrmals ausgeführt werden, so besteht die Möglichkeit diese in einer Datei zu speichern. Mittels

```
source file_path
```

bzw.

```
\. file_path
```

kann die Datei geladen werden, in der sich die Befehle befinden. Hierbei ist zu beachten, dass `file_path` nicht in Hochkommata stehen darf.

Import von CSV-Dateien

Möchte man hingegen Datensätze einfügen, die bereits in einer CSV-Datei stehen, da die Werte beispielsweise aus Excel exportiert wurden, bietet SQL eine Möglichkeit diese direkt zu importieren. Listing 2.4 zeigt die grundlegende Syntax einer solchen Anfrage.

Listing 2.4: Syntax eines `LOAD-INTO`-Statements zum Import von CSV-Dateien

```
1 LOAD DATA INFILE file_path
2 INTO TABLE table_name
3 [{FIELDS | LINES} TERMINATED BY 'string']
4 [IGNORE number {LINES | ROWS}]
```

Die in eckigen Klammern stehenden Ausdrücke sind hierbei optional und können weggelassen werden. Mittels `TERMINATED BY 'string'` wird das Zeilen- bzw. Feldtrennzeichen `'string'` festgelegt. Dieses Trennzeichen muss stets in Hochkommata stehen. Das voranstellte `FIELDS` oder `LINES` gibt dabei an, ob es sich um das

Trennzeichen für die einzelnen Felder oder für die einzelnen Zeilen handelt. Wird dieser Teil weggelassen, so ist das Standardtrennzeichen für die einzelnen Felder ein Tabulator und das der einzelnen Zeilen ein systemabhängiger Zeilenumbruch. `IGNORE number {LINES | ROWS}` hingegen gibt an, ob einzelne Zeilen oder Spalten weggelassen werden können. Dies bietet sich z.B. an, wenn in der ersten Zeile des CSV's die Namen der einzelnen Spalten stehen. Der Pfad `file_path`, an dem sich die CSV-Datei befindet, die importiert werden soll sowie der Tabellenname `table_name`, in den die Datensätze eingefügt werden sollen, müssen stets angegeben werden. Die Tabelle muss hierbei bereits existieren und `file_path` muss in Hochkommata gestellt werden.

Um in die zuvor angelegte Tabelle `lagerorte` neue Datensätze aus der Datei `datensaetze.csv` zu laden, deren Felder mit Hilfe des Zeichens `|` voneinander getrennt sind und bei der die erste Zeile ignoriert werden soll, lautet der Code wie folgt:

```
1 LOAD DATA INFILE 'datensaetze.csv'
2 INTO TABLE lagerorte
3 FIELDS TERMINATED BY '|'
4 IGNORE 1 LINES;
```

Export von CSV-Dateien

Sind die Daten bereits in MySQL und sollen daraus in eine CSV-Datei exportiert werden, so ist dies mit `SELECT`-Statements möglich. Hierfür werden mittels eines `SELECT`'s die zu exportierenden Spalten ausgewählt. Ferner wird neben dem Tabellennamen noch der Pfad des entstehenden CSV-Files angegeben sowie die gewünschten Trennzeichen. Listing 2.5 zeigt die Syntax eines solchen `SELECT`-Statements. [MyS18h]

Listing 2.5: Syntax eines `SELECT`-Statements zum Export von Daten

```
1 SELECT column_1, column_2, column_3, ...
2 INTO OUTFILE output_path
3 [ {FIELDS | LINES} TERMINATED BY 'string' ]
4 FROM table_name;
```

Analog zum Import von CSV-Dateien werden auch hier die gewünschten Trennzeichen `'string'` für Zeilen und Felder in Hochkommata angegeben. Erfolgt dies nicht, so werden die Standardzeichen verwendet. Diese sind der Tabulator sowie der Zeilenumbruch. Ferner müssen im `SELECT`-Teil die Spalten angegeben werden, die

exportiert werden sollen. Zudem wird die Information benötigt, um welche Tabelle `table_name` es sich handelt, sowie in welche Datei `output_path` der Export erfolgen soll.

Für das Exportieren der gesamten Tabelle 2.1 mit dem Namen `lagerorte` in die Datei `export.csv`, mit `|` als Feldtrennzeichen und Zeilenumbruch `\r\n` als Zeilentrennzeichen (unter Windows), lautet die SQL-Anfrage:

```
1 SELECT produkt_id, standort, lagerhalle, position, menge
2 INTO OUTFILE 'export.csv'
3 FIELDS TERMINATED BY '|'
4 FROM lagerorte;
```

2.1.4 Weitere hilfreiche SQL-Befehle

Im Folgenden werden noch einige hilfreiche SQL-Befehle vorgestellt, die das Arbeiten mit MySQL und das Verständnis dieser Thesis vereinfachen. Dazu zählen das Anzeigen von Tabellen und Datenbanken sowie das Löschen selbiger.

Selektieren von Datensätzen

Neben der Manipulation stellt das Abrufen von Datensätzen eine elementare Operation einer Datenbank dar. Dies erlaubt der `SELECT`-Befehl. Bei einer `SELECT`-Anfrage wird eine Tabelle zurückgegeben, die alle Datensätze enthält, welche die Bedingungen aus dem `WHERE`-Teil erfüllen. Eine solche Anfrage hat die Form:

```
1 SELECT column_1, column_2, ...
2 FROM table_name
3 WHERE condition;
```

Im `SELECT`-Teil werden die Spalten angegeben, die ausgegeben werden sollen. Ferner muss die Tabelle `table_name` angegeben werden, aus der die Daten entnommen werden sollen, sowie welche Bedingung die angezeigten Daten erfüllen sollen.
Durch Weglassen des `WHERE`-Teils werden alle Inhalte der ausgewählten Spalten angezeigt. Möchte man alle Spalten auswählen, so kann auf das Aufzählen der einzelnen verzichtet werden und stattdessen ein `*` nach dem `SELECT` eingefügt werden.

Anzeigen der verfügbaren Datenbanken und Tabellen

Mittels des `SELECT`-Befehls lässt sich der Inhalt einer Tabelle anzeigen. Möchte man hingegen die Struktur der Tabelle, alle verfügbaren Datenbanken oder alle Tabellen in einer Datenbank einsehen, so ist dies mittels einer `SHOW`-Anfrage möglich.

Die Anfrage für das Anzeigen der Tabellenstruktur heißt dabei

```
SHOW COLUMNS FROM table_name;
```

Möchte man hingegen alle verfügbaren Datenbanken auflisten, so benötigt man den Befehl

```
SHOW DATABASES;
```

Für das Auflisten der Tabellen hingegen

```
SHOW TABLES FROM db_name;
```

Löschen von Datenbanken und Tabellen

Der `DELETE`-Befehl zum Löschen einzelner Datensätze wurde bereits vorgestellt. Oftmals möchte man jedoch nicht nur einzelne Datensätze, sondern ganze Tabellen oder Datenbanken löschen. Dies ist mittels des `DROP`-Befehls möglich. Zum Löschen einer einzelnen Tabelle lautet der Befehl hierbei:

```
DROP TABLE table_name;
```

Der Befehl für das Löschen einer ganzen Datenbank hingegen lautet

```
DROP DATABASE db_name;
```

2.1.5 Maskieren von Sonderzeichen

Wie in anderen Programmiersprachen gibt es auch in SQL einige besondere Steuerzeichen. Dazu zählen u.a. die Hochkommata `'` sowie die Wildcards `%` und `_`. Genauere Information über die Wildcards sind unter [w3s18] zu finden. Damit beim Einlesen der Daten zwischen den Tabelleninhalten und diesen Sonderzeichen unterschieden werden kann, ist es nötig, diese zu maskieren. Dies geschieht nach folgenden Regeln [MyS18f]:

- **Hochkommata** werden mittels Verdopplung maskiert. Der Datensatz `Ben's` wird somit zu `Ben''s` und `''Dortmund''` zu `''''Dortmund''''`.
- **Backslashes** werden mittels Verdopplung maskiert. Somit wird `12\47` zu `12\\47`.
- **Wildcards** hingegen werden durch ein vorangestelltes `%\` sowie einem abschließenden `%` maskiert. Somit ergibt sich für die Wildcard `_` die Maskierung `%\_%`. Für die Wildcard `%` hingegen lautet die Maskierung `%\%%`. Hierbei ist jedoch darauf zu achten, das der Backslash vorab als Escape-Sequenz in SQL festgelegt wurde.

2.2 CSV-Format

Das folgende Kapitel gibt einen kurzen Überblick über das Dateiformat CSV. Hierbei werden neben der Spezifikation die wichtigsten Regeln anhand eines Beispiels erläutert.

2.2.1 Einführung

Ein weit verbreitetes Dateiformat für Tabellen stellt *CSV* dar. „Dem Einsatz des Kommas als Trennzeichen verdankt das Dateiformat im übrigen auch seinen Namen: Comma Seperated Values. Allerdings ist das Dateiformat nicht standardisiert, so dass es etliche Abwandlungen gibt. Die häufigste Variante ist der Austausch des Kommas durch ein anderes Trennzeichen, weswegen CSV oft auch als Akronym für Character Seperated Values verstanden wird.“ [LM07]

Das Dateiformat wird oftmals zum Austausch von tabellarischen Daten zwischen zwei verschiedenen Programmen oder Betriebssystemen genutzt. Auch die Abspeicherung als Textdatei, die ein direktes Lesen und eine guten Portierbarkeit erlaubt, ist ein großer Vorteil. CSV wurde zwar teilweise von XML abgelöst, erfreut sich aber auch heute noch aufgrund des einfachen Formats großer Beliebtheit. [LM07]

2.2.2 Spezifikation

Zwar existieren im Internet viele Spezifikation für CSV, jedoch keine standardisierte. Nachfolgend sollen einige Regeln aufgeführt werden, die in den meisten Spezifikationen vertreten sind [Sha05, Ray03]:

- Jeder Datensatz wird in eine eigene Zeile geschrieben, die durch ein Zeichen abgeschlossen wird. Dieses Zeichen (Begrenzungszeichen) trennt somit die einzelnen Datensätze voneinander. Meist wird hierfür der Zeilenumbruch des jeweiligen Betriebssystems verwendet. Im Fall von Windows ist dies `\r\n`. Bei Linux hingegen `\n`.
- Jeder Datensatz wird durch ein Begrenzungszeichen beendet. Nur beim letzten ist das Begrenzungszeichen optional.
- Bei der ersten Zeile kann es sich um eine Kopfzeile handeln. Diese beinhaltet die Spaltennamen und hat die selbe Anzahl an Feldern wie die restlichen Zeilen.
- Innerhalb jeder Zeile, auch der Kopfzeile, werden die einzelnen Felder durch ein Feldbegrenzungszeichen voneinander getrennt. Dies ist oftmals ein Komma. Das letzte Feld einer Zeile muss nicht per Feldbegrenzungszeichen abgeschlossen werden.
- Jede Zeile soll die gleiche Anzahl an Feldern haben. Leerzeichen werden hierbei als Teil des Feldes angesehen und sollten nicht ignoriert werden.
- Ein Feld kann optional in Textbegrenzungszeichen gesetzt werden. Falls jedoch Sonderzeichen innerhalb des Feldes stehen, wie beispielsweise Zeilenumbrüche, Anführungszeichen oder Kommas, so sollten die Felder in Textbegrenzungszeichen gesetzt werden. Als Textbegrenzungszeichen werden häufig Anführungszeichen verwendet.
- Falls ein Feld in Textbegrenzungszeichen steht und innerhalb dieses Feldes erneut ein Textbegrenzungszeichen auftritt, muss letzteres mittels eines weiteren Textbegrenzungszeichen maskiert werden.

2.2.3 Beispiel

Wandelt man die Beispieltabelle 2.1 in CSV-Format um, so ergibt sich folgender Dateiinhalt:

```
1 "ProduktID"|"Standort"|"Lagerhalle"|"Position"|"Stückpreis"
2 "0457"|"Dortmund"|"D8E5"|"47\54"|"0.30"
3 "8459"|"Berlin"|"Z8W0"|"02\87"|"77.00"
4 "3458"|"München"|"S7T3"|"02\87"|"2.50"
5 "2934"|"Dortmund"|"D8E5"|"78\54"|"6.10"
6 "8459"|"Dortmund"|"R7T5"|"56\76"|"1000.40"
```

Hierbei dient `|` als Feldtrennzeichen sowie der Zeilenumbruch als Zeilentrennzeichen. Da innerhalb der Felder mitunter Sonderzeichen vorkommen, ist ein Textbegrenzungszeichen notwendig. In obigem Beispiel wurden hierfür Anführungszeichen gewählt.

Handelt es sich bei dem CSV um einen Export aus SQL, so sind die Sonderzeichen wie beispielsweise der Backslash nach den in Abschnitt 2.1.5 vorgestellten Regeln zu maskieren. Demnach würde z.B. das vorletzte Feld der ersten Zeile nicht `"47\54"` lauten, sondern `"47\\54"`.

2.3 Prolog

In diesem Kapitel erfolgt eine kurze Einführung in die logische Programmiersprache Prolog, da die in Kapitel 4 vorgestellten Algorithmen in Prolog implementiert sind. Hierbei wird neben der grundlegenden Syntax und Semantik auch auf das Konzept des Backtrackings eingegangen.

An dieser Stelle sei darauf hingewiesen, dass sich auf die wichtigsten Aspekte beschränkt wird, die zum Verständnis der Arbeit benötigt werden. Eine detaillierte Einführung ist in den Büchern [Bra01] und [CM03] zu finden. Diese bilden zusammen mit [Sei15] die Grundlage dieses Kapitels.

2.3.1 Einführung

Bei Prolog handelt es sich um eine logische Programmiersprache, die 1972 vom französischen Informatiker Alain Colmerauer konzipiert wurde. [Coh04, CR96] Prolog hat sich vor allem einen Namen auf den Gebieten der künstlichen Intelligenz, Expertensysteme, Computerlinguistik und domänenspezifischen Sprachen gemacht. So wird Prolog beispielsweise in der Arbeit [SNA17] eingesetzt, um Expertensysteme mittels domänenspezifische Sprache darzustellen. Aber auch im hier relevanten Fall, den Datenbanken, erfreut sich Prolog großer Beliebtheit. Hierzu zählen vor allem die deduktiven Datenbanken. [Han13]

Bei SWI-Prolog wiederum handelt es sich um eine freie Implementierung von Prolog. Diese steht seit 1987 unter ständiger Weiterentwicklung durch den Hauptautor Jan Wielemarker. SWI-Prolog hat sich vor allem im Bereich der Lehre und des Semantic Webs durchgesetzt. [swi18a]

SWI-Prolog selbst wurde in C implementiert und verfügt daher über eine Vielzahl von Schnittstellen hierzu. Daneben bietet es die Möglichkeit, sich über Interfaces,

wie beispielsweise CAPJA [OFS14] oder ODBC [Wie18], mit anderen Programmiersprachen wie Java bzw. SQL zu verbinden. Neben dieser Eigenschaft verfügt SWI-Prolog auch noch über eine umfangreiche Anzahl von Bibliotheken für Constraint-Programmierung, Multithreating, GUIs und vieles mehr. [swi18a, WSTL12]

2.3.2 Funktionsweise

Bei herkömmlichen Programmiersprachen, wie beispielsweise Java oder C, gibt der Programmierer einen Lösungsweg vor, anhand dessen der Computer dann die Lösungen ermittelt. Prolog hingegen verfolgt den Ansatz, dass lediglich das bereits bekannte Wissen und die logischen Zusammenhänge kodiert werden und Prolog aus diesen selbstständig eine Lösung ermittelt. Das bereits bekannte Wissen wird hierbei durch Fakten repräsentiert, die logischen Zusammenhänge durch Regeln. Mittels Anfragen kann dann auf dieses Wissen zugegriffen werden. Die Gesamtheit von Fakten, Regeln und Anfragen wird als Klauseln bezeichnet. Die Bausteine, aus denen sich diese zusammensetzen, werden Terme genannt. Bei diesen wird zwischen Konstanten, Variablen und Strukturen unterschieden.

Prolog bedient sich bei der Lösungsfindung des Konzepts der Unifizierung und des Backtrackings. Bei der *Unifizierung* wird mittels Variablenersetzung versucht, zwei Terme aneinander anzugleichen, d.h. es werden zwei Terme miteinander verglichen und versucht, geeignete Variablenbelegungen für die ungebundenen Variablen zu finden. Beim *Backtracking* hingegen wird ein Lösungsweg solange verfolgt bis dieser scheitert. Tritt dies ein, so wird zum letzten Entscheidungspunkt zurückgekehrt und von dort aus neu gesucht, bis eine gültige Lösung gefunden oder alle möglichen Wege überprüft wurden.

Eine weitere Besonderheit von Prolog ist der Verzicht auf feste Datentypen. Es gibt jedoch einige Elemente, zwischen denen unterschieden wird. Abbildung 2.2 gibt einen Überblick über die verschiedenen Sprachelemente von Prolog.

2.3.3 Terme

Die gröbste Unterscheidung von Prolog-Sprachelementen besteht zwischen Klauseln und Termen. Bei Termen handelt es sich um die einzelnen Bausteine, aus denen sich die Klauseln zusammensetzen. Hierbei wird zwischen Konstanten, Variablen und Strukturen unterschieden.

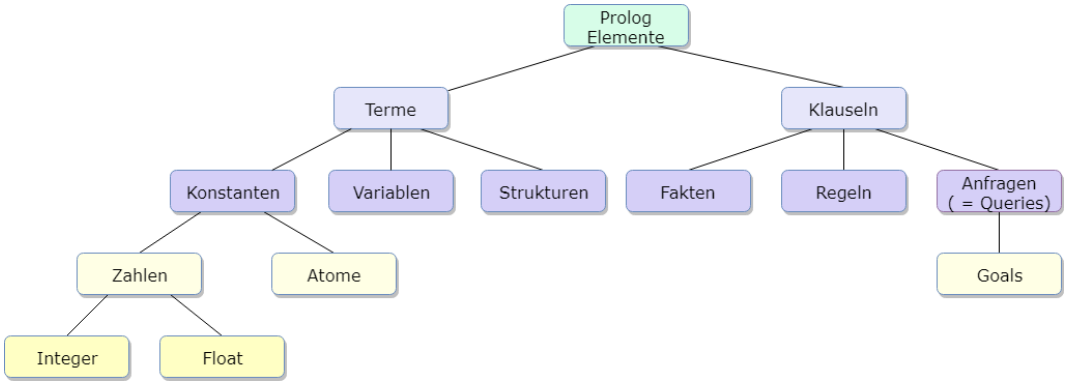

Abbildung 2.2: Übersicht über die verschiedenen PROLOG-Elemente

Konstanten

Die kleinste logische Einheit in PROLOG stellen *Konstanten* dar. Diese kommen zum Einsatz, wenn der Wert des Objektes bereits bekannt ist. Hierbei wird zwischen den Zahlen und den symbolischen Konstanten, den Atomen, unterschieden.

Die *Zahlen* wiederum unterteilen sich in Ganzzahlen (*Integer*) und Gleitkommazahlen (*Float*). Bei letzteren dient der Punkt als Trennzeichen zwischen Vor- und Nachkommastellen. Bei der Zahl `8459` handelt es sich beispielsweise um einen Integerwert. Eine Mengenangabe `0.30` ist ein Float.

Objekte oder Eigenschaften werden hingegen durch die symbolischen Konstanten, die *Atome*, beschrieben. Da in PROLOG auf Datentypen verzichtet wurde, ist die korrekte Benennung der einzelnen Typen von großer Bedeutung. Daher folgt die korrekte Nomenklatur eines Atoms einer der folgenden Regeln :

- Ein Atom ist eine Zeichenfolge aus Großbuchstaben, Kleinbuchstaben, Ziffern und Unterstrichen, die stets mit einem Kleinbuchstaben oder einer Zahl beginnt.
- Ein Atom ist eine Zeichenfolge beliebiger Zeichen, die mit einem Apostroph beginnt und mit einem Apostroph endet.
- Ein Atom besteht nur aus Sonderzeichen. Dazu zählen `!`, `#`, `$`, `%`, `&`, `*`, `+`, `,`, `-`, `.`, `/`, `:`, `;`, `>`, `<`, `?`, `@`, `\`, `~`, `{`, `}`. Hierbei haben die vier Zeichenketten `,`, `;`, `!` und `[]` eine vordefinierte Bedeutung und dürfen daher nicht beliebig verwendet werden.

Somit handelt es sich bei folgenden Beispielen um gültige Konstanten:

```
1 0457
2 0.30
3 dortmund
4 'Dortmund'
```

Bei den ersten beiden Konstanten handelt es sich um Zahlen. Das erste ist ein Integer, das zweite ein Float. Bei Zeile 3 und 4 handelt es sich hingegen um Atome, die das Objekt Dortmund beschreiben. Damit das Objekt in Zeile 4 groß geschrieben werden konnte, wurde es in Hochkommata gesetzt.

`'Dortmund`, `Dortmund` oder `0.30` hingegen sind keine gültigen Atome.

Variablen

Analog zu anderen Programmiersprachen, wie Java oder C, existieren auch in Prolog *Variablen*. Diese sind jedoch keine Speicherzellen mit festem Datentyp, in denen Werte gespeichert und ausgelesen werden können, sondern Platzhalter, an deren Stelle jeder beliebige Term eingesetzt werden kann. Da in Prolog keine Datentypen existieren, wird anhand der Nomenklatur zwischen Variablen und Konstanten bzw. Atomen unterschieden. Variablen beginnen stets mit einem Großbuchstaben oder Unterstrich, gefolgt von beliebig vielen Groß- und Kleinbuchstaben sowie Zahlen.

Beispiele für gültige Variablen sind somit:

```
1 Lagerorte
2 _Lagerorte
3 _LaGeRoRtE_____123
4 _123
5 _
```

Bei `123Lagerorte`, `lagerorte` oder `'Lagerorte'` hingegen handelt es sich nicht um Variablen sondern um Atome.

Das letzte Beispiel, die Variable `_`, hat hierbei eine Sonderstellung. Sie wird als *anonyme Variable* bezeichnet und verwendet, falls der Wert der Prolog Variable nicht ausgegeben werden soll, z.B. da der Wert im weiteren Programmverlauf nicht mehr benötigt wird. Kommt die anonyme Variable innerhalb einer Regel mehrmals vor, so handelt es sich nicht zwangsläufig um die selbe Variable.

Strukturen

Mittels Konstanten und Variablen ist es möglich, einzelne Objekte oder Eigenschaften abzubilden. Allerdings erlauben sie es nicht, Relationen zwischen diesen herzustellen. Um dies zu ermöglichen wird PROLOG um die Strukturen erweitert.

Strukturen erlauben es, Objekte in Relation zueinander zu setzen. Sie haben die Form

```
Funktor(Argument_1, Argument_2, .... Argument_n)
```

Hierbei ist der *Funktor* ein Atom. Bei den Argumenten kann es sich um Konstanten, Variablen oder wieder um Strukturen handeln. Diese stehen in runden Klammern und werden durch Kommata voneinander getrennt. Die Anzahl der Argumente wird als *Stelligkeit* bezeichnet.

Listing 2.6: Beispiele für PROLOG-Strukturen

```
1 artikel('0457','Dortmund', 'D8E5', '47\54', '0.30')
2 artikel('3458', Standort, _, _, Menge)
3 artikel
```

Zeile 1 aus Listing 2.6 beschreibt den exakten Lagerort eines Artikels mit einer Artikelnummer. Struktur 2 beschreibt ebenfalls einen Lagerort, jedoch sind hier Standort und Menge variabel gehalten und Lagerhalle und Position als irrelevant gekennzeichnet. Bei der Struktur, die lediglich aus einem Funktor und keinen Argumenten besteht, wie in Zeile 3, handelt es sich wiederum eine Konstante.

Listen

Eine Sonderform der Struktur stellen die Listen dar. Eine *Liste* ist eine Sammlung geordneter Terme, deren Länge nicht festgelegt werden muss.

Die einfachste Form einer Liste ist die leere Liste, die durch das Atom `[]` repräsentiert wird. Ein- oder mehrelementige Listen werden durch eine Struktur mit dem Funktor `•` dargestellt, wobei das erste Argument der Kopf und das zweite Argument der Rumpf ist. Der *Kopf* ist stets ein Term. Der *Rumpf* hingegen muss wieder eine Liste sein. Bei dem letzten Element der innersten Liste muss es sich stets um die leere Liste handeln. Beispiele für Listen sind somit:

```
1 []
2 •(Berlin, [])
3 •(Berlin, •(München, •(Dortmund, [])))
```

Betrachtet man das letzte Beispiel, so wird ersichtlich, dass diese Schreibweise bei längeren Listen schnell unübersichtlich und somit fehlerträchtig wird. Daher bietet PROLOG noch eine zweite Schreibweise an. Bei dieser wird auf den Funktor verzichtet. Stattdessen werden die Elemente direkt in eckige Klammern geschrieben. Wandelt man die oben aufgeführten Beispiele in diese Notation um, so ergeben sich folgende Schreibweisen:

```
1 []
2 [Berlin]
3 [Berlin, München, Dortmund]
```

Für die weitere Arbeit wird die Infix-Notation verwendet, da diese mehr Übersichtlichkeit bietet.

Ausgehend von dieser Notation erlaubt es eine dritte Schreibweise die Liste in Kopf und Rumpf zu unterteilen. Bei dieser steht die Liste ebenfalls in eckigen Klammern. Kopf und Rumpf werden jedoch durch einen senkrechten Strich voneinander getrennt. Somit hat diese Notation folgende Form: `[Kopf|Rumpf]`. Hierbei können sowohl Kopf als auch Rumpf beliebig viele Elemente enthalten. Der Kopf muss jedoch stets mindestens ein Element enthalten. Die Elemente des Rumpfs müssen jedoch selbst wieder in einer Liste stehen, d.h. der Rumpf selbst ist eine Liste. Für die letzte Liste des obigen Beispiels wären somit die nachfolgenden Notationen gültig:

```
1 [Berlin|[München, Dortmund]]
2 [Berlin, München|[Dortmund]]
3 [Berlin, München, Dortmund|[]]
```

Mittels dieser Notation ist es möglich, auf einzelne Elemente am Anfang der Liste zuzugreifen bzw. Elemente am Kopf der Liste anzuhängen.

Um Elemente ans Ende der Liste anzuhängen, bzw. ganze Listen miteinander zu kombinieren, bietet PROLOG das Prädikat `append/3`. Mittels `delete/3` hingegen ist es möglich, Elemente wieder zu entfernen. Neben diesen bietet PROLOG noch eine Vielzahl anderer Prädikate zum Arbeiten und Manipulieren von Listen. Eine Auflistung dieser ist unter [swi18c] zu finden.

2.3.4 Klauseln

Terme sind die einzelnen Bausteine eines PROLOG-Programms, jedoch alleinstehend keine gültigen Anweisungen. Terme können jedoch zu Klauseln erweitert werden, die gültige PROLOG-Anweisungen darstellen. Hierbei unterteilen sich Klauseln in

Fakten, Regeln und Anfragen (vgl. Abb. 2.2). Damit eine Klausel gültig ist und von PROLOG bearbeitet wird, muss diese stets durch einen Punkt abgeschlossen werden.

Fakten

Die einfachste Erweiterungen stellen Fakten dar. Ein Fakt ist bekanntes Wissen in der Wissensbasis. Es steht für sich allein und nimmt keinen Bezug auf andere Fakten. Syntaktisch stellt es eine Struktur (komplexe Struktur, Konstante, Liste, etc.) dar, die von einem Punkt abgeschlossen wird.

Listing 2.7 zeigt die in Fakten umgewandelten Strukturen aus Abschnitt 2.6.

Listing 2.7: Beispiele für PROLOG-Fakten

```
1 artikel('0457','Dortmund', 'D8E5', '47\54', '0.30').
2 artikel('3458', Standort, _, _, Menge).
3 artikel.
```

Fakten repräsentieren somit die Daten einer Datenbank. Die Gesamtheit der Fakten bildet zusammen mit den Regeln die Datenbank bzw. Wissensbasis.

Regeln

Regeln stellen im Gegensatz zu Fakten kein für sich allein stehendes Wissen dar. Sie beziehen sich stets auf bereits als Fakten oder Regeln repräsentiertes Wissen.

Eine Regel besteht aus einem Regelkopf und einem Regelrumpf, die durch den Funktor `:-` voneinander getrennt sind. Der Kopf besteht hierbei aus einer Struktur. Der Rumpf wiederum besteht aus einem oder mehreren Termen. Diese Terme werden mittels Kommata für Konjunktionen und Strichpunkten für Disjunktionen miteinander verknüpft. Analog zu den Fakten wird eine Regel durch einen Punkt beendet.

Existieren eine oder mehrere Klauseln mit dem selben Funktor, wie beispielsweise in Listing 2.8, werden diese als Prädikat bezeichnet, die Anzahl der Argumente wiederum als Stelligkeit des Prädikats. Die gängige Notation für Prädikate lautet: Funktor/Stelligkeit. Somit ergibt sich für das `member`-Prädikat die Notation `member/2`.

Die Reihenfolge der Prädikate bestimmt die Aufrufreihenfolge. PROLOG ruft stets das erste Prädikat mit der entsprechenden Stelligkeit auf. Erst wenn dieses scheitert, wird mittels Backtracking das nächste zutreffende Prädikat aufgerufen.

Das in Listing 2.8 gezeigte Prädikat `member/2` überprüft, ob ein Element in einer Liste enthalten ist. Hierzu wird im ersten Schritt die übergebene Liste in das erste Element und die restliche Liste unterteilt. Besteht die Liste aus nur einem Element und handelt es sich bei diesem um das gesuchte Element, so liefert die erste Regel `true` zurück. Das Backtracking wird jedoch dennoch angestoßen, da die Möglichkeit besteht, dass das Element sich häufiger in der Liste befindet. Andernfalls liefert die erste Regel `false` zurück und mittels Backtracking wird die zweite Regel aufgerufen. Diese unterteilt die Liste in das erste Element und den Rest. Anschließend überprüft das erste Prädikat, ob es sich bei dem ersten Element um das gesuchte handelt. Ist dies der Fall, so liefert PROLOG das Ergebnis `true` zurück. Anschließend wird wieder das zweite Prädikat aufgerufen, das das erste Element von der Liste trennt und überprüft. Ist das letzte Element erreicht, so bricht die Regel ab.

Listing 2.8: Beispiel einer Regel

```
1 %member(?Element, ?List).
2 member(X, [X|_]).
3
4 member(X, [_|Tail]) :-
5   member(X, Tail).
```

Die darüberstehende, mit `%` beginnende Zeile, stellt ein Kommentar dar. Dieses dient dazu anzuzeigen, welche Argumente übergeben und welche vom Prädikat zurückgegeben werden. Das `?` vor `Element` und `List` steht hierbei für eine ungebundene Variable. Diese kann sowohl als Eingabe als auch als Ausgabe fungieren. Ein `+` hingegen stünde für eine Eingabe, eine `-` wiederum für eine Ausgabe.

Anfragen

Mittels Regeln und Fakten wird bekanntes Wissen abgespeichert. Anfragen erlauben es auf dieses Wissen zuzugreifen. Eine Anfrage besteht hierbei aus einem oder mehreren Goals.

Ein *Goal* ist ein Term, für das PROLOG versucht eine geeignete Unifizierung zu finden. Existiert eine gültige Unifizierung, so gibt PROLOG `true` bzw. die erste gefundene Lösung auf der Konsole aus. Mittels `;` kann das Backtracking angestoßen und somit weitere Lösungen ermittelt werden. Existiert keine gültige Lösung gibt PROLOG `false` aus.

Eine Aneinanderreihung von einem oder mehreren Goals wird als *Anfrage* oder auch *Query* bezeichnet. Hierbei werden die einzelnen Goals mittels Konjunktion bzw.

Disjunktion kombiniert. Die Konjunktion wird durch ein Komma, die Disjunktion durch einen Strichpunkt dargestellt.

Ist ein Einzelargument einer Konjunktion falsch, so ist die gesamte Konjunktion falsch. Sind jedoch alle Elemente wahr, so ist die Konjunktion `true`. Bei einer Disjunktion hingegen genügt es, wenn ein Goal wahr ist, damit die ganze Aussage `true` ist. Nur wenn alle Goals `false` sind, ist die Anfrage `false`.

2.3.5 Weitere hilfreiche Prolog-Prädikate

Nachfolgend sollen einige weitere hilfreiche PROLOG-Prädikate vorgestellt werden, die für das weitere Verständnis der Arbeit hilfreich sind.

Lesen und Schreiben in Dateien

Bevor eine Datei ausgelesen bzw. in eine Datei geschrieben werden kann, muss zunächst ein Stream zu dieser Datei geöffnet werden. Dies geschieht mittels des Prädikats `open(+SrcDes, +Mode, -Stream)`.

`open/3` öffnet einen I/O-Stream `Stream` des Files `SrcDes` im Modus `Mode`. `SrcDes` ist hierbei der Dateiname bzw. der Dateipfad inklusive Dateiname. Dieser wird als Fakt oder String dargestellt. Der Modus kann `read` (lesen), `write` (schreiben), `append` (hinzufügen) oder `update` (aktualisieren) sein. Bei `append` wird die Datei zwar zum Schreiben geöffnet, jedoch wird im Gegensatz zu `write` die Datei nicht überschrieben, sondern die neuen Daten ans Ende der bisherigen Datei angehängt. `update` hingegen fügt die neuen Daten am Anfang der Datei hinzu. Der Modus `read` öffnet die Datei nur zum Lesen, d.h. es können keine Änderungen an der Datei vorgenommen werden.

Ferner existiert noch das Prädikat `open/4`. Bei diesem können als viertes Argument noch Optionen übergeben werden. Eine Liste aller möglichen Optionen ist unter [swi18d] zu finden. Anschließend ist die Datei bereit zum Auslesen bzw. Hineinschreiben. Zum Auslesen der Daten bietet PROLOG eine Reihe von Prädikaten.

Im Nachfolgenden soll nur das in der Implementierung in Kapitel 4 vorgestellte Prädikat `read_line_to_codes(+Stream, -Codes)` eingeführt werden. Dieses liest die nächste Zeile des Inputstreams `Stream` und unifiziert diese mit der Variablen `Codes`. Eine Zeile wird dabei durch den betriebssystemabhängigen Zeilenumbruch oder dem Atom `end-of-file` abgeschlossen. Das Atom `end-of-file` steht hierbei für das Dateiende.

Abschließend muss der `Stream` mittels `close(+Stream)` wieder geschlossen werden. Hierbei ist Stream die zuvor in `open` festgelegte Variable.

Das Prädikat `writeln(+Stream, +Text)` ermöglicht das zeilenweise Schreiben in eine Datei. Als erstes Argument wird der zuvor mittels `open` geöffnete Stream `Stream` übergeben und als zweites der zu schreibende Text `Text`. Abschließend wird der Stream mittels `close` geschlossen.

Listing 2.9 zeigt ein Beispiel zum Schreiben des Faktes `row(a,b,c)` in die Datei `C:\Users\File.txt`.

Listing 2.9: Beispiel für das Schreiben in eine Datei

```
1 ?- open('C:\\Users\\File.txt', write, Out),
2    writeln(Out, [row(a,b,c)]),
3    close(Out)).
```

Anwenden eines Prädikates auf eine Liste

In der PROLOG-Programmierung wird umfangreich vom Prinzip der Rekursion Gebrauch gemacht. Diese kann für einfachere Anfragen, wie beispielsweise das Anwenden eines Prädikats auf eine ganze Liste, sehr schnell viel Code erzeugen und somit sehr unübersichtlich werden, was wiederum zu einer höheren Fehleranfälligkeit führt. Um dies zu vermeiden bietet PROLOG das Meta-Prädikat `maplist(:Goal, ?List)`. Dieses wendet das Prädikat `Goal` auf die Liste `List` an.

Lisiting 2.10 zeigt die Funktionsweise von `maplist` am Beispiel der Ausgabe einer Liste auf die Konsole.

Listing 2.10: Beispielaufruf von maplist/3

```
1 ?- maplist(writeln, ['Berlin', 'Dortmund']).
2 Berlin
3 Dortmund
4 true.
```

Als Goal wird das Prädikat `writeln/2` auf die beiden Prädikate der Liste angewandt. Als Ergebnis werden dann die beiden Atome aus der Liste auf die Konsole geschrieben.

Finden ausgewählter Fakten

Analog zum `SELECT`-Statement in SQL existiert in PROLOG das `findall`-Prädikat. `findall(+Template, :Goal, -Bag)` liefert alle Fakten in Form einer Liste zurück, die das gewünschte Kriterium erfüllen. Bei `Template` handelt es sich um den gesuchten PROLOG-Term. Dieser kann eine einzelne Variable, eine Struktur oder eine Liste von Variablen sein. Das Suchkriterium, das den Term erfüllen soll, wird wiederum in Goal angegeben. `Goal` kann hierbei jeder aufrufbare Term sein. Das Ergebnis dieser Suche wird als Liste in `Bag` ausgegeben. Wird keine Lösung für `Goal` gefunden, wird `Bag` mit der leeren Liste unifiziert.

Eine Anfrage zum Ermitteln aller vorhandener Artikel, könnte wie folgt heißen:

Listing 2.11: Beispielaufruf von findall/3

```
1 ?- findall(Artikel, artikel(Artikel, _, _, _, _), Liste).
```

Da eine Auflistung aller Artikel gesucht ist, wird als `Template` die Variable `Artikel` gesetzt. Als Suchkriterium `Goal` wird das Fakt `artikel` verwendet. Hierbei wird für das erste Argument die Variable `Artikel` gesetzt, da dies das gesuchte Element ist. Für die anderen wird die anonyme Variable eingesetzt, da diese bei der Suche keine Rolle spielen.

Neben dem Prädikat `findall` existieren noch die beiden Prädikate `bagof` und `setof` zum Durchsuchen von Fakten.

`bagof` und `findall` ähneln sich zwar im Aufbau, unterscheiden sich jedoch in der Ergebnisausgabe. Während bei `findall` alle gefundenen Ergebnisse in einer Liste ausgegeben werden, wird bei `bagof` das gesuchte Ergebnis in mehreren Listen sortiert nach den freien Variablen ausgegeben.

Ein Beispiel (vgl. Listing 2.12) soll die Unterschiede, sowie die Funktionsweise von `bagof` verdeutlichen. Hierfür wurde das Beispiel aus Tabelle 2.1 etwas vereinfacht. Statt fünf Feldern existieren nur die drei Felder Artikelname, Standort und Lagerhalle. Gesucht sind alle existierenden Artikel.

Die neuen Fakten sind in Zeile 2 bis 6. Die Zeilen 9 und 12 beinhalten Anfragen mittels des `findall`-Prädikats, Zeile 15 das Äquivalent mit `bagof`-Prädikat. Bei `findall` werden die gefundenen Ergebnisse bzw. Konstanten wie im vorherigen Beispiel in einer Liste ausgegeben. Hierbei spielt es keine Rolle, ob die Variablen gesetzt sind (vgl. Zeile 9) oder einige davon anonyme Variablen sind (vgl. Zeile 12), ausgeschlossen die gesuchte Variable.

Bei `bagof` hingegen macht dies in der Ergebnisausgabe einen Unterschied (vgl. Zeile 15 und 20). Werden bei `bagof` alle Variablen gesetzt (Zeile 15) so wird das Suchergebnis nach allen ungebundenen Variablen sortiert. Setzt man jedoch die Lagerhalle auf die anonyme Variable, so wird diese zwar in der Ausgabe nicht angezeigt, die Sortierung ändert sich jedoch nicht.

Möchte man jedoch eine Sortierung der Artikel nach den Standort, d.h. unabhängig von den Lagerhallen, so bietet `bagof` den Infix-Operator `^`. Dieser wird zwischen der zu ignorierenden Variable und den Suchterm gestellt (vgl. Zeile 25).

Listing 2.12: Vergleich von findall/3 und bagof/3

```
1  ?- listing(data).
2  data('Schraube','Dortmund','D8E5').
3  data('Mutter','Dortmund','D8E5').
4  data('Reifen','Dortmund','S7T3').
5  data('Radlager','Berlin','S7T3').
6  data('Felge','Berlin','S7T3').
7  true.
8
9  ?- findall(A, data(A, B, C), As).
10 As = ['Schraube', 'Mutter', 'Reifen', 'Radlager', 'Felge'].
11
12 ?- findall(A, data(A, B, _), As).
13 As = ['Schraube', 'Mutter', 'Reifen', 'Radlager', 'Felge'].
14
15 ?- bagof(A, data(A, B, C), As).
16 B = 'Berlin', C = 'S7T3', As = ['Radlager', 'Felge'] ;
17 B = 'Dortmund', C = 'D8E5', As = ['Schraube', 'Mutter'] ;
18 B = 'Dortmund', C = 'S7T3', As = ['Reifen'].
19
20 ?- bagof(A, data(A, B, _), As).
21 B = 'Berlin', As = ['Radlager', 'Felge'] ;
22 B = 'Dortmund', As = ['Schraube', 'Mutter'] ;
23 B = 'Dortmund', As = ['Reifen'].
24
25 ?- bagof(A, C^data(A, B, C), As).
26 B = 'Berlin', As = ['Radlager', 'Felge'] ;
27 B = 'Dortmund', As = ['Schraube', 'Mutter', 'Reifen'].
```

Neben `findall/3` und `bagof/3` existiert noch das Prädikat `setof/3`. Dieses verhält sich äquivalent zu `bagof/3`, sortiert jedoch die Ergebnisliste und filtert Duplikate heraus.

2.3.6 Arbeiten mit externen Datenbanken in Prolog

Zwar kann PROLOG als eigenständige Datenbanksprache eingesetzt werden, dennoch bietet es die Möglichkeit, auf bereits vorhandene Datenbanken im CSV- oder SQL-Format zuzugreifen und mit diesen zu arbeiten. Nachfolgend sollen die wichtigsten Prädikate hierfür vorgestellt werden.

Einlesen von Daten mittels ODBC

Liegen die SQL-Datenbanken auf einem Server, so bietet SWI-PROLOG die Möglichkeit mittels einer ODBC-Verbindung direkt auf diese zuzugreifen. Hierzu muss zuerst mittels des PROLOG-Prädikates `odbc_connect(+DSN, -Connection, +Options)` eine ODBC-Verbindung aufgebaut werden. `DSN` gibt hierbei den Data Source Name an. Der Alias zu dieser Verbindung wird in der Variable `Connection` gespeichert. Falls zusätzliche Informationen, wie beispielsweise ein Passwort, benötigt werden, so können diese mittels der letzten Variable `Options` übergeben werden. Eine Liste der möglichen Optionen ist unter [Wie18] zu finden. Wird keine Option benötigt, so wird die leere Liste übergeben.

Anschließend kann mittels `odbc_query(+Connection, +SQL, -RowOrAffected)` eine Anfrage in Form eines SQL-Statements gestellt werden. Mittels des `findall`-Prädikates ist es anschließend möglich, alle Werte als Strukturen mit dem Funktor `row` in einer Liste abzuspeichern. Ein Beispiel hierfür befindet sich in Listing 2.13.

Listing 2.13: Aufbau einer Datenverbindung mittels ODBC

```
1 getAllValues(SQL) :-
2   odbc_connect(dsn, C, []),
3   findall(Row, odbc_query(C, 'SELECT * FROM lagerorte', Row), SQL).
```

Verzichtet man auf das `findall` und verwendet textvergleichend `odbc_query/2`, so erhält man als Rückgabe anstelle einer Liste die einzelnen Datensätze als Fakten, die mittels Backtracking durchlaufen werden müssen. Da bei dieser Variante jedoch entweder eine komplette Liste durchlaufen werden muss, bzw. für jede neue Anfrage wieder eine Verbindung mit der Datenbank aufgebaut werden muss, ist diese Variante sehr zeitaufwendig. Daher wird sie in dieser Arbeit nur der Vollständigkeit halber erwähnt.

Lesen und Schreiben von CSV-Dateien

Ferner stellt SWI-Prolog die Bibliothek `library(csv)` zur Verfügung, die das Parsen und Generieren von CSV-Daten in Prolog ermöglicht. Die CSV-Daten werden hierbei als Liste von Spalten repräsentiert. Jede Spalte ist hierbei ein kombinierter Term, bei dem alle Spalten den selben Namen und die selbe Stelligkeit haben. [swi18b]

Mittels der Prädikate `csv_read_file/2` bzw. `csv_read_file/3` erfolgt das Einlesen in diese Liste. Als erstes Argument wird das einzulesende File bzw. der Pfad des einzulesenden Files übergeben. An zweiter Position folgt die Variable, in der die Liste mit den Zeilen gespeichert werden soll. Als drittes Argument kann dann optional eine Liste mit den gewünschten Optionen übergeben werden. Zu diesen zählen u.a. die Prädikate `functor/1` oder `separator/1`, die es erlauben, den Standardfunktor bzw. den Standardseparator auf einen gewünschten Wert zu setzen. Werden keine Optionen gesetzt, so ist der Standardseparator `','` und der Standardfunktor `row`. Eine vollständige Liste der Optionen ist unter [swi18b] zu finden. Abschließend kann die Liste mittels `maplist` der Wissensbasis hinzugefügt werden.

Listing 2.14 zeigt einen Beispielcode zum Einlesen einer existierenden CSV-Datei.

Listing 2.14: Beispiel für das Lesen aus einer CSV-Datei

```
1 ?- csv_read_file('C:\\temp.csv', Rows, [functor(data), separator('|')]),
2    maplist(assert, Rows).
```

Das obige Beispiel liest das CSV aus dem Dateipfad `C:\tmp.csv` aus und und speichert dieses als Liste in die Variable `Rows`. Da das Ursprungsfile als Separator `|` hat, wird mittels `separator('|')` dieser für das Einlesen geändert. `functor(data)` ändert den Funktor von `row` auf `data`. Mittels `maplist` werden dann die als Liste in der Variablen `Rows` abgespeicherten Datensätze der Faktenbasis einzeln hinzugefügt.

Neben einem Prädikat zum Einlesen von CSV bietet SWI-Prolog auch ein Prädikat `csv_write_file/2` bzw. `csv_write_file/3` zum Schreiben eines CSV's in eine Datei. Hierzu wird zuerst ein Stream mit dem Atom `write` geöffnet zu der Datei, in die geschrieben werden soll. Mittels des Prädikats `csv_write_file/3` wird anschließend der Befehl zum Schreiben gegeben. Das Prädikat erhält an der ersten Position den zuvor geöffneten Stream. Als zweites Argument wird der zu schreibende Text in Form einer Liste übergeben. Anschließend wird der Stream mittels `close/1` wieder geschlossen.

Listing 2.15: Beispiel für das Schreiben in eine CSV-Datei

```
?- open(File, write, Out),
   csv_write_file(Out, [row(a,b,c)], []),
   close(Out).
```

Neben den vorgestellten Prädikaten gibt es noch eine Reihe anderer Prädikate, die das Arbeiten mit CSV's vereinfachen. Eine Auflistung aller Prädikate zum Arbeiten mit CSV's unter Prolog ist unter [swi18b] zu finden.

3 Performanzbetrachtung von SQL-Befehlen

Im vorherigen Kapitel wurden die wichtigsten SQL-Befehle vorgestellt, darunter auch `INSERT`, `DELETE` und `UPDATE`. Dieses Kapitel gibt einen Einblick in den Laufzeitbedarf der einzelnen Befehle. Auf dieser Basis soll der möglichst zeiteffiziente Einsatz erläutert werden. Die daraus resultierenden Ergebnisse sollen dann im PROLOG-Programm in Kapitel 4 zum Einsatz kommen. Zudem werden einige weitere allgemeine Optimierungsmöglichkeiten aufgezeigt, die jedoch nicht im Rahmen dieser Arbeit realisiert wurden.

3.1 INSERT

Bevor die Optimierung der Statements vorgestellt wird, soll kurz ein Einblick über die Verteilung des Laufzeitbedarfs gegeben werden. Dies geschieht exemplarisch am Beispiel des `INSERT`-Statements. Dies geschieht in Anlehnung an das offizielle MySQL-Manual [MyS18d].

Der Zeitaufwand für das Einfügen eines neuen Datensatzes mittels `INSERT` setzt sich wie folgt zusammen (der näherungsweise Anteil ist dabei jeweils in Klammern dahinter angegeben):

- Aufbau der Verbindung zum Server (3)
- Abschicken der Anfrage an den Server (2)
- Übersetzung der Anfrage (2)
- Einfügen der Zeile (1 x Datensatzlänge)
- Einfügen der Indizes (1 x Anzahl der Indizes)
- Schließen der Verbindung zum Server (1)

Hierbei ist der einmalige Aufwand für das Öffnen der Tabelle nicht berücksichtigt. Dieser fällt für jede auszuführende Anfrage einmal an.

Ein weiterer wichtiger Einflussfaktor ist – neben der bereits genannten Datensatzlänge sowie der Anzahl der Indizes – die bereits vorhandene Anzahl an Datensätzen in der Tabelle.

Berücksichtigt man diese Faktoren, so sieht eine optimaler Einfügeprozess wie folgt aus: Einzelstatements werden zu möglichst großen Blöcken zusammengefasst. Anschließend wird einmalig eine Verbindung zum Server hergestellt. Dann erfolgt das Übermitteln des Statements. Erst wenn alle Daten eingefügt sind, erfolgt das Aktualisieren der Indizes sowie die Konsistenz-Überprüfung.

Als Ergebnis für die Generierung der Statements in Kapitel 4 folgt somit, dass ein kombiniertes `INSERT`-Statement vielen einzelnen Statements vorzuziehen ist. Um dies noch einmal zu überprüfen, sowie den Grad der Effizienzsteigerung zu ermitteln, wurde dennoch eine Variante implementiert, in der Einzelstatements generiert werden. In Kapitel 7 werden dann die beiden Methoden bezüglich ihres Zeitbedarfs verglichen. Zudem existieren auch Vorteile von Einzelstatements. So erlauben sie es beispielsweise eine Abarbeitung der Statements in mehreren Threads. Diese Untersuchung ist jedoch nicht Teil dieser Arbeit.

Standard-SQL-`INSERT`-Statements lassen sich jedoch nur kombinieren, wenn diese vom selben Client stammen. Dies ist im Rahmen dieser Arbeit gegeben. Sind große Datenmengen von verschiedenen Clients einzufügen, so bietet MySQL die `INSERT DELAYED`-Anweisung an. Diese bündelt Einfügeoperationen von verschiedenen Clients und behandelt diese dann wie ein kombiniertes `INSERT`-Statement. [MyS18a]

Befinden sich die Datensätze hingegen in einer Textdatei, rät das offizielle MySQL-Manual [MyS18d] von der Verwendung von `INSERT` ab, und empfiehlt stattdessen die Verwendung von `LOAD-INTO`, da dieses um den Faktor 20 schneller sein soll.

3.2 DELETE

Analog zu `INSERT`-Statements existiert auch für `DELETE`-Statements die Möglichkeit, mehrere Einzelanweisungen zu einem kombinierten zusammenzufassen.

Über den zeitlichen Vorteil eines kombinierten Statements gegenüber von Einzelstatements ist zwar im offiziellen Manual [MyS18e] nichts Konkretes zu finden, jedoch ist anzunehmen, dass dieser ähnlich zu dem bei `INSERT`-Statements ist. Dies soll im Rahmen der in Kapitel 7 durchgeführten Messungen genauer betrachtet werden.

Das Manual [MyS18c] nennt jedoch für Einzelstatements eine Optimierungsmöglichkeit mittels des Zusatzes `LIMIT 1`. Damit hört die Suche nach dem zu löschenden

Element auf, sobald ein Datensatz gefunden wurde, auf den diese Bedingung zutrifft. Andernfalls würde MySQL auch nach dem Löschen des gefundenen Datensatzes noch die verbleibenden Datensätze durchsuchen, um sicherzustellen, dass die Bedingung nicht noch auf einen zweiten Datensatz zutrifft. Dies funktioniert jedoch nur bei einem Einzelstatement. Bei einem kombinierten Statement würde nach der ersten erfüllten Bedingung abgebrochen werden und die anderen Teile der Konjunktion würden nicht mehr gelöscht werden.

Ferner können in der `WHERE`-Bedingung statt einer Bedingung für alle Felder eine nur die Schlüsselfelder umfassende gebildet werden. Dies erspart den Aufwand die anderen Spalten zu überprüfen.

3.3 UPDATE

Im Gegensatz zu `INSERT`- und `DELETE`-Statements gibt es bei `UPDATE` keine Möglichkeit diese zusammenzufassen. Jedoch nennt das Manual [MyS18g] auch hier zwei Optimierungsmöglichkeiten. Zum einen besteht analog zu `DELETE` die Möglichkeit mittels `LIMIT 1` die Suche nach dem ersten gefundenen Datensatz abzubrechen. Zudem ist auch hier eine `WHERE`-Bedingung von Vorteil, die lediglich die Schlüsselfelder umfasst.

Aber auch im `SET`-Teil kann eine Optimierung vorgenommen werden, indem lediglich die Felder aufgeführt werden, die sich geändert haben. Zwar überprüft MySQL selbst, ob sich der Wert eines Feldes geändert hat und aktualisiert diesen nur dann, jedoch kostet auch diese Überprüfung Zeit.

3.4 Zusammenfassung

In diesem Kapitel wurde der möglichst effiziente Einsatz der drei MySQL-Statements `INSERT`, `DELETE` und `UPDATE` vorgestellt. Dieser besteht für `INSERT`-Statements darin, diese zu möglichst großen Blöcken zusammenzufassen statt viele Einzelstatements zu bilden.

Selbiges gilt für `DELETE`-Statements. Soll zudem ein Element gelöscht werden, das eindeutig über seine Primärschlüssel identifizierbar ist, so sollen auch nur diese Primärschlüsselfelder in der `WHERE`-Bedingung aufgelistet werden. Handelt es sich um ein Einzelstatement, das über seine Primärschlüssel definiert ist, so kann mittels des Zusatzes `LIMIT 1` der Löschvorgang beschleunigt werden.

`LIMIT 1` kann auch stets bei `UPDATE`-Statements angewendet werden, die einen einzelnen Datensatz betreffen. Das gleiche wie bei `DELETE` gilt auch hier für die `WHERE`-Bedienung. Zudem kann ein `UPDATE`-Statement schneller ausgeführt werden, wenn im `SET`-Teil nur die Felder aufgelistet werden, die sich ändern.

4 Synchronisation zweier CSV-Dateien mittels des Shell-Scripts tablediff

In Kapitel 3 wurde die Optimierung von SQL-Befehlen behandelt. In diesem Kapitel soll nun u.a. dieses Wissen eingesetzt werden, um einen Satz von SQL-Statements zu ermitteln, der für die optimierte Synchronisation von zwei CSV-Dateien notwendig sind. Das PROLOG-Programm diff2sql, das dies übernimmt, bildet den Mittelpunkt dieser Arbeit. Hierzu wird eine kurze Einführung in das Shell-Skript tablediff gegeben, das der Vorverarbeitung der Daten dient. Zudem werden die grundlegenden Algorithmen von diff2sql vorgestellt sowie dessen wichtigsten Module. Abschließend werden noch die bei der Implementierung aufgetretenen Probleme sowie deren Lösung aufgezeigt.

4.1 Gesamtablauf des Synchronisationsprozesses

Ausgangspunkt sind zwei SQL-Tabellen: Die Ausgangstabelle, sowie die, an der Änderungen vorgenommen wurden. Diese beiden gilt es zu synchronisieren. Hierzu werden im ersten Schritt beide Tabellen mittels eines `SELECT`-Statements (vgl. Listing 2.5) in ein CSV-File exportiert. tablediff errechnet anschließend aus den beiden ihm übergebenen CSV-Dateien ein Patch-File, das die Unterschiede sowie die Art der Änderung zwischen den beiden Dateien aufzeigt (vgl. Abschnitt 4.2.3). Die Ausgabe kann jedoch - vor allem bezüglich der change-Abschnitte - noch optimiert werden. Daher werden vor allem diese erneut mittels PROLOG ausgewertet und die entsprechenden SQL-Statements generiert. Diese werden dann auf die Ausgangstabelle angewendet, sodass diese identisch mit der geänderten SQL-Tabelle ist.

Der Datenfluss des kompletten Vorgangs ist in Abbildung 4.1 mittels eines Datenflussdiagramms verdeutlicht.

4.2 Die Shell-Skriptsammlung tablediff

Zur Vorverarbeitung der CSV-Dateien wird das freie Tool tablediff eingesetzt. Bei *tablediff* [Nog18] handelt es sich um Shell-Skripte zur Synchronisation zweier CSV-

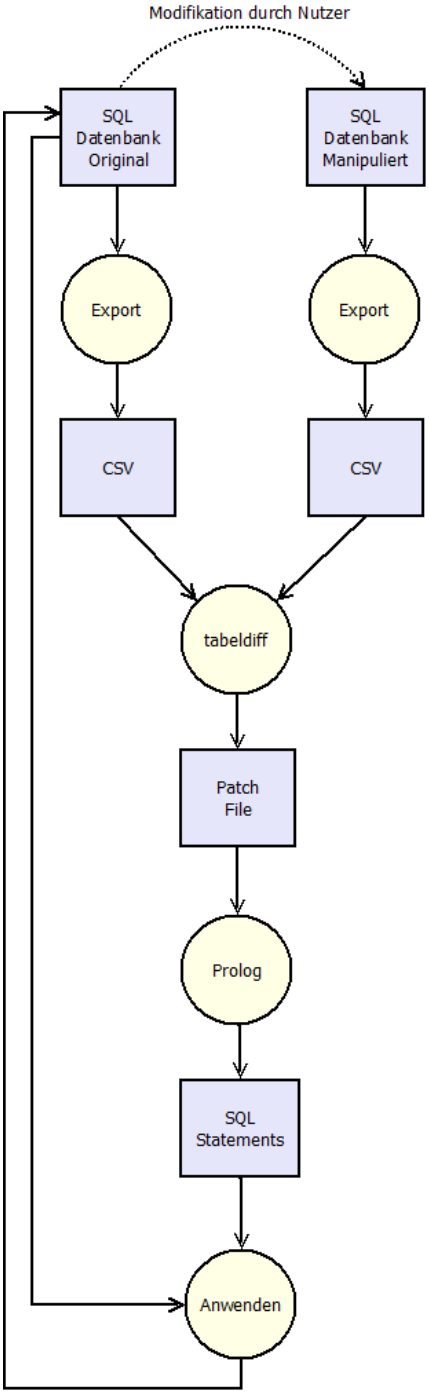

Abbildung 4.1: Datenflussdiagramm des gesamten Synchroniationsprozesses

Dateien. Es besteht aus den beiden Skripten diff.sh und patch.sh. *diff.sh* erstellt hierbei ein File, das die minimalen Änderungen zwischen den zwei CSV-Dateien sowie die Art der Änderung beinhaltet. Dies geschieht auf Basis der angegebenen Primärschlüssel-Spalten. *patch.sh* erstellt anhand des von diff.sh erstellten Files die dazugehörigen SQL-Statements, die nötig sind, um die beiden Tabellen ineinander zu überführen. Im Nachfolgenden soll eine genauere Betrachtung des Skriptes diff.sh stattfinden. Um hierbei eine Verwechslung mit dem Linux-eigenen diff auszuschließen, wird in der weiteren Arbeit tablediff als Synonym für diff.sh verwendet.

4.2.1 Algorithmus

Der exakte Sourcecode von diff.sh ist unter [Nog18] zu finden. An dieser Stelle soll nur kurz auf die Idee hinter dem Algorithmus eingegangen werden.

diff.sh bietet zwei Variationen an. Diese werden bereits beim Aufruf des Skriptes ausgewählt. Im simplen Modus findet lediglich eine Sortierung der Dateieinträge anhand der Primärschlüssel statt. Anschließend erfolgt mittels des Linux-Befehls diff ein zeilenweiser Vergleich der beiden sortierten CSV-Dateien. Dies Variante hat den Nachteil, dass sie zeitlich ineffizient ist.

Die zweite Variante ist etwas komplexer, jedoch auch schneller. Bei dieser wird vor der Sortierung eine Aufteilung nach dem Teile-und-herrsche-Verfahren vorgenommen. Anschließend werden Datensätze entfernt, die in beiden Dateien vorkommen. Daraufhin werden die einzelnen Teile sortiert und wieder zusammengefügt. Abschließend erfolgt analog zum simplen Algorithmus ein zeilenweiser Vergleich mittels diff.

4.2.2 Nomenklatur

Nachfolgend sollen einige Begriffe für die weitere Arbeit festgelegt werden. Abbildung 4.2 soll die Begrifflichkeit noch einmal verdeutlichen.

Bei Zeilen der Form `3d2`, `1c1,2` und `0a1` handelt es sich um eine *Zuordnungszeile*. Diese geben an, um welche Art von Änderung es sich bei dem jeweils nachfolgenden Block handelt. Hierbei steht ein `d` für delete (löschen), ein `a` für add (hinzufügen) und ein `c` für change (ändern). Im Gegensatz zum normalen Linux-diff kann hier die Information der Zeilennummer nicht genutzt werden, da diese durch das Sortieren nicht mehr mit den Eingabedateien übereinstimmt.

Bei einer Zeile der Form `---` handelt es sich hingegen um eine *Trennzeile*. Diese dient allein der besseren Übersichtlichkeit und existiert nur in den change-Blöcken.

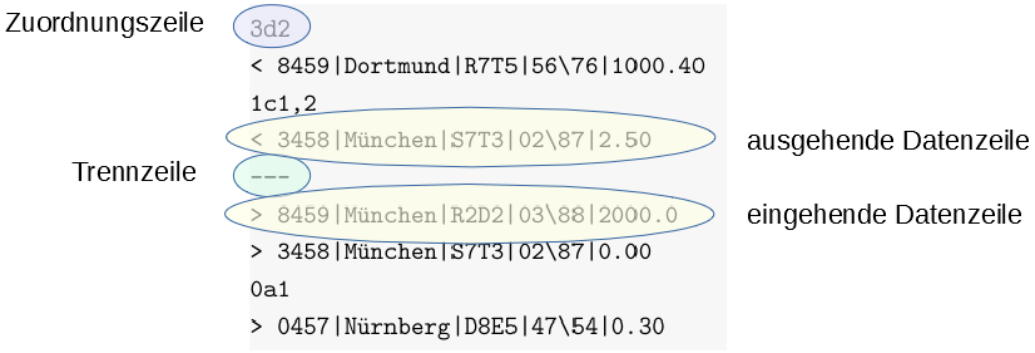

Abbildung 4.2: Begriffserklärung der tablediff-Ausgabe

Die dritte Art von Zeile ist die *Datenzeile*. Diese enthält den eigentlichen Datensatz. Zudem enthält sie die Information, aus welcher der beiden CSV-Dateien die Zeile stammt. Beginnt die Zeile mit einem `>` handelt es sich um einen *eingehende Datenzeile*, d.h. die Zeile stammt aus der geänderten CSV-Datei . Bei `<` hingegen um eine *ausgehende Datenzeile*. Diese bezieht sich auf die Ausgangs-CSV-Datei.

4.2.3 Funktionsweise

Beide Varianten von tablediff haben die übliche Ausgabe eines per diff generierten Patch-Files. Listing 4.1 zeigt eine Beispiels-CSV-Datei, die in 4.2 überführt werden soll [1]. Der Primärschlüssel setzt sich aus dem zweiten, dem dritten und dem fünften Feld zusammen.

Die Änderungen sind hierbei:

- Zeile 1 in Listing 4.2 **eingefügt**.
- Zeile 6 in Listing 4.2 **eingefügt**.
- Zeile 5 aus Listing 4.1 wurde **entfernt**.
- In Zeile 3 in Listing 4.1 wurde die Menge von 2.50 auf 0 **geändert**.

[1] Der Befehl für die Überführung lautet:
`./diff.sh --empty --delimiter='|' --primary=2,3,4 orignial.csv manipuliert.csv`

Listing 4.1: Originaldatei

```
1 0457|Dortmund|D8E5|47\54|0.30
2 8459|Berlin|Z8W0|02\87|77.00
3 3458|München|S7T3|02\87|2.50
4 2934|Dortmund|D8E5|78\54|6.10
5 8459|Dortmund|R7T5
      |56\76|1000.40
```

Listing 4.2: Manipulierte Datei

```
1 1111|Nürnberg|A1A1|00\00|100.0
2 0457|Dortmund|D8E5|47\54|0.30
3 8459|Berlin|Z8W0|02\87|77.00
4 3458|München|S7T3|02\87|0.00
5 2934|Dortmund|D8E5|78\54|6.10
6 8459|München|R2D2|03\88|2000.0
```

Das Ergebnis dieser Überführung ist in Listing 4.3 abgebildet.

Listing 4.3: Ausgabe von tablediff

```
1 3d2
2 < 8459|Dortmund|R7T5|56\76|1000.40
3 1c1,2
4 < 3458|München|S7T3|02\87|2.50
5 ---
6 > 8459|München|R2D2|03\88|2000.0
7 > 3458|München|S7T3|02\87|0.00
8 0a1
9 > 1111|Nürnberg|A1A1|00\00|100.0
```

Aus dieser geht hervor, dass die Zeile `8459|Dortmund|R7T5|56\76|1000.40` (entspricht der 5 Zeile der Ausgangsdatei) entfernt wurde.

Zudem muss die Zeile `3458|München|S7T3|02\87|2.50` (Zeile 3 der Ausgangsdatei) geändert werden, so dass diese in die Zeilen `8459|München|R2D2|03\88|2000.0` und `3458|München|S7T3|02\87|0.00` (Zeile 6 bzw. 4 der geänderten Datei) überführt wird.

Zeile 8 und 9 besagen, dass die Zeile `1111|Nürnberg|A1A1|00\00|100.0` (Zeile 1 der geänderten Datei) ergänzt werden muss.

Problem

Vergleicht man die von tablediff gefunden Änderungen mit den tatsächlich vorgenommen, wird das Problem ersichtlich. Das Einfügen von Zeile 1 und das Löschen von Zeile 5 wird zwar erkannt, jedoch wird das Einfügen der 6. Zeile nicht als Einfügen, sondern als Änderung gekennzeichnet. Dies ist zwar nicht falsch, da Einfügen auch eine Form der Änderung darstellt. Da jedoch die minimale Anzahl an Änderungen

anzustreben ist, sollen diese change-Blöcke noch per PROLOG erneut analysiert werden.

4.3 Das Prolog-Programm diff2sql

Das Programmflussdiagramm aus Abbildung 4.3 zeigt den prinzipiellen Ablauf des PROLOG-Programms. Für die vorliegende Masterarbeit wurden hierbei verschiedene Varianten des Programms implementiert. Die Abbildungen 4.3, 4.4 sowie 4.5 zeigen nicht die effizienteste Variante, sondern die, welche die grundlegende Idee hinter dem Algorithmus am besten darstellt. Die anderen Varianten werden an den entsprechenden Stellen im Text erläutert.

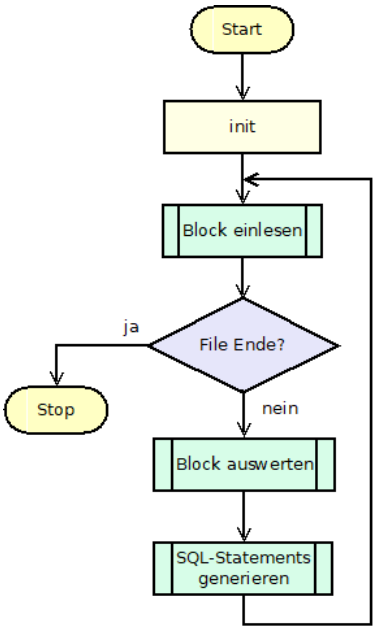

Abbildung 4.3: Programmflussdiagramm von diff2sql

Der Ablauf lässt sich in vier Abschnitte unterteilen: Initialisierung, Einlesen des Patch-Files, Auswertung und Erzeugung der SQL-Statements.

- **Initialisierung**: Bei dieser werden die vom Benutzer festgelegten Parameter eingelesen, sowie das Berechnen und Hinzufügen einiger Fakten durchgeführt.
- **Einlesen**: Anschließend erfolgt das blockweise Einlesen des mittels tablediff generierten Patch-Files.
- **Auswerten**: Der eingelesene Block wird im nächsten Schritt ausgewertet, d.h. es wird überprüft, um welche Art von Block es sich handelt und wie dieser weiterverarbeitet werden muss.

- **Generierung**: Danach erfolgt die Generierung der SQL-Statements für diesen Block.

Anschließend wird der nächste Block eingelesen, ausgewertet und die entsprechenden Statements generiert, bis das Dateiende erreicht ist.

4.3.1 Initialisierung

Zu Beginn erfolgt die Initialisierung einiger Parameter durch den Benutzer, indem er folgende Fakten in die Datei initialize.pl einträgt:

- **input/1** gibt den Dateipfad des mittels tablediff generierten Patch-Files an. Dieses dient als Grundlage des Weiteren Algorithmus.
- **output/1** gibt den Dateipfad an, in den die vom Programm generierten SQL-Statements abgespeichert werden sollen.
- **tablename/1** gibt den Namen der SQL-Tabelle an, die manipuliert werden soll. Dieser wird zur Generierung der SQL-Statements benötigt.
- **field/2** gibt an, an welcher Spaltennummer, im Folgenden Position genannt, sich ein Feld befindet und wie dieses benannt ist. Hierbei wird für jedes Feld ein Fakt benötigt. Die Reihenfolge der Parameter ist hierbei: Name, Position.
- **key/1** gibt die Position eines einzelnen Schlüsselelementes an. Hierbei wird für jedes Schlüsselfeld ein Fakt benötigt.
- **separator/1** gibt das in der CSV-Datei verwendete Feldtrennzeichen an.

Ferner muss für die `LOAD-INTO`-Variante noch folgender Parameter initialisiert werden:

- **output_csv/1** gibt den Dateipfad an, unter dem die einzufügenden Datensätze im CSV-Format gespeichert werden sollen. Diese Datei wird dann später von MySQL mittels des `LOAD-INTO`-Statements eingelesen.

Der Schritt zur Initialisierung entfällt, wenn das später in Kapitel 5 vorgestellte Skript csv2sql verwendet wird.

Zudem werden in bei der Initialisierung einige weitere Fakten berechnet, die im weiteren Programmverlauf des öfteren benötigt und daher zu Beginn einmal berechnet und dann der Wissensbasis hinzugefügt werden. Hierzu zählen:

- **positions/1** berechnet eine sortierte Liste aller Positionsnummern.

- **keys/1** berechnet eine sortierte Liste aller Schlüsselpositionen.
- **insert_string/1** berechnet einen Formatstring, der für die `INSERT`-Statements verwendet wird. Bei diesem wird der Tabellenname bereits eingetragen. Statt der konkreten Werte, die in die Felder eingefügt werden sollen, werden Platzhalter gesetzt. Für einen Block der Länge 1, dem Tabellennamen `lagerorte` und fünf Feldern hätte der Formatstring z.B. die Form

  ```
  INSERT INTO lagerorte VALUES (~w, ~w, ~w, ~w, ~w)
  ```

- **update_string/1** berechnet einen Formatstring, der für die `UPDATE`-Statements verwendet wird. Bei diesem wird der Tabellenname sowie die Spaltennamen bereits eingetragen. Statt der konkreten Werte, die in die Felder eingefügt werden sollen, werden Platzhalter gesetzt. Für einen Block der Länge 1, dem Tabellennamen `lagerorte` und fünf Feldern, bei denen die ersten drei Felder Primärschlüsselfelder sind, hätte der Formatstring z.B. die Form

  ```
  UPDATE lagerorte
  SET produkt_id ~w, menge ~w
  WHERE standort ~w, lagerhalle ~w, position ~w LIMIT 1
  ```

- **delete_string/1** berechnet einen Formatstring, der für die `DELETE`-Statements verwendet wird. Tabellenname, die Spaltennamen und die Feld-Werte werden analog zu `update_string/1` behandelt. Für einen Block der Länge 1, dem Tabellennamen `lagerorte` und fünf Feldern, bei denen die ersten drei Felder Primärschlüsselfelder sind, hätte der Formatstring z.B. die Form

  ```
  DELETE FROM lagerorte
  WHERE standort ~w, lagerhalle ~w, position ~w LIMIT 1
  ```

Ferner wird überprüft, ob die Liste der Schlüsselpositionen `keys/1` bereits aufsteigend sortiert ist. Ist dies nicht der Fall, so wird sie entfernt und sortiert neu hinzugefügt.

4.3.2 Einlesen der Patch-Datei

Das Einlesen des Patch-Files erfolgt durch einen Datenstrom, bei dem Zeile für Zeile eingelesen wird, bis das Blockende erreicht ist. Sobald ein Block vollständig eingelesen ist, erfolgt die Auswertung des jeweiligen Blocks. Abbildung 4.4 zeigt das Grundprinzip des Einleseprozesses.

Zuerst wird die erste Zeile eingelesen. Hierfür wird das PROLOG-eigene Prädikat `read_line_to_codes(+Stream, +Codes)` eingesetzt. Dieses liest den übergebenen

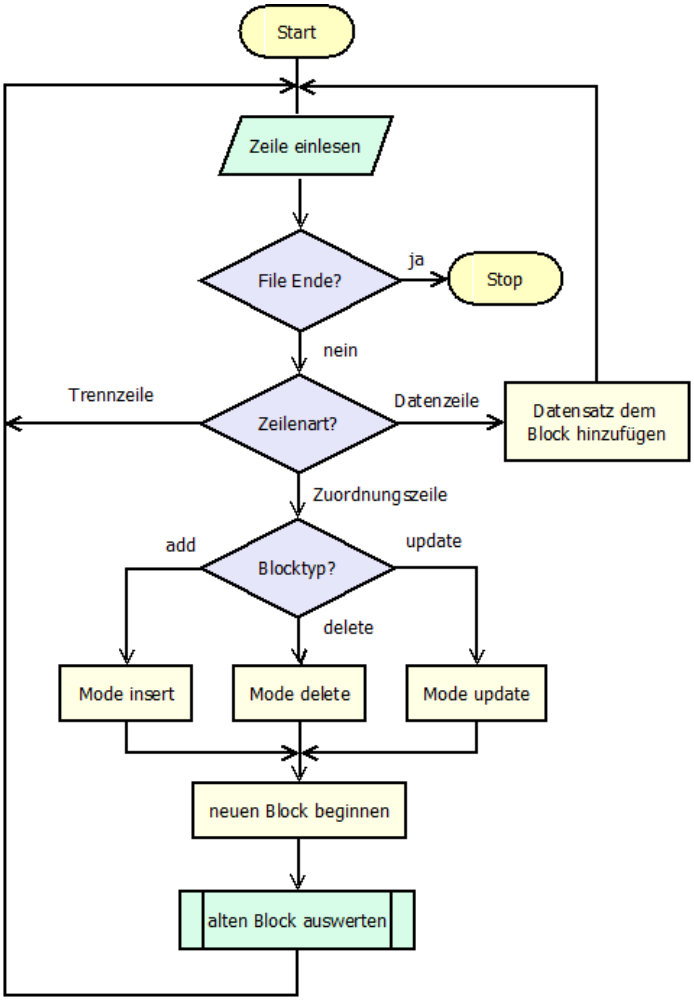

Abbildung 4.4: Programmflussdiagramm des Einleseprozesses

`Stream` zeilenweise ein. Eine Zeile wird hierbei durch einen Zeilenumbruch abgeschlossen. Zudem entfernt das Prädikat unnötige Zeilenumbrüche selbstständig.

Sobald das Dateiende erreicht ist, wird anstelle von `Codes` das Atom `end_of_file` zurückgegeben. Ist dies der Fall, so stoppt der Algorithmus an dieser Stelle. Andernfalls wird überprüft, um welche Art es sich bei der eingelesenen Zeile handelt. Hierbei kann es sich entweder um eine Trennzeile, eine Datenzeile oder eine Zuordnungszeile handeln (vgl. Abbildung 4.2).

- Liegt eine **Trennzeile** vor, so wird diese übersprungen und die nächste Zeile eingelesen.
- Bei einer **Zuordnungszeile** wird ein neuer Block begonnen. Zudem wird festgestellt, um welche Art von Block es sich handelt. Hierfür wird überprüft, ob ein `a`, `c` oder `d` Element der Zeile ist. Da diese Zeichen jedoch auch in anderen Zeilen auftreten können, wird zuvor sichergestellt, dass die Zeile nicht mit einem `>` oder `<` beginnt. Dies wird dadurch erreicht, dass Backtracking unterbunden wird und zuvor in anderen Prädikaten alle Zeilen herausgefiltert wurden, die mit `>` oder `<` beginnen. Der daraus hervorgehende Modus (`add`, `delete` oder `change`) wird daraufhin zurückgegeben. Zudem erfolgt die Auswertung des vorherigen Blocks.
- Handelt es sich hingegen um eine **Datenzeile**, so wird diese dem aktuellen Block hinzugefügt. Ein Block wird hierbei in Form einer Liste abgebildet. Bei einem change-Block muss hierbei beachtet werden, ob es sich bei der Zeile um eine eingehende oder eine ausgehende Datenzeile handelt. Daher werden hier zwei Listen benötigt.

4.3.3 Auswerten der change-Blöcke

Nach dem Einlesen eines Blockes erfolgt das Auswerten. Dies besteht für einen add- bzw. delete-Block in der Generierung der entsprechenden Statements. Hierbei kann entweder für jede Zeile ein Statement erstellt werden oder für jeden Block ein kombiniertes Statement.

Für das Auswerten der change-Abschnitte existieren ebenfalls verschiedene Algorithmen. Zwei davon wurden im Rahmen dieser Thesis implementiert. Diese sind:

- **Einfache Auswertung:** Zum einen besteht die Möglichkeit eines zeilenweisen Vergleichs der Primärschlüssel.

- **Auswertung mittels Merge:** Eine etwas ausgeklügeltere Variante besteht darin, nicht jede Zeile gegen jede zu testen, sondern die Sortierung des Patch-Files zu nutzen. Diese ist in `merge.pl` zu finden.

Einfache Auswertung

Der wohl simpelste Algorithmus besteht darin, Zeile für Zeile zu überprüfen, ob die Primärschlüssel übereinstimmen. Ist dies der Fall, so wird ein `UPDATE` Statement daraus generiert und beide Zeilen aus den Listen gelöscht. Wurde dies für jede Zeile überprüft, bilden die noch verbleibenden Elemente in der alten Liste dann die zu löschenden Datensätze, die der neuen Liste die einzufügenden.

Diese Variante ist jedoch sehr zeitaufwendig. Daher bietet es sich an, die Sortierung des Patch-Files nach den Primärschlüsseln auszunutzen.

Auswertung mittels Merge

Eine Variante die dies ausnutzt, ist der Merge-Algorithmus. Dieser ist in Abbildung 4.5 abgebildet. Hierbei wird untersucht, ob die aktuelle Zeile der alten Tabelle bzgl. der Primärschlüssel mit der aktuellen Zeile der neuen Tabelle übereinstimmt. Ist dies der Fall, so muss ein `UPDATE`-Statement für die beiden Datensätze generiert werden. Trifft dies jedoch nicht zu, so wird für jedes Feld des Schlüssels überprüft, ob der Wert des alten kleiner ist, als der des neuen. Sobald ein solches Feld gefunden wurde, wird ein `DELETE`-Statement generiert. Ist der Wert hingegen größer, so wird ein `INSERT`-Statement generiert.

Dabei wird im Detail wie folgt vorgegangen:

Analog zum einfachen Algorithmus erfolgt das Auswerten der einzelnen Blöcke. Für delete- und add-Blöcken müssen die entsprechenden SQL-Statements erstellt werden. Die Auswertung der change-Blöcke erfordert einigen Mehraufwand.

Hierbei wird die Hilfsliste `position_help/1` verwendet. Diese hat die gleiche Anzahl an Elementen wie es Felder gibt. Handelt es sich bei dem Feld um ein Schlüsselfeld, so wird die aktuelle Position des Feldes in die Liste eingetragen, andernfalls hat es den Wert 0. Für das Beispiel aus Tabelle 2.1 ergibt sich somit folgende Liste: `[0,2,3,4,0]`. Diese erlaubt eine schnelle Unterscheidung zwischen Schlüsselfeldern und Nichtschlüsselfelder, ohne dabei jedes Mal mittels `member/2` überprüfen zu müssen, ob es sich bei einer Position um ein Schlüsselfeld handelt.

Es wird zuerst von der Liste mit den alten Datensätzen sowie der Liste mit den neuen Datensätzen das erste Element betrachtet. Mittels der zuvor angelegten Hilfsliste

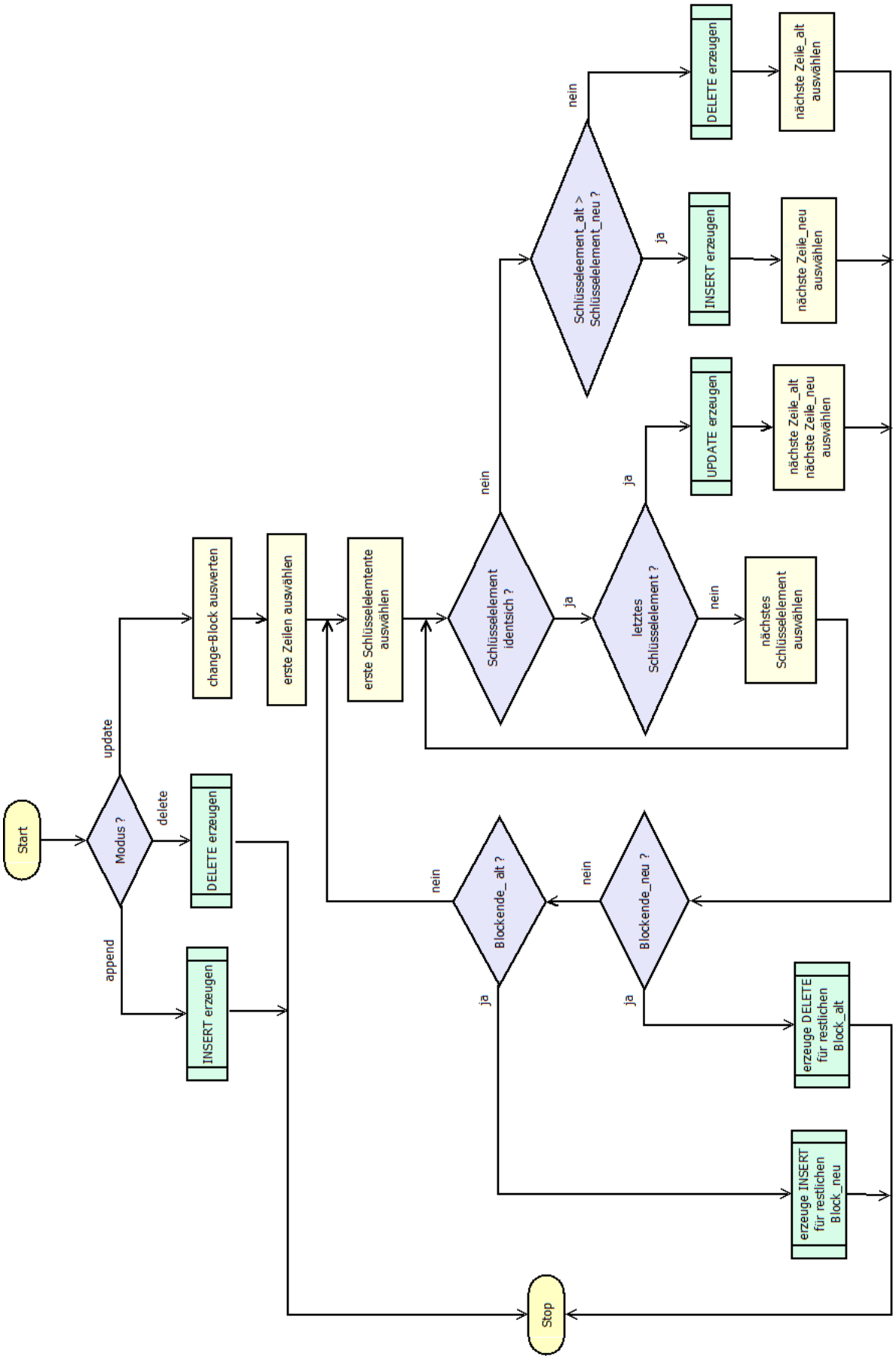

Abbildung 4.5: Programmflussdiagramm des Merge-Algorithmus

`position_help/1` wird nun für jedes Element überprüft, ob es ungleich 0 und somit ein Schlüssel ist. Hat das Element den Wert 0, so geschieht nichts und das nächste Element wird rekursiv untersucht. Ist der Wert hingegen ungleich 0, so werden beide Schlüsselfelder mittels `string_codes/2` jeweils in eine Liste von Codes umgewandelt.

Die beiden dadurch entstehenden Listen von Codes werden daraufhin wieder Element für Element gegeneinander überprüft. Sobald ein Element gefunden ist, das ungleich ist, wird entschieden ob dieses größer oder kleiner ist. Dementsprechend wird als Zustand 'greater' oder 'smaller' gesetzt. Sind alle Elemente gleich, so wird der Zustand 'equals' zurückgegeben.

Anhand dieses Zustandes wird dann bestimmt, wie weiter mit der Zeile verfahren wird. Ist der Zustand

- `equals`, so wird der aktuelle Datensatz aus beiden Listen entfernt und ein `UPDATE`-Statement muss erzeugt werden
- `greater`, so wird der Datensatz aus der Liste der der geänderten gelöscht und ein `INSERT`-Statement generiert werden
- `smaller`, so wird der Datensatz aus der Ausgangsliste gelöscht und ein `DELETE`-Statement generiert werden

4.3.4 Generierung der SQL-Statements

Sobald die zu ändernden Datensätze ermittelt sind, erfolgt das Generieren der SQL-Statements. Hierfür existieren mehrere Möglichkeiten:

1. Zum einen kann für jede Zeile ein eigenes Statement generiert werden.
2. Zum anderen können Zeilen gebündelt werden, für die die selbe Art von Statement auszuführen ist. Sind z.B. fünf Zeilen einzufügen, so wird nicht pro Zeile ein `INSERT`-Statement generiert, sondern diese in einem Statement zusammengefasst (vgl. Listing 2.2).

Hierbei haben beide Varianten ihre Vor- und Nachteile. Bei Variante 1 wird eine erneute Iteration über die Block-Liste vermieden und ein Ausführen der SQL-Statements in mehreren Threads ermöglicht. Bei Variante 2 hingegen muss beispielsweise die Verbindung zum Server nur einmal hergestellt werden. Eine genauere Betrachtung der verschiedenen Vor- und Nachteile ist in Kapitel 3 zu finden, konkrete Messungen hierzu in Kapitel 7.

Bei beiden Varianten wird bereits bei der Initialisierung einmalig pro Statement ein Format-String angelegt und als Fakt hinzugefügt. Dies dient der Performanzsteigerung, da so viele Ersetzungen, wie beispielsweise das Einfügen der Feld- und Tabellennamen, nur einmalig durchgeführt werden müssen.

Nach der Auswertung können dann die entsprechenden Fakten angefragt und mittels `format/3` die konkreten Werte eingesetzt werden.

Auch beim Generieren der `UPDATE`-Statements gibt es zwei Möglichkeiten. Zum einen kann ein Statement generiert werden, das alle Werte neu setzt, unabhängig davon, ob sich der Wert geändert hat oder nicht. Bei einer etwas eleganteren Lösung werden nur die Werte neu gesetzt, die sich auch geändert haben. Letztere wurde in dieser Arbeit realisiert.

Auch hier haben beide Varianten ihre Vor- und Nachteile. Werden nur die geänderten Werte neu gesetzt, führt dies zu einem Mehraufwand in Prolog, da für jedes Feld überprüft werden muss, ob eine Änderung vorliegt. Werden hingegen standardmäßig alle Werte neu gesetzt, erspart dies die Überprüfung in Prolog, führt jedoch zu einem Mehraufwand in SQL.

Abschließend werden alle generierten Statements in eine Textdatei geschrieben.

4.3.5 Variationen des Algorithmus

Das Einfügen neuer Datensätze in eine existierende Tabelle erfolgt in SQL für gewöhnlich mittels `INSERT`. Jedoch besteht auch die Möglichkeit neue Datensätze mittel eines `LOAD-INTO`-Statement einzufügen. Listing 2.4 zeigt die Syntax eines solchen Statements. Eine Gegenüberstellung der beiden Varianten ist in Kapitel 7 zu finden.

Um diese Variante zu realisieren, müssen jedoch einige Änderungen am zuvor vorgestellten Algorithmus vorgenommen werden. Statt die Daten komplett in Prolog-Listen zu verwalten, müssen die einzufügenden Zeilen in ein CSV-File gespeichert werden. Hierzu wird die Zeile normal eingelesen. Statt ein `INSERT`-Statement zu generieren, wird die Zeile jedoch mittels `write_csv/2` in die Textdatei geschrieben. Die Syntax hierfür ist in Listing 2.15 zu finden.

Zudem wird in die selbe Datei, in die auch die übrigen SQL-Statements geschrieben werden, ein einzelner `LOAD-INTO`-Befehl geschrieben. Dieser verweist auf die CSV-Datei, in der die Datensätze gespeichert sind. Die Syntax dieses `LOAD-INTO`-Befehls ist in Listing 2.5 zu finden.

4.4 Die Module von diff2sql

Der PROLOG-Code von diff2sql besteht aus vier verschiedenen Modulen: `diff2sql`, `read_in`, `merge` und `create_statements`. `read_in` übernimmt das zeilenweise Einlesen des mittels tablediff erstellten Patch-Files. Mittels `merge` werden die change-Abschnitte noch einmal genauer untersucht. Die fertigen SQL-Statements werden mittels `create_statements` erzeugt.

Im Folgenden werden die wesentlichen Prädikate der einzelnen Module vorgestellt. Hierbei wird die Stelligkeit auf Grundlage des in den Abschnitten 4.3.2, 4.3.3 und 4.3.4 vorgestellten Algorithmus aufgeführt. Diese kann sich in einzelnen Varianten unterscheiden. Die Module heißen jedoch stets gleich und haben die gleiche Funktion.

Modul: diff2sql

- **main**
  Startet das Programm und nimmt die Initialisierung vor.

Modul: read_in

Das Modul `read_in` übernimmt das Einlesen des mittels tabeldiff erstellten Patch-Files.

- **read_until_stop(+File, +Mode, +List_In, +List_Out)**
  Liest das mittels tablediff generierte Patch-File zeilenweise ein, bis das Ende des Files erreicht ist.
- **read_line(+Line_Char_List, +Mode, -Mode_New, +List_In, +List_Out, -List_In_New, -List_Out_New)**
  Überprüft, um welche Art von Zeile es sich handelt und setzt den Mode auf `add`, `delete` oder `change`.
- **check_status_changed(+Mode, +List_In, +List_Out)**
  Wertet einen Block aus, sobald dieser vollständig eingelesen ist.

Modul: merge

Das Modul `merge` untersucht anhand des Merge-Algorithmus (vgl. Abschnitt 4.3.3) die change-Abschnitte des mittels tablediff generierten Patch-Files genauer, d.h. es nimmt eine erneute Unterteilung in neu hinzugefügte, aktualisierte und entfernte Datensätze vor.

- **evaluate_change(+List_In, +List_Out)**
  Wertet den change-Block aus und entscheidet darüber, welche Zeilen gelöscht, geändert oder hinzugefügt werden müssen.
- **compare(+Positions, +Key_In, +Key_Out, -Mode)**
  Vergleicht den Schlüssel `Key_In` mit dem Schlüssel `Key_Out`.
- **compare_codes(+Xs, +Ys, -Mode)**
  Vergleicht alphabetisch die Codelisten `Xs` und `Ys`. Sind beide Codes identisch, so wird als Mode `equals` zurückgegeben, ist `Xs` kleiner `smaller`, andernfalls `greater`
- **check_lines(+List_In, +List_Out)**
  Überprüft, ob eine Zeile eingefügt, gelöscht oder geändert werden muss.

Modul: create_statements

Das Modul `create_statements` erzeugt die finalen SQL-Statements.

- **create_format_string(+Mode)**
  Erzeugt einen String mit Platzhalter an Stelle der zu setzenden Werte. In die Platzhalter werden dann mit dem Prädikat `create_statement/4` die spezifischen Werte eingesetzt. Für `Mode` existieren hierbei die Belegungen `insert`, `update` und `delete`, die die entsprechenden `INSERT`-, `DELETE`- bzw. `UPDATE`-Statements generieren.

  Beispiel:
  Für einen Datensatz aus Beispieltabelle 2.1 mit fünf Feldern, bei denen die ersten drei Schlüsselfelder sind, lautet der generierte `INSERT`-String

  ```
  INSERT INTO lagerorte VALUES (~w, ~w, ~w, ~w, ~w)
  ```

  der `DELETE`-String

  ```
  DELETE FROM lagerorte
  WHERE standort ~w, lagerhalle ~w, position ~w LIMIT 1
  ```

und der UPDATE-String

```
UPDATE lagerorte SET produkt_id ~w, menge ~w,
WHERE standort ~w, lagerhalle ~w, position ~w LIMIT 1
```

- **create_statement(+Mode, +Line_List_In, +Line_List_Out)**
  Erzeugt das finale SQL-Statement durch Einsetzen der Werte aus den Listen `Line_List_In` und `Line_List_Out` und des Verbindungszeichen zwischen Feldname und Wert in den mittels `create_format_string/1` generierten String. Hierbei ist der Modus `Mode` analog zu `create_format_string/1`. Bei `INSERT` kann hierbei auf das Verbindungszeichen verzichtet werden. Bei `DELETE` und `UPDATE` wird für den Wert Null ein `is` eingefügt, für jeden anderen Wert ein `=`.

Zusätzlich zu den oben aufgeführten Prädikaten existiert für die `LOAD-INTO`-Variante noch folgendes Prädikat:

- **write_lines_to_file(+Line_List)**
  Schreibt die Liste der einzelnen einzufügenden Zeilen `Line_List` in eine CSV-Datei.

Zudem existiert bei `create_statement` anstelle des Modus `insert` der Modus `load`. Bei diesem wird ein `LOAD-INTO`-Statement erzeugt.

4.5 Schwierigkeiten und deren Lösung

Im Folgenden sollen einige Schwierigkeiten, die bei der Implementierung von diff2sql aufgetreten sind, erläutert und deren Lösung aufgezeigt werden.

NULL

Eine Schwierigkeit liegt in der Sonderbehandlung von `NULL`. In MySQL werden Werte standardmäßig mittels `feldname = Wert` abgefragt. Bei Null liegt jedoch ein Sonderfall vor. Hier findet der Vergleich nicht mittels `=` statt, sondern mit `is`. Die Abfrage lautet somit `feldname IS NULL`. Dies muss bei der Generierung von `DELETE`- und `UPDATE`-Statements beachtet werden. Hierdurch ist es nicht möglich, einen String mit Platzhalter der Form `feldname = ~w` zu generieren und in diesen den konkreten Wert einzusetzen. Stattdessen wird der Platzhalter später durch einen String ersetzt, der neben dem Wert auch das „Vergleichszeichen“ enthält. Somit hat der String die Form: `feldname ~w`. Erst beim Einsetzen erfolgt dann die Überprüfung, ob es sich bei dem Wert um ein `NULL` handelt.

Escapen von Sonderzeichen

Eine weitere Schwierigkeit liegt in der Verarbeitung der in Abschnitt 2.1.5 aufgelisteten Sonderzeichen. Diese können nicht ohne Weiteres in SQL eingelesen werden, sondern müssen zuvor maskiert werden.

Hierfür existieren drei Möglichkeiten:

- **Ersetzung mittels Prolog**
  Zum einen ist eine Ersetzung mittels PROLOG möglich. Bei dieser Variante findet die Maskierung beim Generieren der Statements statt, mittels des Prädikats `re_replace/3`. Eine Maskierung während des Einlesens wäre zwar auch möglich, hätte aber den Nachteil, dass evtl. Felder maskiert würden, die in den finalen SQL-Statements nicht vorkommen (z.B. ein Nichtschlüsselfeld bei einem `DELETE`). Dies würde zu einem unnötigen zeitlichen Mehraufwand führen. Zudem würde es die Anzahl der zu überprüfenden Zeichen im Merge-Teil erhöhen, was ebenfalls einen erhöhten Zeitbedarf zur Folge hätte.

- **Ersetzung mittels Shell-Skript**
  Eine Alternative hierzu stellt die Ersetzung mittels Shell-Skript dar. Bei dieser Variante wird der sed-Operator benutzt, um die Sonderzeichen zu finden und zu ersetzen. Dieser kann entweder auf das von tablediff generierte Patchfile oder nach der Generierung der SQL-Statements auf diese angewendet werden.

  Ersteres hat den Vorteil, dass hierbei weniger Zeichen durchsucht werden müssen, da das Textfile mit den SQL-Statements zusätzlich zu den Informationen über die Änderungen noch die SQL-Befehle beinhaltet. Daher wurde diese Variante in der dieser Masterarbeit zu Grunde liegenden Implementierung gewählt.

- **Ersetzung mittels SQL**
  Eine weitere Alternative wäre es, bereits vor dem Export der SQL-Tabelle in CSV eine Maskierung der Sonderzeichen vorzunehmen. Diese bietet jedoch die meisten Nachteile und ist daher nur der Vollständigkeit halber erwähnt. Der massivste Nachteil bestünde darin, das die Daten im SQL-Format vorliegen müssen. Liegen sie hingegen im CSV-Format vor, müssten sie zunächst in SQL eingespielt, die Anfrage durchgeführt und dann wieder als CSV-Format extrahiert werden. Zudem müssten alle Zeichen untersucht werden, d.h. auch die, die in beiden Tabellen identisch sind.

Zudem muss bekannt sein, ob die an tablediff übergebenen CSV-Dateien nicht bereits maskiert sind. Dies wäre z.B. der Fall, wenn die Dateien mittels XAMPP aus SQL exportiert worden wären. Dies wird mittels des Faktes `escaped/1` realisiert.

Hat dieses Fakt den Wert `true`, so ist keine Ersetzung der Sonderzeichen notwendig, bei `false` hingegen schon.

Überprüfung, ob es sich bei einem Feld um einen Schlüssel handelt

Eine weitere Schwierigkeit lag in der Überprüfung, ob es sich bei einem Feld um ein Schlüsselfeld handelt. Im Grunde stellt dies in PROLOG kein Problem dar, da die Positionen der Schlüsselfelder in einer Liste gespeichert sind und somit mittels des Prädikats `member/3` abgefragt werden kann, ob ein Feld ein Schlüsselfeld ist. Da hierbei jedoch stets die Liste bis zu dem gesuchten Element oder im schlimmsten Fall sogar die ganze Liste durchlaufen werden muss, gestaltet sich ein Lösung mittels des `member`-Prädikats als sehr zeitaufwendig.

PROLOG ist jedoch sehr schnell bei der Unifizierung von Fakten. Daher ist ein zeiteffizienterer Ansatz, die Schlüsselelemente in Fakten `keys/1` abzuspeichern, die bereits in der Initialisierungsdatei init.pl festgelegt werden.

Variable Feldanzahl

Der Vorteil der schnellen Unifizierung durch Fakten ist jedoch in diesem Fall stark beschränkt. Dies liegt daran, dass durch die variablen Feldanzahlen und Schlüsselpositionen keine Struktur hierfür festgelegt werden kann. Daher ist eine Bearbeitung mit Listen notwendig.

4.6 Zusammenfassung

In diesem Kapitel wurde das PROLOG Programm diff2sql vorgestellt, das aufbauend auf tablediff die zur Synchronisation zweier CSV-Dateien nötigen SQL-Statements ermittelt und generiert. Hierzu wurde neben dem auf Sortierung beruhenden Algorithmus von tablediff auch dessen Funktionsweise beschrieben. Das Hauptaugenmerk lag jedoch auf dem Algorithmus von diff2sql und dessen Modulen. Zudem wurden die Schwierigkeiten bei der Realisierung des Algorithmus erläutert und Lösungen skizziert.

5 Das Shell-Skript csv2sql

Im vorherigen Kapitel wurden das Prolog-Programm diff2sql sowie das Shell-Skript tablediff, das zur Vorverarbeitung der Daten dient, vorgestellt. Da für eine reibungslose Zusammenarbeit der beiden Tools einige Zwischenschritte nötig sind (z.B. Entfernung der Textbegrenzungszeichen, Maskieren von Sonderzeichen, etc.), wurden die Tools in ein Shell-Skript eingebunden, das in diesem Kapitel vorgestellt werden soll. Zudem wendet das Skript die generierten Statements auf die SQL-Tabelle an und misst den Zeitbedarf der einzelnen Schritte. Um die Bedienung für den Endnutzer noch weiter zu vereinfachen, wurde für die Verwaltung der Vielzahl an benötigten Parametern die komfortable Lösung einer Konfigurationsdatei gewählt, die in Abschnitt 5.2 vorgestellt wird.

5.1 Funktionsweise

Das Shell-Skript csv2sql.sh verfolgt das Ziel, zwei CSV-Dateien zu synchronisieren. Dazu werden zuerst die zwei CSV-Dateien mittels tablediff verglichen und die Unterschiede zwischen diesen ermittelt. Anschließend werden die benötigten SQL-Statements mit Prolog generiert, die zur Angleichung der beiden CSV-Dateien benötigt werden. Abschließend werden diese SQL-Statements dann auf die bereits existierende MySQL-Tabelle angewendet (vgl. Abbildung 4.1).

Hierbei erfüllt csv2sql im Wesentlichen folgende Aufgaben:

- Entfernen der Textbegrenzungszeichen
- Erstellen der Patch-Datei mittels tablediff
- Bereinigen der Patch-Datei
- Maskieren der Sonderzeichen
- Erstellen der Prolog-Konfigurationsdatei init.pl auf Basis der Konfigurationsdatei config.cfg
- Generierung der SQL-Statements mittels des Prolog-Programms diff2sql

- Anwenden der generierten Statements auf die bereits existierende MySQL-Tabelle
- Aufräumen der nicht mehr benötigten Dateien
- Zeitmessung von tablediff, diff2sql und Ausführen der Statements in MySQL

Im ersten Schritt müssen – falls vorhanden – die Textbegrenzungszeichen der CSV-Dateien mittels des sed-Kommandos entfernt werden, da diese sowohl in tablediff als auch bei der Generierung der SQL-Statements zu Fehlern führen würden. Bei dem *Stream Editor*, kurz *sed*, handelt es sich um einen Linux-Editor, der sich zur skriptgesteuerten Manipulation eignet. Dazu zählen u.a. das Suchen und Ersetzen von Zeichen sowie das Einfügen von Zeilen. Im Gegensatz zu anderen Linux-Editoren wie vim oder Emacs wird der Datenstrom nur aus der Datei gelesen und die Ursprungsdatei bleibt unverändert. Daher ist es notwendig, den manipulierten Datenstrom wieder in eine neue Datei zu schreiben. [Wol10]

Das in der CSV-Datei verwendete Textbegrenzungszeichen sowie die anderen benötigten Parameter werden in der Konfigurationsdatei config.cfg definiert. Die Bedeutung der einzelnen Parameter wird in Abschnitt 5.2 vorgestellt.

Da innerhalb der Konfigurationsdatei als Trennzeichen zwischen Aufzählungen der Strichpunkt verwendet wird (vgl. Abschnitt 5.2), tablediff jedoch bei der Position der Schlüsselfelder ein einfaches Komma benötigt, wird eine Ersetzung mittels des tr-Operators vorgenommen. Der tr-Operator ersetzt alle Vorkommen eines Zeichens durch ein festgelegtes anderes Zeichen. Im Vergleich zu sed kann er jedoch nur auf einzelnen Zeichen operieren und nicht auf Zeichenfolgen oder regulären Ausdrücken. [SS04]

Anschließend kann tablediff mit den entsprechenden Variablen aus der Konfigurationsdatei parametriert und ausgeführt werden.

Da hierbei ein Echo des diff-Befehls vor einigen Blöcken ausgegeben wird, müssen diese Zeilen mittels grep herausgefiltert und gelöscht werden. Das grep-Kommando sucht nach Zeilen, auf die ein bestimmtes Muster zutrifft und gibt diese aus. Zudem erlaubt grep die Suche zu negieren, d.h. anstatt der Zeilen, die die Bedingung erfüllen, werden alle Zeilen ausgegeben, die das Muster nicht erfüllen. [Wol10]

Da für eine spätere zeitliche Evaluation der einzelnen Prozesse des Skriptes eine Zeitmessung notwendig ist, wird der Aufruf von tablediff um diesen ergänzt. Die Zeitmessung wird hierbei mittels `command time -f "%e"` vorgenommen und in eine Zwischendatei abgespeichert. Im Gegensatz zu dem herkömmlichen time-Kommando

wird bei diesem nur der Wert der `real-time` ausgegeben sowie eine Command-Zeile. Da letztere jedoch unerwünscht ist, muss sie bei dieser sowie der folgenden Messungen mittels grep herausgefiltert werden.

Die zuvor mittels tablediff generierte und bereinigte Patch-Datei kann nun mittels des PROLOG-Programms diff2sql (vgl. Kapitel 4) weiterverarbeitet werden. Hierzu muss zunächst die PROLOG-Initialisierungsdatei init.pl (vgl. Abschnitt 4.3.1) auf Basis der Konfigurationsdatei config.cfg erstellt werden. Zudem muss das PROLOG-Programm, bevor es aus einem Skript heraus aufgerufen werden kann, in ein PrologScript umgewandelt werden. Die hierfür nötigen Schritte sind in Abschnitt 5.3 zu finden.

Abschließend müssen die mittels PROLOG generierten Statements auf die bereits existierende SQL-Tabelle angewendet werden. Hierfür muss die Datei mit den generierten Statements noch mittels sed um den Befehl `USE database_name` ergänzt werden, um MYSQL mitzuteilen, auf welche Datenbank es die Befehle anwenden soll. Die Information über die zu verwendende Tabelle ist bereits in den einzelnen Statements enthalten. Anschließend muss MYSQL die Datei mit den Statements übergeben werden. Hierbei, sowie bei diff2sql, findet eine Zeitmessung analog zu tablediff statt.

5.2 Die Konfigurationsdatei config.cfg

Da csv2sql einige Parameter benötigt, wurde auf eine Parameterübergabe über die Konsole verzichtet. Stattdessen wurde eine für den Endnutzer komfortablere Lösung mittels einer Konfigurationsdatei gewählt.

In dieser Konfigurationsdatei müssen zum einen Informationen über die Struktur der Datenbank angegeben werden. Dies geschieht mittels der folgenden Parameter:

- **database_name** gibt den Namen der MYSQL-Datenbank an, auf die die generierten SQL-Statements angewendet werden sollen.
- **table_name** gibt den Namen der Tabelle an, auf die die generierten SQL-Statements angewendet werden sollen
- **positions_keys** gibt die Position der Schlüsselfelder an. Diese werden untereinander durch Strichpunkte getrennt.
- **fields** gibt den Name der einzelnen Felder der Tabelle an. Hierbei werden alle Felder durch Strichpunkt getrennt in einer Variablen aufgezählt.

Neben den Informationen über die Tabellen bzw. Datenbank werden auch einige Information bezüglich der CSV-Dateien benötigt, die ausgewertet werden sollen. Dazu zählen:

- **input_csv_in** gibt den Pfad der eingehenden CSV-Datei an.
- **input_csv_out** gibt den Pfad der ausgehenden CSV-Datei an.
- **field_separator** gibt das in `input_csv_in` sowie `input_csv_out` verwendete Feldbegrenzungszeichen an.
- **text_separator** gibt das in `input_csv_in` sowie `input_csv_out` verwendete Textbegrenzungszeichen an. Wurde in den beiden CSV-Dateien kein Textbegrenzungszeichen verwendet, ist an dieser Stelle ein leerer String einzutragen.

Ferner werden der Ausgabepfad der generierten SQL-Statements sowie die Information benötigt, ob die beiden CSV-Dateien bereits maskiert sind. Dies geschieht mittels der folgenden Parameter:

- **output_statements** gibt den Pfad der Datei an, in die die mittels Prolog generierten SQL-Statements geschrieben werden sollen.
- **already_escaped** gibt an, ob die Sonderzeichen in den vorliegenden CSV-Dateien bereits maskiert sind. Ist dies der Fall, so wird der Wert der Variablen auf `true` gesetzt, andernfalls auf `false`.

5.3 Umwandlung eines Prolog-Programms in ein PrologScript

Das Shell-Skripte csv2sql.sh erfordert den Aufruf des Prolog-Programms diff2sql.pl. Um dieses aus einem Skript heraus aufzurufen, muss es zuvor erst in ein Skript umgewandelt werden. Hierfür gibt es zwei Möglichkeiten. Zum einen kann das Prolog-Programm in ein Skript umgewandelt werden. Die andere Möglichkeit besteht darin, aus dem Prolog-Programm ein PrologScript zu erzeugen. Letzteres wurde für die dieser Masterarbeit zu Grunde liegenden Implementierung gewählt und soll im Folgenden kurz vorgestellt werden. [Pro18]

Für die Umwandlung eines Prolog-Programms in ein PrologScript sind im Wesentlichen folgende Änderungen nötig:

- Automatischer Start bei Konsultierung der Datei
- Sicherstellung der Terminierung

- Umwandlung in ein Linux-Programm

Ein automatischer Start des Programms bei Aufruf wird mittels des Prädikats `initialization/1` ermöglicht. Dieses Prädikat ruft das ihm übergebene Prädikat automatisch beim Konsultieren der Datei auf. Dadurch wird das Start-Prädikat automatisch beim Konsultieren der Datei aufgerufen. [ini18]

Für die Sicherstellung der Terminierung müssen zwei Schritte erfolgen. Zum einen muss das aktuelle Prädikat mittels `halt/0` angehalten werden, sobald es am Ende ist, und zum anderen muss eine zweite Regel das Prädikat `halt/1` aufrufen. Dieses schließt PROLOG. [hal18]

Mittels der HashBang-Zeile `#!/usr/bin/env swipl` am Anfang der Datei kann das Programm dann in ein Linux-Programm umwandelt werden. Dies ist möglich, da PROLOG in einem File, dessen erstes Zeichen `#` ist, die erste Zeile als Kommentar interpretiert. [Pro18]

Ein PrologScript-Datei (ohne Parameter) hat somit folgende Form:

Listing 5.1: Aufbau einer PrologScript-Datei

```
1  \#!/usr/bin/env swipl
2
3  :- initialization main.
4
5  main:-
6      Befehle,
7      halt.
8
9  main:-
10     halt(1).
```

Ist ein Aufruf mit Parameter notwendig, müssen noch einige weitere Schritte vorgenommen werden. Diese sind unter [Pro18] beschrieben.

5.4 Zusammenfassung

In diesem Kapitel wurde das Shell-Script csv2sql vorgestellt. Dieses verbindet das Shell-Skript tablediff und das PROLOG-Programm diff2sql.pl. Somit ermittelt es den Unterschied zwischen zwei CSV-Dateien und generiert basierend auf diesen die SQL-Statements, die für die Überführung notwendig sind. Zudem wendet es die generierten Statements auf die MYSQL-Tabelle an. Ferner wurden die Parameter der Kon-

figurationsdatei config.cfg sowie die zur Umwandlung eines PROLOG-Programms in ein PrologScript erläutert.

6 Testdatengenerierung

In diesem Kapitel erfolgt eine Einführung in das Tool CSV-Creator zur Testdatengenerierung. Dieses Tool wurde im Rahmen der vorliegenden Masterarbeit realisiert, um Testdatenbanken zu entwickelt, die für die Evaluation in Kapitel 7 benötigt werden. Nachfolgend erfolgt eine kurze Vorstellung der Zielsetzung sowie der Benutzeroberfläche des Tools. Zudem wird der dem Tool zu Grunde liegende Algorithmus näher erläutert. Ferner wird anhand eines Klassendiagramms und eines Sequenzdiagramms die Struktur und der Ablauf des Programms aufzeigt.

6.1 Zielsetzung

Ziel des Programms ist eine möglichst zeiteffiziente Erstellung von CSV-Files zum Testen und Evaluieren des Zeitbedarfs des in Kapitel 4 vorgestellten Algorithmus. Dies geschieht unter Zuhilfenahme der objektorientierten Programmiersprache Java. Zur Vereinfachung des Programms wird für die generierten CSV's nur das Datenformat String verwendet. Die Strings wiederum bestehen aus Zahlen und für SQL relevanten Sonderzeichen. Diese Sonderzeichen sind in Abschnitt 2.1.5 zu finden. Auf die Berücksichtigung des Sonderfalls von `NULL` (einem leeren Feld) wurde hierbei verzichtet, da dies zeitlich bezüglich des zu testenden Algorithmus keine Relevanz hat.

Anfangs erstellt das Programm ein zufälliges Ausgangs-CSV. Dabei kann der Nutzer die Anzahl der zu generierenden Datensätze, die Anzahl der Felder sowie die Positionen der Schlüsselfelder festlegen. Ferner können die zu verwendenden Feld-, Datensatz- und Textbegrenzungszeichen angegeben werden.

Anschließend wird das generierte CSV manipuliert. Hierfür können die Anzahl der zu löschenden, zu ändernden und neu einzufügenden Daten anhand der Prozentzahl festgelegt werden.

Abschließend kann das manipulierte CSV noch auf verschiedene Weisen vermischt werden. Hierzu zählt zum einen das Invertieren. Hierbei wird die Reihenfolge der Datensätze umgedreht, d.h. das erste Element wird das letzte, das zweite das vorletzte, usw. Ferner können die Datensätze zufällig vermischt werden. Eine Variation

hiervon ist die blockweise Vermischung. Bei dieser wird das manipulierte CSV in Blöcke einer gewünschten Länge zerlegt. Anschließend werden diese zufällig neu angeordnet.

6.2 Algorithmus des CSV-Generators

Der Algorithmus zur Erstellung eines Zufalls-CSV's sowie den dazugehörigen Manipulationen besteht maßgeblich aus drei Teilen. Hierbei werden jedoch nicht direkt CSV-Dateien erzeugt, sondern Listen von Listen. Die Umwandlung in das CSV-Format erfolgt erst am Ende des Programms und ist nicht Teil des vorgestellten Algorithmus. Die äußerste Liste stellt das CSV dar. Die inneren Listen bilden die Datensätze, die einzelnen Listenelemente die Felder.

- **Generierung:** Im ersten Teil des Algorithmus muss zunächst eine Ausgangsdatenbank generiert werden. Hierzu werden als Input-Parameter die Anzahl der zu generierenden Datensätze (Zeilenanzahl), die Anzahl der benötigten Felder, sowie die Position der Primärschlüssel benötigt.
- **Manipulation:** Diese Datenbank wird im nächsten Schritt manipuliert. Hierzu werden neue Datensätze eingefügt, alte gelöscht und existierende geändert. Um dies zu berechnen, werden die entsprechenden Prozentsätze der einzelnen Manipulationen benötigt.
- **Vermischung:** Im letzten Teil werden dann die Zeilen des manipulierten CSV's vertauscht. Hierbei wird als Input-Parameter lediglich bei der blockweise zufälligen Vermischung die gewünschte Blocklänge benötigt.

6.2.1 Generierung der Ausgangsdatenbank

Bei der Generierung einer Datenbank mit zufälligen Einträgen liegt das Hauptproblem darin, eindeutige Schlüssel zu generieren - und dies auf effiziente Art und Weise. Die Effizienz ist hierbei wichtig, da der Test des Algorithmus aus Kapitel 4 mit großen Datenmengen (überwiegend 10.000 Datensätzen) stattfinden soll. Besteht der Primärschlüssel nur aus einem Feld, stellt dies kein großes Problem dar. Bei mehreren Feldern muss jedoch sichergestellt werden, dass jede Schlüsselkombination eindeutig ist. Die einfachste Idee dies zu realisieren, wäre so viele Sets zu erzeugen, wie Schlüsselfelder benötigt werden. Da das Einfügen einer neuen Zahl in ein Set jedoch sehr zeitaufwendig ist, weil für jeden neuen Wert überprüft werden muss, ob dieser bereits im Set vorhanden ist, wäre dies zwar eine einfache, jedoch nicht sonderlich effiziente Lösung. Zudem erhöht sich der Zeitbedarf mit steigender Größe

des Sets stark überproportional (mit Fakultät n). Um kombinierte Primärschlüssel dennoch möglichst zeiteffizient zu realisieren wird folgende Idee anwendet: Statt mehrere Schlüssel pro Datensatz zu generieren, wird lediglich ein Wert generiert, der anschließend in mehrere Werte zerlegt und auf die Primärschlüsselfelder verteilt wird.

Hierbei wird wie folgt vorgegangen:

1. Im ersten Schritt wird eine Zufallszahl zwischen *0* und *Anzahl zu generierender Datensätze multipliziert mit einem gewählten Faktor* erzeugt. Diese werden einer `Collection`-Klasse *Set* hinzugefügt, da diese sicherstellt, das keine Duplikate innerhalb des Sets vorkommen. Dies geschieht solange, bis das Set die Größe *Anzahl Datensätze* besitzt. Die Multiplikation mit dem Faktor trägt zu einer verbesserten Zeitperformanz bei. Diese ist gegeben, da so Zufallszahlen mit einem größeren Wertebereich generiert werden, wodurch die Wahrscheinlichkeit sinkt, die selbe Zahl mehrmals zu generieren. Dies reduziert wiederum die Anzahl an Fehlversuchen beim Einfügen in das Set.

   Sollen beispielsweise 4 Datensätze generiert werden, so werden für die Schlüssel, bei einem Faktor von 10, Zufallszahlen zwischen 0 und 40 (= Anzahl Datensätze*10) generiert. Somit ergibt sich z.B. die Liste `[30, 5, 12, 22]`.

2. Da nur ein Set generiert wird, ergibt sich das Problem, dass es vorkommen kann, dass die Anzahl der Ziffern der generierten Zufallszahl nicht ausreichend ist, um diese auf die verschiedenen Felder des Primärschlüssels zu verteilen. Um dies zu umgehen, werden linksbündig Nullen eingefügt, bis die Anzahl der Ziffern der Anzahl der benötigten Zeichen der Primärschlüsselfelder entspricht. Ist die Anzahl Ziffern bereits zu Beginn *größer Anzahl der benötigen Felder*, so entfällt dieser Schritt.

   Wurde z.B. die Zufallszahl `30` generiert und werden drei Primärschlüsselfelder benötigt, ergibt sich somit der Wert `030`.

3. Anschließend wird der kombinierte Schlüssel auf die einzelnen Primärschlüsselfelder verteilt. Hierzu wird zunächst die Anzahl der Ziffern pro Schlüsselfeld berechnet. Diese ergibt sich aus dem *abgerundeten Wert von Gesamtlänge des kombinierten Schlüssels geteilt durch die Anzahl an Schlüsseln.* Ist dies nicht restlos teilbar, so wird die Länge des ersten Feldes entsprechend erweitert, sodass alle Ziffern aufgeteilt werden können.

   Wurde beispielsweise der kombinierte Primärschlüssel `234` generiert, aber werden nur zwei Schlüsselfelder benötigt, so wird dieser wie folgt aufgeteilt: `[23][4]`.

Für das obige Beispiel ergibt sich die Liste:
`[[0,3,0], [0,0,5], [0,1,2], [0,2,2]]`.

4. Nun müssen noch die Nichtschlüsselfelder gefüllt werden. Für diese werden beliebige Zufallszahlen gewählt.

   Da der Zeitaufwand des zu testenden Algorithmus durch Sonderzeichen erhöht wird, sollen noch einige eingefügt werden. Da es jedoch keine Rolle spielt, welches der in Abschnitt 2.1.5 befindlichen Sonderzeichen bearbeitet wird, wurde exemplarisch der Backslash gewählt. Zur Entscheidungsfindung, bei welchen Feldern ein Sonderzeichen eingefügt werden soll, wird aus Effizienzgründen auf eine erneute Zufallszahlgenerierung verzichtet. Stattdessen wird hierfür die generierte Zufallszahl selbst herangezogen: Es wird geprüft, ob diese ein Vielfaches eines gewählten Divisors ist. Hierfür wird der Modulo-Operator verwendet. Ergibt Zahl modulo Divisor 0, so wird ein Backslash angehängt.

In der für diese Masterarbeit realisierten Implementierung wurden für den Divisor 13 und für den Faktor nur 100 gewählt, um eine Überschreitung des Wertebereichs des gewählten Datentypens zu vermeiden [2].

6.2.2 Manipulation der Datenbank

Für die Manipulation des CSV's müssen Datensätze entfernt, geändert und eingefügt werden. Hierbei ist auf diese Reihenfolge zu achten, da sonst geänderte Datensätze gelöscht bzw. neu eingefügte Datensätze verändert werden könnten, was zu einer Verfälschung der gewünschten Prozentsätze führen würde.

1. Zuerst wird das Löschen vorgenommen. Dies muss als erster Schritt erfolgen, da es sonst vorkommen könnte, dass ein neu eingefügter Datensatz bzw. ein bereits veränderter Datensatz gelöscht wird. Würde dies geschehen, so wäre zwar der zu löschende Prozentsatz erfüllt, jedoch nicht die für Einfügen und Ändern, was wiederum eine Verfälschung der Messergebnisse der Evaluation zur Folge hätte.

   Zum Löschen der Datensätze wird im ersten Schritt die Anzahl der zu löschenden Datensätze anhand des Prozentsatzes ermittelt. Anschließend werden die zu löschenden Datensätze ausgewählt. Hierzu wird eine Zufallszahl zwischen *0* und *Länge Liste - 1* gewählt. Diese Zufallszahl wird erst wieder einem Set

[2] Bei dem in Kapitel 4 vorgestellte Algorithmus findet neben der Überprüfung auf Sonderzeichen auch noch eine Überprüfung auf `NULL`-Felder statt. Da die Verarbeitung dieser `NULL`-Felder jedoch zeitlich keinen Unterschied aufweist, wurde auf die Abbildung dieser Felder aus Performanzgründen im Testprogramm verzichtet

hinzugefügt, um Duplikate zu vermeiden. Dieser Vorgang wird so lange wiederholt, bis die zu löschende Anzahl an Datensätzen erreicht ist. Abschließend werden diese Zeilen aus der Liste entfernt.

2. Nach dem Löschen der Datensätze erfolgt das Ändern. Auch hier gilt, dass dieser Schritt vor dem Einfügen vorgenommen werden muss, damit keine neu eingefügten Datensätze manipuliert werden.

   Zuerst werden anlog zum Löschen die Anzahl der zu änderten Datensätze ermittelt, ausgewählt und einem Set hinzugefügt. Danach werden für die ausgewählten Listen für die Nichtschlüsselelemente neue Zufallswerte berechnet und die alten Werte ersetzt. Hierbei werden zufällig Nichtschlüsselelemente ersetzt.

3. Abschließend müssen noch die neuen Datensätze hinzugefügt werden. Dies stellt den zeitaufwendigsten Schritt bei der Manipulation dar, da bei diesem Schritt neue Primärschlüssel generiert werden müssen, d.h. es muss überprüft werden, ob die neuen Werte nicht bereits als Primärschlüssel existieren. Um hierbei den Abgleich mit den alten Primärschlüsseln zu vermeiden, werden die neuen Zufallszahlen zwischen dem *maximalen Wert der bereits generierten kombinierten Primärschlüsseln + 1* und *(maximalen Wert der bereits generierten kombinierten Primärschlüsseln + Anzahl einzufügender Datensätze) * Faktor* gewählt. Die Vorgehensweise ist hierbei analog zum Generieren der Datenbank. Zuerst wird ein kombinierter Schlüssel erzeugt, dieser wird anschließend mit Nullen erweitert, geteilt und abschließend die Nichtschlüsselfelder mit Zufallswerten aufgefüllt. Die neue generierten Datensätze werden abschließend der bereits existierenden Liste hinzugefügt.

6.2.3 Vermischung der manipulierten Datenbank

Im letzten Schritt soll die zuvor manipulierte Liste noch auf verschiedene Arten vermischt werden. Zu den Varianten zählen: invertieren, zufällig vermischen und blockweise zufällig vermischen.

- Bei der Invertierung wird auf die `reverse`-Methode des Sets zurückgegriffen und so die Reihenfolge der Liste umgekehrt, d.h. es werden letztes mit erstem Element, zweites mit vorletztem usw. ausgetauscht.
- Bei dem zufälligen Vermischen wird auf die `shuffle`-Methode des Sets zurückgegriffen.

- Bei der blockweisen zufälligen Vermischung wird die Listen in kleinere Listen einer festen Blocklänge unterteilt. Diese kleineren Listen werden wieder zu einer großen Liste hinzugefügt, die dann mittels der `shuffle`-Methode zufällig vermischt wird.

6.3 Software-Design

Im folgenden Kapitel wird das Design und der Ablauf des Programms mittels Klassen- und Sequenzdiagramm erläutert. Es wird exemplarisch nur die Vermischungsvariation des zufälligen Vermischens abgebildet. Die beiden anderen Mischmethoden haben eine analoge Kommunikation.

6.3.1 Aufbau

Abbildung 6.1 zeigt anhand eines vereinfachten Klassendiagramms den Aufbau von CSV_Generator. Dieses setzt sich aus den Klassen `GUI`, `Values`, `Calculator`, `DataBase_Creator`, `DataBase_Manipulator`, `DataBase_Permutator` sowie `CSV` zusammen.

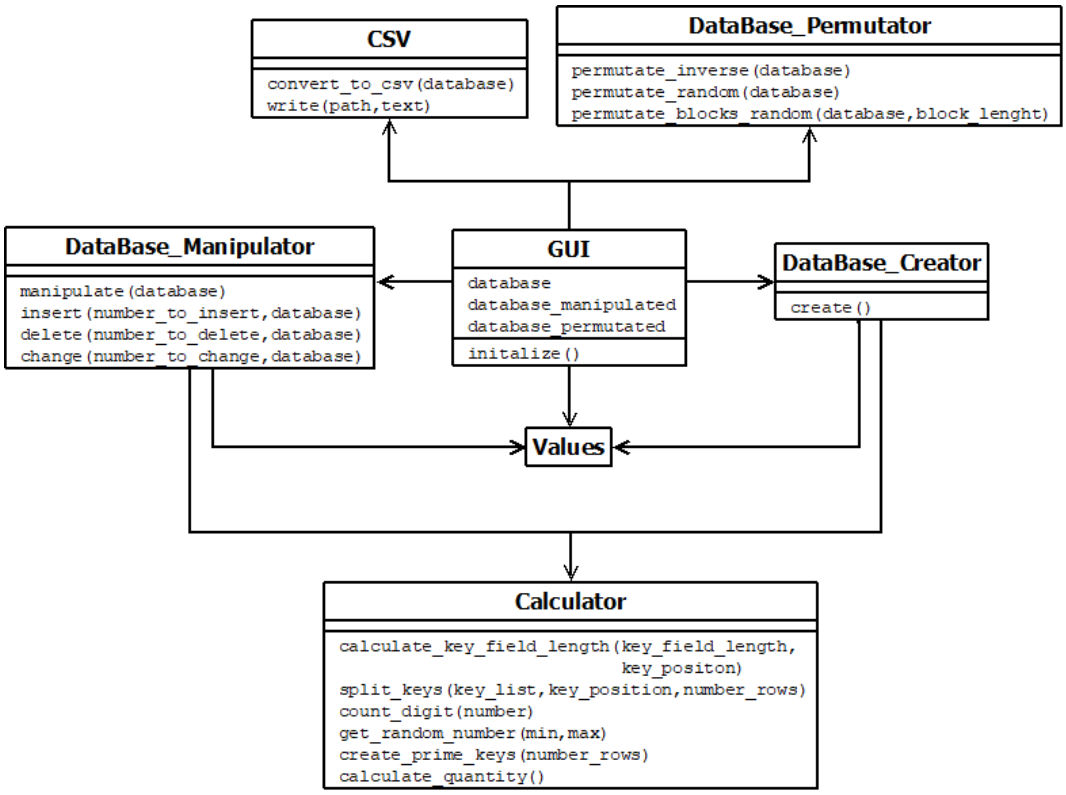

Abbildung 6.1: Klassendiagramm des CSV-Generators

Die Klasse `Values` verwaltet mittels Gettern und Settern die über die GUI eingelesenen Werte. `DataBase_Creator` erzeugt unter Zuhilfenahme der Methode `create` und der Klasse `Values` die Ausgangsdatenbank `database`, die in `GUI` zwischengespeichert wird.

`DataBase_Manipulator` manipuliert die von `DataBase_Creator` generierte Datenbank mittels der Methode `manipulate`. Die Hilfsmethode `insert` übernimmt hierbei das Einfügen neuer Datensätze in die Datenbank. Analog hierzu löscht `delete` und `change` ändert Datensätze.

Beide Klassen bedienen sich der Methoden der Klasse `Calculator`. Hierbei handelt es sich um eine Hilfsklasse, die Hilfsmethoden bereitstellt (wie Aufteilen von Schlüsseln auf Felder, Generierung von Primärschlüsseln, etc.).

Die Klasse `DataBase_Permutator` dient der Permutation von übergebenen Datenbanken. Hierfür bietet die Klasse drei verschiedene Methoden. `permutate_inverse` kehrt die Reihenfolge der Einträge der Datenbank um. Mittels `permutate_random` werden die Datensätze zufällig gemischt, wohingegen `permutate_blocks_random` die Datenbank in Blöcke der Länge `block_length` unterteilt und diese untereinander wieder zufällig mischt.

Den Schreibvorgang in eine Datei übernimmt die Klasse `csv`. Hierfür wandelt die Methode `convert_to_csv` die übergebene Datenbank in einen String in CSV-Format um. Die Methode `write` schreibt diesen dann an den angegebenen Pfad.

6.3.2 Kommunikation

Das Sequenzdiagramm aus Abbildung 6.2 verdeutlicht die Kommunikation zwischen den einzelnen Klassen. Hierbei wurde aus Gründen der Übersichtlichkeit auf die Hilfsklasse `Calculator` verzichtet. Ferner wird nur bei `GUI` der Aufruf der eigenen Klasse gezeigt. Bei den anderen Klassen wird aus Gründen der Übersichtlichkeit hierauf verzichtet.

Nach dem Start des Programms durch den Nutzer wird die `GUI` mittels der Methode `initialize()` erzeugt und aufgerufen. Anschließend kann der Nutzer alle von ihm gewünschten Optionen anwählen und die entsprechenden, hierzu benötigten Werte eintragen.

Mit Bestätigung der Eingabe mit *Start* beginnt die eigentliche Kommunikation zwischen den Klassen. Die Klasse `GUI` übermittelt mittels Settern ihre Werte an die Hilfsklasse `Values`. Danach ruft sie über die Methode `create()` die Klasse `DataBase_Creator` auf. Diese holt sich die benötigten Werte aus der Klasse

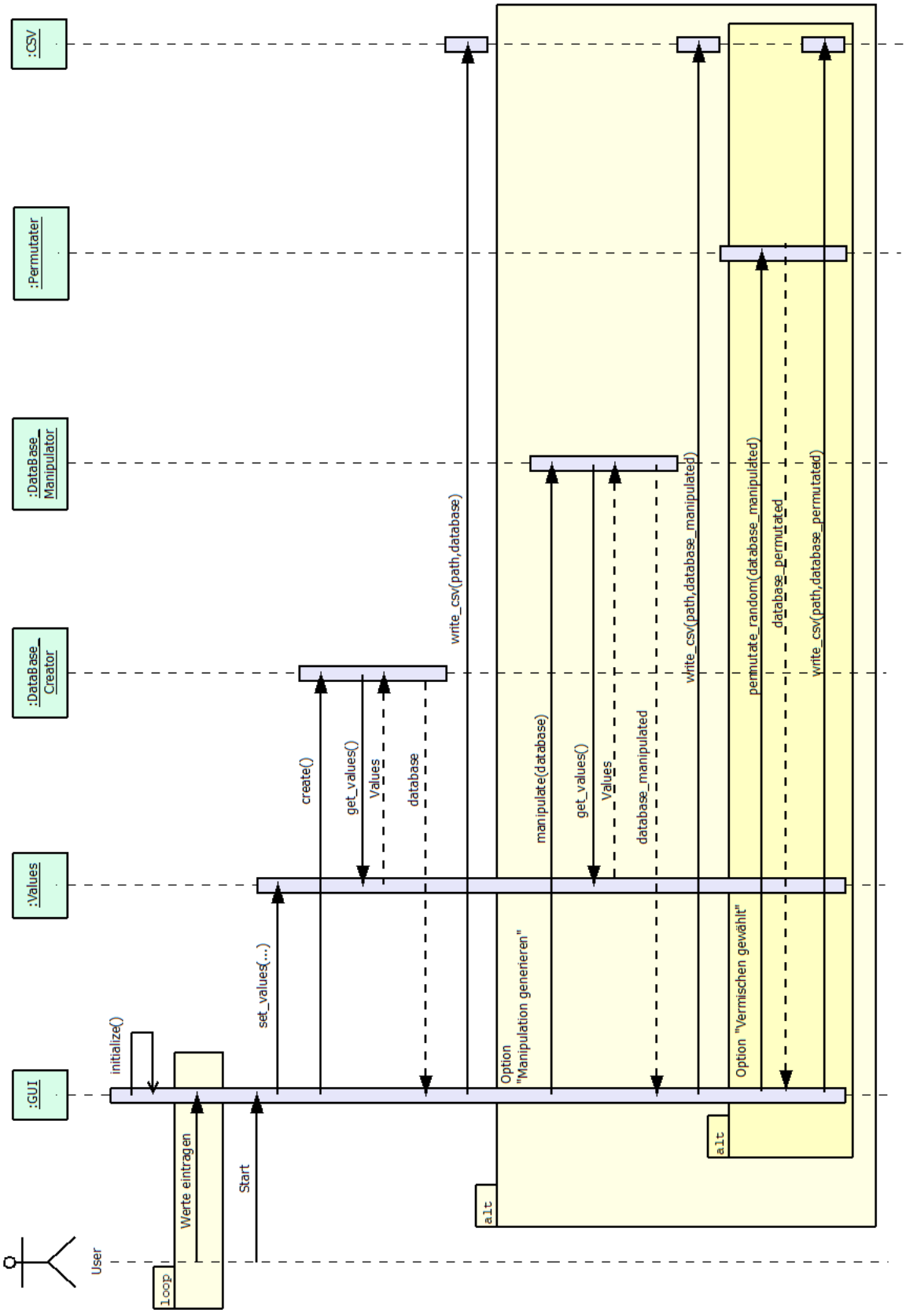

Abbildung 6.2: Sequenzdiagramm des CSV-Generators

`Values` und liefert anschließend die generierte Datenbank `database` an `GUI` zurück. `GUI` wiederum übermittelt die Datenbank an die Klasse `CSV`, die diese mittels der Methode `write(path, database)` in ein Textfile schreibt.

Anschließend wird überprüft ob die Option *Manipulation generieren* gewählt wurde. Ist dies nicht der Fall, so endet die Kommunikation hier. Anderenfalls ruft die GUI mittels der Methode `manipulate(database)` die Klasse `DataBase_Manipulator` auf. Diese fordert analog zu `DataBase_Creator` die benötigten Werte von `Values` an und übermittelt die nun manipulierte Datenbank `database_manipulated` zurück an die `GUI`, die wiederum mittels `write` die Klasse `CSV` kontaktiert.

Abschließend wird überprüft ob die Option *Zufälliges Vermischen* gewählt wurde. Trifft dies zu, so wird mittels der Funktion `permuatate_random` der Klasse `Permutator` die Datenbank `database_permutated` erzeugt. `Permutator` schickt die vermischte Datenbank zurück an `GUI`, die wiederum anlog zum Generieren und Manipulieren mittels `write(path, database_permutated)` an die Klasse `CSV`.

6.4 GUI-Beschreibung

Abbildung 6.3 zeigt die Benutzeroberfläche des CSV-Generators. Die obere Hälfte enthält die zur Generierung benötigten Felder, die unteren die zur Manipulation.

Mittels *Anzahl Datensätze* wird die Anzahl der zu generierenden Datensätze (Zeilen) festgelegt. *Anzahl Felder* wiederum gibt an, wie viele Felder (Spalten) jeder Datensatz besitzen soll. Die Position der Schlüssel wird im Feld *Position Schlüssel* festgelegt. Die einzelnen Werte werden mittels Semikolon voneinander getrennt. Ist die Position der Schlüssel beispielsweise 3 und 5, so muss `3;5` eingetragen werden.

Ferner müssen noch *Feldtrennzeichen*, *Datensatztrennzeichen* sowie *Textbegrenzungszeichen* gesetzt werden.

Die Box *Manipulation generieren* ermöglicht es, aus dem zuvor generierten CSV ein neues CSV zu generieren. Hierbei werden mittels der Felder *Insert*, *Delete*, *Update* die Prozentsätze festgelegt, in denen sich das neue generierte CSV vom alten unterscheidet. Werden z.B. alle drei Werte auf 10 gesetzt so wird bei 10 Datensätzen einer neu generiert, einer gelöscht und einer geändert. *Dateipfad Ausgabe* gibt an, unter welchem Pfad und Namen das neu generierte CSV abgespeichert werden soll.

Ferner bietet das Programm die Möglichkeit, die zuvor generierte Manipulation mittels Vertauschung von Zeilen erneut zu manipulieren. Die Art der Vermischung kann durch die drei Check-Boxen *Zufall*, *Invertiert* und *Blockweise* ausgewählt werden. Hierbei können beliebig viele ausgewählt werden. Bei *Zufall* werden die einzelnen

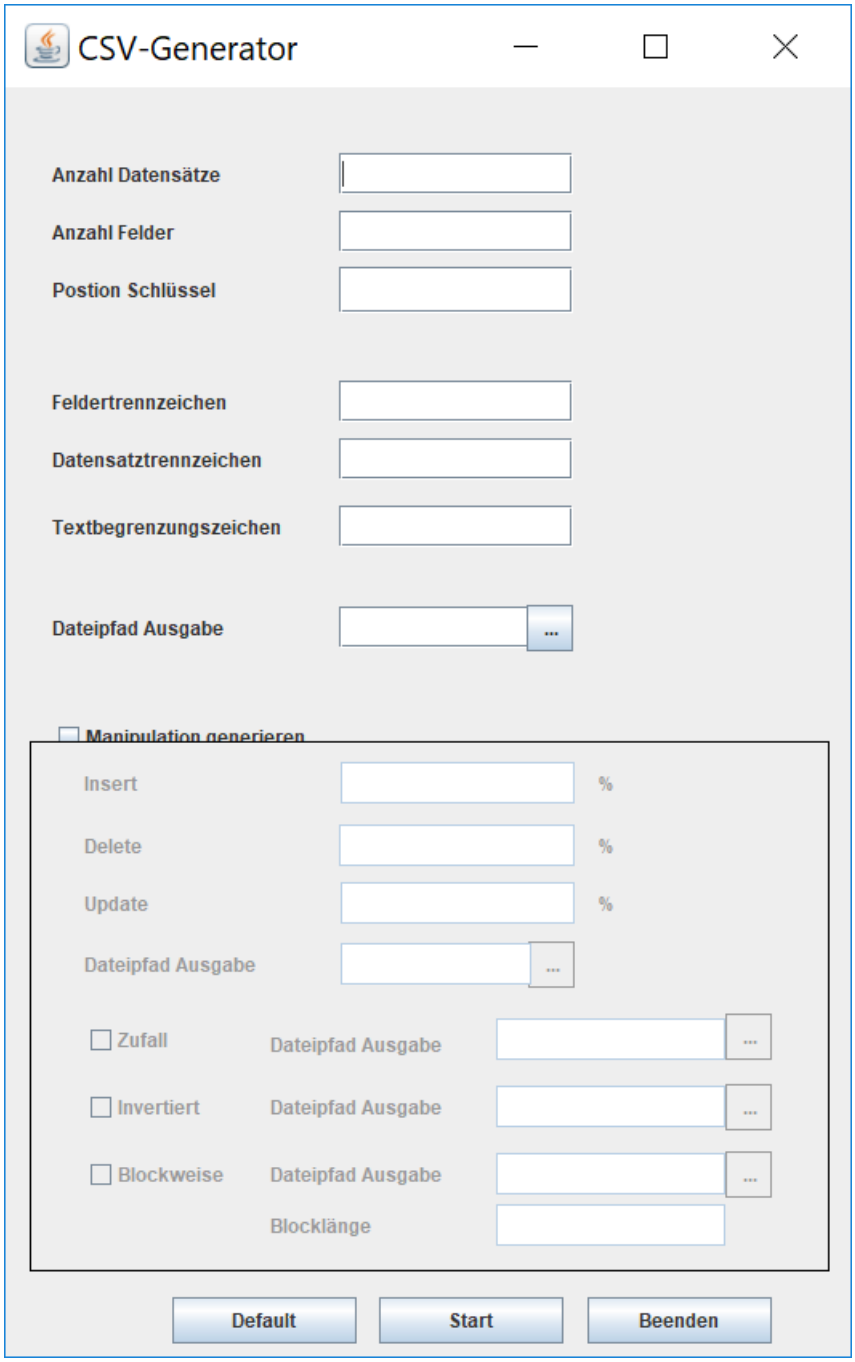

Abbildung 6.3: Benutzeroberfläche des CSV-Generators

Zeilen des CSV's zufällig in einer neuen Reihenfolge angeordnet. Bei *Invertiert* wird das bestehende CSV herumgedreht, d.h. die erste Zeile wird zur letzten, die letzte zur ersten, die zweite zur vorletzten, usw. Bei der Option *Blockweise* wiederum wird das CSV in Blöcke der Länge *Blocklänge* unterteilt. Diese Blöcke werden dann in zufälliger Reihenfolge zu einem neuen CSV angeordnet. Bei dieser Option muss die gewünschte Blocklänge angegeben werden.

Weiterhin muss für jedes zu generierende CSV noch in *Dateipfad Ausgabe* der Dateipfad sowie der Name angegeben werden, unter dem es gespeichert werden soll. Dies kann entweder direkt über das Textfeld oder unter Zuhilfenahme des daneben befindlichen Buttons erfolgen. Dieser öffnet einen Pfaddialog.

Die *Start*-Schaltfläche startet die Generierung der ausgewählten CSV's. Mittels der *Beenden*-Schaltfläche wird das Programm beendet. Die *Default*-Schaltfläche hingegen setzt alle Felder auf einen vorinitialisierten Wert.

6.5 Ausblick

Für die Optimierung des CSV-Generators existieren im Wesentlichen drei Ansatzpunkte. Zum einen können weitere Varianten der Vermischung hinzugefügt werden.

Der zweite Ansatzpunkt ist eine andere Verteilung bei der Primärschlüsselerstellung. Der vorgestellte Algorithmus erzeugt gleichverteilte Primärschlüssel. Hier könnte ein Ansatz gewählt werden, der bei der Generierung kombinierter Schlüssel nur ein Feld variieren lässt, bzw. eine prozentuale Verteilung der Primärschlüssel vornimmt. Ein Beispiel soll diese Idee verdeutlichen.

Bislang werden für ein CSV mit fünf Zeilen und fünf Spalten, wobei die ersten drei den kombinierten Primärschlüssel bilden, beispielsweise eine Datei folgender Form geniert.

```
1 4|45|12|564/
2 5|56|23|434
3 3|02|45|34
4 1|80|34|56
5 9|99|45|34
```

Hierbei sind alle Primärschlüsselfelder komplett zufällig und gleich verteilt.

Der neue Algorithmus soll hingegen Dateien folgender Form liefern:

```
1 01|02|12|564/
2 01|02|23|434
3 01|02|45|34
4 01|02|34|56
5 02|02|45|34
```

Bei dieser Variante variieren in der ersten Spalte nur 20 Prozent der Primärschlüsselwerte. Bei der zweiten sind sogar alle konstant. Lediglich die dritte ist komplett zufällig gewählt.

Ferner kann das Programm um eine Einlesemöglichkeit für bereits existierende CSV's erweitert werden. Diese könnten dann mit Hilfe des CSV-Generators manipuliert werden.

6.6 Zusammenfassung

In diesem Kapitel wurde der Java-basierte CSV-Generator vorgestellt. Dieser dient der Generierung von CSV-Dateien zu Testzwecken sowie zur Evaluation des in Kapitel 4 vorgestellten Algorithmus. Die Generierung der CSV-Dateien geschieht u.a. in Abhängigkeit der gewünschten Primärschlüsselpositionen. Zudem kann die Verteilung der Manipulation beeinflusst werden.

Der Fokus liegt auf der dem Algorithmus zu Grunde liegenden Idee sowie dem konzeptionellen Aufbau des Programms. Dieses kann in Zukunft noch durch weitere Möglichkeiten der Vermischung sowie der Generierung nicht gleich verteilter Primärschlüssel erweitert werden. Zudem soll das Einlesen bereits existierender CSV-Dateien ermöglicht werden.

7 Evaluation

Nachdem in Kapitel 6 der Testdatenbankgenerator und in Kapitel 4 sowie 5 die zu evaluierenden Algorithmen vorgestellt wurden, beschreibt dieses Kapitel deren Evaluation. Dabei werden zunächst die verwendeten PROLOG-Varianten noch einmal vorgestellt. Zudem wird ein kurzer Einblick in das verwendete Messskript sowie die verwendeten Messdaten gegeben. Abschließend werden die Messergebnisse sowie die daraus resultierenden Erkenntnisse diskutiert.

7.1 Evaluierte Prolog-Varianten

Nachfolgend soll eine kurze Übersicht über die getesteten PROLOG-Algorithmen erfolgen. Diese sollen anhand der in Listing 7.1 gezeigten Beispielausgabe eines Blocks von tablediff erläutert werden.

Listing 7.1: Beispielausgabe eines tablediff-Blocks

```
1c1
< 5555|Berlin|Z8W0|02\87|77.00
< 6666|München|S7T3|02\87|2.50
< 1111|Dortmund|D8E5|78\54|6.10
< 2222|Dortmund|R7T5|56\76|1000.40
---
> 1111|Dortmund|D8E5|78\54|0.00
> 2222|Dortmund|R7T5|56\76|0.00
> 7777|Nürnberg|Z8W0|02\87|77.00
> 8888|Hamburg|S7T3|02\87|2.50
```

Die Algorithmen lassen sich grob in zwei Kategorien unterteilen. Zum einen gibt es die Kategorien von Algorithmen, die `INSERT`-, `DELETE`- und `UPDATE`-Statements generieren. Zu diesen zählen:

- **block**
  Pro Manipulation (Anweisung) wird ein Statement generiert. Die Auswertung eines **Blocks** erfolgt, sobald dieser komplett eingelesen wurde. Bei dieser Variante handelt es sich um den in Abbildung 4.5 gezeigten Algorithmus.

Für das Beispiellisting 7.1 werden bei dieser Variante zwei `UPDATE`-, zwei `DELETE`- und zwei `INSERT`-Statements generiert.

- **block_1_statement**
  Pro **Block** und Anweisungsart wird **ein Statement** generiert. Die Auswertung eines Blocks und die Generierung der Statements erfolgt, sobald dieser komplett eingelesen wurde.

  Hier werden die beiden `INSERT`-Statements zu einem kombinierten zusammengefasst, ebenso die `DELETE`'s. Die beiden `UPDATE`-Statements bleiben erhalten.

- **1_statement**
  Es werden insgesamt nur maximal **ein** `INSERT`, ein `DELETE` und n `UPDATE`-**Statements** generiert. Die Generierung erfolgt hierbei, sobald das ganze File eingelesen wurde.

Bei der zweiten Kategorie von Algorithmen wurde auf `INSERT`-Statements verzichtet. Stattdessen werden die einzufügenden Daten in ein CSV-File zwischengespeichert und anschließend mittels `LOAD-INTO` in die Datenbank eingelesen. Zu dieser Kategorie zählen:

- **load**
  Pro Block werden ein `DELETE` und n `UPDATE`-Statements generiert. Die einzufügenden Datensätze werden in eine separate Datei geschrieben und mittels `LOAD-INTO` (**load**) geladen.

- **load_del**
  Es werden nur `DELETE`-Statements generiert (**del**); pro Block je ein Statement. Anstelle von `UPDATE`-Statements werden die alten Daten gelöscht, die geänderten in die CSV-Datei geschrieben und anschließend zusammen mit den einzufügenden Daten geladen (**load**).

- **load_1_statement**
  Kombination aus **load** und **1_statement**. Die einzufügende Daten werden in eine Datei geschrieben und per `LOAD-INTO` geladen. Für die zu löschenden Daten wird für das gesamte File ein `DELETE`-Statement generiert. Pro Änderung erfolgt ein `UPDATE`-Statement.

- **load_del_1_statement**
  Kombination aus **load_del** und **1_statement**. Die zu aktualisierenden Daten werden zuerst gelöscht, in ein File geschrieben und dann per `LOAD-INTO` geladen. Hierbei wird für das gesamte File ein `DELETE`-Statement generiert.

7.2 Verwendete Messdatenbanken

Für eine möglichste umfassende Untersuchung des Einflusses der Eigenschaften der Datenbanken und des Umfangs der Änderungen der beiden zu synchronisierenden Datenbanken auf den Zeitbedarf der Synchronisation zu untersuchen, wurde eine Reihe von Testdatenbanken generiert. Grundgedanke war, dass jeweils nur ein Parameter pro Untersuchung variiert wird. „Man nennt dies in der Wissenschaft »die Variable isolieren«.“ [ZTZ+09] Diese Datenbanken wurden mittels des in Kapitel 6 vorgestellten CSV-Generators erstellt.

Um hierbei eine möglichst praxisnahe Anwendung zu simulieren, aber die Laufzeit dennoch im Rahmen zu halten, haben sich folgende Basis-Parameter im Laufe der Evaluation ergeben:

- 10.000 Datensätze je Datenbank
- 6 Felder pro Datensatz
- 3 Primärschlüsselfelder (an den ersten drei Positionen)
- 1% neu eingefügte Daten (`INSERT`)
- 1% gelöschte Daten (`DELETE`)
- 1% geänderte Daten (`UPDATE`)
- Sonderzeichen sollen enthalten sein

Bei den für die Evaluation verwendeten Datenbanken wurden hierbei folgende Parameter variiert:

- Anzahl Datensätze
- Anzahl Felder
- Anteil Primärschlüssel
- Prozentsatz `INSERT`
- Prozentsatz `DELETE`
- Prozentsatz `UPDATE`

Auf eine Untersuchung des Einflusses der Position der Primärschlüssel sowie der Permutationen der manipulierten Datenbank gegenüber der Ausgangsdatenbank wurde auf Grund des beschränkten Umfangs dieser Arbeit verzichtet. Der CSV-Generator kann jedoch auch für die Erzeugung dieser genutzt werden.

7.3 Das Messskript test.sh

Für die Evaluation wurde neben dem in Kapitel 6 vorgestellten Java-basierten CSV-Generator ein eigens hierfür konzipiertes Messskript erstellt. Dieses erfüllt im wesentlichen folgende Aufgaben:

- Durchführen einer gewünschten Anzahl an Messdurchläufen
- Iteration über verschiedene Datenbanken und Algorithmen
- Laden der erforderlichen Datenbanken
- Synchronisation der Datenbanken mittels csv2sql
- Protokollieren der Messergebnisse der Zeitmessungen
- Überprüfung der Korrektheit der Synchronisation

Die Hauptfunktion des Messskriptes ist die Anwendung verschiedener Varianten des Prolog-Algorithmus mit verschiedenen Datenbanken. Hierfür wird zunächst die benötigte Originaldatenbank in MySQL geladen. Zudem wird für den späteren Test die manipulierte Datenbank in eine zweite Tabelle geladen. Danach wird der Aufruf des Produktivskriptes csv2sql vorbereitet. Hierzu werden die in dieser Iteration verwendeten Prolog-Dateien, die Datenbanken und die dazugehörige Konfigurationsdatei umkopiert. Anschließend erfolgt der Aufruf von csv2sql. In Rahmen dessen wird tablediff sowie der entsprechende Prolog-Algorithmus angewendet. Die daraus resultierenden Statements werden anschließend auf die Originaltabelle angewendet und die hierfür benötigte Zeit gemessen. Die Einbettung des Produktivskriptes in das Testskript hat den Vorteil, dass dies bei jedem Durchlauf automatisch mit getestet wird. Anschließend erfolgt eine Überprüfung, ob die beiden Datenbanken korrekt synchronisiert wurden. Die genaue Funktionsweise dieses Tests wird in Abschnitt 7.4 erläutert. Dies geschieht mehrmals, da es bei den Messungen zu zufälligen zeitlichen Schwankungen kommt. Wurde jede Datenbank mit jedem Prolog-Algorithmus mehrmals durchlaufen, endet das Skript.

7.4 Verifikationstest

Wie in Abschnitt 7.3 erwähnt, umfasst die Evaluation auch noch einen Verifikationstest, der überprüft, ob die beiden Datenbanken korrekt ineinander überführt wurden. Die Grundidee, auf der dieser Softwaretest basiert, soll in diesem Abschnitt erläutert werden.

Wurde die Originaldatenbank korrekt in die manipulierte Datenbank überführt, so haben die beiden Datenbanken die selbe Anzahl an Einträgen. Werden anschließend alle Einträge, die in beiden Datenbanken identisch sind, aus der manipulierten Datenbank entfernt, so hat diese die Größe 0. Trifft eine dieser beiden Bedienungen nicht zu, so wurden die Datenbanken nicht korrekt synchronisiert.

7.5 Messergebnisse

Im Rahmen der Messungen ergaben sich starke zufällige Schwankung der benötigten Ausführungszeiten. Dieser Effekt war vor allem bei den absoluten Ausführungszeiten der MySQL-Statements sichtbar. Dies ist ein in der Literatur bekannter Effekt, der auf etliche, schwer zu beeinflussende Einflüsse zurückzuführen ist, z.B. Defragmentierung oder Aufteilung auf der Festplatte [ZTZ+09].

Damit diese Schwankungen die Messergebnisse nicht zu stark verfälschen, wurden sowohl in der Messumgebung als auch in der Auswertung der Ergebnisse einige Gegenmaßnahmen getroffen. So wurden während der Messung möglichst viele Störfaktoren eliminiert, z.B. das WLAN deaktiviert, alle offenen Fenster außer dem ausführenden Terminalfenstar geschlossen und der Login-Screen aktiviert. Zudem wurde im Testskript eine Pause von 10 s vor dem Start des Produktivskripts eingebaut, damit Betriebssystemvorgänge, z.B. aufgrund der vorhergehenden Kopier- und Löschvorgänge vor der Messung abgeschlossen sind. Da es dennoch zu Extremwerten bei einzelnen Messungen kommen kann, wurden die Messungen mehrmals durchgeführt und anstelle des Mittelwertes der Median verwendet.

Des Weiteren wurde bewusst auf Messungen auf einem Server verzichtet und ein langsameres System gewählt, damit die Unterschiede zwischen den einzelnen Algorithmen stärker ersichtlich werden. Die Messungen erfolgten daher unter Linux (Ubuntu 16.04), auf einem Rechner mit einem AMD Athlon(tm) II P340 Dual-Core-Prozessor und 4 GB RAM. Es wurde die MySQL-Version 5.7 sowie die Prolog-Version 7.2.3 verwendet.

7.5.1 Anzahl Datensätze

Um eine geeignete Datenbankgröße zu finden, die sich als Basis für die anderen Messungen eignet, wird im ersten Schritt der Einfluss der Datenbankgröße auf die Laufzeit untersucht.

Es ergab sich hierbei eine Größe von 10.000 Datensätzen als optimal. Zum einen stellt dies eine praxisrelevante Datenbankgröße dar. Zum anderen ergaben sich hiermit

adäquate Laufzeiten der Messungen, die aufgrund der Wiederholungen und der zahlreichen Kombination von Datenbanken und Algorithmen durchaus im zweistelligen Stundenbereich lagen. Zudem ergeben sich hierbei Messwerte in einer Größenordnung, die eine gute Darstellbarkeit ermöglichen und es somit erlauben, die Unterschiede zwischen den einzelnen Algorithmen anschaulich aufzuzeigen.

Bei der Auswertung der Messungen ergab sich, dass sich ein Single-Component-Benchmarking [ZTZ+09] anbietet, da dies die Übersichtlichkeit der Diagramme verbessert. Bei dieser wird nicht die Laufzeit der Gesamtanwendung (tablediff, Prolog und MySQL), sondern nur die von einer einzelnen Komponente betrachtet, in diesem Fall MySQL. Dies bietet sich an, da bei einer Datenbankgröße von 10.000 sich im Mittel folgendes Verhältnis zwischen den einzelnen Komponenten ergab:

- tablediff: 1%
- Prolog: 1%
- MySQL: 98%

Mit steigender Anzahl an Datensätzen verschiebt sich dieses Verhältnis noch weiter zu Gunsten von tablediff und Prolog. Auch bei den anderen Messungen und Datenbanken ergaben sich ähnliche Verhältnisse.

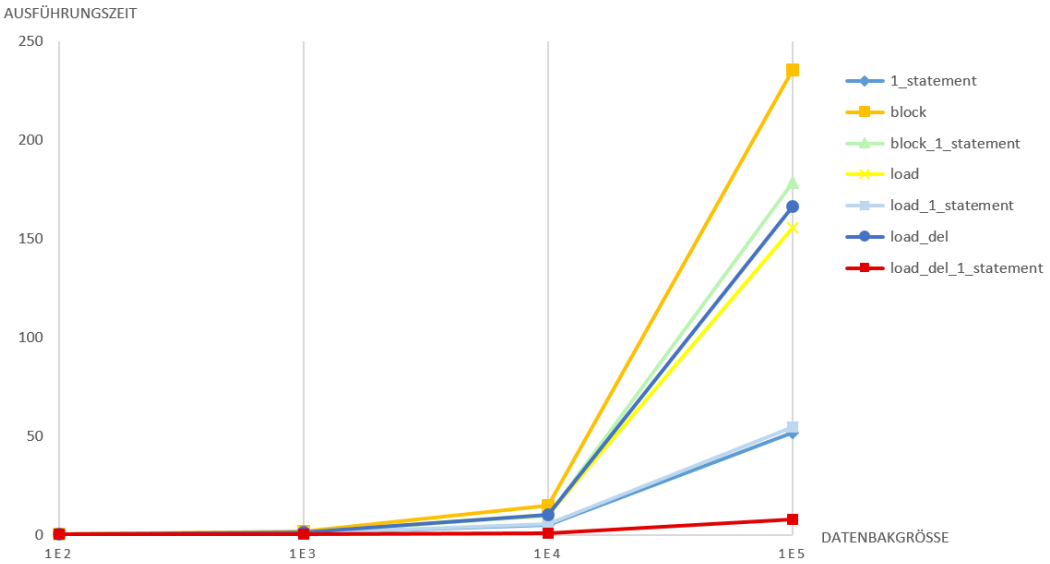

Abbildung 7.1: Laufzeitdiagramm der verschiedenen Varianten von csv2sql in Abhängigkeit der Anzahl der Datensätze

Abbildung 7.1 zeigt die benötigte Laufzeit in Abhängigkeit von der Datenbankgröße. Hieraus ist ersichtlich, dass der Anstieg bei allen Algorithmen überproportional erfolgt.

Erwartungsgemäß ist *block* der ineffizienteste Algorithmus, da bei diesem die meisten Einzelstatements generiert werden und u.a. für jedes Statement eine erneute Verbindung zur Datenbank aufgebaut werden muss.

Die nächste Gruppe stellen *block_1_statement*, *load_del* und *load* dar. Dies ist darin begründet, dass bei den *load*-Varianten viele Einzelstatements durch ein `LOAD-INTO`-Statement ersetzt bzw. bei *block_1_statement* bereits einige Statements zusammengefasst werden.

Da sich bereits große Vorteile durch das Zusammenfassen von Statements gezeigt hatten, wurden im nächsten Schritt die Algorithmen dahingehend optimiert, dass sämtliche einzufügende Daten zu einem `INSERT`-Statement bzw. alle zu löschenden Daten zu einem `DELETE`-Statement zusammengefasst wurden. Bei den *load*-Varianten wurde wieder anstelle des `INSERT`-Statements ein `LOAD-INTO`-Statement generiert. Somit entstanden die drei Algorithmen *load_del_1_statement*, *load_1_statement* und *1_statement*. Zwischen *load_1_statement* und *1_statement* ist im Rahmen dieser Messung keine signifikanter Unterschied ersichtlich.

load_del_1_statement stellt im Rahmen dieser Messung die beste Alternative dar.

Einige Firmen sehen jedoch aufgrund von Sicherheitsüberlegungen, z.B. Problemen mit den Zugriffsrechten, von einer Lösung mittels `LOAD-INTO` ab. Betrachtet man diesen Aspekt, so ist *1_statement* die beste Lösung.

7.5.2 Anzahl Felder

Nachdem der Einfluss der Anzahl der Datensätze auf die Laufzeit untersucht wurde, erfolgt im nächsten Schritt eine Betrachtung des Einflusses der Anzahl der Felder.

Für diese Messung wurden als Basis-Parameter 10.000 Datensätze, 1% `INSERT`, 1% `DELETE` und 1% `UPDATE` sowie 50% Primärschlüsselfelder gewählt. Die Anzahl der Felder wurde schrittweise erhöht, wobei der Anteil der Primärschlüsselfelder auf 50% konstant gehalten wurde.

Abbildung 7.2 zeigt den Einfluss der Feldanzahl auf die Laufzeit. Hieraus ist ersichtlich, dass die Feldanzahl bei gleichbleibendem Primärschlüsselfelder-Verhältnis bei allen Algorithmen keinen Einfluss auf die Laufzeit hat.

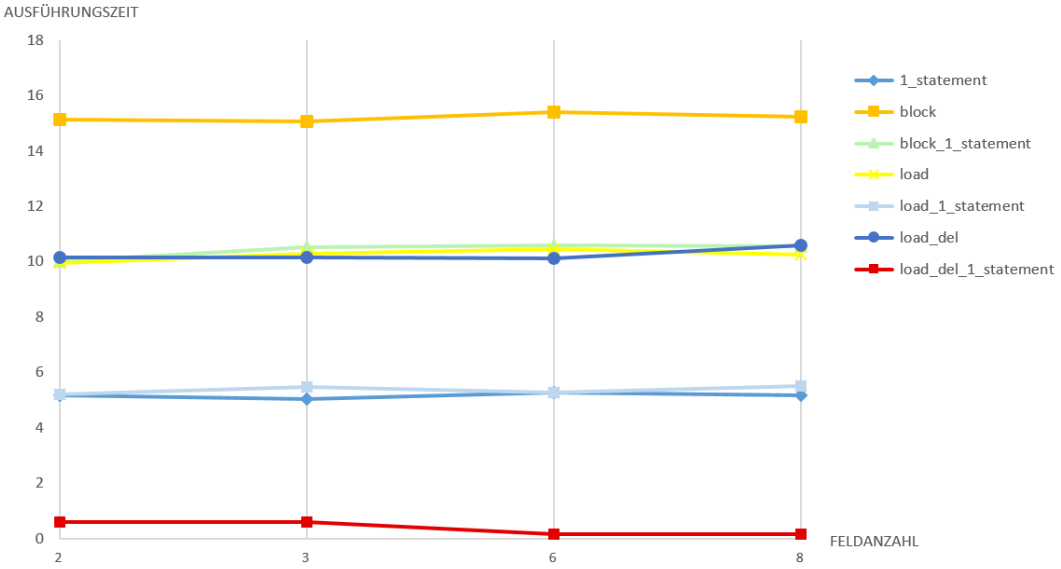

Abbildung 7.2: Laufzeitdiagramm der verschiedenen Varianten von csv2sql in Abhängigkeit der Anzahl der Felder

7.5.3 Anteil Primärschlüssel

Im nächsten Schritt wurde die Anzahl der Felder konstant gehalten und der Anteil der Primärschlüssel variiert. Die restlichen Basis-Parameter wurden auf 10.000 Datensätze, 1% `INSERT`, 1% `DELETE` und 1% `UPDATE` sowie sechs Felder gesetzt.

Auch hier ist aus Abbildung 7.3 ersichtlich, das dies bei keinem der Algorithmen Einfluss auf die Laufzeit hat. Bei *load_1_statement* ist zwar ein Ausschlag ersichtlich, dies ist jedoch auf bereits erwähnten Schwankungen seitens MySQL zurückführbar (siehe Abschnitt 7.5).

7.5.4 Prozentsatz neu eingefügte Datensätze

Nach der Untersuchung des Einflusses der einzelnen Parameter der Datenbank auf die Laufzeit werden in den nächsten Schritten der Einfluss der Manipulationen auf die Laufzeit betrachtet.

Hierzu wird im ersten Schritt die Anzahl der eingefügten Daten variiert. Dabei wurde ein Rahmen zwischen 1% und 5% gewählt, was 100 bis 500 Änderungen pro Synchronisation entspricht. Als Basis-Parameter wurden hierbei 10.000 Datensätze gewählt, sowie sechs Felder, bei denen die ersten drei Felder Primärschlüsselfelder sind.

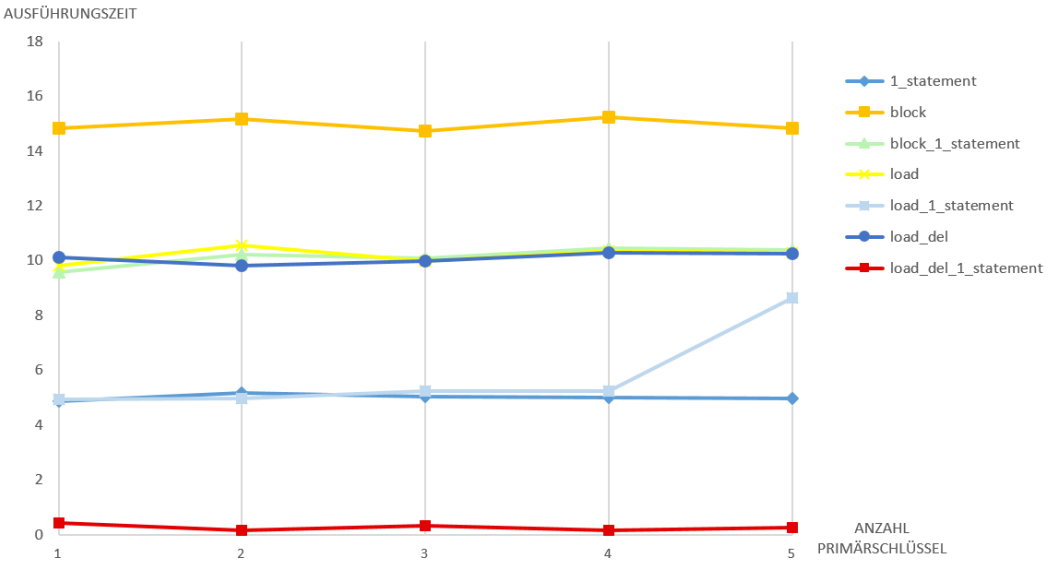

Abbildung 7.3: Laufzeitdiagramm der verschiedenen Varianten von csv2sql in Abhängigkeit des Anteils der Primärschlüssel

Da hier gezielt der Einfluss der neu eingefügten Daten untersucht werden soll, wurden `DELETE` und `UPDATE` auf 0% gesetzt. Hierdurch verhalten sich *load_1_statement*, *load*, *load_del* und *load_del_1_statement* identisch, da diese sich nur in der Verarbeitung von `DELETE`- und `UPDATE`-Statements unterscheiden, die aber in dieser Messreihe nicht vorkommen. Somit sind nur Unterschiede zwischen *block*, *block_1_statement*, *1_statement* und *load_del_1_statement* zu erwarten.

Die Messung in Abbildung 7.4 zeigt für den Algorithmus *block* eine lineare Abhängigkeit. Im Gegensatz hierzu ist bei den anderen Varianten kein merklicher Einfluss sichtbar. Zudem zeigt sich, dass das Zusammenfassen aller Statements bezüglich der eingefügten Daten zeitlich äquivalent zu einer Lösung mittels `LOAD-INTO` ist. Dies wird auch die vorhergehenden Messreihen bestätigt, die ergaben, dass *1_statement* und *load_1_statement* nahezu gleiche Messwerte liefern.

Des Weiteren ist ersichtlich, das *block_1_statement* zeitlich identisch zu *1_statement* sowie *load_del_1_statement* ist. Dies erklärt sich bei genauerer Betrachtung der generierten Daten: Bei der verwendeten Datenbank lagen die einzufügenden Daten nahe beieinander, wodurch sich relativ große Blöcke ergaben. Hierdurch unterscheiden sich in diesem Fall *1_statement* und *block_1_statement* kaum.

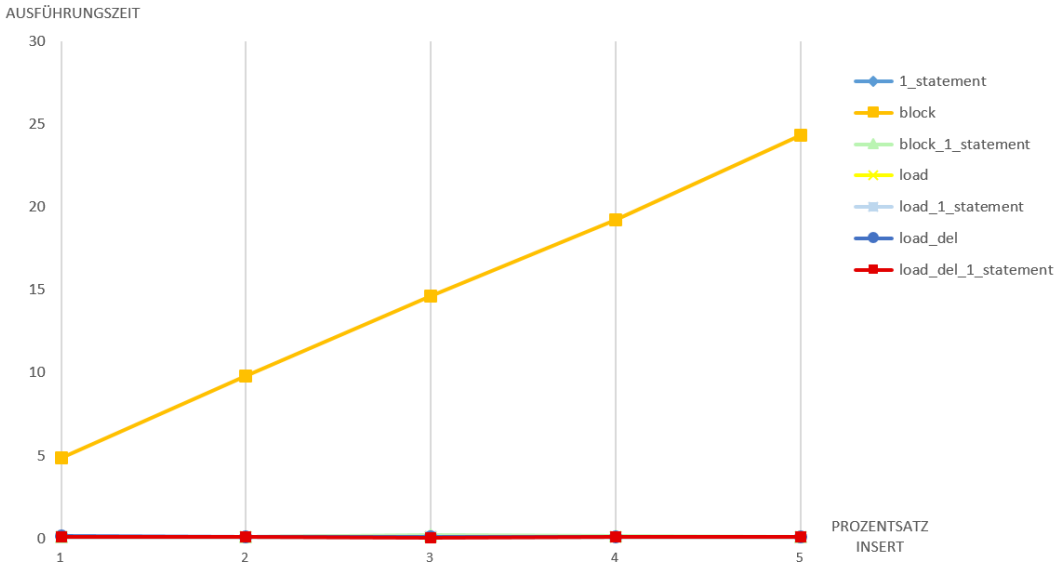

Abbildung 7.4: Laufzeitdiagramm der verschiedenen Varianten von csv2sql in Abhängigkeit des Prozentsatzes der eingefügten Datensätze

7.5.5 Prozentsatz gelöschte Datensätze

In dieser Messreihe wird die Anzahl der zu löschenden Daten variiert. Dabei wurde wieder ein Bereich von 1% bis 5% gewählt. Die Basis-Parameter sind wie in Abschnitt 7.5.4, nur wurden nun `INSERT` und `UPDATE` auf 0% gesetzt.

Hierbei verhalten sich folgende Varianten identisch: *1_statement*, *load_1_statement*, *load_del_1_statement*, da diese die `DELETE`-Blöcke jeweils zu einem Statement zusammenfassen.

Weiterhin zeigen *bock*, *load* und *load_del* ein gleiches Verhalten (ausgenommen die Abweichung bei 5%, die auf Messschwankungen zurückzuführen ist), da diese für jede `DELETE`-Manipulation ein Einzel-Statement bilden. Daneben existiert noch der Algorithmus *block_1_statement*, der pro tablediff Block die `DELETE`-Statements zusammenfasst. Somit müssen die drei Varianten *load_del_1_statement*, *block_1_statement* und *block* differenziert werden.

Betrachtet man das Diagramm aus Abbildung 7.5 so zeigt sich ein deutlicher Unterschied zwischen *block* und *load_del_1_statement*. *block* wächst im Vergleich zu *load_del_1_statement* linear. *load_del_1_statement* hingegen weist kaum eine signifikante Abhängigkeit auf. *block_1_statement* zeigt ein ähnliches Verhalten wie *block*. Dies liegt darin begründet, dass aufgrund der Blockgröße kaum Statements zusammengefasst werden können und somit vorwiegend Einzelstatements generiert werden.

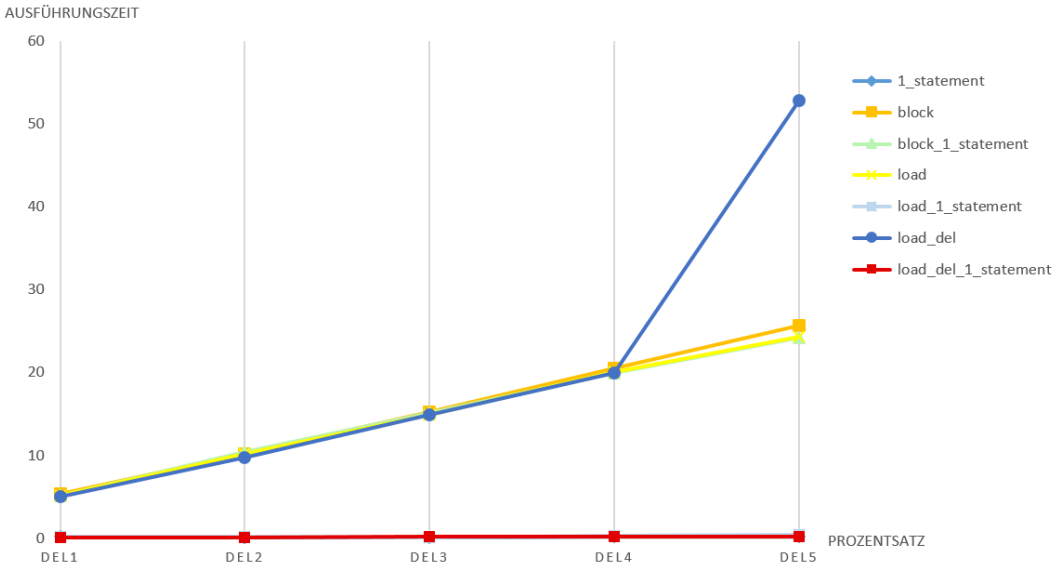

Abbildung 7.5: Laufzeitdiagramm der verschiedenen Varianten von csv2sql in Abhängigkeit des Prozentsatzes der gelöschten Datensätze

7.5.6 Prozentsatz geänderte Datensätze

Abschließend erfolgt eine Untersuchung der Abhängigkeit der Laufzeit von der Anzahl der geänderten Daten. Bei dieser Messreihe finden außer der Änderung von existierenden Daten keine weiteren Manipulationen statt. Somit werden vom Algorithmus nur `UPDATE`-Statements generiert. Wieder wurden Änderungen von 1% bis 5% vorgenommen. Die restlichen Basis-Parameter sind identisch zu denen der eingefügten Datensätze in Abschnitt 7.5.4.

Da `UPDATE`-Statements nicht zusammengefasst werden können, existiert hier kein Unterschied zwischen den Varianten. Nur *load_del* und *load_del_1_statement* bilden eine Ausnahme. Bei diesen werden die Daten nicht mittels `UPDATE`-Statements geändert, sondern die zu ändernden Daten werden zuerst gelöscht und dann mittels `LOAD-INTO` wieder hinzugefügt. Bei *load_del* werden hierbei für das Löschen Einzelstatements gebildet, bei *load_del_1_statement* werden diese zu einem kombinierten Statement zusammengefasst.

Der Abbildung 7.6 ist zu entnehmen, das *load_del_1_statement* gegenüber den anderen Algorithmen zeitlich klar gewinnt. Werden die `DELETE`-Statements jedoch nicht zu einem einzelnen Statement zusammenfasst, so ergibt sich kein nennenswerter Unterschied zwischen einer Lösung mittels `LOAD-INTO` im Vergleich zu einer Lösung mittels `UPDATE`.[3]

[3]Die auffällige Abweichung von *load* bei 5% ist hierbei auf Messschwankungen zurückzuführen.

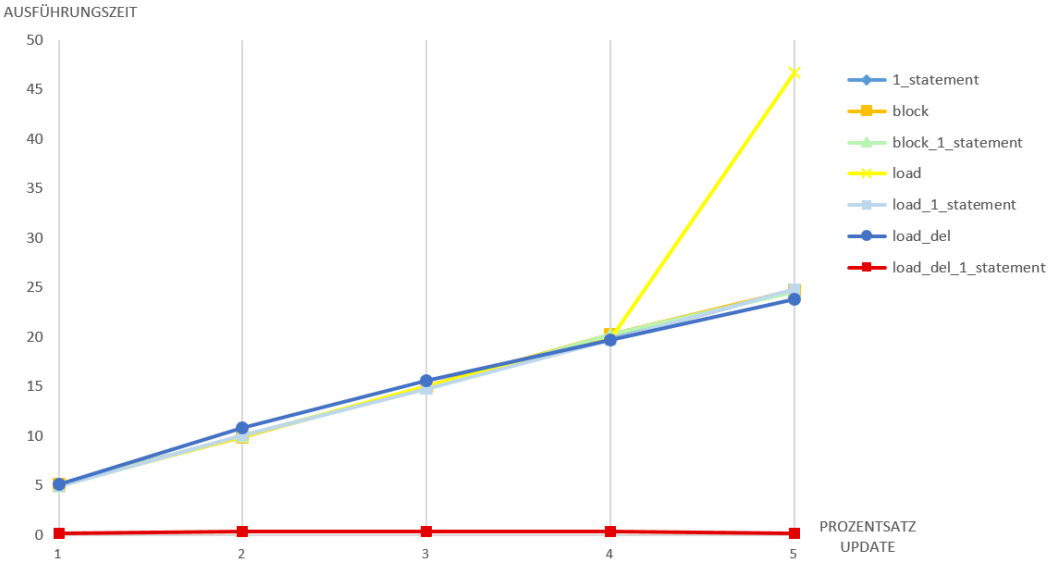

Abbildung 7.6: Laufzeitdiagramm der verschiedenen Varianten von csv2sql in Abhängigkeit der Prozentsatzes der geänderten Datensätze

7.6 Zusammenfassung

In diesem Kapitel erfolgte eine Evaluation der verschiedenen Algorithmen mit dem Ziel, den besten für die jeweiligen Ausgangsbedingung zu finden, sowie den Einfluss der einzelnen Parameter auf die Laufzeit zu untersuchen.

Hierbei wurden zunächst die getesteten PROLOG-Varianten vorgestellt. Diese unterscheiden sich zum einen darin, ob Einzelstatements oder kombinierte Statements gebildet werden. Zudem verwenden einige Varianten für das Einfügen von Daten statt eines `INSERT`-Statements ein `LOAD-INTO`. Zwei Varianten verwenden dies kombiniert mit einem vorherigen Löschen den geänderten Daten auch als eine Alternative für `UPDATE`.

Abschließend wurde der Einfluss der einzelnen Datenbankparameter sowie der Manipulationen auf die Laufzeit untersucht, um anhand dessen einen optimalen Algorithmus zu finden. Hierbei zeichnete sich der Algorithmus *load_del_1_statement* als klarer Sieger ab. Dieser verwendet anstelle von `INSERT`- und `UPDATE`-Statements eine Lösung mittels `LOAD-INTO`. Hierzu werden zunächst für die geänderten Daten `DELETE`-Statements generiert. Anschließend werden diese zusammen mit den einzufügenden Daten in ein CSV-File zwischengespeichert, welches später in MYSQL mittels `LOAD-INTO` geladen wird. Der zeitliche Vorteil ergibt sich maßgeblich aus der Zusammenfassung der Statements. Da bei dieser Variante alle Statements (auch indirekt die `UPDATE`'s) zusammengefasst werden, ist dies der schnellste Algorithmus.

Eine weitere Erkenntnis ist, dass es keinen nennenswerten Unterschied zwischen einer Lösung mittels `LOAD-INTO` und einer mittels einen einzigen `INSERT` gibt.

Des Weiteren zeigten die Messungen, das erwartungsgemäß die Datenbankgröße der erheblichste Einflussfaktor ist. Feldanzahl sowie Primärschlüsselverhältnis fallen nicht ins Gewicht. Zudem zeigte sich, dass die Anzahl der Manipulationen bei kombinierten Statements keine nennenswerte Rolle spielt. Bei Einzelstatements steigt diese hingegen linear.

8 Zusammenfassung und Ausblick

Dieses Kapitel gibt noch einmal einen Überblick über die Arbeit und fasst die Ergebnisse zusammen. Abschließend werden die Ergebnisse dieser Arbeit diskutiert und darauf basierend ein Fazit gezogen. Zudem wird ein Ausblick auf weitere mögliche Analysen und Entwicklungsmöglichkeiten gegeben.

8.1 Zusammenfassung

Diese Arbeit befasst sich mit der zeitoptimierten Synchronisation von Datenbanken. Dies geschieht unter Verwendung des Datenbanksystems MySQL, der logischen Programmiersprachen SWI-Prolog sowie Shell-Skripten.

Hierzu wurden in Kapitel 2 zunächst die grundlegenden SQL-Statements vorgestellt. Dabei lag der Schwerpunkt vor allem auf den für die Manipulation einer Datenbank nötigen Statements: `INSERT`, `DELETE` und `UPDATE`. Diese wurden am Beispiel einer Lagerortverwaltung näher erläutert, welches auch in der weiteren Arbeit verwendet wurde. Ferner wurde das für den Austausch der Daten verwendete Datenformat CSV vorgestellt und ein kurzer Einblick in die logische Programmiersprache Prolog gegeben.

Nachdem die grundlegenden SQL-Statements für die Manipulation einer Datenbank vorgestellt wurden, erfolgte in Kapitel 3 eine nähere Betrachtung dieser Datenbankbefehle. Hierbei lag der Fokus auf deren möglichst zeiteffizienten Einsatz. Um diesen zu verdeutlichen, wurde die Funktionsweise der Statements näher erläutert. Dabei zeigte sich, dass kombinierte Statements den Einzelstatements vorzuziehen sind. Zudem sollten innerhalb des Statements keine unnötigen Felder angegeben werden. Wurde beispielsweise bei einem Datensatz nur ein Feld geändert, sollte nur dieses im `SET`-Teil des `UPDATE`-Statements genannt werden. Soll ein Statement nur einen Datensatz manipulieren, sollte dies mittels des Zusatzes `LIMIT 1` auch angegeben werden.

Diese Erkenntnisse wurden dann in Kapitel 4 eingesetzt, um das Prolog-Programm diff2sql zu konzipieren, welches ausgehend von einer Patch-Datei die zur Synchronisation zweier Datenbanken nötigen SQL-Statements berechnet und generiert. Hier-

bei wurde auch die Shell-Skriptesammlung tablediff vorgestellt, die die benötigte Patch-Datei erzeugt. Nach der Erläuterung des Grundalgorithmus wurden darauf aufbauend einige Variationen des selbigen beschrieben. Diese unterscheiden sich zum einen darin, ob Einzelstatements oder kombinierte Statements gebildet werden. Zudem verwenden einige Varianten für das Einfügen von Daten statt eines `INSERT`- ein `LOAD-INTO`-Statement. Zwei Varianten verwenden dies kombiniert mit einem vorherigen Löschen der geänderten Daten auch als eine Alternative für `UPDATE`. Des Weiteren wurden einige Schwierigkeiten, die während des Entwicklungsprozesses auftraten, sowie deren Lösung beschrieben.

Um dem Endnutzer eine möglichst komfortable Lösung zu bieten, wurden das Prolog-Programm diff2sql sowie die Shell-Skriptesammlung tablediff im Skript csv2sql zusammengefasst, das in Kapitel 5 vorgestellt wurde. Dieses wendet zudem die generierten Statements auf die Datenbank an und synchronisiert diese somit.

In Kapitel 6 wurde der Java-basierte CSV-Generator vorgestellt. Dieser ermöglicht die Generierung von zufälligen CSV-Dateien entsprechend der gewünschten Schlüsselpositionen, sowie der Zeilen- und Spaltenzahl. Zudem kann der Benutzer über die Eingabe eines Prozentsatzes die Anzahl der einzufügenden, zu löschenden bzw. zu ändernden Datensätze der manipulierten CSV-Datei bestimmen. Dies erlaubt eine exakte Untersuchung jedes Einflussfaktors auf die Laufzeit im Rahmen der Evaluation.

Mit diesen generierten Messdaten erfolgt abschließend in Kapitel 7 die Evaluation der einzelnen Algorithmen. Diese verfolgt hierbei zwei Ziele: Zum einen soll der Einflussfaktor der verschiedenen Parameter einer Datenbank sowie der Anzahl der Manipulationen auf die Laufzeit analysiert werden. Basierend darauf soll anschließend der beste Algorithmus in Abhängigkeit dieser Parameter bestimmt werden.

Hierbei zeichnete sich der Algorithmus *load_del_1_statement* als klarer Sieger ab. Dieser verwendet anstelle von `INSERT`- und `UPDATE`-Statements eine Lösung mittels `LOAD-INTO`. Hierzu werden zunächst die zu änderden Daten gelöscht. Anschließend werden diese zusammen mit den einzufügenden Daten in eine CSV-Datei zwischengespeichert und danach mittels `LOAD-INTO` geladen. Der zeitliche Vorteil ergibt sich maßgeblich aus der Zusammenfassung der Statements. Da bei dieser Variante alle Statements (auch indirekt die `UPDATE`'s) zusammengefasst werden, ist dies der schnellste Algorithmus. Eine weitere Erkenntnis ist, dass kein nennenswerter Unterschied zwischen einer Lösung mittels `LOAD-INTO` und einer mittels eines einzigen `INSERT`-Statements existiert.

Des Weiteren ergaben die Messungen, dass die Datenbankgröße der erheblichste Einflussfaktor ist und die Feldanzahl sowie das Primärschlüsselverhältnis keine Rolle

spielen. Zudem zeigte sich, dass die Anzahl der Manipulationen bei kombinierten Statements keinen nennenswerten Einfluss haben. Bei Einzelstatements steigt die Laufzeit mit zunehmender Anzahl an Manipulationen hingegen linear.

8.2 Ausblick

Neben den in dieser Arbeit verfolgten Ansätzen können noch folgende Maßnahmen und Messungen durchgeführt werden:

- **Ausbau von 1_statement**
  Das Ergebnis der Messungen zeigt, dass *load_del_1_statement* stets der schnellste Algorithmus war. Bei diesem werden die geänderten Datensätze nicht mittels `UPDATE`-Statements aktualisiert, sondern gelöscht und mittels `LOAD-INTO` neu hinzugefügt. Werden hingegen `UPDATE`-Statements verwendet, so zeigt sich kein signifikanter Unterschied zwischen einem einzelnen kombinierten `INSERT`-Statement und einer Lösung mittels `LOAD-INTO`. Daher wäre eine Erweiterung von *1_statement*, die auch auf `UPDATE` verzichtet, in Betracht zu ziehen.

- **Reine Prolog-Lösung**
  In der vorliegenden Masterarbeit wurde eine Vorverarbeitung der Daten mittels der Shell-Skriptesammlung tablediff vorgenommen. Dieses gibt ein Patch-File zurück, dass dann als Textdatei eingelesen wurde. Aufgrund der vielseitigen Einsetzbarkeit, umfangreichen Bibliotheken, sowie der Fähigkeit von Prolog mit großen Datenmengen umzugehen, wäre eine reine Prolog-Lösung erstrebenswert. Diese hätte zudem den Vorteil, dass nicht zwangsläufig Linux-Kommandos verwendet werden müssten.

- **Untersuchung weiterer Datenbank-Einflussfaktoren**
  Im Rahmen der Messungen ergab sich, dass die Anzahl der Primärschlüssel keinen Einfluss auf die Laufzeit von csv2sql hat. Der Einfluss der Position der Primärschlüssel auf die Laufzeit wurde hierbei noch nicht betrachtet. Dies könnte in weiteren Messungen untersucht werden. Ebenso kann der Grad der Permutation zwischen der Ausgangs-CSV-Datei sowie der manipulierten näher betrachtet werden. Die Generierung dieser Datenbanken unterstützt der CSV-Generator bereits jetzt.

- **Untersuchen der System-Einflussfaktoren**
  Neben den Einflussfaktoren der Datenbank auf die Laufzeit könnten auch noch systemabhängige Faktoren wie CPU und Arbeitsspeicher analysiert werden. Hierbei ist vor allem der Einfluss des Arbeitsspeicher interessant, da dieser eventuell die Laufzeit von *1_statement*-Algorithmen beeinflussen könnte.

- **Messungen auf Servern**
  Im Rahmen dieser Arbeit wurde für die Messungen bewusst ein langsameres System gewählt. In der Praxis verfügen Firmen, Behörden, etc. jedoch meist über Server. Daher wäre eine Messung auf einem Server ein weiterer interessanter Punkt für eine Untersuchung.

- **Übertragbarkeit auf andere SQL-Varianten**
  Des Weiteren wurde durch die bewusste Verwendung von Wildcards bei der Generierung der SQL-Statements in PROLOG sichergestellt, dass sich eine Portierung auf andere SQL-Systeme einfach realisieren lässt. Daher müssen nur diese Format-Strings geändert werden, um weitere Messungen diesbezüglich durchführen zu können.

Literaturverzeichnis

[BJK11] C. Brücher, F. Jüdes, and W. Kollmann. *SQL Thinking - Vom Problem zum SQL-Statement*. mitp, 2011. 1.1, 2.1.1

[Bra01] I. Bratko. *Prolog (3rd Ed.): Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 2.3

[CB74] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM. 2.1.1

[Cen18] Oracle Help Center. *Database Performance Tuning Guide*. https://docs.oracle.com/cd/B19306_01/server.102/b14211/perf_overview.htm, 2018. 1.2.1

[CLS00] M. Chan, H. V. Leong, and A. Si. Incremental update to aggregated information for data warehouses over internet. In *Proceedings of the 3rd ACM International Workshop on Data Warehousing and OLAP*, DOLAP '00, pages 57–64, New York, NY, USA, 2000. ACM. 1.2.2

[CM03] W. F. Clocksin and C. S. Mellish. *Programming in Prolog: [using the ISO standard]*. Springer-Verlag, 5th ed edition, 2003. 2.3

[Coh04] J. Cohen. A tribute to alain colmerauer. *CoRR*, cs.PL/0402058, 2004. 2.3.1

[CR96] A. Colmerauer and P. Roussel. History of programming languages—ii. chapter The Birth of Prolog, pages 331–367. ACM, New York, NY, USA, 1996. 2.3.1

[DBCK98] F. Douglis, T. Ball, Y.F. Chen, and E. Koutsofios. *The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web*, volume 1, pages 27–44. 03 1998. 1.2.2

[Eib14] J. Eibl. Kdiff3. http://kdiff3.sourceforge.net/, 2014. 1.2.1

[EN09] R. A. Elmasri and S. B. Navathe. *Grundlagen von Datenbanksystemen*, volume 3. Pearson Studium, 2009. 2.1

[hal18] *SWI-Prolog - User Top-level Manipulation*. http://www.swi-prolog.org/pldoc/man?section=toplevel, 2018. 5.3

[Han13] M. Hanus. *Problemlösen mit PROLOG*. MikroComputer-Praxis. Vieweg+Teubner Verlag, 2013. 2.3.1

[HM76] J. W. Hunt and M. D. MacIlroy. An algorithm for differential file comparison. Bell Laboratories, 1976. 1.2.1, 1.2.2

[HV01] M.J. Hernandez and J. Viescas. *Go to SQL*. Addison Wesley, 2001. 2.1.1

[ini18] *SWI-Prolog - Predicate initialization/1*. http://www.swi-prolog.org/pldoc/doc_for?object=(initialization)/1, 2018. 5.3

[LM07] D. Louis and P. Müller. *Das Java 6 codebook*. Premium codebook. Addison-Wesley, 2007. 2.2.1

[Mul02] C. Mullins. Coding db2 sql for performance: The basics. http://www.ibm.com/developerworks/data/library/techarticle/0210mullins/0210mullins.html, 2002. 1.2.1

[MyS18a] MySQL. *MySQL 5. Referenzhandbuch - Geschwindigkeit von INSERT-Anweisungen*. http://download.nust.na/pub6/mysql/doc/refman/5.1/de/insert-speed.html, 2018. 3.1

[MyS18b] MySQL. *MySQL 5.7 Reference Manual - Data Types*. https://dev.mysql.com/doc/refman/5.7/en/data-types.html, 2018. 2.1.2

[MyS18c] MySQL. *MySQL 5.7 Reference Manual - DELETE Syntax*. https://dev.mysql.com/doc/refman/5.7/en/delete.html, 2018. 3.2

[MyS18d] MySQL. *MySQL 5.7 Reference Manual - Optimizing INSERT Statements*. https://dev.mysql.com/doc/refman/5.7/en/insert-optimization.html, 2018. 3.1

[MyS18e] MySQL. *MySQL 5.7 Reference Manual - SQL Statement Syntax*. https://dev.mysql.com/doc/refman/5.7/en/sql-syntax.html, 2018. 1.2.1, 2.1, 3.2

[MyS18f] MySQL. *MySQL 5.7 Reference Manual - String Literals*. https://dev.mysql.com/doc/refman/5.7/en/string-literals.html, 2018. 2.1.5

[MyS18g] MySQL. *MySQL 5.7 Reference Manual - UPDATE Syntax*. https://dev.mysql.com/doc/refman/5.7/en/update.html, 2018. 3.3

[MyS18h] MySQLTutorial. Mysql export table to csv. http://www.mysqltutorial.org/mysql-export-table-to-csv/, 2018. 2.1.3

[Nog18] F. Nogatz. tabeldiff. https://github.com/fnogatz/tablediff, 2018. 1.2.1, 4.2, 4.2.1

[OFS14] L. Ostermayer, F. Flederer, and D. Seipel. Capja- a connector architecture for prolog and java. In *Proceedings of the 10th International Conference on Knowledge Engineering and Software Engineering - Volume 1289*, KESE'14, pages 59–70, Aachen, Germany, Germany, 2014. CEUR-WS.org. 2.3.1

[onl18] Encyclopædia Britannica online. Stichwort: Sql. https://www.britannica.com/technology/SQL, 2018. 1.1, 2.1.1

[Pro18] *SWI-Prolog - For running the result*. http://www.swi-prolog.org/pldoc/man?section=runcomp, 2018. 5.3, 5.3

[Ray03] E.S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional Computing Series. Pearson Education, 2003. 2.2.2

[Sch07] B. Schwartz. An algorithm to find and resolve data differences between mysql tables. http://www.xaprb.com/blog/2007/03/05/an-algorithm-to-find-and-resolve-data-differences-between-mysql-tables/, 2007. 1.2.1

[Sei11] D. Seipel. *Scriptum - Datenbanken*. Universität Würzburg, 2011. 2.1

[Sei15] D. Seipel. *Scriptum - Deduktive Datenbanken und Logikprogrammierung*. Universität Würzburg, 2015. 2.3

[Sha05] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, October 2005. 2.2.2

[SNA17] D. Seipel, F. Nogatz, and S. Abreu. Prolog for expert knowledge using domain-specific and controlled natural languages. pages 138–140, 2017. 2.3.1

[Spi07] A. Spiers. mysqldiff – a utility for comparing mysql database structures. https://github.com/aspiers/mysqldiff, 2007. 1.2.1, 1.2.2

[SS04] W. Schorn and J. Schorn. *Scriptprogrammierung für Solaris & Linux: Systemverwaltung und Automatisierung mit nawk, Korn-/Bourne-Shell und Perl*. open source library. Addison Wesley Verlag, 2004. 5.1

[swi18a] *SWI-Prolog*. http://www.swi-prolog.org/, 2018. 2.3.1

[swi18b] *SWI-Prolog - library(csv): Process CSV (Comma-Separated Values) data.* http://www.swi-prolog.org/, 2018. 2.3.6, 2.3.6

[swi18c] *SWI-Prolog - library(lists): List Manipulation.* http://www.swi-prolog.org/pldoc/man?section=lists, 2018. 2.3.3

[swi18d] *SWI-Prolog - Predicate open/4.* http://www.swi-prolog.org/pldoc/doc_for?object=open/4, 2018. 2.3.5

[w3s18] w3schools. *SQL Wildcards.* https://www.w3schools.com/sql/sql_wildcards.asp, 2018. 2.1.5

[WDC03] Y. Wang, D.J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *Proceedings - International Conference on Data Engineering*, pages 519– 530, 04 2003. 1.2.2

[Wie18] J. Wielemaker. *SWI-Prolog ODBC Interface*, 2018. 2.3.1, 2.3.6

[Win12] M. Winand. *SQL Performance Explained.* 2012. 1.2.1

[Wol10] J. Wolf. *Shell-Programmierung.* Galileo Press. Galileo Press, 2010. 5.1

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. 2.3.1

[Yan91] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, June 1991. 1.2.2

[ZTZ+09] P. Zaitsev, V. Tkachenko, J.D. Zawodny, A. Lentz, and D.J. Balling. *High Performance MySQL.* O'Reilly Verlag, 2009. 7.2, 7.5, 7.5.1

Erklärung

Ich, Sandra Lederer, Matrikel-Nr. 1740405, versichere hiermit, dass ich meine Masterarbeit mit dem Thema

> *Analyse und Optimierung verschiedener Algorithmen zur Synchronisation von SQL-Datenbanken*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Masterarbeitzusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Universität abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Würzburg, den 12.6.2018

Sandra Lederer