FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Masterarbeit

Entwurf, Implementierung und Analyse einer datenbankgestützten Datenverwaltung in der internen Ablaufsteuerung des Workflow-Tools JUBE

Fachbereich Medizintechnik und Technomathematik
im Studiengang Angewandte Mathematik und Informatik

Jülich, den 10. August 2023

Autor: Julia Wellmann (Matr. Nr.: 3209301)
1. Prüfer: Prof. Dr. rer. nat. Philipp Rohde
2. Prüfer: Thomas Breuer

Firma: Forschungszentrum Jülich GmbH
Institut: Jülich Supercomputing Centre

JÜLICH
Forschungszentrum

Eidesstattliche Erklärung

Hiermit versichere ich, Julia Wellmann, dass ich die Masterarbeit mit dem Thema

Entwurf, Implementierung und Analyse einer datenbankgestützten Datenverwaltung in der internen Ablaufsteuerung des Workflow-Tools JUBE

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Masterarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Jülich, den 10. August 2023

Ort und Datum

Unterschrift des Autors

Vorwort

Die vorliegende Masterarbeit beschäftigt sich mit der Implementierung und Einarbeitung einer datenbankgestützten Datenhaltung in dem Workflow-Tool JUBE. Diese Arbeit habe ich als Abschlussarbeit meines Studiums der angewandten Mathematik und Informatik M.Sc. an der Fachhochschule Aachen verfasst. Ziel dieser Arbeit war es die aktuelle Datenhaltung zu analysieren, die Anforderungen an die neue Datenhaltung herauszuarbeiten und den Entwurf zur Implementierung dieser neuen Datenhaltung vorzustellen.

Im Rahmen meiner Tätigkeit im Jülich Supercomputing Centre des Forschungszentrum Jülich entwickelte ich, zusammen mit meinen Betreuern Thomas Breuer und Wolfgang Frings, die Aufgabenstellung für diese Abschlussarbeit, basierend auf einer ebenfalls gemeinsam erstellten Fragestellung meiner Bachelorarbeit ***Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien anhand von verschiedenen Benchmarks auf HPC Systemen***[29]. In dieser Arbeit wurden unter anderem die Grundfunktionen und die Beschreibung der Eingabedatei des Workflow-Tools JUBE herausgearbeitet. In der vorliegenden Abschlussarbeit sind diese Thematiken mit eingegangen, sodass in einzelnen Kapiteln dieser Arbeit auf die Bachelorarbeit verwiesen wird.

Julia Wellmann
Jülich, 10. August 2023

Abstract

Die Speicherung von Informationen spielt unter anderem bei der Automatisierung von Arbeitsabläufen und der Analyse von Ergebnissen eine entscheidende Rolle. Dabei erfordert die Komplexität und Vielfalt dieser Informationen eine effiziente und zuverlässige Datenhaltung, um den reibungslosen Betrieb eines Systems zu gewährleisten.

Das Workflow-Tool JUBE verwendet eine solche Datenhaltung in Form von XML-Dateien, um Workflow-Konfigurationen für eine spätere Fortsetzung oder Reproduktion zu speichern. Ziel der vorliegenden Masterarbeit ist es, einen Entwurf für die Ablösung der XML-basierten Datenhaltung des Workflow-Tools JUBE zu erarbeiten, der die Probleme der derzeitigen Datenhaltung adressiert. Dazu wird folgende Forschungsfrage gestellt: Welche Datenhaltungsoption bietet eine konsistentere, persistentere und performantere Datenhaltung bei gleichzeitiger Reduzierung des Speicherbedarfs und Verbesserung der Leistung als die derzeitige XML-Datenhaltung und wie kann diese in JUBE implementiert werden?

Um die Probleme der derzeitigen XML-Datenhaltung zu identifizieren, wird diese im aktuellen Softwarezustand von JUBE analysiert, wobei sich herausstellt, dass die Probleme vor allem in der Datenredundanz und der nicht optimalen Aktualisierungsfunktion liegen. Darauf aufbauend werden Anforderungen an die neue Datenhaltung definiert, die diese Probleme vor allem durch eine neue Strukturierung der Daten und ein geeignetes Datenmanagementsystem beheben sollen. Neben diesen funktionalen Anforderungen werden auch qualitative Anforderungen wie Leistungsfähigkeit und Wartbarkeit definiert, die ebenfalls die Nachteile der XML-Datenhaltung beheben sollen.

Auf der Grundlage der zuvor identifizierten Anforderungen wird die Verwendung einer relationalen Datenbank mit dem Datenbankmanagementsystem SQLite3 ausgewählt und auf dieser Basis ein Implementierungsentwurf entwickelt. Der erarbeitete Entwurf für die Datenhaltung, die Schnittstelle und das Konzept für die Implementierung in JUBE wird prototypisch implementiert, so dass der Entwurf anhand dieses Prototyps evaluiert werden kann. Die Evaluierung ergab, dass der Entwurf dieser Arbeit einen vielversprechenden Ansatz zur Lösung der identifizierten Probleme darstellt, der die Hauptprobleme löst, dass aber noch einige Punkte für den Einsatz in Produktionsumgebungen berücksichtigt werden können, um die Datenverwaltung weiter zu verbessern.

Inhaltsverzeichnis

1. Motivation

Die Speicherung von Informationen in komplexen Anwendungsprogrammen durch eine Datenhaltung hat in den letzten Jahren stark an Bedeutung gewonnen. Die dadurch ermöglichte Weiterverarbeitung von Daten erlaubt es Anwendungen, ihre Daten über die Laufzeit hinaus zu nutzen, z.B. zur Datenanalyse oder zur Ausgabe von Ergebnissen. Für eine effektive und effiziente Datenverwaltung werden Datenmanagementsysteme benötigt, die es ermöglichen, die Daten zu erfassen, zu speichern, zu organisieren, zu aktualisieren und abzurufen.

Das Jülich Supercomputing Centre (JSC) des Forschungszentrum Jülich (FZJ) entwickelt aktiv an dem Workflow-Tool JUBE (Jülich Benchmarking Environment), das eine Datenhaltung verwendet, um Workflow-Konfigurationen für eine spätere Fortsetzung oder Reproduktion zu speichern. Dazu verwendet JUBE derzeit eine textbasierte Datenhaltung im XML-Format, die mit Hilfe des *ElementTree*-Python-Moduls verwaltet wird. Da diese Datenhaltung einige Nachteile mit sich bringt, wie z.B. die fehlende Konsistenzprüfung oder die erhöhte Laufzeit aufgrund fehlender Aktualisierungsfunktionen, liegt der Schwerpunkt dieser Masterarbeit auf der Frage, wie diese Datenhaltung durch eine schnellere, zuverlässigere und speicherorientiertere Datenhaltung ersetzt werden kann.

Um dieses Problem der aktuellen Datenhaltung anzugehen, wird ein systematischer Ansatz gewählt, der sich aus folgenden Punkten zusammensetzt: Der Analyse des Ist-Zustandes der Datenhaltung in JUBE, der Formulierung der Ziele der neuen Datenhaltung, dem Entwurf eines Implementierungskonzeptes und der abschließenden Bewertung dieses Entwurfes. Der Schwerpunkt liegt dabei auf dem Konzeptentwurf. Die Realisierung einer vollständigen, produktionsreifen Implementierung wird in dieser Arbeit durch eine prototypische Implementierung vorbereitet, anhand derer die Evaluierung durchgeführt wird.

2. Grundlagen

Um eine Argumentationsbasis für die weitere Arbeit zu schaffen werden in diesem Kapitel zunächst die Grundlagen der für diese Arbeit relevanten Themen und Softwarepakete dargestellt.

2.1. Workflow-Tool JUBE

Ein Workflow ist eine definierte Abfolge von Arbeitsschritten, die erforderlich sind, um ein bestimmtes Ziel zu erreichen. Beispielsweise kann ein Workflow die Konfiguration, Kompilierung und Ausführung eines Programms auf verschiedenen Rechnerarchitekturen sowie die Überprüfung und Analyse der Ergebnisse umfassen. In einem anderen Einsatzbereich kann ein Workflow auch die Ausführung einer Aneinanderreihung von mehreren Simulationsprogrammen und Skripten beschreiben, die z.B. dazu dienen eine umfangreiche Parameterstudie durchzuführen. Beispiele für diese Art von Workflows sind unter anderem unter [22] und [5] veröffentlicht. Die manuelle Ausführung eines solchen Workflows mit unterschiedlichen Parameterkombinationen und der Vergleich der Ergebnisse kann einen hohen Verwaltungsaufwand und eine hohe Fehleranfälligkeit mit sich bringen. [28]

Ein Workflow-Tool bietet dem Benutzer die Möglichkeit, Workflows zu konfigurieren, auszuführen, zu verwalten und zu optimieren. Darüber hinaus können Abhängigkeiten zwischen den einzelnen Arbeitsschritten eines Workflows definiert werden, so dass ein Workflow-Tool die Schritte automatisch in der richtigen Reihenfolge und unter Berücksichtigung der vorherigen Ergebnisse sequentiell oder parallel ausführen kann. Dies erleichtert die Koordination und Verwaltung komplexer Arbeitsschritte. Außerdem bietet es die Möglichkeit, den Fortschritt der einzelnen Arbeitsschritte zu überwachen. [28]

Der Einsatz eines Workflow-Tools ermöglicht es daher, einen Workflow hinsichtlich verschiedener Fragestellungen automatisiert auszuführen und die erzielten Ergebnisse zu vergleichen. Beispielsweise kann die Verteilung der Prozesse auf die Prozessorkerne des Rechners, das so genannte Prozess-Pinning, anhand verschiedener Parameterkombinationen konfiguriert und somit die Leistung eines Programms in Abhängigkeit vom Pinning ermittelt werden. Darüber hinaus ermöglicht es die Reproduzierbarkeit der Workflows, da die Konfigurationen, mit denen die konkreten Workflows ausgeführt werden, gespeichert und damit dokumentiert werden. Durch die feste Struktur, nach der die Workflows innerhalb des Workflow-Tools definiert werden, wird zudem die Anpassung der Workflows und damit die Übertragbarkeit und Wiederverwendbarkeit auf andere Anwendungsfälle vereinfacht. [29]

Die Software JUBE ist ein solches Workflow-Tool, das am JSC des FZJ entwickelt wurde und aktiv weiterentwickelt wird. Es bietet die Möglichkeit, benutzerdefinierte Abläufe flexibel zu konfigurieren, diese in definierter Reihenfolge auszuführen, sowie die Ergebnisse zu analysieren und in verschiedenen Formaten auszugeben. Das Haupteinsatzgebiet von JUBE sind High-Performance Computing (HPC) Systeme, wie z.B. das am JSC installierte JUWELS-System[1], auf dem es bereits als Kommandozeilenprogramm vorinstalliert ist. Die Ausführung von JUBE ist jedoch nicht auf HPC-

[1] https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/juwels

Systeme beschränkt, sondern kann auch auf anderen Linux-Systemen wie z.B. einfachen Laptops erfolgen, da die einzige Abhängigkeit die Python-Standardbibliothek ist. [13] [22, S. 2877-2878] [29, S. 15]

Zur Nutzung des Workflow-Tools JUBE muss der Anwender eine Eingabedatei mit den gewünschten Konfigurationen, Schritten, Analysen und Ergebnisformaten erstellen. Diese wird von JUBE eingelesen und weiterverarbeitet. Mit Hilfe verschiedener Kommandozeilenoptionen können z.B. die in der Eingabedatei definierten Workflows ausgeführt oder die daraus resultierenden Ergebnisse angezeigt werden. [12] [13] [29, S. 15]

Um alle Daten eines Workflows für eine Fortsetzung oder eine spätere Reproduktion zu speichern, verwendet JUBE eine textbasierte Datenhaltung im XML-Format, deren genaue Struktur und Aufbau im Kapitel 3 näher erläutert wird. Diese Datenhaltung soll im Rahmen dieser Arbeit durch ein konsistentere, persistentere und performantere Datenhaltung ersetzt werden.

2.1.1. Python

Das Workflow-Tool JUBE ist in der Programmiersprache Python implementiert. Python ist eine plattformunabhängige, objektorientierte Programmiersprache, die für ihre einfache Syntax und Handhabbarkeit bekannt ist. Python verfügt über eine umfangreiche Standardbibliothek und eine große Anzahl zusätzlich installierbarer Pakete, die aus verschiedenen Modulen bestehen. Diese unterstützen viele gängige Programmieraufgaben, wie z.B. die Textsuche mit regulären Ausdrücken oder das Lesen und Verändern von Dateien. Der Python-Interpreter ist frei verfügbar. [30]

Um Daten typgerecht zu speichern, gibt es in Python, wie auch in anderen Programmiersprachen, verschiedene Datentypen. Datentypen sind spezielle Klassifizierungen, die bestimmen, wie Daten verarbeitet und gespeichert werden. Die grundlegenden Datentypen von Python für Zahlen sind *int* für ganze Zahlen, *float* für Fließkommazahlen und *complex* für komplexe Zahlen. Es gibt auch *Strings*, die eine Zeichenkette beschreiben, oder *Booleans*, die einen Wahrheitswert beschreiben. Eine Sammlung dieser Typen kann wiederum durch *Tupel*, *Listen*, *Mengen* oder ein *Dictionary* definiert werden: *Tupel* ist eine unveränderliche Sammlung von Elemente, *Listen* eine Sammlung geordneter Elemente, *Mengen* eine Sammlung ungeordneter Elemente und *Dictionaries* eine Sammlung geordneter Schlüssel-Wert-Paare. [19]

Für die Nutzung von JUBE ist mindestens die Python-Version 3.2 erforderlich. JUBE ist objektorientiert implementiert und verwendet neben anderen Python-Bibliotheken das *ElementTree* Modul des *xml*-Pakets[1], das eine einfache Möglichkeit bietet, XML-Dokumente zu erstellen, zu bearbeiten und zu analysieren, indem es eine Baumstruktur für XML-Dokumente bereitstellt. Dieses Paket wird verwendet, um die JUBE-Eingabedatei zu lesen und die Konfigurationen in der aktuellen Datenhaltung im XML-Format zu speichern. [20] [21][24, S. 2-3]

2.1.2. XML

Die Definition eines Workflows in der JUBE-Eingabedatei und die Speicherung der Konfigurationen der intenern Ablaufsteuerung erfolgt im XML-Format. Die Extensible Markup Language (XML), ist eine plattformunabhängige Auszeichnungssprache, mit

[1] `https://docs.python.org/3/library/xml.etree.elementtree.html`

der hierarchische Daten in Form von Textdateien dargestellt werden können. Diese Dateien sind sowohl von Menschen als auch von Maschinen lesbar. [7] [14]

Ein XML-Dokument besteht aus einer Reihe von Elementen, die durch Verschachtelung eine hierarchische Baumstruktur bilden können. Jedes Element hat ein Start- und ein End-Tag, zwischen denen der Inhalt des Elements steht. Darüber hinaus können Elemente verschiedene Attribute besitzen. [2, S. 10-14] [8]

Diese gültigen Elemente und Attribute können nach strukturellen und inhaltlichen Vorgaben definiert werden, so dass die Struktur der Datei an beliebige Daten angepasst werden kann. Dazu werden sogenannte XML-Schemata verwendet, die diese Anforderungen definieren und somit auch eine Validierung der Struktur und des Inhalts der Daten ermöglichen. [2, S. 89-90] [26]

Innerhalb eines XML-Dokuments können Kommentare gemäß der Syntax in Listing 2.1 eingefügt werden. [27]

```
<!-- Kommentar -->
```

Listing 2.1: Kommentar-Syntax einer XML-Datei

2.1.3. Aufbau und Struktur der JUBE-Eingabedatei

Um einen Workflow und die darin enthaltenen Schritte und Konfigurationen zu definieren, erstellt der Benutzer eine Eingabedatei im XML- oder YAML-Format, die im Folgenden als (JUBE-)Eingabedatei bezeichnet wird. Inhalt, Struktur und Verwendung der Optionen der Eingabedatei werden in diesem Unterkapitel beschrieben.

JUBE unterstütz für beide Eingabeformate, XML und YAML, den gleichen Funktionsumfang. Um die weiteren Ausführungen im Kapitel 3 nachvollziehbarer zu machen, beschränkt sich dieses Kapitel auf das XML-Format der Eingabedaten, da das ursprüngliche Eingabeformat XML ist und auch bei einer YAML-Eingabedatei die interne Verarbeitung und Speicherung der Daten im XML-Format erfolgt.[1] [12] [29]

2.1.3.1. Allgemeine Struktur

Zunächst wird anhand der allgemeinen Struktur der Eingabedatei ein Überblick über die möglichen Optionen zur Definition eines Workflows gegeben. Die allgemeine Struktur der Eingabedatei sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <!-- Optional: Hinzufuegen von externen Dateien -->
  <include-path>
    <path>...</path>
    ...
  </include-path>
  <!-- Optional: Ein-/Ausschliessen von ausgewählten Benchmarks -->
  <selection>
    <only>...</only>
    <not>...</not>
    ...
  </selection>
```

[1] Namen und Attribute der einzelnen Optionen sind in beiden Formaten gleich, nur Struktur und Aufbau unterscheiden sich.

```
    <!-- Optional: Globale Sets -->
    <parameterset name="">...</parameterset>
    <substitutionset name="">...</substitutionset>
    <fileset name="">...</fileset>
    <patternset name="">...</patternset>

    <benchmark name="" outpath="">
      <comment>...</comment>
      <!-- Lokale Sets -->
      <parameterset name="">...</parameterset>
      <fileset name="">...</fileset>
      <substituteset name="">...</substituteset>
      <patternset name="">...</patternset>

      <!-- Ausfuehrungsschritt -->
      <step name="">...</step>

      <!-- Analyseschritt -->
      <analyser name="">...</analyser>

      <!-- Ergebnisausgabe-->
      <result>...</result>
    </benchmark>
    ...
</jube>
```

Listing 2.2: Allgemeine Struktur einer JUBE-Eingabedatei im XML-Format

Jede XML-basierte JUBE-Eingabedatei beginnt (nach dem allgemeinen XML-Header) mit dem Basis-Tag `<jube>`.

Innerhalb des `<jube>`-Tags können optional zunächst globale Informationen definiert werden, die für alle nachfolgend definierten Workflows gelten. Mit dem `<include-path>`-Tag können externe Dateien importiert werden und mit dem `<selection>`-Tag kann die Ausführung auf bestimmte Workflows beschränkt werden. Zusätzlich können optional verschiedene globale Sets definiert werden, die identisch zu den lokalen Sets eines Workflows sind, die im Laufe dieses Kapitels beschrieben werden.

Die Definition der Workflow-spezifischen Konfiguration beginnt mit dem *<benchmark>*-Tag. Die Wortwahl ist historisch bedingt, da JUBE ursprünglich eine Benchmarking-Umgebung war. Da jedoch die Funktionalität von JUBE im Laufe der Jahre erweitert wurde, ist JUBE nicht mehr nur auf Benchmarks beschränkt, sondern hat sich zu einem Workflow-Tool entwickelt. Der *<benchmark>*-Tag ist jedoch geblieben.

Das `outpath`-Attribut des *<benchmark>*-Tags legt das Ausführungsverzeichnis des Workflows fest, welches von JUBE automatisch erstellt wird, falls es noch nicht existiert. In diesem Verzeichnis werden alle für diesen Workflow relevanten Daten abgelegt. Der genaue Aufbau der JUBE-Ordnerstruktur wird im Kapitel 3 näher erläutert. Das `name`-Attribut deklariert den Namen des Workflows und mit dem `<comment>`-Tag können Kommentare zum Workflow definiert werden. Diese Daten können zur Unterscheidung verschiedener Workflows verwendet werden.

Zusätzlich zu den globalen Sets können innerhalb einer Workflow-Definition auch lokale Sets definiert werden, die nur für den spezifischen Workflow gültig sind. Darüber hinaus können innerhalb eines Workflows Ausführungs-, Analyse- und Ergebnisschritte definiert werden. Die Funktionalitäten der genannten globalen Informationen, lokalen

und globalen Sets sowie der Ausführungs-, Analyse- und Ergebnisschritte werden in den folgenden Unterkapiteln näher beschrieben. [12] [29]

2.1.3.2. Externe Dateien einbinden

Zur besseren Strukturierung innerhalb einer JUBE-Eingabedatei können bereits vorhandene Eingabedateien wiederverwendet werden. Es gibt mehrere Möglichkeiten, den Pfad einer wiederzuverwendenden Datei anzugeben. Die folgenden Beispiele zeigen zwei Möglichkeiten:

```
<include-path>
   <path>...</path>
</include-path>
```

Listing 2.3: Importieren von externen Dateien mit Hilfe von `<include-path>` in JUBE

```
<include from="..." path="..." />
```

Listing 2.4: Importieren von externen Dateien mit Hilfe von `<include>` in JUBE

Innerhalb des `<include-path>`-Tags, das im Listing 2.3 zu sehen ist, kann mit Hilfe des *<path>*-Tags ein Pfad angegeben werden, in dem nach zu importierenden Dateien gesucht wird. Das *<path>*-Tag, das den Pfad zur Datei angibt, kann innerhalb des `<include-path>`-Tags mehrfach verwendet werden. Eine weitere Möglichkeit zum Einbinden einer Eingabedatei bietet der im Listing A.2 dargestellte `<include>`-Tag, in dem der Benutzer mit Hilfe des *path*-Attributs den Pfad zur Eingabedatei angeben kann.

Als zusätzliche Option zur Verwendung externer Dateien kann unabhängig von der Eingabedatei in der Shell-Umgebungsvariablen `JUBE_INCLUDE_PATH` eine Liste von Pfaden zu Verzeichnissen definiert werden, die zu importierenden XML-Dateien enthalten. [12] [29]

2.1.3.3. Benchmarks selektieren

Es ist möglich, mehrere Workflows in derselben Eingabedatei zu speichern. JUBE führt die Workflows nacheinander aus und weist jedem Workflow eine aufsteigende ID zu. Um nur bestimmte Workflows zu verwenden oder einzelne Workflows auszuschließen, wird das `<selection>`-Tag verwendet:

```
<selection>
   <only>...</only>
   <not>...</not>
   <tag>...</tag>
</selection>
```

Listing 2.5: Aufbau des `<selection>`-Tags in JUBE

Workflows werden über ihren Namen an- oder abgewählt. Workflows, deren Namen im `<only>`-Tag aufgelistet sind, werden für den Lauf ausgewählt, während andere Workflows durch das `<not>`-Tag ausgeschlossen werden können. Die jeweiligen Tags können mehrfach verwendet werden oder eine durch Kommata getrennte Liste von Workflow-Namen enthalten.

Zusätzlich bietet JUBE die Möglichkeit, sogenannte Tags zu definieren. Um Verwechslungen mit XML-Tags zu vermeiden, werden diese im Folgenden als JUBE-Tags

bezeichnet. Diese JUBE-Tags können, wie im Listing 2.5 dargestellt, durch den <*tag*>-Tag angegeben werden. Wenn das JUBE-Tag angegeben wird, wird dem JUBE-Tag der Wahrheitswert *true* zugewiesen und kann in der weiteren Workflow-Konfiguration als Attribut verwendet werden. Im weiteren Verlauf dieses Kapitels werden einige Beispiele für die Verwendung des JUBE-Tags erläutert. [12] [29]

2.1.3.4. Parameterräume

Eine der Hauptfunktionen von JUBE ist die flexible und einfache Erstellung der Parameterräume. Die Parameterräume, auch Parametersets, enthalten die Definition der Parameter, die in einem Ausführungsschritt verwendet werden können.

Die Struktur eines Parametersets mit einem Beispielparameter ist im folgenden Listing dargestellt.

```
<parameterset name="..." init_with="..." duplicate="...">
  <parameter name="..." mode="..." type="..." separator="..." export="..."
      update_mode="..." duplicate="..." tag="...">...</parameter>
</parameterset>
```

Listing 2.6: Schematische Darstellung eines Parametersets in JUBE

Ein Parameterset wird durch das `<parameterset>`-Tag eingeleitet, und eine beliebige Anzahl zugehöriger Parameter wird jeweils durch das `<parameter>`-Tag eingeleitet.

Der Name des Parametersets wird durch das `name`-Attribut definiert und muss für den aktuellen Workflow eindeutig sein. Der Name eines Parameters wird ebenfalls durch das *name*-Attribut definiert. Ist der Parametername innerhalb des Sets nicht eindeutig, so wird der zuletzt vorkommende Parameter mit diesem Namen verwendet. Auf den Wert des Parameters kann an verschiedenen Stellen über `$parameter_name` zugegriffen werden.

Der Inhalt des <parameter>-Tags definiert die Parameterwerte. Es können mehrere Werte angegeben werden, indem diese durch Komma (,) oder das, durch das *separator*-Attribut definierte Trennzeichen, getrennt werden.

Zudem kann das *tag*-Attribut eine Liste verschiedener JUBE-Tags enthalten, die mittels Boolescher Algebra verknüpft werden können. Ergibt der boolesche Ausdruck den Wahrheitswert *false*, so wird dieser Parameter ausgeblendet und im weiteren Workflow nicht verwendet. Auf diese Weise können z.B. gleichnamige Parameter mit unterschiedlichen Tags definiert werden, so dass der Wert des Tags und damit der zu verwendende Parameter bei der Ausführung des Workflows festgelegt werden kann. [12] [29]

Die Verwendung der Parametersets in den Ausführungsschritten und die Erläuterung der dafür relevanten Attribute werden im Kapitel 2.1.3.6 näher erläutert. Eine Erläuterung aller Attribute der `<parameterset>`- und `<parameter>`-Tags und deren Funktion findet sich im Anhang unter den Auflistungen A.2.1 und A.2.2.

2.1.3.5. Dateien und Substitutionen

Jeder Ausführungsschritt wird in verschiedenen Verzeichnissen ausgeführt, die automatisch von JUBE erzeugt und verwaltet werden. Um externe Dateien in diesen Verzeichnissen zu verwenden, kann ein Ausführungsschritt ein sogenanntes Fileset einbinden, das die Möglichkeit bietet, Dateien oder Ordner zu verlinken oder zu kopieren.

Das folgende Listing zeigt die Struktur eines Filesets.

```
<fileset name="files">
   <copy>...</copy>
   <link>...</link>
   <prepare>...</prepare>
</fileset>
```

Listing 2.7: Schematische Darstellung eines Filesets in JUBE

Das <*copy*>-Tag bietet die Möglichkeit eine Datei in ein Verzeichnis des Ausführungsschrittes zu kopieren, damit diese Datei dort verwendet werden kann. Dafür wird innerhalb des <*copy*>-Tags der Pfad zur gewünschten Datei angegeben.

Der `<link>`-Tag steht zur Verfügung, um einen symbolischen Link auf die angegebene Datei oder den Ordner in einem Verzeichnis des Ausführungsschrittes zu erstellen.

Wenn zusätzliche Operationen erforderlich sind, um die Dateien vorzubereiten (z.B. um eine Zip-Datei zu entpacken), kann das <*prepare*>-Tag verwendet werden.

Die kopierten oder verlinkten Dateien eines Filesets können mit Hilfe von Substitutionssets weiterverarbeitet werden. Beispielsweise kann ein Platzhalter in einer Datei ersetzt werden.

Das folgende Listing zeigt die Struktur der Substitutionssets.

```
<substituteset name="...">
   <iofile in="..." out="..." out_mode="..." />
   <sub source="..." dest="..." />
</substituteset>
```

Listing 2.8: Schematische Darstellung eines Substitutesets in JUBE

Das `<iofile>`-Tag enthält den Namen der Eingabe- (*in*) und Ausgabedatei (*out*). Zusätzlich kann durch das `out_mode`-Attribut spezifiziert werden, ob die angegebene Datei direkt verändert werden soll (*w*) oder ob Änderungen an die Eingabedatei angehangen werden soll (*a*). Innerhalb des `<sub>`-Tag wird die Ersetzung definiert. Dabei werden alle Vorkommen des `source`-Wertes durch den Wert in `dest` ersetzt, z.B. durch den Wert eines Parameters. [12] [29]

2.1.3.6. Ausführungsschritte und deren Arbeitspakete

Mit Hilfe eines Ausführungsschrittes (<*step*>-Tag) werden die Shell-Kommandos definiert und ausgeführt. Beim Ausführen dieser Ausführungsschritte erstellt JUBE automatisch Arbeitsverzeichnisse, in dem die Informationen eines Ausführungsschrittes gespeichert sind.

Der allgemeine Aufbau eines Ausführungsschrittes ist wie folgt:

```
<step name="..." depend="..." work_dir="..." suffix="..." shared="..."
    active="..." export="..." max_async="..." iterations="..." cycles="..."
    procs="..." do_log_file="...">
   <use from="">...</use>
   <do stdout="..." stderr="..." work_dir="..." active="..." done_file="..."
   error_file="..." break_file="..." shared="...">...</do>
</step>
```

Listing 2.9: Schematische Darstellung eines Ausführungsschrittes in JUBE

Der `<use>`-Tag eines Ausführungsschritts enthält alle zu verwendenden Sets und kann innerhalb eines `<step>`-Tags mehrfach verwendet werden oder eine Liste aller

zu verwendenden Sets enthalten. Über das *from*-Attribut können auch externe Sets verwendet werden. Beim Einbinden von File- und Substitutesets werden die Dateien automatisch eingebunden und die angegebenen Platzhalter ersetzt. Die Parameter innerhalb der eingebundenen Parametersets können im Ausführungsschritt und in den anderen eingebundenen Sets verwendet werden.

Ein Shell-Kommando wird durch den `<do>`-Tag definiert. Dabei wird das Shell-Kommando innerhalb des Tags ausgeschrieben. Der `<do>`-Tag kann innerhalb eines Ausführungsschrittes mehrfach verwendet werden. Die Standard-Shell zur Ausführung der Befehle ist */bin/sh*. Diese kann allerdings durch die Umgebungsvariable *JUBE_EXEC_SHELL* geändert werden. [12] [29]

Eine Erläuterung zu den Attributen und deren Funktionen der `<step>`- und `<do>`-Tags ist in Anhang unter den Auflistungen A.2.3 und A.2.4 zu finden.

Um die Daten eines Ausführungsschrittes intern zu speichern, verwendet JUBE sogenannte Arbeitspakete. Ein Ausführungsschritt kann dabei durch mehrere Arbeitspakete beschrieben werden, die eine Kombination aus einem Ausführungsschritt und einer Parameterauswahl darstellen. Die Parameterauswahl wird dabei durch die im Kapitel 2.1.3.4 beschriebenen möglichen Inhalte von Parametern bestimmt. Wird beispielsweise ein Parameterset verwendet, das einen Parameter mit drei möglichen Inhalten und einen Parameter mit zwei möglichen Inhalten definiert, so werden insgesamt sechs Arbeitspakete erzeugt, die alle Kombinationen der möglichen Parameterinhalte enthalten. Eine genauere Beschreibung der Arbeitspakete folgt im Kapitel 3.3.

Ein wichtiges Attribut des Parametersets bzw. der Parameter aus Kapitel 2.1.3.4, das für die Arbeitspakete relevant ist, ist das *update_mode*-Attribut. Es regelt die Handhabung der Neuauswertung des Parameters bei der Ausführung eines Ausführungsschrittes, der diesen Parameter verwendet. Es kann auf *never*, *use*, *step*, *cycle* oder *always* gesetzt werden. Der Standardwert ist *never*, so dass dieser Parameter nie neu ausgewertet wird. *Use* bewirkt eine Neuauswertung bei Verwendung des zugehörigen Parametersatzes durch einen Ausführungsschritt, und *step* bewirkt eine Neuauswertung bei jedem neuen Ausführungsschritt. Eine Neuauswertung bei jeder Wiederholung eines *do*-Befehls eines Ausführungsschritts ist mit *cycle* und eine Neuauswertung für jeden Schritt und jede Wiederholung mit *always* möglich. Mit diesem Attribut ist es somit möglich, dass sich die Parameterwerte und damit die Arbeitspakete zur Laufzeit ändern. [12] [29]

2.1.3.7. Statistische Patternräume

Neben Parametersets können auch Patternsets erstellt werden, die z.B. zum Auslesen von Ergebnisgrößen verwendet werden können. Dabei wird ein Patternset mit den zugehörigen Pattern analog zur Erstellung eines Parametersets definiert.

Das Schema einer Patternset-Definition ist folgend abgebildet:

```
<patternset name="..." init_with="...">
   <pattern name="..." default="..." unit="..." mode="..." type="..."
       dotall="...">...</pattern>
</patternset>
```

Listing 2.10: Schematische Darstellung eines Patternsets in JUBE

Innerhalb des *<pattern>*-Tags kann das zu suchende Pattern mit Hilfe von regulären Ausdrücken deklariert werden. Zusätzlich stellt JUBE einige Standardpattern zur

Verfügung, wie z.B. *$jube_pat_int*, welches nach ganzzahligen Werten sucht. Wird das verwendete Pattern mehrfach gefunden, generiert JUBE automatisch statistische Werte wie z.B. den Durchschnitt, das Minimum oder das Maximum der gefundenen Werte. [12] [29]

Die Attribute des Patternsets und der zugehörigen Pattern, sowie deren Funktionen sind der Vollständigkeit halber in den Auflistungen A.2.5 und A.2.6 erläutert.

2.1.3.8. Analyse und Ergebnisse

Die folgenden Tags bietet die Möglichkeit, Dateien zu durchsuchen, um relevante Daten zu extrahieren und diese in vorgegebenen Formaten auszugeben. Dazu wird der Inhalt der Dateien unter Verwendung der angegebenen Patternsets durchsucht und die Ergebnisse in gewünschter Form ausgegeben.

Mit Hilfe des `<analyser>`-Tags, welches durch das *name*-Attribut eindeutig definiert ist, können beliebige Ergebnisse (z.B. aus der Standardausgabedatei) extrahiert werden:

```
<analyser name="..." reduce="...">
   <use>...</use>
   <analyse step="...">
      <file use="...">...</file>
   </analyse>
</analyser>
```

Listing 2.11: Schematische Darstellung des `<analyser>`-Tags in JUBE

Das *reduce*-Attribut des *<analyser>*-Tags kann die Ergebniszeilen der Analyse reduzieren, falls das *iterations*-Attribut beim angegebenen Ausführungsschritt genutzt wurde. Der Standardwert ist *true* und sorgt dafür das alle Ergebniszeilen kombiniert werden. Beim Wert *false* wird pro Iteration eine Zeile erstellt.

Mit dem `<use>`-Tag können Patternsets deklariert werden, die für alle angegebenen `<analyse>`-Tags verwendet werden. Das *step*-Attribut des `<analyse>`-Tags deklariert den Ausführungsschritt, dessen Daten durchsucht werden sollen. Der `<file>`-Tag definiert eine bestimmte Datei, die durchsucht werden soll, und das *use*-Attribut kann optional ein Patternset deklarieren, das ausschließlich auf diese Datei angewendet wird.

Zur Darstellung der bei der Analyse ermittelten Werte kann der *<result>*-Tag verwendet werden. Es bestehen die Möglichkeiten der Darstellung in einer Tabelle, der Ausgabe in eine Datenbank oder des Sendens an einen Syslog-Server. [12] [29]

Die allgemeine Struktur des *<result>*-Tags, einschließlich der Ausgabe in Tabellenform, ist im folgenden Listing dargestellt:

```
<result result_dir="...">
   <use from="...">..</use>
   <table name="..." style="..." sort="..." separator="..." filter="...">
      <column colw="..." format="..." title="...">...</column>
   </table>
</result>
```

Listing 2.12: Schematische Darstellung einer Ergebnistabelle in JUBE

Mit dem optionalen *result_dir*-Attribut kann ein Verzeichnis angegeben werden, in dem die formatierten Ergebnisse als Textdatei gespeichert werden. Mit dem `<use>`-Tag kann ein `<analyser>` eingebunden werden, dessen Daten zur Darstellung der Ergebnisse verwendet werden.

Mit dem <table>-Tag kann eine einfache ASCII-basierte Tabellenausgabe definiert werden. Der <column>-Tag definiert eine Spalte der Ergebnistabelle und kann Parameter- oder Patternnamen enthalten. [12] [29]

Durch die Attribute der <table>- und <column>-Tags kann die Formatierung der Tabelle angepasst werden. Im Anhang sind die Attribute der Tags und deren Funktion unter den Auflistungen A.2.7 und A.2.8 erläutert.

Darüber hinaus gibt die Möglichkeit, die Werte der Analyse in eine sqlite3-Datenbank zu speichern.

```
<result result_dir="...">
   <use from="...">..</use>
   <database name="..." primekeys="..." file="..." filter="...">
      <key format="..." title="...">...</key>
   </database>
</result>
```

Listing 2.13: Schematische Darstellung einer Datenbank-Ergebnistabelle in JUBE

Anstelle einer Tabellen Darstellung kann durch das <database>-Tag die Erstellung einer Datenbank-Tabelle definiert werden. Die Spalten der Datenbank-Tabelle werden mit dem <key>-Tag definiert. Der <key>-Tag kann einzelne Parameter- oder Patternnamen beinhalten.

Auch bei der Datenbank können durch Attribute bestimmte Formatierungen angegeben werden. Die Attribute der <database>- und <key>-Tags und dessen Funktionen sind im Anhang unter A.2.9 und A.2.10 erläutert.

Außerdem gibt es eine weitere Ausgabemöglichkeit, den Syslog:

```
<syslog name="..." address="..." host="..." port="..." sort="..."
    format="..." filter="...">
   <key format="..." title="...">...</key>
</syslog>
```

Listing 2.14: Beispiel für ein Datenbank-Ergebnistabelle in JUBE

Mit dem *address*-Attribut oder einer Kombination aus den *host*- und *port*-Attributen können die auszugebenden Ergebnisdaten als formatiertes Log-Protokoll an einen Syslog-Server gesendet werden. Der <key>-Tag kann einzelne Parameter- oder Patternnamen beinhalten. [12] [29]

Die weiteren Attribute der <syslog>-Tags sind der Vollständigkeit halber im Anhang unter Auflistung A.2.11 aufgeführt. Die Attribute des <*key*>-Tags sind identisch mit denen des Datenbank-Keys.

2.2. Grundlagen zur Datenverwaltung

Der Begriff Datenverwaltung beschreibt den Prozess der Speicherung von Daten, unabhängig vom Speichermedium und bezieht sich auf den reinen Speicheraspekt, der wiederum unabhängig von der Struktur und Organisation der Daten ist. Die Datenverwaltung wird auch als Datenmanagement bezeichnet und deren grundsätzliche Bedeutung wurde bereits in der Motivation (Kapitel 1) dargestellt.

Der Begriff Datenhaltung wiederum bezieht sich auf den Prozess der Speicherung von Daten in einem bestimmten Format oder einer bestimmten Struktur. In diesem Un-

terkapitel werden verschiedene Datenhaltungsmodelle vorgestellt, die für diese Arbeit in Frage kommen. [4] [18]

Anhand dieser Beschreibung und der im Kapitel 4 erläuterten Anforderungen an die neue Datenhaltung kann im Kapitel 5 eine konkrete Datenhaltungsoption ausgewählt werden.

2.2.1. Textbasierte Datenhaltung

Textbasierte Datenspeicherung ist die Speicherung von Daten in einer textbasierten Kodierung. Dieses Format unterscheidet sich von anderen Formaten, wie z.B. den im nächsten Unterkapitel vorgestellten binären Formaten, durch die Art der Interpretation, da der Inhalt einer Textdatei als sequentielle Folge von Zeichen interpretiert wird, während bei binären Dateien eine beliebige andere Interpretation des Inhalts möglich ist. Textdateien können daher ohne spezielle Programme mit einem Texteditor gelesen und bearbeitet werden. [14]

Neben dem Vorteil der Bearbeitbarkeit sind textbasierte Daten für den Menschen leichter lesbar, da es sich um Klartext handelt. Dies erleichtert auch die Fehlersuche und das Verständnis des Inhalts. Darüber hinaus bieten sie den Vorteil der Portabilität durch eine universelle Kodierung (z.B. ASCII oder UTF-8) und können so zwischen verschiedenen Betriebssystemen und Anwendungen ausgetauscht werden. Diese Zeichenkodierung erfordert jedoch eine bestimmte Anzahl von Bits, um jedes Zeichen darstellen zu können, was im Vergleich zu Binärdateien, die die interne binäre Darstellung der Daten speichern, zu einem höheren Speicherplatzbedarf führt. [14]

JUBE verwendet derzeit eine textbasierte Datenhaltung im XML-Format (siehe Kapitel 2.1.2), das für hierarchische Datenstrukturen geeignet ist. Dazu nutzt JUBE das *ElementTree*-Python-Paket, mit dem die textbasierte Datenhaltung gelesen und bearbeitet werden kann.

2.2.2. Binäre Datenhaltung

Die binäre Datenspeicherung ist die Speicherung von Daten in einem binären Format, das aus einer Folge von Nullen und Einsen besteht. Im Gegensatz zur textbasierten Datenhaltung sind diese Daten nur maschinenlesbar und nicht von Menschen interpretierbar. Binäre Dateien können beispielsweise zur Speicherung von Text, Bildern, Audio, Video oder ausführbaren Programmen verwendet werden. [14]

Wie bereits im vorhergehenden Unterkapitel beschrieben, ist die Datenspeicherung in Binärdateien zwar effizienter in Bezug auf Speicherplatz und Verarbeitungsgeschwindigkeit, erschwert jedoch die Fehlersuche aufgrund der fehlenden menschlichen Lesbarkeit. Darüber hinaus hängt die Struktur von Binärformaten vom verwendeten System ab. Beispielsweise kann sich die Darstellung einer Binärdatei zwischen den Betriebssystemen Windows und Linux unterscheiden, was die Portabilität einschränkt. Diese Nachteile machen diese Art der Datenspeicherung für die Anwendung in dieser Arbeit ungeeignet. [14]

2.2.3. Temporäre Datenhaltung

Die temporäre Datenhaltung beschreibt die temporäre Speicherung von Daten während der Laufzeit eines Prozesses. Diese Datenhaltung dient der Zwischenspeicherung von Daten und kann z.B. über den Arbeitsspeicher erfolgen. [25]

Diese Möglichkeit der Datenhaltung ist für den Anwendungsfall dieser Arbeit nicht geeignet, da die Daten auch über die Laufzeit hinaus zugänglich sein müssen, um z.B. bei der Fortsetzung eines Workflows diese Daten wiederverwenden zu können.

2.2.4. Datenbanken

Die bekannteste und am weitesten verbreitete Lösung zur Datenspeicherung ist die Datenbank. Eine Datenbank ist eine organisierte Sammlung einer Menge von Daten, die je nach Art der Datenbank in strukturierter oder unstrukturierter Form gespeichert werden. Mit einem durchdachten Datenbankdesign, das an die Struktur der Daten angepasst ist, können Vorteile wie Datenintegrität, -konsistenz und -sicherheit erreicht werden. [6, S. 51-52]

In den folgenden Abschnitten werden die verschiedenen Arten von Datenbanken erläutert und was ein Datenbankmanagementsystem ist.

2.2.4.1. Relationale Datenbanken

Das relationale Datenbankmodell organisiert Daten in festen Schemata mit Hilfe von Tabellen, die Relationen genannt werden. Diese Relationen bestehen aus einer beliebigen Anzahl von Spalten, die die Attribute repräsentieren, und Zeilen, die jeweils einen Datensatz repräsentieren. Um einen Datensatz anhand eines oder mehrerer Attribute eindeutig identifizieren zu können oder um einen Verweis auf andere Datensätze einzufügen, werden sogenannte Schlüssel verwendet. Ein oder mehrere Attribute können als Schlüssel oder als Kombination von Schlüsseln definiert werden, so dass ein Datensatz anhand dieser Attribute eindeutig identifizierbar ist, z. B. durch eine ID. [6, S. 52-53]

Die eindeutige Identifizierung der Daten ist für die Abfrage, Ergänzung oder sonstige Verwaltung der Datensätze relevant. Die Verwaltung dieser Daten erfolgt über die Abfragesprache Structured Query Language (SQL), die basierend auf den Schlüsseln eine Datensatzselektion vornehmen kann. [6, S. 52-53]

Nicht nur die zu speichernden Informationen, sondern auch festgelegte Datenbanknormalisierungen bestimmen die Struktur der Tabellen. Dabei werden die Daten so strukturiert, dass weniger redundante oder anomale Daten auftreten und die Tabellen in kleinere, zusammenhängende Einheiten aufgeteilt werden. Die Normalisierungen und ihre Anwendung in dieser Arbeit werden in Kapitel 2.3 näher erläutert.

2.2.4.2. NoSQL Datenbanken

Not-only-SQL-Datenbanken, kurz NoSQL-Datenbanken, sind im Gegensatz zu klassischen relationalen Datenbanken schemafrei. Es gibt also keine feste Struktur der in der Datenbank gespeicherten Daten. Die Daten können nicht nur als Tabellen, sondern beispielsweise auch als Dokumente oder Grafiken dargestellt werden. Dies ermöglicht eine flexible Speicherung der Daten in unterschiedlichen Formaten.

Zur Verwaltung der Daten von NoSQL-Datenbanken ist die für relationale Datenbanken verwendete Abfragesprache SQL ungeeignet, da diese eine feste Strukturierung der Daten vorsieht, die bei NoSQL-Datenbanken allerdings nicht gegeben ist. Daher benötigt jeder Datenbanktyp der NoSQL-Datenbanken eine eigene API zur Verwaltung der Datensätze, da keiner dieser Datenbanktypen einem gemeinsamen Schema folgt. [15, S. 299-303]

2.2.4.3. Datenbankmanagementsysteme

Datenbankmanagementsysteme (DBMS) dienen der Verwaltung von Datenbanken. Es stellt die Schnittstelle zwischen der Anwendung und der Datenbank dar und ermöglicht das Erstellen, Ändern, Abfragen und Löschen von Daten in der Datenbank. Darüber hinaus kann ein DBMS die Integrität, Sicherheit und Effizienz der Datenbank gewährleisten. [6, S. 51]

Wenn für die neue Datenhaltung in JUBE eine Datenbank verwendet werden soll, kommen verschiedene DBMS-Modelle für die beiden Datenbanktypen in Frage. Dabei ist zu beachten, dass es eine Vielzahl von DBMS gibt, die jeweils unterschiedliche Funktionen und Anwendungsbereiche bieten. Im Rahmen dieser Arbeit wird sich jedoch auf eine Auswahl von DBMS konzentriert, die für den speziellen Einsatz in dieser Arbeit als besonders geeignet erachtet werden. Diese werden im Folgenden näher beschrieben und es wird begründet, warum sie für diese Arbeit als geeignet angesehen werden.

Zwei der bekanntesten DBMS für relationale Datenbanken sind MySQL[1] und SQLite[2].[3] MySQL ist ein leistungsfähiges, serverbasiertes DBMS, das für den Einsatz in großen Anwendungen und Systemen entwickelt wurde. Es unterstützt mehrere parallele Verbindungen von der Anwendung zur Datenbank und ist für große Datenmengen ausgelegt. SQLite hingegen ist ein leichtgewichtiges, serverloses DBMS, das sich besonders für den Einsatz in eingebetteten Systemen eignet und sehr performant mit kleineren Datenmengen umgehen kann. [17] [23]

Diese beiden DBMS sind von den relationalen Datenbanktypen für den Anwendungsfall dieser Arbeit besonders geeignet, da beide eine weitreichende Unterstützung für Python-Anwendungen bieten. Es gibt umfangreiche Python-Bibliotheken und Module, die speziell für die Interaktion mit diesen Datenbanken entwickelt wurden. Beide DBMS verfügen über offizielle Python-Implementierungen (*MySQL Connector/Python* für MySQL und *sqlite3* für SQLite), die eine nahtlose Integration mit Python-Anwendungen ermöglichen. Das Paket *sqlite3* ist zudem bereits in der Standardbibliothek von Python enthalten und muss nicht zusätzlich installiert werden. [17] [23]

Die NoSQL-DBMS MongoDB[4] und BaseX[5] eignen sich ebenfalls gut für den Anwendungsfall dieser Arbeit. Sie haben unterschiedliche Ansätze, um die Speicherung und Verwaltung der Daten zu realisieren. Während MongoDB ein Dokumentenmodell verwendet, bei dem die Daten in JSON-ähnlichen Dokumenten gespeichert werden, konzentriert sich BaseX ausschließlich auf XML-Daten. Durch die JSON-ähnlichen Dokumente und die sogenannten Collections, in denen die Daten organisiert sind, haben MongoDB-Datenbanken eine ähnliche Struktur wie relationale Datenbanken. Ihr Ziel ist es, eine flexible, skalierbare und performante Datenspeicherung zu ermöglichen. BaseX wurde speziell für die Verwaltung von XML-Daten entwickelt und bietet eine intuitive Benutzeroberfläche für die Interaktion mit der Datenbank. Es unterstützt leistungsfähige XQuery-Funktionen zur Abfrage und Transformation von XML-Daten. [1] [16]

Beide DBMS verfügen auch über offizielle Python-Implementierungen (*PyMongo*

[1] https://www.mysql.com/de/
[2] https://www.sqlite.org/index.html
[3] siehe https://db-engines.com/de/ranking
[4] https://www.mongodb.com/de-de
[5] https://basex.org/

für MongoDB und *BaseXClient* für BaseX). Während für MongoDB die Unterstützung durch eine große Community spricht, bietet BaseX mit XML ein geeignetes Dateiformat für den Einsatz in JUBE. [1] [16]

Im Kapitel 5.1 werden die aufgeführten DBMS anhand der Anforderungsanalyse und dem aktuellen Stand von JUBE evaluiert und somit ein geeignetes DBMS für den Einsatz in JUBE ausgewählt.

2.3. Normen und Standards

Normen und Standards sind festgelegte Regeln, Richtlinien oder Kriterien, um sicherzustellen, dass Produkte, Dienstleistungen, Verfahren oder Prozesse bestimmte Anforderungen erfüllen und eine gewisse Qualität, Kompatibilität, Interoperabilität oder Sicherheit aufweisen. Sie können von nationalen und internationalen Organisationen entwickelt und festgelegt werden und gewährleisten Sicherheit, Leistung, Konsistenz und Vergleichbarkeit. [9]

Im folgenden Unterkapitel werden die verwendeten Normen und Standards kurz erläutert. Zur genaueren Erläuterung der angewandten Verfahren wird auf die angegebenen Quellen verwiesen.

2.3.1. ISO-Norm 25010

Die Internationale Organisation für Normung (ISO) entwickelt freiwillige Normen, die dazu dienen, bestimmte Anforderungen sicherzustellen. Diese Normen gibt es für verschiedene Bereiche wie Qualitätssicherung oder Design.

Die ISO-Norm 25010 ist ein Modell zur Beschreibung der Qualitätsmerkmale eines Computersystems oder eines Softwareprodukts. Sie umfasst insgesamt 8 Merkmale, die die Spezifikation, Messung und Bewertung der Qualität von Systemen und Softwareprodukten festlegen. Das Modell enthält keine Qualitätsmerkmale für die funktionalen Bedingungen, schließt aber allgemeine funktionale Anforderungen ein. [10]

Für die Strukturierung der Anforderungsanalyse im Kapitel 4 wurde diese ISO-Norm verwendet.

2.3.2. Standards der Datenmodellierung

Um die Daten der 'realen' Welt in eine abstrakte, formale Beschreibung und Darstellung der Datenbank zu überführen, werden Datenmodelle verwendet. Für die Formulierung dieser Datenmodelle stehen verschiedene Modellierungsmethoden zur Verfügung. Im Folgenden werden die in dieser Arbeit verwendeten Modellierungsmethoden beschrieben.

Das Entity-Relationship-Modell (ERM), wird in der Regel für den Entwurf von - meist relationalen - Datenbanken verwendet. In diesem Modell werden Objekte der realen Welt als Entitätstypen durch Rechtecke dargestellt. Die zugehörigen Attribute werden durch Ellipsen um den Entitätstyp beschrieben. Zu den Attributen kann auch ein sogenanntes identifizierendes Attribut gehören, das eine Entität unter den Entitätstypen eindeutig identifizierbar macht. Dieses wird auch Primärschlüssel genannt und im ERM durch Unterstreichung gekennzeichnet. [6, S. 59-61]

Die folgende Abbildung zeigt ein Beispiel für ein einfaches ERM:

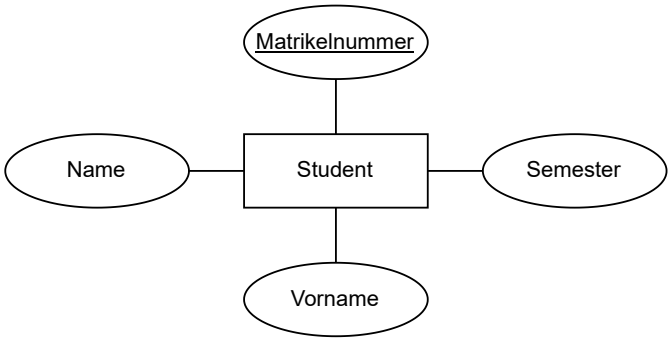

Abbildung 2.1.: Einfaches Beispiel für ein ERM von einem Studenten

Das Objekt der realen Welt, der Student, wird durch ein Rechteck als Entitätstyp repräsentiert. Es besitzt die Attribute Matrikelnummer, Vorname, Nachname und Semester. Da jeder Student eindeutig über seine Matrikelnummer identifiziert werden kann, ist diese das identifizierende Attribut und aus diesem Grund unterstrichen.

Darüber hinaus können zwischen den verschiedenen Entitätstypen Beziehungen bestehen, die durch Linien und Rauten mit der entsprechenden Beschreibung der Beziehung dargestellt werden. Es gibt verschiedene Arten von Beziehungen zwischen Entitäten, z.B. 1-zu-1-, 1-zu-N- und N-zu-N-Beziehungen. Bei der 1-zu-1-Beziehung kann ein Entitätstyp maximal mit einer Entität eines anderen Entitätstyps verknüpft werden. Bei der 1-zu-N-Beziehung wird ein Entitätstyp mit beliebig vielen Entitäten eines anderen Entitätstyps verknüpft und bei der N-zu-N-Beziehung werden beliebig viele Entitäten mit beliebig vielen anderen Entitäten verknüpft. In der Darstellung kann die Art der Beziehung durch die Beschriftung der Verbindungslinien zwischen den Entitäten identifiziert werden. [6, S. 59-61]

Die folgende Abbildung ist eine Weiterführung des vorherigen Beispiels und soll die Beziehungen zwischen den Entitätstypen zum besseren Verständnis an einem Beispiel veranschaulichen.

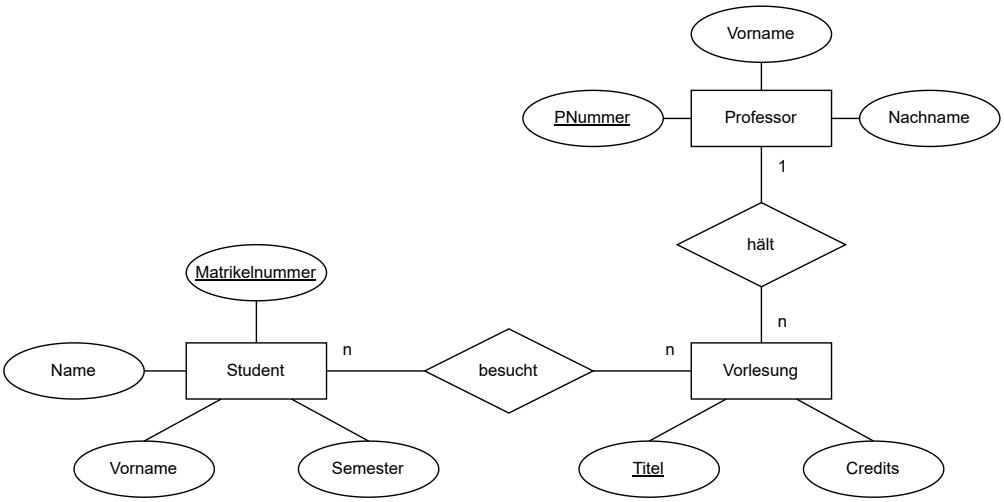

Abbildung 2.2.: Einfaches Beispiel für ein ERM mit Beziehungen

In diesem Beispiel werden zusätzlich die Entitätstypen Vorlesung und Professor hinzugefügt. Diese haben die Attribute Titel und Credits bzw. PNummer, Vorname und

Nachname. PNummer steht für die Personalnummer, mit der ein Professor eindeutig identifiziert werden kann. Gleiches gilt für den Titel einer Vorlesung. Diese drei Entitätstypen stehen in Beziehung zueinander. So können beliebig viele Studenten beliebig vielen Vorlesungen hören. Es handelt sich also um eine N-zu-N-Beziehung, die durch die Raute und eine Beschriftung gekennzeichnet ist. Ein Professor kann beliebig viele Vorlesungen halten, während eine Vorlesung (im Normalfall) nur von einem Professor gehalten wird. In diesem Fall handelt es sich um eine 1-zu-N-Beziehung, die ebenfalls durch die Raute und die Beschriftung gekennzeichnet ist.

Bei diesem und dem Beispiel 2.1 ist zu beachten, dass es sich nicht um ein vollständiges ERM handelt, da es weitere sinnvolle Beziehungen und Attribute geben kann und es nur als Beispiel dienen soll.

Das ERM dient als Grundlage für das Datenbankdesign und beschreibt die benötigten Daten in abstrakter Form grafisch. Das Relationenmodell (RM), stellt die gleichen Daten wie das zugrunde liegende ERM dar, beschreibt aber zusätzlich die Datentypen der Attribute und stellt die Beziehungen tabellarisch dar. Somit kann das RM im Gegensatz zum ERM direkt in eine Datenbank überführt werden.

Um das ERM in ein RM zu überführen, werden die Entitätstypen des ERM im RM in sogenannten Relationen, d.h. Tabellen, dargestellt. Ausgehend von den Entitäten als eindeutige Datensätze in der Tabelle werden die Attribute der Entitäten in die Spalten der Tabelle übertragen, wobei die Primärschlüssel analog zum ERM ebenfalls unterstrichen werden. Die Primärschlüssel werden im RM mit der zusätzlichen Anmerkung **PK** markiert. Dies steht für ”Primary Key”, das englische Wort für Primärschlüssel. [3]

Folgende Abbildung zeigt das Beispiel des ERM aus Abbildung 2.1 umgewandelt in ein RM:

Student	
PK	**Matrikelnummer integer**
	Vorname text
	Nachname text
	Semester Integer

Abbildung 2.3.: Einfaches Beispiel für ein RM von einem Studenten

Die Realisierung der Beziehungen, also der Relationen, erfolgt über so genannte Fremdschlüssel, den Primärschlüssel einer anderen Tabelle, der als Referenz in der eigenen Tabelle gespeichert wird. Die 1-zu-N-Beziehung wird durch einen Fremdschlüssel dargestellt, indem die Relation mit der Ner-Beziehung einen Fremdschlüssel zur Relation der 1er-Beziehung erhält. Bei der 1-zu-1-Beziehung wird einer der beteiligten Relationen ein Fremdschlüssel hinzugefügt. Schließlich wird eine N-zu-N-Beziehung durch eine weitere Tabelle realisiert, die die beiden Fremdschlüssel innerhalb der Tabelle verbindet und die restlichen Attribute der Beziehung enthält. Der neue Primärschlüssel dieser Tabelle ist die Kombination der beiden Fremdschlüssel. [3]

Das folgende RM zeigt das umgewandelte ERM aus dem Beispiel 2.2.

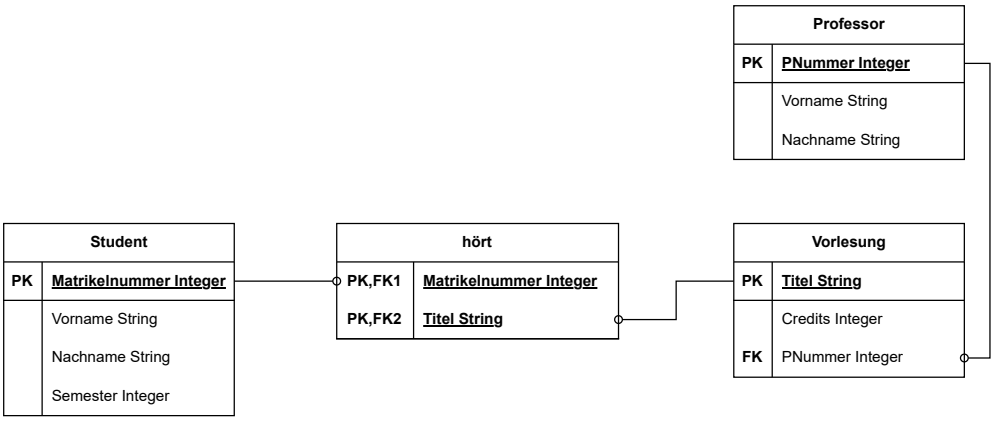

Abbildung 2.4.: Einfaches Beispiel für ein RM mit Beziehungen

Die anderen Entitätstypen werden ebenfalls als Tabelle mit den entsprechenden Zeilen dargestellt. Die N-zu-N-Beziehung zwischen Student und Vorlesung wird durch eine weitere Tabelle dargestellt, die beide Fremdschlüssel speichert. Fremdschlüssel werden mit **FK** für "foreign key" gekennzeichnet. Da die Kombination der beiden Fremdschlüssel den Primärschlüssel der Tabelle ergibt, werden sie zusätzlich mit **PK** gekennzeichnet. Die 1-zu-N-Beziehung zwischen Professor und Vorlesung kann durch einen Fremdschlüssel auf die PNummer des zugehörigen Professors in der Vorlesungs-Tabelle realisiert werden. [3]

In den Kapiteln 5.2.1 und 5.2.2 werden sowohl das ERM als auch das RM zur Erstellung des konzeptuellen und des relationalen Datenmodells verwendet.

2.3.3. Normalisierung der Datenbank

Die Normalisierung einer Datenbank dient dazu, die redundante Speicherung von Informationen und die daraus resultierenden Inkonsistenzen und Anomalien zu vermeiden. Die Daten werden nach bestimmten Regeln, auch Normalformen genannt, in das gewünschte Schema überführt. Derzeit gibt es insgesamt sechs gebräuchliche Normalformen: Die erste bis fünfte Normalform und die Boyce-Codd-Normalform, die als Erweiterung der dritten Normalform gilt. [6, S. 55] [11, S. 91]

Trotz der Vorteile, die sich aus der Vermeidung von Redundanzen, Inkonsistenzen und Anomalien ergeben, hat die vollständige Normalisierung einige Nachteile, die bei der Benutzung der Normalformen berücksichtigt werden sollten. Beispielsweise erhöht die Normalisierung die Komplexität der Datenbank und kann zu Performanceeinbußen führen, da Abfragen über mehrere Tabellen hinweg durchgeführt werden müssen. Außerdem erhöht sich der Aufwand für die Datenmanipulation, da auch das Aktualisieren, Einfügen oder Löschen von Daten über mehrere Tabellen hinweg erfolgen muss, was die Abfragen sehr komplex und aufwendig macht.

Im Folgenden werden die verwendeten Normalformen und die Gründe für ihre Verwendung kurz erläutert, und begründet, warum weitere Normalformen nicht verwendet werden.

Die erste Normalform dient der atomaren Trennung der Attribute einer Tabelle. Dies bedeutet, dass die Tabelle nur einfache, unteilbare Werte enthält, was zur Vermeidung von Datenredundanz und -inkonsistenz beiträgt. Außerdem erleichtert es die Manipulation und Abfrage der Daten. Die erste Normalform ist die grundlegendste Normalform

und sollte in jeder gut strukturierten Datenbank erreicht werden. Sie wird daher auch für den Datenbankentwurf dieser Arbeit verwendet. [6, S. 55-56] [11, S. 93-96]

Die zweite Normalform verlangt, dass jedes Nicht-Schlüsselattribut funktional vom vollständigen Schlüssel abhängig ist. Dies bedeutet, dass jedes Attribut in einer Tabelle nur von der vollständigen Kombination der Schlüsselattribute abhängen darf und nicht nur von einem Teil davon. Dies vermeidet Redundanz bei teilweise abhängigen Attributen und trägt zur Effizienz und Konsistenz der Datenbank bei. Aus diesen Gründen wurde diese Normalform auch für den Datenbankentwurf dieser Arbeit verwendet. [6, S. 56] [11, S. 97-103]

Die Entscheidung gegen die Anwendung der weiterführenden Normalformen wurde aufgrund von steigender Komplexität der Abfragen und den Anforderungen der Arbeit getroffen, da die nächsten Normalformen weitere Aufspaltungen der Tabellen verursachen. So werden höhere Normalformen in der Regel für die Erfüllung spezieller Anforderungen genutzt, die in diesem Anwendungsfall nicht nötig sind.

Die Anwendung dieser Normalformen wird anhhand der Überführung der Informationen in das ERM und RM in Kapitel 5.2.1 verdeutlicht und die Entscheidung zwischen der zweiten und dritten Normalform anhand eines Entwurfsschritt näher beschrieben.

3. Ausgangssituation des Workflow-Tools JUBE

Im vorherigen Kapitel wurden die Grundlagen des Workflow-Tools JUBE erläutert, die nun weiter vertieft werden. Dazu wird zunächst die Funktion und der Programmablauf anhand der einzelnen Kommandozeilenbefehle von JUBE erläutert. Anschließend wird die aktuelle Datenhaltung anhand der Dateien *workpackages.xml* und *configuration.xml* auf ihren Aufbau und ihre Struktur hin untersucht, um die Frage beantworten zu können, welche Daten in der neuen Datenhaltung gespeichert werden sollen. Mit den daraus gewonnenen Informationen wird JUBE hinsichtlich der Nachteile der textbasierten Datenhaltung analysiert, um eine Ablösung der aktuellen Datenhaltung begründen zu können.

3.1. Allgemeiner Programmaufbau und -ablauf

JUBE kann, wie bereits erwähnt, innerhalb einer Linux Konsole bzw. Terminal über die Kommandozeile ausgeführt werden. Dabei verfügt JUBE über verschiedene Kommandozeilenargumente, um z.B. ausgewählte Workflows auszuführen oder zu analysieren. Mit Hilfe einer Eingabedatei werden die gewünschten Workflows konfiguriert. Der Aufbau dieser Eingabedatei wurde bereits im Kapitel 2.1.3 näher erläutert.

3.1.1. Der *run*-Befehl

Der wichtigste Befehl ist der *run*-Befehl. Dieser startet einen neuen Workflow, indem die angegebene Eingabedatei (*input.xml*) verarbeitet wird. JUBE extrahiert die Daten der Eingabedatei, speichert die nötigen Informationen und führt die in der Eingabedatei angegebenen Ausführungsschritte aus. Das Kommando wird wie folgt ausgeführt:

```
jube run input.xml [-t TAG] [-o OUTPATH]
```

Als Optionen des *run*-Befehls können mit *-t* oder *--tag* JUBE-Tags angegeben werden. Zusätzlich kann mit der *-o-* oder *--outpath*-Option ein Ordner angegeben werden, in dem der Wokflow ausgeführt wird und die Daten gespeichert werden sollen. Wird bei der Ausführung des *run*-Befehls kein Ordner angegeben, so wird der in der Eingabedatei angegebene Ordnerpfad verwendet. [1]

Die Ausführung des *run*-Befehls mit der Ordnerangabe *bench_run* kann zu folgender Ordnerstruktur führen:

[1] Es gibt weitere Kommandozeilenoptionen, auf die im Folgenden nicht weiter eingegangen wird, da sie für das Thema dieser Arbeit nicht relevant sind.

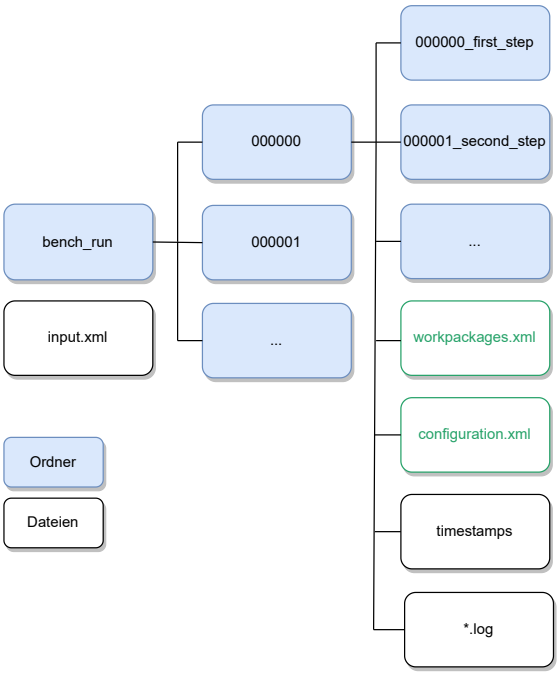

Abbildung 3.1.: Darstellung einer möglichen von JUBE erstellten Ordnerstruktur

Die Abbildung 3.1 zeigt das der Ordner mit dem Beispielnamen *bench_run* auf der gleichen Ebene wie die angegebene Eingabedatei erstellt wurde. Die Position des Ordners wird durch die oben genannte Option *-o* bzw. durch die Eingabedatei bestimmt. Er enthält für jeden mit *jube run* gestarteten Lauf einen Unterordner mit einer aufsteigenden ID. Innerhalb des Unterordners wird für jedes Arbeitspakete ein Ordner mit einer aufsteigenden ID und dem Namen des Schritts angelegt. Zusätzlich zu den Ordnern der Arbeitspakete werden verschiedene Dateien zur Protokollierung der Ausführung und zur Speicherung der Workflow- und Arbeitspaket-Daten angelegt.

Zur besseren Veranschaulichung wurden die Dateien *configuration.xml* und *workpackages.xml*, die die aktuelle Datenhaltung bilden, in der Abbildung 3.1 grün markiert. Sie werden im Folgenden als Konfigurations- bzw. Workpackagedatei bezeichnet. Diese Dateien werden für jeden Workflow erstellt und enthalten die Informationen des Workflows und der Arbeitspakete. Die genaue Struktur der beiden Dateien wird in Kapitel 3.2 und 3.3 näher erläutert.

3.1.2. Der *continue*-Befehl

Es kann vorkommen, dass nicht alle Ausführungsschritte eines Workflows durch den *run*-Befehl vollständig ausgeführt werden. Um den Workflow und die damit nicht abgeschlossenen Ausführungsschritte fortzusetzen, wird der *continue*-Befehl verwendet:

```
jube continue ordnername [-i ID]
```

Der Platzhalter *ordnername* spezifiziert den Ordnernamen, in dem sich der fortzusetzende Workflow befindet. Mit der Kommandozeilenoption *-i* kann die ID des Workflows innerhalb des Ordners angegeben werden. Wird keine ID angegeben, wird der letzte im Verzeichnis gefundene Workflow fortgesetzt.

3.1.3. Der *analyse*-Befehl

Der *analyse*-Befehl analysiert die in der Eingabedatei aufgeführten Ausführungsschritte mit Hilfe der im <*analyse*>-Tag angegebenen Patternsets:

```
jube analyse ordnername [-i ID] [-u UPDATE_FILE]
```

Der Platzhalter *ordnername* gibt den Namen des Ordners an, in dem sich der zu analysierende Workflow befindet, und die Option *-i* wird wie bei einem *continue*-Aufruf behandelt. Beim Aufruf des *analyse*-Befehls mit der Option *-u* oder *--update* können mit Hilfe der Angabe einer weiteren Datei die Daten der bisherigen Analyser nachträglich geändert werden. Diese Datei entspricht dem Aufbau der JUBE-Eingabedatei oder eine modifizierte Version der Eingabedatei, die neue Informationen über die Patternsets, den Analyser und die Ausgabe der Ergebnisse enthalten kann. Die aktuell gespeicherten Daten des Workflows zu den Patternsets, dem Analyser und den Ergebnisausgabe werden mit den angegebenen Informationen aktualisiert. Dadurch kann ein erneuter Durchlauf des Workflows vermieden werden und es werden nur die benötigten Daten angepasst.

Die Analyse kann auch automatisch direkt im Anschluss an den *run-* oder *continue*-Befehl mit der *-a-* bzw. *--analyse*-Kommandozeilenoption nach der Ausführung der Schritte ausgeführt werden. Dies geschieht analog zum *analyse*-Befehl.

3.1.4. Der *result*-Befehl

Der *result*-Befehl kann die aus der Analyse extrahierten Daten in verschiedenen Formen ausgeben. Der Aufruf des Befehls sieht wie folgt aus:

```
jube result [-i ID] [-u UPDATE_FILE] ordnername
```

Der Platzhalter *ordnername* gibt den Namen des Ordners an, in dem sich der Workflow befindet. Zusätzlich besteht die Möglichkeit, wie beim *analyse*-Befehl, mit *-u* bzw. *--update* die Daten der Patternsets, der Analyse und der Ergebnisausgabe zu aktualisieren. Zudem kann die Ergebnistabelle direkt beim *run-* bzw. *continue*-Befehl mit der *-r* bzw. *--result*-Option ausgegeben werden. Mit dem *result*-Befehl ist es auch möglich, vor der Ausgabe der Ergebnisse mit der *-a-* oder *--analyse*-Option eine Analyse durchzuführen.

3.1.5. Weitere Befehle

Mit dem *info*-Befehl können allgemeine Informationen des Workflows wie ID, Name, Kommentar, angegebene Tags und Zeitpunkt der letzten Ausführung bzw. Änderung ausgegeben werden. Ohne Angabe einer ID mit der Option *-i* werden alle Workflows des angegebenen Ordners ausgegeben.

Der *status-* bzw. *log*-Befehl kann den Status bzw. die protokollierten Daten zur Ausführung eines ausgewählten Workflows ausgeben.

Mit dem *remove*-Befehl kann ein ausgewählter Workflow und damit der gesamte Unterordner oder ein zugehöriges Arbeitspaket, d.h. ein Teil der Workpackagedatei, gelöscht werden.

Mit dem *comment*-Befehl kann der aktuelle Workflow-Kommentar innerhalb der Konfigurationsdatei geändert werden.

3.2. Aufbau der Konfigurationsdatei

Wie bereits in Abbildung 3.1 gezeigt, wird bei der Ausführung des *run*-Befehls die abgebildete Datenstruktur einschließlich der Konfigurationsdatei (*configuration.xml*) erzeugt. Diese Konfigurationsdatei speichert die Workflowdaten, um die programminternen Strukturen im Programmspeicher aufbauen zu können. Dadurch kann der Ablauf eines Workflows bei einem Folgebefehl wie z.B. *continue* oder *analyse* wiederhergestellt werden. Um die Workflow-Daten zu speichern, werden bei der Ausführung des *run*-Befehls zunächst alle Daten aus der Eingabedatei extrahiert, ausgewertet und in die Konfigurationsdatei geschrieben. Dieser Vorgang wird in Kapitel 3.4 ausführlich beschrieben. In diesem Kapitel wird die Struktur der Konfigurationsdatei erläutert, um einen Überblick über die zu speichernden Daten zu erhalten.

Das folgende Beispielschema soll einen Überblick über den Aufbau der Konfigurationsdatei und die inhaltliche Ähnlichkeit der Konfigurationsdatei mit der Eingabedatei geben.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube version="...">
   <selection>
      <tag>...</tag>
      ...
   </selection>
   <benchmark name="..." file_path_ref="..." outpath="...">
      <comment>...</comment>
      ...
      <parameterset name="..." duplicate="...">
         <parameter name="..." type="..." separator="..." duplicate="..."
             mode="...">...</parameter>
      </parameterset>
      <patternset name="...">
         <pattern name="..." type="..." dotall="..." mode="...">...</pattern>
      </patternset>
      <step name="...">
         <use>...</use>
         <do>...</do>
      </step>
      <analyser name="..." reduce="...">
         <use>...</use>
         <analyse step="...">
            <file>...</file>
         </analyse>
      </analyser>
      ...
   </benchmark>
```

```
</jube>
```

Listing 3.1: Beispielschemata einer Konfigurationsdatei

Da die Daten aus der Eingabedatei extrahiert, verarbeitet und in die Konfigurationsdatei geschrieben werden, ähneln die Dateien sich im Aufbau. Die Konfigurationsdatei speichert somit alle Angaben zu einem Workflow, die in der Eingabedatei gemacht wurden.

Die Konfigurationsdatei speichert zunächst innerhalb des <*jube*>-Tags mit dem *version*-Attribut die in dieser Ausführung verwendete JUBE-Version.

Das erste Element, der innerhalb des <*jube*>-Tags vorkommen kann, ist der *selection*-Tag. Dieser enthält eine Liste aller JUBE-Tags, die bei der Ausführung des *run*-Befehls angegeben wurden. Jeder angegebene JUBE-Tag wird mit <*tag*> umschlossen.

Globale Elemente sind in der Konfigurationsdatei nicht vorhanden. Das Einbinden von externen Daten durch das globale <*include-path*>-Tag wird vor der Ausschreibung der Konfigurationsdatei ausgewertet, so dass die an anderer Stelle wiederzuverwendenden Daten auch in der Konfigurationsdatei ausgeschrieben werden. Die <*not*>- und <*only*>-Tags des <*selection*>-Tags (Kapitel 2.1.3.3) werden vor der Ausschreibung der Konfigurationsdatei ausgewertet und erscheinen daher nicht in der Konfigurationsdatei. Die globalen Sets werden mit den lokalen Sets des Workflows zusammengeführt und innerhalb der Konfigurationsdatei als ein Gesamtset interpretiert.

Innerhalb eines <*benchmark*>-Tags werden alle Angaben wie Kommentar, Sets, Ausführungsschritte, Analyser und Ergebnisse, die in der Eingabedatei für einen Workflow definiert sind, angegeben. Im Gegensatz zur Eingabedatei werden jedoch die Werte aller Attribute, inklusive der optionalen, innerhalb des jeweiligen Tags entweder mit den Standardwerten oder den benutzerdefinierten Werten angegeben. Ist beispielsweise das *separator*-Attribut des <*parameter*>-Tags in der Eingabedatei nicht gesetzt, so wird in der Konfigurationsdatei der Standardwert Komma (,) angegeben.

Neben den ausgefüllten Attributwerten unterscheiden sich auch die Pfadangaben der Dateien. Da die Pfadangaben in der Eingabedatei relativ sind und die Konfigurationsdatei an einem anderen Ort in der Ordnerstruktur liegt als die Eingabedatei (siehe Abbildung 3.1), werden die Pfade in der Konfigurationsdatei entsprechend angepasst.

Schließlich ist zu beachten, beim Schreiben vom XML-Dateien im UTF-8-Format Sonderzeichen maskiert werden. So werden beispielweise doppelte Anführungszeichen aus der Eingabedatei (Listing 3.2) durch *"* in der Konfigurationsdatei (Listing 3.3) ersetzt.

```
<step name="write_number">
   <use>param_set</use> <!-- use existing parameterset -->
   <do>echo "Number: $number"</do> <!-- shell command -->
</step>
```

Listing 3.2: Beispiel einer Step-Deklarierung in einer Eingabedatei

```
<step name="write_number">
   <use>param_set</use>
   <do>echo &quot;Number: $number&quot;</do>
</step>
```

Listing 3.3: Konfigurationsdatei zu Beispiel 3.2

Zusammengefasst repräsentiert die Konfigurationsdatei den erweiterten und verarbeiteten Inhalt der Eingabedatei und speichert damit die Grundstruktur des Workflows, so dass JUBE mit Hilfe dieser Datei die programminternen Strukturen im Programmspeicher aufbauen kann.

3.3. Aufbau der Workpackagedatei

Neben der Konfigurationsdatei ist in Abbildung 3.1 auch die Workpackagedatei (*workpackages.xml*) zu sehen, die bei der Ausführung des *run*-Befehls erzeugt wird. Die Workpackagedatei speichert die Daten der erstellten Arbeitspakete für die Ausführungsschritte des Workflows. Diese Arbeitspakete wurden bereits im Kapitel 2.1.3.6 kurz vorgestellt und sind die Kombination aus einem Ausführungsschritt und einer Parameterauswahl. Die Funktion und der Aufbau eines Arbeitspakets werden in diesem Kapitel anhand der Workpackagedatei näher erläutert.

Um die Erstellung der Arbeitspakete im Folgenden besser erklären zu können, zeigt Listing 3.4 eine Eingabedatei, auf deren Grundlage die Workpackagedatei aufgebaut werden soll.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="steps" outpath="bench_run">
      <comment>A full step example</comment>

      <parameterset name="param_set">
         <parameter name="number" type="int">1,2</parameter>
         <parameter name="text">Hello,World</parameter>
      </parameterset>
      <parameterset name="export_set">
         <parameter name="EXPORT_ME" export="true">VALUE</parameter>
      </parameterset>

      <step name="first_step">
         <use>param_set</use>
         <do>echo $number</do>
         <do>echo $text</do>
      </step>

      <step name="second_step" iterations="2" depend="first_step">
         <use>export_set</use>
         <do>export SHELL_VAR=Hello</do>
      </step>
   </benchmark>
</jube>
```

Listing 3.4: Beispiel einer Eingabedatei

Das Beispiel deklariert zunächst ein Parameterset mit dem Namen *param_set* und den beiden Parametern *number* und *text*. Diese haben den Wert *1*, oder *2* bzw. *Hello* oder *World*. Ein zweites Parameterset mit dem Namen *export_set* und dem Parameter *EXPORT_ME* mit dem Wert *VALUE*. Das *export*-Attribut ist auf *true* gesetzt, wodurch dieser Parameter in die Shell-Umgebung exportiert wird.

Zusätzlich zu den beiden Parametersets werden zwei Ausführungsschritte mit den Namen *first_step* und *second_step* deklariert. Der Ausführungsschritt *first_step* ver-

wendet das Parameterset *param_set* und soll mit Hilfe der beiden <*do*>'s die Werte der Parameter ausgeben. Der Ausführungsschritt *second_step* ist abhängig vom Ausführungsschritt *first_step*, d.h. JUBE stellt sicher, dass der erste Ausführungsschritt vor dem zweiten Ausführungsschritt ausgeführt wird. Außerdem soll der zweite Ausführungsschritt insgesamt zweimal ausgeführt werden (*iterations="2"*) und erhält somit die doppelte Anzahl an Arbeitspaketen, da diese doppelt ausgeführt werden. Darüber hinaus verwendet der zweite Ausführungsschritt das Parameterset *export_set* und soll die Variable *SHELL_VAR* mit dem Wert *Hello* in die Shell-Umgebung exportieren.

Am Beispiel der Eingabedatei aus Listing 3.4 und den daraus resultierenden Arbeitspaketen wird im Folgenden der Aufbau der Workpackage-Datei Schritt für Schritt erläutert.

Da der Ausführungsschritt *first_step* zwei Parameter mit jeweils zwei Werten verwendet, ergeben sich für diesen Ausführungsschritt insgesamt vier Arbeitspakete, eines pro Parameterkombination. Die jeweiligen Parameterkombination lauten (*1*, *Hello*), (*1*,*World*), (*2*, *Hello*) und (*2*, *World*).

Listing 3.5 zeigt die Speicherung des ersten Arbeitspaketes des Ausführungsschrittes *first_step* innerhalb der Workpackagedatei.

```
<workpackage id="0">
   <step iteration="0" cycle="0">first_step</step>
   <parameterset>
      <parameter name="number" type="int" separator="," duplicate="none"
          mode="text">
         <value>1,2</value>
         <selection idx="0">1</selection>
      </parameter>
      <parameter name="text" type="string" separator="," duplicate="none"
          mode="text">
         <value>Hello,World</value>
         <selection idx="0">Hello</selection>
      </parameter>
   </parameterset>
   <parents/>
   <iteration_siblings>0</iteration_siblings>
   <environment/>
</workpackage>
```

Listing 3.5: Ausschnitt der Workpakagedatei

Die Daten der Arbeitspakete befinden sich innerhalb eines <*workpackage*>-Tags mit aufsteigender ID, das wiederum die Informationen zum Ausführungsschritt durch das <*step*>-Tag enthält. Die Werte und Bedeutungen der *iteration*- und *cycle*-Attribute der Arbeitspakete werden im Zusammenhang mit dem zweiten Ausführungsschritt näher erläutert.

Zudem werden innerhalb des <*workpackage*>-Tags die Parameter der verwendeten Parametersätze in einem <*parameterset*>-Tag ausgeschrieben. Die Parameter erhalten den Namen und die Werte, die in der Eingabedatei angegeben sind oder die Standardwerte für die *type*, *separator*, *duplicate* und *mode*-Attribute. Innerhalb des Parameters gibt es das <*value*>-Tag, welches alle Parametervarianten enthält, und das <*selection*>-Tag, das den ausgewählten Wert und seinen Index enthält. Im Beispiel (Listing 3.5) wurde für das erste Arbeitspaket der Wert *1* für den Parameter *number* und der Wert *Hello* für den Parameter *text* gewählt.

Außerdem enthält das <*workpackage*>-Tag den <*parents*>-, <*iteration_siblings*>- und <*environment*>-Tag, die hier nur kurz vorgestellt und am Beispiel des nächsten Ausführungsschrittes näher erläutert werden. Der <*parents*>-Tag speichert die Abhängigkeiten zu anderen Arbeitspaketen. Der <*iteration_siblings*>-Tag enthält die IDs der Arbeitspakete, die Wiederholungen dieses Arbeitspaketes sind (*iterations*-Attribut des Ausführungsschrittes). Da für den *first_step*-Schritt auch keine weiteren Iterationen ausgeführt werden sollen, steht innerhalb des <*iteration_siblings*>-Tags nur die ID des eigenen Arbeitspakets. Da keine Parameter in die Shell-Umgebung exportiert wurden, ist auch der <*environment*>-Tag leer.

Die Struktur der anderen drei Arbeitspakete desselben Ausführungsschritt ist nahezu identisch, abgesehen von der aufsteigenden ID und der entsprechenden Parameterauswahl.

Nach der Erläuterung der Arbeitspakete des ersten Ausführungsschrittes werden nun die Arbeitspakete des zweiten Ausführungsschrittes (*second_step*) näher betrachtet. Zunächst besitzt dieser Ausführungsschritt einen *iterations*-Wert von 2, d.h. er wird insgesamt zweimal pro Durchlauf ausgeführt. Außerdem besitzt dieser Ausführungsschritt eine Abhängigkeit zum Ausführungsschritt *first_step*. Da dieser insgesamt viermal mit den jeweiligen Daten der Arbeitspakete ausgeführt wurde, wird der Ausführungsschritt *second_step* in Abhängigkeit von jedem dieser Arbeitspakete ausgeführt und für jedes zweimal wiederholt. Auf diese Weise entstehen acht Arbeitspakete dieses Ausführungsschrittes, die ebenfalls in die Workpackagedatei übertragen werden.

Das Listing 3.6 zeigt die Beschreibung der ersten beiden Arbeitspakete des Ausführungsschrittes *second_step* innerhalb der Workpackagedatei.

```
<workpackage id="4">
   <step iteration="0" cycle="0">second_step</step>
   <parameterset>
      <parameter name="EXPORT_ME" type="string" separator=","
          duplicate="none" mode="text" export="true">
        <value>VALUE</value>
      </parameter>
   </parameterset>
   <parents>0</parents>
   <iteration_siblings>4,5</iteration_siblings>
   <environment>
      <env name="SHELL_VAR">'Hello'</env>
      <env name="EXPORT_ME">'VALUE'</env>
   </environment>
</workpackage>
<workpackage id="5">
   <step iteration="1" cycle="0">second_step</step>
   <parameterset>
      <parameter name="EXPORT_ME" type="string" separator=","
          duplicate="none" mode="text" export="true">
        <value>VALUE</value>
      </parameter>
   </parameterset>
   <parents>0</parents>
   <iteration_siblings>4,5</iteration_siblings>
   <environment>
      <env name="SHELL_VAR">'Hello'</env>
      <env name="EXPORT_ME">'VALUE'</env>
```

```
    </environment>
</workpackage>
```

Listing 3.6: Weiterer Ausschnitt der Workpakagedatei

Diese Arbeitspakete werden ab der ID vier aufsteigend nummeriert, da die ID null bis drei durch die Arbeitspakete des Ausführungsschrittes *first_step* belegt sind. Die nachfolgenden Arbeitspakete erhalten eine aufsteigende ID.

Da das Arbeitspaket mit der ID 4 die erste Wiederholung und das Arbeitspaket mit der ID 5 die zweite Wiederholung des gesamten Ausführungsschrittes ist, enthält das *iteration*-Attribut des <*step*>-Tags die entsprechenden Werte *0* und *1*. Das *cycle*-Attribut, das die Wiederholung der Shell-Befehle angibt, enthält die Anzahl der tatsächlich ausgeführten Wiederholungen der Shell-Befehle. Da in der beispielhaften Eingabedatei (Listing 3.4) kein Wert für das *cycles*-Attribut gesetzt wurde, werden keine Wiederholungen der Shell-Befehle ausgeführt und das *cycle*-Attribut des <*step*>-Tags bekommt den Wert 0 zugewiesen.

Wie bereits beim Ausführungsschritt *first_step* beschrieben, folgt nach dem <*step*>-Tag die Deklaration der Parameter. Da der Parameter *EXPORT_ME* nur einen möglichen Wert hat, gibt es keinen <*selection*>-Tag. Das zusätzliche *export*-Attribut wurde ebenfalls in das Arbeitspaket aufgenommen.

Nach der Spezifikation der Parameter folgt die Beschreibung der Abhängigkeiten zu anderen Arbeitspaketen. Da der Ausführungsschritt *second_step* eine Abhängigkeit zum Ausführungsschritt *first_step* besitzt, stammen die Arbeitspakete mit den IDs 4 und 5 (Arbeitspakete des *second_step*) vom Arbeitspaket 0 (Arbeitspaket des *first_step*) ab. Daher wird in den Arbeitspaketen mit den IDs 4 und 5 mit Hilfe des <*parents*>-Tags die ID des übergeordneten Arbeitspakets, also für diesen Fall 0, angegeben. Da es sich bei den Arbeitspaketen mit der ID 4 und ID 5 lediglich um Wiederholungen der gleichen Parameterauswahl handelt, werden diese durch den <*iteration_siblings*>-Tag als "Geschwister" definiert, indem dort die IDs der beiden Arbeitspakete als Wert eingetragen werden.

Abschließend werden innerhalb des <*workpackage*>-Tags die Umgebungsvariablen durch das <*environment*>-Tag beschrieben. Die Eingabedatei im Listing 3.4 zeigt zwei Möglichkeiten um eine Variable in die Shell-Umgebung zu exportieren. Zum einen durch das *export*-Attribut eines Parameters, zum anderen durch den *export*-Befehl innerhalb eines <*do*>-Tags. Die beiden so exportierten Variablen werden innerhalb des <*environment*>-Tags durch <*env*>-Tags definiert. Der <*env*>-Tag wird für Nicht-Betriebssystemvariablen verwendet, wie z.B. die Variablen *SHELL_VAR* und *EXPORT_ME*, die im Beispiel 3.6 zu sehen sind. Neben dem <*env*>-Tag gibt es noch den <*nonenv*>-Tag innerhalb des <*environment*>-Tags, der in diesem Beispiel nicht vorkommt. Es beschreibt die Betriebssystemvariablen, die in die Shell-Umgebung exportiert werden.

Der Vollständigkeit halber ist die resultierende Workpackagedatei aus der in Listing 3.4 dargestellten Eingabedatei im Anhang unter Listing A.1 zu finden.

3.4. Erstellen, Auslesen und Aktualisieren der Datenhaltung

In den vorangegangenen Unterkapiteln wurde die allgemeine Funktionsweise der Kommandozeilenbefehle des Workflow-Tools JUBE und der Aufbau der Konfigurations- und Workpackagedatei erläutert. Da die Dateien bei bestimmten Befehlen wie *run* oder *continue* erstellt oder aktualisiert werden, wird in diesem Kapitel analog zum Kapitel 3.1

die Vorgehensweise des Erstellens, des Lesens und des Aktualisierens der Datenhaltung anhand der Kommandozeilenbefehlen von JUBE erläutert.

Im Laufe der Ausführung des *run*-Befehls wird, wie bereits erläutert, die in Abbildung 3.1 dargestellte Ordnerstruktur und damit die Workpackage- und Konfigurationsdatei erzeugt. Da für die Erstellung dieser Dateien die Daten aus der Eingabedatei benötigt werden, extrahiert JUBE zunächst die Informationen aus der Eingabedatei mit Hilfe des *ElementTree*-Pakets. Die genaue Struktur der programminternen Speicherung dieser Informationen und die Umsetzung in die Datenhaltung wird im Folgenden erläutert.

Um die Informationen der angegebenen Sets, Ausführungsschritte etc. strukturiert abzulegen, werden programmintern Objekte der gleichnamigen Klassen angelegt, die die in den Kapiteln 3.2 und 3.3 aufgeführten Attribute speichern können. Darüber hinaus verfügen diese Klassen über Methoden, um z.B. die Daten in das XML-Format zu schreiben oder, spezifisch für den Analyser, die Ausführungsschritte mit den zugehörigen Patternsets zu analysieren. Unter anderem wird ein Objekt der *Benchmark*-Klasse erstellt, die zur Repräsentation eines Workflows verwendet wird. Diese speichert sowohl die Attribute wie Name, Kommentar etc. als auch die zugehörigen Sets, Ausführungsschritte, Analyser und Ergebnisausgaben.

Bei der Erzeugung dieser Objekte bzw. bei der Speicherung der Attribute der Objekte treten einige Sonderfälle auf, die im Folgenden näher erläutert werden. Der erste Sonderfall tritt bei den Sets auf. Wie bereits im Kapitel 3.2 erwähnt, werden globale und lokale Sets eines Workflows zusammengefasst, da die globalen Sets lediglich dazu dienen, die Eingabedatei kompakter zu schreiben, wenn mehrere Workflows innerhalb einer Datei definiert sind und diese zumindest teilweise auf denselben Daten basieren. Auch die Klasse der Sets unterscheidet nicht zwischen globalen und lokalen Sets. Es wird also für jedes Set, egal ob global oder lokal, ein Objekt der jeweiligen Setklasse erzeugt, das dem zugehörigen Benchmark-Objekt zugeordnet wird. Ein weiterer Sonderfall ist die Speicherung der *iofiles* und *subs* der Substitutionssets. Die Klasse Substitutionsset speichert alle spezifizierten *iofiles* in einer Liste. Die Spezifikation eines einzelnen *iofiles*, d.h. eines Elements dieser Liste, wird durch ein Tupel mit den Werten *in*, *out* und *outmode* gespeichert. Die Daten von *subs* werden durch ein Python-Dictionary gespeichert, welches *source* als Schlüssel und *dest* als zugehörigen Wert enthält.

Nach der Extraktion der Daten und der Erzeugung der Objekte erfolgt die eigentliche Funktionalität des *run*-Befehls, die Ausführung der Workflows. Dies wird durch die *new_run()*-Methode[1] der Benchmark-Klasse realisiert.

Zuerst werden die benötigten Verzeichnisse angelegt. Wenn das Verzeichnis, das als Ausgabeordner für die Workflows definiert wurde, noch nicht existiert, wird dieses Verzeichnis mit dem angegebenen Namen erstellt. Innerhalb dieses Verzeichnisses wird dann für jeden in der Eingabedatei definierten Workflow ein Ordner angelegt. Der Name des Ordners entspricht der jeweiligen Workflow-ID (siehe Abbildung 3.1). Für jeden dieser Workflow-Ordner wird zunächst die Konfigurationsdatei entsprechend der oben beschriebenen Struktur erstellt. Die Erstellung der Konfigurationsdatei erfolgt über die *write_benchmark_configuration()*-Methode[2] der Benchmark-Klasse. Diese speichert zunächst die Version und, falls vorhanden, die spezifizierten JUBE-Tags

[1] https://github.com/FZJ-JSC/JUBE/blob/REL-2.5.1/jube2/benchmark.py#L560
[2] https://github.com/FZJ-JSC/JUBE/blob/REL-2.5.1/jube2/benchmark.py#L787

im XML-Format. Anschließend werden die Workflow-spezifischen Daten *comment*, *name*, *file_path_ref* und *outpath* sowie die Daten der zu diesem Workflow gehörenden Sets, Ausführungsschritte, Analysatoren und Ergebnisausgaben im XML-Format gespeichert. Alle Daten werden dann in eine Datei mit dem Namen *configuration.xml* geschrieben.

Nach dem Schreiben der Konfigurationsdatei und im weiteren Ablauf der *new_run()*-Methode der Benchmark-Klasse wird im Workflow-Objekt ein Python-Dictionary angelegt, das zu den jeweiligen Namen der Ausführungsschritte eine Liste der zugehörigen Workpackage-Objekte speichert. Um die Workpackage-Objekte zu erzeugen, wird für jeden Ausführungsschritt die *create_workpackages()*-Methode[1] der Step-Klasse aufgerufen und die aus dem Ausführungsschritt resultierenden Workpackage-Objekte zurückgegeben.

Nach der Erstellung der Workpackages wird die Workpackagedatei mit Hilfe der *write_workpackage_information()*-Methode[2] der Benchmark-Klasse erstellt. Die Methode schreibt zunächst für jedes Arbeitspaket die zugehörigen Informationen im XML-Format aus. Die Informationen zu einem Arbeitspaket umfassen, wie bereits im Kapitel 3.3 beschrieben, das *id*-Attribut, die Informationen zum zugehörigen Ausführungsschritt, die Informationen zu den verwendeten Parametersätzen, den abhängigen Arbeitspaketen und den Umgebungsvariablen. Diese Daten werden ebenfalls in eine Datei mit dem Namen *workpackage.xml* geschrieben.

Im weiteren Verlauf der Workflow-Ausführung wird die *run()*-Methode[3] der Workflow-Klasse aufgerufen. Diese stellt die zuvor erzeugten Workpackage-Objekte in eine Warteschlange und prüft, ob diese bereits ausgeführt wurden. Ist dies nicht der Fall, wird die Ausführung seriell oder parallel durchgeführt. Dabei können sich einige Parameter durch den im Kapitel 2.1.3 erwähnten *update_mode* ändern, was zu Änderungen der Workpackage-Objekte während der Laufzeit führt. Daher werden die Informationen in der erstellten Workpackagedatei am Ende der Ausführung jedes Workpackages aktualisiert. Da die Aktualisierung bestimmter Informationen aufgrund des XML-Formats nicht möglich ist, muss die gesamte Datei mit der oben beschriebenen *write_workpackage_information()*-Methode der Benchmark-Klasse neu geschrieben werden. Die Daten in der Konfigurationsdatei bleiben hingegen unverändert.

Ein weiterer Befehl, der die Datenhaltung aktualisiert, ist der *continue*-Befehl. Bei seiner Ausführung wird zunächst der Ordner mit der angegebenen ID oder, wenn keine ID angegeben ist, der Ordner mit der letzten Workflow-ID gesucht. Aus diesem Ordner werden die Workflow-Informationen der Konfigurationsdatei extrahiert und die programminterne Struktur mit den Objekten der gleichnamigen Klasse neu aufgebaut. In gleicher Weise werden anschließend die Daten aus der Workpackagedatei gelesen und die Workpackage-Objekte der zugehörigen Ausführungsschritte erzeugt. Nach der Extraktion der Daten wird die *run()*-Methode wie oben beschrieben ausgeführt und die Workpackagedatei wie beschrieben aktualisiert.

Auch für den *analyse*-Befehl muss zunächst der Ordner mit den gewünschten Workflow-Daten gesucht werden. Anschließend werden die Daten aus der Konfigurationsdatei und der Workpackagedatei extrahiert, um die benötigten Objekte zu erzeugen. Ist

[1] https://github.com/FZJ-JSC/JUBE/blob/REL-2.5.1/jube2/step.py#L215
[2] https://github.com/FZJ-JSC/JUBE/blob/REL-2.5.1/jube2/benchmark.py#L819
[3] https://github.com/FZJ-JSC/JUBE/blob/REL-2.5.1/jube2/benchmark.py#L597

eine Update-Datei angegeben, so werden die zugehörigen Objekte der Patternsets, Analyser und Ergebnisausgaben mit den neuen Daten aktualisiert bzw. neu erzeugt und die Daten in der Konfigurationsdatei mit der *write_benchmark_configuration()*-Methode der Benchmark-Klasse neu geschrieben. Wird keine Aktualisierungsdatei angegeben, werden die Originaldaten verwendet. Anschließend werden die angegebenen Ausführungsschritte mit Hilfe der Patternsets analysiert. Abschließend werden die Ergebnisse der Analyse ebenfalls im XML-Format in eine Datei geschrieben.

Der *result*-Befehl sucht gleichermaßen zunächst das entsprechende Workflow-Verzeichnis und extrahiert die Daten aus der Konfigurations- und Workpackagedatei sowie die Ergebnisse der Analyse. Auch hier können die aktuellen Objekte der Patternsets, Analyser und Ergebnisausgaben durch Angabe einer Update-Datei aktualisiert bzw. neu erzeugt werden. In diesem Fall muss die Konfigurationsdatei mit der *write_benchmark_configuration()*-Methode der Benchmark-Klasse neu geschrieben werden. Die Ergebnisse werden im angegebenen Format ausgegeben und im *result*-Ordner gespeichert.

Der *remove*-Befehl bietet die Möglichkeit einzelne Workpackages oder ganze Workflows zu löschen. Beim Löschen des Workflows wird der Ordner mit der angegebenen Workflow-ID gesucht und das gesamte Verzeichnis gelöscht. Beim Löschen eines oder mehrerer spezifischer Workpackage-Objekte werden die spezifischen Workpackage-Objekte aus dem Workflow-Objekt entfernt und die Workpackagedatei wird ohne die gelöschten Workpackage-Objekte mit der *write_workpackage_information()*-Methode der Benchmark-Klasse neu geschrieben.

Beim Editieren des Kommentars mit dem *comment*-Befehl werden auch die Daten der Konfigurationsdatei aus dem gewünschten Workflow-Ordner geladen. Der Kommentar des Workflow-Objekts wird angepasst und anschließend die Konfigurationsdatei mit dem neuen Kommentar über die *write_benchmark_configuration()*-Methode der Benchmark-Klasse neu geschrieben.

Der *info*- und *status*-Befehl laden die Daten der Konfigurations- und Workpackagedatei im angegebenen Ordner und lesen die auszugebenden Informationen und den Status aus den erzeugten Workflow- und Workpackage-Objekten aus.

Der *log*-Befehl durchsucht den angegebenen Ordner nach Protokolldateien und gibt diese aus.

3.5. Problemanalyse

Um darzulegen, warum die Datenhaltung ersetzt werden sollte, werden in diesem Unterkapitel die Nachteile und Probleme der derzeitigen Datenhaltung erläutert. So kann im nächsten Kapitel auf die gestellten Anforderungen eingegangen werden.

Die in den Kapiteln 3.2 und 3.3 beschriebene Datenstruktur der Konfigurations- und Workpackagedatei weist zunächst einige Nachteile hinsichtlich der Aufbau der Daten auf. Zum einen ist festzustellen, dass einige Attribute der beispielsweise Parametersets oder Ausführungsschritte eines Workflows in beiden Dateien und damit doppelt ge-

speichert werden. Das Schreiben und Lesen der mehrfach gespeicherten Daten kostet Laufzeit und Speicherplatz. Zudem kann es durch Abhängigkeiten der Ausführungsschritte, wie im Kapitel 3.3 beschrieben, dazu kommen, dass mehrere Workpackages die gleichen Daten speichern, was ebenfalls zu einem erhöhten Speicherplatzbedarf führt. Außerdem werden alle Werte der Attribute sowohl in der Eingabedatei als auch in der Konfigurations- und der Workpackagedatei als Text und nicht in einem passenden Datentyp gespeichert. Dies hat zur Folge, dass beim späteren Ein- und Auslesen diese Werte wieder in den richtigen Datentyp umgewandelt werden müssen, was zu einem erhöhten Programmier- und Zeitaufwand führt. Darüber hinaus ermöglicht das im Kapitel 2.1.2 beschriebene, einfach zu handhabende XML-Format eine vereinfachte Manipulation durch den Benutzer mittels Texteditor. Fehlerhafte Manipulationen können die Datenspeicherung für JUBE unlesbar machen, sodass Workflows unbrauchbar gemacht werden und neu ausgeführt werden müssen.

Auch aus dem vorherigen Kapitel 3.4 werden Nachteile durch die Erstellung, Auslesung und Aktualisierung der Datenhaltung erkennlich.

Zunächst ist beim Erstellen der XML-Dateien zu sehen, dass die gesamte baumartige Struktur mit Hilfe des *ElementTree*-Pakets aufwendig erzeugt werden muss, um diese dann auszuschreiben. Dieser Punkt sorgt für erhöhte Laufzeit und weiteren Speicherplatzbedarf. Auch beim Einlesen der Daten muss das XML-Format in eine baumartige Struktur formatiert werden und einzelne Attribute müssen innerhalb dieser Struktur gesucht und, wie oben beschrieben, in den passenden Datentyp formatiert werden. Das sorgt für erhöhten Zeit- und Speicheraufwand. Außerdem werden die Dateien nicht in einzelnen Punkten aktualisiert, da dies aufgrund der hierarchischen Struktur sehr zeitaufwendig und nicht praktikabel wäre. Da es sich um eine textbasierte Datenhaltung in XML handelt, wird die gesamte Datei neu geschrieben. Auch dies führt zu längeren Laufzeiten.

Zusammenfassend lässt sich sagen, dass sowohl die Art der textbasierten Datenhaltung, als auch die Struktur der Datenhaltung durch das XML-Format und dessen Aufbau zur Speicherung der Daten einige Nachteile mit sich bringt, bei denen im Folgenden evaluiert wird, ob sie durch eine andere Datenhaltung behoben werden können.

4. Anforderungen und Zielsetzung

Ziel dieses Kapitels ist es, die Anforderungen an die neue Datenhaltung herauszuarbeiten, um im nächsten Kapitel die möglichen Datenhaltungsoptionen auf ihre Praxistauglichkeit zu validieren und auszuwählen. Darüber hinaus sollen die hier erarbeiteten Anforderungen in das im Kapitel 5 beschriebene Datenbankentwurfs- und Integrationskonzept in JUBE einfließen.

Die Anforderungen sind in Anlehnung an die im Kapitel 2.3 beschriebene ISO-Norm 25010 strukturiert. Diese Norm umfasst acht Merkmale: funktionale Anforderungen, Leistung, Sicherheit, Kompatibilität, Zuverlässigkeit, Benutzerfreundlichkeit, Wartbarkeit und Übertragbarkeit. [10]

4.1. Funktionale Anforderungen

Funktionale Anforderungen sind spezifische Beschreibungen der Funktionen und des Verhaltens, die die Datenhaltung erfüllen muss, um die gewünschte Funktionalität zu bieten. In diesem Unterkapitel werden diese funktionalen Anforderungen an die neue Datenhaltung definiert.

Dazu werden zunächst die Informationsanforderungen, d.h. die Daten, die in der Datenhaltung vorhanden sein müssen, definiert, um darauf aufbauend die Anforderungen an die Eingabe dieser Daten in die Datenhaltung zu erläutern. Des Weiteren werden die Anforderungen, die sich bei der Verarbeitung der Daten ergeben, z.B. bei der Aktualisierung oder Löschung von Daten in der Datenhaltung, näher beschrieben. Abschließend werden die Integritätsanforderungen an die Datenhaltung, d.h. die funktionalen Anforderungen an die Genauigkeit, Konsistenz und Gültigkeit der Daten vertieft.

4.1.1. Informationsanforderungen

Die Informationsanforderung an eine Datenhaltung beschreibt die Art der Dateninhalte, die gespeichert werden sollen. Somit wird der Umfang und die Struktur der Daten erfasst.

Die bereits in den Kapiteln 3.2 und 3.3 erläuterten Inhalte und Strukturen der Daten der aktuellen Datenhaltung werden auch in der neuen Datenhaltung benötigt. Im Folgenden wird kurz auf diese Daten eingegangen.

Die in Kapitel 3.2 beschriebene Konfigurationsdatei enthält die Daten des Workflows. Für diesen wird die ID, der Name, der Kommentar, der Pfad zum Ordner der Workflows und das in Kapitel 3.2 beschriebenen *file_path_ref*-Attribut benötigt. Zusätzlich müssen die JUBE-Tags, die für diesen Workflow genutzt wurden, gespeichert werden.

Die Sets, die in diesem Workflow definiert wurden, müssen jeweils einen Namen speichern. Zusätzlich speichert das Parameterset das *duplicate*-Attribut.

Die Elemente der Sets speichern die in Kapitel 3.2 beschriebenen Attribute. Für die Parameter des Parametersets sind das das *name*-, *seperator*-, *type*-, *mode*-, *update_mode*-, *export*-, *duplicate*-Attribut und den Inhalt des Parameters als *value*. Die Pattern des Patternsets speichern das *name*-, *mode*-, *type*-, *unit*-, *dotall*-, *dafault*-Attribut und den Inhalt des Patterns als *value*. Die Files des Filesets speichern das *path*-, *source_dir*-, *target_dir*-, *name*-, *file_path_ref*-, *active*-, *is_internal_ref*- und *type*-Attribut.

Außerdem müssen die Substitutionsfiles und die Substitutionen der Substitutesets gespeichert werden. Die Substitutionsfiles speichern das *in*-, *out*- und *out_mode*-Attribut und die Substitutionen das *source*- und *dest*-Attribut.

Zudem müssen die Werte eines Ausführungsschrittes gespeichert werden. Ein Ausführungsschritt muss das *name*-, *depend*-, *iterations*-, *work_dir*-, *export*-, *max_async*-, *active*-, *suffix*-, *cycles*-, *procs*-, *do_log_file*- und *shared*-Attribut speichern. Zusätzlich müssen die genutzten Parameter-, File- und Substitutionssets gespeichert werden.

Die zu den Ausführungsschritten zugehörigen <*do*>-Tags können als Operationen beschrieben werden und speichern das *error_filename*-, *async_filename*-. *break_filename*-, *stdout_filename*-, *stderr_filename*-, *actived*-, *shared*- und *work_dir*-Attribut. Der Shell-Befehl einer Operation kann unter dem *do*-Attribut gespeichert werden.

Außerdem sollten die Analyser und die Ergebnisausgaben eines Workflows gespeichert werden. Die Analyse muss den Namen, die genutzten Patternsets und das *reduce*-Attribut speichern. Zudem muss gespeichert werden welcher Ausführungsschritt mit welcher Analyse-Datei analysiert wird und den Pfad dieser Analyse-Datei und ob und welche weitere Patternsets von der Analyse-Datei genutzt werden.

Für die Ergebnisausgabe eines Workflow sollten zunächst allgemein dessen Name, die genutzten Analyser und den Pfad zur Speicherung der Ergebnisausgaben gespeichert werden. Für die Tabellenausgabe muss der Name der Tabelle und das *style*-, *separator*-, *transpose*- und *sort*-Attribut gespeichert werden. Zudem muss der Namem das Format, der Titel und der Inhalt der zu der Tabelle gehörigen Spalten gespeichert werden. Für eine Datenbankausgabe muss der Name der Datenbank, die Primärschlüssel und der Dateiname gespeichert werden. Die Ergebnisausgabe über das Syslog muss das *name*-, *address*-, *host*-, *port*-, *sort*- und *format*-Attribut speichern. Die zum Syslog und zur Datenbank gehörigen Schlüssel müssen das *format*- und *title*-Attribut und den zueghörigen Pattern- oder Parameternamen speichern.

Zusätzlich zu den Werten die aufgrund der Konfigurationsdatei gespeichert werden müssen, müssen auch die Inhalte der Workpackagedatei gespeichert werden. Wie in Kapitel 3.3 bereits zu sehen ist, speichert diese Daten zu den Arbeitspaketen, den Ausführungsschritten, den Parametern und den Umgebungsvariablen.

Zunächst müssen daher das *ID*-, *iteration_sibling*- und *parents*-Attribut für ein Arbeitspaket gespeichert werden. Zudem sollte gespeichert werden zu welchem Ausführungsschritt dieses Arbeitspaket gehört, in der wievielte *iteration* sich dieses gerade befindet und wieviele *cycles* ausgeführt wurden.

Zudem werden in der Workpackagedatei die Daten der genutzten Parameter beschrieben. Da diese bereits gespeichert werden, muss nur noch die Information über den genutzten Parameterinhalt und dessen Index gespeichert werden.

Für die Umgebungsvariablen des Arbeitspaketes muss das *name*- und *value*-Attribut gespeichert werden. Zudem kann mit Hilfe eines *env*-Attributes gespeichert werden, ob es sich um eine Betriebssystem- oder Nicht-Betriebssystemvariable handelt.

Ausgehend von den hier beschriebenen Attributen für Sets, Ausführungsschritte, Analyser, Ergebnisausgaben und Arbeitspakete muss die neue Datenhaltung die Möglichkeit haben, die Daten in geeigneter Weise zu strukturieren und mit Hilfe von Verweisen Relationen speichern zu können. Beispielsweise kann so festgelegt werden, welcher Parameter zu welchem Parameterset oder welches Arbeitspaket zu welchem Ausführungsschritt gehört.

4.1.2. Dateneingabeanforderungen

Die Dateneingabe beschreibt die Erfassung und Eingabe von Daten in das Datenhaltungssystem. Die Anforderungen an die Dateneingabe legen fest, wie die Daten erfasst, strukturiert, validiert und gespeichert werden müssen.

Die erstmalige Datenerfassung durch den *run*-Befehl erfolgt aktuell durch einen XML-Parser in JUBE, der die Daten aus der Eingabedatei extrahiert. Die Daten sind bereits durch die vorgegebene Struktur der Eingabedatei strukturiert und werden ebenfalls durch den XML-Parser auf gültige Strukturierung geprüft und als strukturiertes Objekt einer Klasse zurückgegeben. Die Strukturierung und Validierung der Daten ist also bereits durch den weiterhin zu verwendenden XML-Parser gegeben.

Bei der Speicherung dieser bereits strukturierten und validierten Daten ist auf die Einhaltung der Datentypen zu achten. Dazu muss sowohl die programminterne Speicherung der Daten als auch die Speicherung in der neuen Datenhaltung an die jeweiligen Datentypen angepasst werden.

Um schließlich die Datenspeicherung durchführen zu können, muss zunächst die Datenhaltung erstellt und die gewünschten Daten eingefügt werden. Das Einfügen der Daten erfordert die Implementierung einer oder mehrerer Funktionen, die die extrahierten Objekte des XML-Parsers entsprechend der Datenhaltung hinzufügen können.

Für die Anforderungen an die Dateneingabe ist nicht nur die erstmalige Datenerfassung durch den *run*-Befehl relevant, sondern auch die Erfassung von Daten in bereits bestehenden Datenhaltungen, z.B. durch den *continue*- oder *analyse*-Befehl. Die notwendige Handhabung für den Fall, dass die neue Datenhaltung bereits vorhanden ist, wird im Kapitel 4.1.3 näher erläutert.

Sollte jedoch die bisherige Datenhaltung über XML-Dateien noch vorhanden sein, die Anwendung JUBE aber bereits auf die neue Datenhaltung umgestellt worden sein, so muss auch die Kompatibilität mit der alten Datenhaltung gewährleistet sein. D.h. die Daten müssen aus der vorhandenen Konfigurations- bzw. Workpackagedatei ausgelesen und in die neue Datenhaltung überführt werden können. Das Auslesen der Daten erfolgt derzeit ebenfalls über den XML-Parser, wodurch die Strukturierung und Validierung dieser Daten gegeben ist. Das Auslesen und Speichern in der neuen Datenhaltung führt somit zu einer Überführung der alten Datenhaltung in die neue Datenhaltung. Diese sogenannte Versionskompatibilität zwischen der Version der XML-Datenhaltung und der Version der neuen Datenhaltung wird im Kapitel 4.2.3 näher erläutert.

Die Anforderungen an die Dateneingabe erfordern daher die Implementierung von Funktionen zur Erstellung der Datenhaltung, zur Speicherung der Datensätze und zur Integration bestehender Daten in die neue Datenhaltung.

4.1.3. Datenverarbeitungsanforderungen

Die grundlegenden Datenverarbeitungen können durch sogenannte CRUD-Operationen beschrieben werden, die in diesem Kapitel definiert werden. CRUD steht für Create, Read, Update und Delete, also das Anlegen, Lesen, Aktualisieren und Löschen von Daten. Da das Anlegen bereits im vorherigen Unterkapitel behandelt wurde, wird in diesem Kapitel nur auf die anderen Operationen eingegangen.

Das Auslesen der Daten aus der bestehenden Datenhaltung erfolgt beim Fortsetzen eines bestehenden Workflows, der mit einer Version erstellt wurde, die noch nicht auf die neue Datenhaltung umgestellt wurde z.B. durch den *continue* oder *analyse*-Befehl. Dabei ist, wie bereits in Kapitel 4.1.2 erläutert, darauf zu achten, ob in diesem Workflow bereits die neue Datenhaltung integriert ist oder noch die alte Datenhaltung durch die Konfigurations- und Workpackagedatei vorliegt. Ist letzteres der Fall, werden die Daten durch den bisher verwendeten XML-Parser ausgelesen.

Für die neue Datenhaltung sollte eine Funktion implementiert werden, die die Daten aus der neuen Datenhaltung ausliest und in eine strukturierte Form bringt. Damit sollen wie bisher die programminternen Objekte der Klassen erzeugt werden können, die für die weitere Ausführung der Workflows benötigt werden.

Während der Ausführung des Workflows kann es zu Änderungen der Informationen kommen, so dass diese aktualisiert werden müssen. Da die fehlende Aktualisierungsfunktion einer der Hauptpunkte für die Ablösung der aktuellen Datenhaltung ist, soll die neue Datenhaltung eine Funktion zur effizienten Aktualisierung von Datensätzen besitzen. Effizient bedeutet, dass nur die Informationen, die sich geändert haben, aktualisiert werden müssen und nicht der gesamte Datensatz bzw. alle Datensätze.

Schließlich ist auch das Löschen von Datensätzen z.B. mit dem *remove*-Befehl relevant. Beim Löschen eines oder mehrerer Arbeitspakete sollen nicht nur die programminternen Objekte der Arbeitspakete gelöscht werden, sondern auch die zugehörigen Datensätze in der neuen Datenhaltung.

Die sich daraus ergebenden Anforderungen an die Datenverarbeitung erfordern daher die Implementierung von Funktionen zum Auslesen, Aktualisieren und Löschen von Datensätzen, die diese Aufgaben so effizient wie möglich ausführen.

4.1.4. Datenintegritätsanforderungen

Die Datenintegrität bezieht sich auf die Korrektheit, Vollständigkeit und Konsistenz der Daten in einer Datenverwaltung und ist ein wichtiger Aspekt, um die Zuverlässigkeit der Daten zu gewährleisten. Im Folgenden werden die Funktionen der Datenverwaltung beschrieben, die eingehalten werden müssen, um diese Eigenschaften der Daten zu gewährleisten.

Zur Gewährleistung der Korrektheit der Daten, müssen diese den richtigen Datentyp haben, innerhalb ihres Wertebereichs liegen und eindeutig sein. Beispielsweise muss der Name eines Parametersets eindeutig sein, und der *iterations*-Wert eines Ausführungsschritts muss eine Zahl größer als Null sein. Da die Prüfung auf die hier beschriebenen Anforderungen bereits bei der Extraktion der Daten aus der Eingabedatei mit Hilfe des XML-Parsers erfolgt und somit nur gültige Datensätze in die Datenbank aufgenommen werden, ist keine weitere Prüfung erforderlich.

Um die Vollständigkeit der Daten zu gewährleisten, müssen alle im Kapitel 4.1.1 beschriebenen Informationen für alle Datensätze vorhanden sein. Darüber hinaus wurde in diesem Kapitel auch die Verwendung von Referenzen zur Speicherung von Bezie-

hungen behandelt. Diese können nur verwendet werden, wenn für jeden Datensatz in der neuen Datenhaltung auch ein eindeutiges Attribut existiert, das den Datensatz selektierbar macht. Daher ist aus den Informationen im Kapitel 4.1.1 für die Workflows, Ausführungsschritte etc. ein Attribut auszuwählen, das für jeden der Datensätze eindeutig ist, oder ein Attribut in Form einer ID hinzuzufügen, das den Datensatz wiederum eindeutig definiert. Beispielsweise kann der Name eines Parametersatzes als eindeutiges Attribut verwendet werden, da er für einen Workflow eindeutig sein muss.

Um die Daten in der Datenhaltung konsistent zu halten, sollen in der neuen Datenhaltung Transaktionen eingesetzt werden. Transaktionen beschreiben eine Abfolge von Operationen auf der Datenhaltung, die nach fehlerfreier Ausführung die Datenhaltung in einem konsistenten Zustand hinterlassen bzw. im Fehlerfall den vorherigen, ebenfalls konsistenten Zustand wiederherstellen. Um Konsistenz durch Transaktionen zu erreichen, werden die so genannten ACID-Prinzipien eingehalten. Sie stellen sicher, dass Datenänderungen atomar, konsistent, isoliert und dauerhaft (engl. atomicity, consistency, isolation, and durability) durchgeführt werden, kurz ACID. [15, S. 209-210]

Die Atomarität stellt sicher, dass Transaktionen als eine Einheit betrachtet werden und entweder vollständig oder gar nicht ausgeführt werden. Sollte es also bei den Operationen einer Transaktion zu einem Fehler kommen, wird der Zustand der Datenhaltung vor Beginn der Transaktion wiederhergestellt. Ansonsten werden die Änderungen aller Operationen ausgeführt. Die Datenverwaltung der neuen Datenhaltung muss somit sicherstellen, dass bei Fehlern während einer Transaktion diese abgebrochen wird und die Daten zurückgesetzt werden. [15, S. 209-210]

Um die Konsistenz einer Transaktion zu gewährleisten, muss der Zustand der Datenhaltung am Ende einer Transaktion alle Konsistenzvorschriften erfüllen.

Außerdem müssen die Transaktionen voneinander isoliert sein. Das bedeutet, dass bei parallelen Zugriffen auf die Datenbank ein gegenseitiges Überschreiben oder Löschen von Datensätzen nicht möglich sein darf. Damit sich die Transaktionen nicht gegenseitig beeinflussen, soll ein sogenanntes Sperrverfahren eingesetzt werden, das bei einer laufenden Transaktion den Zugriff auf die Datenhaltung bis zum Abschluss dieser Transaktion sperrt. [15, S. 209-210]

Die dauerhafte Speicherung der Daten nach einer Transaktion soll durch die Verwendung von Protokollen der ausgeführten Operationen gewährleistet werden. So können z.B. im Falle eines Systemabsturzes, der zu einem Verlust der Daten führen würde, die Daten wiederhergestellt werden.

Zusammengefasst soll die Datenintegration durch die Einführung von Primär- und Fremdschlüsseln und die Verwendung von Transaktionen unter Anwendung der ACID-Prinzipien gewährleistet werden.

4.2. Nicht-funktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben im Gegensatz zu den funktionalen Anforderungen (Kapitel 4.1) bestimmte Eigenschaften, Leistungen oder Verhaltensweisen, die die Datenhaltung erfüllen soll und die in diesen Kapiteln in Anlehnung an die bereits beschriebene ISO-Norm 25010 dargestellt werden.

Dazu werden in den folgenden Unterkapiteln die verbleibenden sieben Merkmale

Leistung, Sicherheit, Kompatibilität, Zuverlässigkeit, Benutzerfreundlichkeit, Wartbarkeit und Übertragbarkeit der ISO-Norm 25010 anhand der gewünschten Datenhaltung beschrieben.

4.2.1. Leistung

Die Anforderungen an die Leistung einer Datenhaltung beschreiben im Allgemeinen ein schnelles Zeitverhalten und die sinnvolle Nutzung der vorhandenen Kapazitäten. Für JUBE bedeutet dies, dass die Datenhaltung so zu entwickeln ist, dass sowohl der Speicherplatzbedarf als auch die Laufzeit für die Erstellung und Verwaltung der Datenhaltung minimiert werden.

Ein erster Aspekt zur Minimierung der Laufzeit ist die Entwicklung einer effektiven Aktualisierungsfunktion für die Datenhaltung. Im Kapitel 3 wurde gezeigt, dass aufgrund der fehlenden Aktualisierungsfunktion bei Änderungen einzelner Daten alle Dateien der derzeitigen textbasierten Datenhaltung neu geschrieben werden müssen, was sehr zeitaufwendig ist, da selbst bei Änderung eines einzelnen Attributs die gesamte Datei gelöscht und neu geschrieben werden muss. Daher sollte die Aktualisierungsfunktion der neuen Datenhaltung in der Lage sein, die Daten der Datenhaltung punktuell zu aktualisieren, ohne den gesamten Datensatz oder die gesamte Datenhaltung neu schreiben zu müssen.

Ein weiterer Aspekt für eine effiziente Laufzeit ist die Verwendung einer Datenhaltungsoption, die einen parallelen Zugriff auf die Daten erlaubt. So könnte die im Kapitel 3.4 beschriebene parallele Abarbeitung der Arbeitspakete auch parallel auf die Datenhaltung zugreifen und dort die Daten der Arbeitspakete aktualisieren.

Schließlich kann eine sinnvolle Strukturierung der Daten innerhalb der Datenhaltung eine doppelte Datenhaltung vermeiden und die Komplexität der Datenabfragen reduzieren. Die Strukturierung hängt von der Datenhaltungsoption ab und kann z.B. bei relationalen Datenbanken durch Normalisierung erfolgen.

Die Datenhaltung sollte daher die punktuelle Aktualisierung von Daten, den parallelen Zugriff auf Daten und die einfache Abfrage von Daten durch eine sinnvolle Strukturierung der Datenhaltung ermöglichen, um die Laufzeit und den Speicherplatzbedarf zu minimieren und somit die Datenhaltung effizient betreiben zu können.

4.2.2. Sicherheit

Sicherheit ist ein wichtiger Aspekt der Datenverwaltung, um zu gewährleisten, dass die Daten vor unbefugtem Zugriff, Verlust oder Beschädigung geschützt sind.

Im Hinblick auf den unberechtigten Zugriff spielen Aspekte wie Authentifizierung und Autorisierung eine wichtige Rolle. Zunächst muss es ein sicheres Verfahren zur Identifizierung des Benutzers geben und es muss sichergestellt werden, dass der Benutzer nur auf die für ihn relevanten Daten zugreifen kann. Ein unberechtigter Zugriff kann zum Verlust, Beschädigung oder Diebstahl der Daten führen.

Bei einer serverseitiger Datenhaltung sollte daher eine Identifizierung z.B. durch eine Kombination aus Benutzername und Passwort erfolgen, die sicherstellt, dass die gespeicherten Daten einem bestimmten Benutzer zugeordnet werden können. Zusätzlich kön-

nen Benutzerrollen definiert werden, die den Zugriff auf die Daten und die Ausführung von Aktionen erlauben.

Bei lokaler Datenhaltung können diese Aspekte vernachlässigt werden, da die Authentisierung durch das Betriebssystem erfolgt und die Benutzer somit nur Zugriff auf ihre eigenen bzw. die für sie freigegebenen Daten haben.

Die lokale Datenhaltung verringert zwar die Möglichkeit des Zugriffs durch unbefugte Dritte, erhöht aber die Wahrscheinlichkeit der Manipulation durch den Nutzer selbst. Eine Veränderung der Daten kann durch verschiedene Maßnahmen verhindert werden, z.B. durch eine Zugriffskontrolle mittels Passwort auf die Datenhaltung selbst. Oder die Daten können innerhalb der Datenhaltung verschlüsselt werden, um ebenfalls ein Abfangen oder Manipulieren zu verhindern.

Schließlich kann der Totalverlust durch z.B. Programmabbruch oder Verbindungsfehler zumindest teilweise durch die im Kapitel 4.1.4 beschriebenen Transaktionsprotokolle verhindert werden. Damit kann sichergestellt werden, dass die Daten auf dem vorherigen Stand sind und somit höchstens der abgebrochene Befehl erneut ausgeführt werden muss und nicht der gesamte Lauf neu gestartet werden muss.

Die geforderten Sicherheitsaspekte sprechen in diesem Anwendungsfall für die Beibehaltung der lokalen Datenhaltung, da diese besser vor äußeren Einflüssen geschützt ist und keine neue Identifikationsschnittstelle implementiert werden muss. Darüber hinaus sollten die Daten z.B. durch Passwörter oder Verschlüsselung geschützt und Transaktionsprotokolle eingesetzt werden, um einen Totalverlust der Daten zu verhindern.

4.2.3. Kompatibilität

Die Kompatibilität einer Datenhaltung beschreibt das Zusammenspiel mit anderen Systemen, Datenquellen oder Anwendungen, um einen reibungslosen Datenfluss zu gewährleisten.

Wichtig ist zum einen die Kompatibilität mit den gewünschten Datentypen. Die im Kapitel 4.1.1 beschriebenen Informationen sollen in geeigneten Datentypen gespeichert werden können, so dass eine Formatierung der Informationen wie bisher nicht mehr notwendig ist. Die Datenhaltung sollte daher über geeignete Datentypen für Zahlen, Zeichenketten und Wahrheitswerte verfügen.

Eine Datenhaltung sollte zudem plattformübergreifend sein und somit auf allen eingesetzten Plattformen laufen. Da JUBE sowohl auf Hoechstleistungsrechnern als auch auf lokalen Systemen, wie Laptops, eingesetzt werden kann, ist es wichtig, dass die Datenhaltung auf verschiedenen linuxbasierten Betriebssystemen arbeiten kann. Aus diesem Grund wurde bereits im Kapitel 2.2.2 die Nutzung einer binären Datenhaltung ausgeschlossen.

Die Verwendung von standardisierten Abfragesprachen, wie z.B. SQL für Datenbanken, trägt ebenfalls zur Plattformunabhängigkeit der Datenhaltung bei, da die Syntax und Semantik der Abfragesprache in der Regel unverändert bleibt, unabhängig davon, ob das Betriebssystem Windows, Linux, macOS oder ein anderes ist.

Schließlich sollte die neue JUBE-Version, wie bereits im Kapitel 4.1.3 näher beschrieben, mit der alten Datenhaltung kompatibel, d.h. versionskompatibel sein. Dies stellt sicher, dass Workflows mit der alten Datenhaltung nicht neu aufgebaut werden müssen, sondern dass die alte Datenhaltung insofern unterstützt wird, dass diese in die

neue Datenhaltung überfuehrt werden kann, im Falle der Nutzung von alten Workflows mit der neuen JUBE-Version.

Um die Kompatibilität der neuen Datenhaltung zu gewährleisten, ist daher eine Speicherung der passenden Datentypen, eine plattformunabhängige Datenhaltungsmöglichkeit und Versionskompatibilität erforderlich.

4.2.4. Zuverlässigkeit

Eine zuverlässige Datenhaltung gewährleistet einen kontinuierlichen und fehlerfreien Betrieb. Dabei sind Aspekte wie Verfügbarkeit, Fehlerbehandlung und Wiederherstellbarkeit zu berücksichtigen.

Um die Verfügbarkeit der Datenhaltung zu gewährleisten, müssen die Daten jederzeit zugänglich sein. Daher muss bei einer serverseitigen Datenhaltung mit Ausfallsicherungen bzw. redundanten Systemen gearbeitet werden, während bei einer lokalen Datenhaltung die Verfügbarkeit der Datenhaltung gewährleistet werden kann, solange die entsprechende Plattform der Datenhaltung verfügbar ist.

Ein weiterer sehr relevanter Punkt, der für die lokale Datenhaltung spricht, ist die Unabhängigkeit von einer Netzwerkverbindung. So können die Daten auch ohne Internetverbindung genutzt werden und sind somit freier zugänglich. Dies ist besonders vorteilhaft, da auch die Anwendung von JUBE nicht an eine Netzwerkverbindung gebunden ist und somit auch die Datenhaltung unabhängig davon sein sollte.

Außerdem ist zu beachten, dass bei der Verwendung eines serverseitigen Datenbankmodells Wartungsarbeiten auf dem Server durchgeführt werden müssen, so dass die Benutzer für einen bestimmten Zeitraum keinen Zugriff auf die Funktionalität von JUBE haben.

Zur Zuverlässigkeit der Datenhaltung gehört nicht nur die bereits beschriebene Verfügbarkeit, sondern auch die Fehlertoleranz der Datenhaltung. Die Fehlertoleranz der Anwendung soll durch ein geeignetes Fehlermanagement erhöht werden. Dieses sollte in der Lage sein, Anwendungsfehler frühzeitig zu erkennen und somit vor Datenverlust zu schützen und dem Anwender aussagekräftige Informationen darüber zu liefern.

Schließlich trägt auch die Wiederherstellbarkeit zur Zuverlässigkeit der Datenhaltung bei. Im Falle eines Programmabbruchs ist die Wiederherstellung der vorherigen Daten notwendig, um die Zuverlässigkeit eines Workflows zu gewährleisten und keine falschen Daten zu erzeugen. Dieser Punkt kann durch die bereits in den vorherigen Kapiteln beschriebenen Transaktionsprotokolle erfüllt werden.

Die Zuverlässigkeit der Datenhaltung in JUBE kann somit durch eine lokale Datenhaltung mit entsprechender Fehlerbehandlung und Transaktionsprotokollierung besser gewährleistet werden als durch eine serverseitige Datenhaltung.

4.2.5. Benutzerfreundlichkeit

Die Benutzerfreundlichkeit ist ein weiterer Punkt der nichtfunktionalen Anforderungen im Sinne der ISO 25010 und bezieht sich in der Regel auf optimale Erkennbarkeit, leichte Erlernbarkeit und Bedienbarkeit, sowie leichten Zugang zur neuen Datenhaltung.

Da es sich bei der neuen Datenhaltung um einen Aspekt von JUBE handelt, der für den Benutzer nicht erkennbar oder erlernbar sein soll, gelten diese Punkte für dieses Kapitel für die Entwickler von JUBE. Das bedeutet, dass die neu zu implementierende Datenhaltung für andere Entwickler der JUBE leicht erkennbar und erlernbar sein muss. Zu diesem Zweck wird zusätzlich zu dieser Arbeit eine ausführliche Dokumentation der neu implementierten Datenhaltung von JUBE erstellt und der implementierte Code in angemessenem Umfang kommentiert.

Um die Bedienbarkeit der neuen Datenhaltung für den Anwender zu gewährleisten, sollte darauf geachtet werden, ob und welche weitere Software für die neue Datenhaltung benötigt wird. Um die Installation zusätzlicher Software für den Nutzer zu vermeiden, sollten Pakete verwendet werden, die entweder bereits in JUBE verwendet werden oder in der Standardbibliothek von Python vorhanden sind.

Darüber hinaus sollte wie bisher eine Versionsinformation für die neue JUBE-Version mit der neuen Datenhaltung veröffentlicht werden, die den Benutzer darüber informiert, dass die neue Datenhaltung verwendet wird, jedoch Workflows mit der Datenhaltung über XML-Dateien weiterhin verwendet werden können, da diese in das neue Datenhaltungsformat konvertiert werden.

Schließlich kann zur besseren Fehlererkennung die Verwaltung der neuen Datenhaltung, durch Dateien protokolliert werden. In diesen Dateien werden alle an der Datenhaltung durchgeführten Operationen und die dabei aufgetretenen Fehler aufgelistet, so dass der Benutzer oder Entwickler den Fehler erkennen kann.

Die Erstellung einer Dokumentation, die Kommentierung des Codes, die Verwendung von Python-Paketen, die nicht zusätzlich installiert werden müssen, Versionshinweise und Protokolldateien tragen zur Benutzerfreundlichkeit der neuen Datenhaltung bei und sollten bei der Erstellung der neuen Datenhaltung beachtet werden.

4.2.6. Wartbarkeit

Die Wartbarkeit einer Datenhaltung bezieht sich auf die Einfachheit, mit der eine Datenhaltung gewartet, aktualisiert und erweitert werden kann.

Um die Weiterentwicklung der Datenhaltung zu erleichtern, sei es die Wartung, Aktualisierung oder Erweiterung der Datenhaltung, sollte eine Schnittstelle zur Verwaltung der Datenhaltung implementiert werden. Dadurch werden die Funktionen, die auf die Datenhaltung zugreifen, an einer zentralen Stelle im Code zusammengefasst und können von den Entwicklern leichter gewartet, aktualisiert und erweitert werden. Außerdem erhöht eine solche Schnittstelle die Wiederverwendbarkeit der Funktionen, da diese Funktionen im Rest des Programms verwendet werden können und der Code für die Operationen nicht mehrfach geschrieben werden muss.

Darüber hinaus können umfangreiche Tests dieser Schnittstelle die Wartung der Datenhaltung vereinfachen, da Fehler schneller erkannt und leichter behoben werden können. Außerdem stellen die Tests die Korrektheit und Funktionsfähigkeit der Anwendung sicher, so dass neue Funktionalitäten oder Änderungen das bestehende Programm nicht beeinträchtigen.

Auch für die Schnittstelle und diese Tests sind Dokumentationen und Kommentare im Code erforderlich, um die Funktionalität und Wartbarkeit der neuen Datenhaltung für alle Entwickler leicht erlernbar zu machen und übersichtlich zu gestalten.

Um die Wartbarkeit zu erleichtern, sollte daher eine dokumentierte Schnittstelle zur Verwaltung der Datenhaltung existieren, die durch ebenfalls dokumentierte Tests auf Korrektheit überprüft wird.

4.2.7. Übertragbarkeit

Übertragbarkeit ist die Fähigkeit einer Datenhaltung, Daten zwischen verschiedenen Umgebungen, Plattformen oder Technologien zu übertragen.

Dabei spielt zunächst die Portabilität eine wichtige Rolle, d.h. die Unabhängigkeit der Datenhaltung von der Plattform, auf der sie laufen soll, so dass sie auf verschiedenen Plattformen und Systemen einheitlich funktioniert. Aus diesem Grund ist z.B. die Verwendung einer Binärdatei als neue Form der Datenhaltung nicht sinnvoll, da die Struktur der Binärdatei vom verwendeten Betriebssystem abhängig ist und somit nicht einheitlich funktionieren würde.

Darüber hinaus sollte die Datenhaltung leicht konfigurierbar und modifizierbar sein. Die Konfigurierbarkeit ermöglicht die Anpassung der Datenhaltung an die Bedürfnisse des Systems und der Anwendungen, während die Modifizierbarkeit eine möglichst einfache und schnelle Reaktion auf Änderungen der Umgebung oder der Anforderungen der Anwendung erlaubt. Beides kann durch die Datenhaltungsschnittstelle realisiert werden, da diese eine schnelle und einfache Anpassung der Daten ermöglicht.

Außerdem kann eine Datenhaltung durch die Verwendung von Standardtechnologien oder -formaten die Übertragbarkeit sicherstellen. Beispielsweise unterstützt die Verwendung der standardisierten Abfragesprache SQL oder die Verwendung von Dateiformaten wie XML oder JSON die Portabilität einer Datenhaltung, da diese von vielen Plattformen unterstützt werden.

Die Verwendung einer Datenverwaltungsschnittstelle und die Anwendung einer portablen Datenhaltung unter Nutzung von Standardtechnologien tragen somit zur Gewährleistung der Portabilität bei.

5. Implementierungsentwurf

Aufbauend auf den Anforderungen des Kapitels 4 wird in diesem Kapitel die Wahl der Datenhaltungsoption argumentiert und ein Konzeption der zu implementierenden neuen Datenhaltung und deren Verwaltung erarbeitet.

Dazu wird zunächst anhand der Vor- und Nachteile der vorgestellten Datenhaltungsoptionen eine dieser Optionen ausgewählt und anschließend die Informationen, die sie enthalten soll, in formaler Sprache im Sinne eines konzeptionellen Datenhaltungsmodells dargestellt. Im logischen Datenhaltungsentwurf wird die Struktur der Datenhaltung durch die geeignete Darstellungsform des Datenhaltungsmodells abgebildet und die Daten werden normalisiert. Der physische Datenhaltungsentwurf beschreibt die Implementierung der Datenhaltung im gewählten Datenhaltungsmodell. Abschließend wird das Konzept für die Datenhaltungsverwaltung zur Implementierung in JUBE vorgestellt.

5.1. Lösungsanalyse

Auf Basis der Problemanalyse aus Kapitel 3.5 und der in Kapitel 4 definierten Anforderungen werden die Vorteile einer Ablösung und die für diesen Anwendungsfall praktikablen Datenhaltungsoptionen bewertet und und eine dieser Option für die Implementierung ausgewählt.

Die eingangs erläuterten Probleme des Formats und der Struktur der XML-Dateien können durch eine neue Datenhaltung vermieden werden. Die doppelte Speicherung von Informationen in zwei Dateien kann verhindert werden, indem die Daten der Konfigurations- und der Workpackagedatei gemeinsam gespeichert werden. Außerdem kann der aufwendige Zugriff auf die Daten aufgrund des XML-Formats und der dafür notwendigen Konvertierung in eine baumartige Struktur durch eine Datenhaltung gelöst werden, deren Datensätze punktuell aktualisiert werden können. Diese Datenhaltung sollte darüber hinaus über die Möglichkeit einer entsprechenden Datentypspeicherung verfügen, um eine typgerechte Speicherung zu ermöglichen.

Um nun auf Basis dieser und der im Kapitel 4 getroffenen Anforderungen an die Datenhaltung eine Datenhaltungsoption zu wählen, muss zunächst die Speicherort Positionierung der Datenhaltung gewählt werden.

Zum einen muss zwischen einer serverseitigen und einer lokalen Datenhaltung und zum anderen zwischen einer Datenhaltung pro Workflow wie bisher oder einer Datenhaltung pro Ordner, der mehrere Workflows enthalten kann, gewählt werden.

Aufgrund der bereits im Kapitel 4 erläuterten Vorteile in Bezug auf Sicherheit, Zuverlässigkeit und Übertragbarkeit der Datenhaltung sollte die bisherige lokale Datenhaltung beibehalten werden. Auch die Speicherung der Daten pro Workflow sollte beibehalten werden, da die Ausführungen der einzelnen Workflows unabhängig sind und parallel ablaufen können. Darüber hinaus können unterschiedliche Sets und Daten verwendet werden, was die Komplexität der gemeinsamen Datenhaltung deutlich erhöhen kann.

Die Wahl einer Datenhaltungsoption kann auf der Grundlage der Speicherort Positionierung erleichtert werden, da die Datenhaltungsoptionen je nach Positionierung unterschiedliche Vor- und Nachteile bieten.

Da die Datenhaltung über XML-Dateien angesichts der in Kapitel 3.5 beschriebenen Nachteile und die alternativen textbasierten Datenhaltungen in Datenformaten wie z.B. CSV, JSON oder YAML aufgrund der gleichen Nachteile nicht in Frage kommen, verbleiben die Optionen der temporären, der binären und der Datenhaltung über eine Datenbank. Da auch die temporäre und die binäre Datenhaltung in Anbetracht der in Kapitel 4 geforderten Anforderungen an Kompatibilität und Zuverlässigkeit nicht in Frage kommen, bleibt nur die Option der Datenhaltung über eine Datenbank, da diese alle Anforderungen erfüllt und nicht die Nachteile der anderen Datenhaltungen aufweist.

Der Einsatz einer Datenbank und des dazugehörigen DBMS bietet viele verschiedene Möglichkeiten und Modelle der Nutzung. Im Kapitel 2.2.4.3 wurden bereits die vier für den Anwendungsfall dieser Arbeit geeigneten DBMS beschrieben. Dabei handelt es sich bei den DBMS für relationale Datenbanken um MySQL und SQLite und bei den NoSQL-DBMS um MongoDB und BaseX. Im Folgenden werden diese im Hinblick auf die herausgearbeiteten Anforderungen an die neue Datenhaltung analysiert.

Die in Kapitel 4.1 aufgeführten funktionalen Anforderungen können alle von den vorgestellten Datenbanken erfüllt werden. Sie sind alle in der Lage, Daten einzufügen und zu validieren und verfügen auch über die Funktionalität der anderen CRUD-Operationen. Darüber hinaus erfüllen alle Optionen die Anforderungen an die Datenintegrität.

Bei der BaseX-Datenbank ist jedoch zu beachten, dass die im Kapitel 4.1.1 erarbeiteten Informationsanforderungen eine neue Strukturierung des XML-Formats erzwingen würde. Dies hätte zur Folge, dass der eigentliche Vorteil der BaseX-Datenbank, dass die Daten bereits im gewünschten Format vorliegen, nicht genutzt werden kann, da die Formatierung geändert werden müsste.

Dahingegen ist bei der MongoDB Datenbank zu beachten, dass es sich um eine dokumentenorientiertes Datenhaltungssystem handelt, das zwar die Möglichkeit von Referenzen, ähnlich den Fremdschlüsseln in relationalen Datenbanken, bietet, diese jedoch eine komplexere Abfrage erfordern. Auch Abfragen über mehrere Dokumente hinweg können sehr komplex werden. Anhand der beschriebenen Informationsanforderungen in 4.1.1 wird jedoch klar, dass diese nötig sein werden, um an spezifische Daten zu kommen.

Darüber hinaus kann die nicht standardisierte Abfragesprache von NoSQL-Datenbanken insbesondere für Entwickler, die mit relationalen Datenbanken vertraut sind, zu einer anspruchsvollen Lernkurve und damit zu einem hohen Arbeitsaufwand führen. Aus diesen und den oben beschriebenen DBMS-spezifischen Gründen scheidet die Verwendung der beiden NoSQL-Datenbanken für die Datenhaltung in JUBE aus.

Nach der Analyse und dem Ausschluss der NoSQL-Datenbanken werden nun die beiden relationalen Datenbankmodelle hinsichtlich der im Kapitel 4.2 definierten nichtfunktionalen Anforderungen analysiert.

Die Leistung der beiden DBMS MySQL und SQLite hängt von der Datenmenge ab, mit der sie arbeiten. MySQL ist bekannt für seine hohe Performance, insbesondere bei großen Datenmengen und komplexen Abfragen, während SQLite durch den Zugriff auf nur eine Datei eine effiziente Performance bei kleineren Datenmengen aufweist. Da es sich in diesem Fall durch die Speicherung der Daten pro Workflow um kleine Datenmengen handelt, ist daher die SQLite-Datenbank hinsichtlich der Performance besser geeignet als Datenhaltung für JUBE.

Im Hinblick auf die Sicherheit bietet MySQL umfangreiche Sicherheitsfunktionen wie Benutzerverwaltung, Zugriffskontrolle und Verschlüsselungsoptionen. SQLite hin-

gegen bietet nur grundlegende Sicherheitsfunktionen wie Passwortschutz und Dateiverschlüsselung, die für diese Datenhaltung, wie im Kapitel 4.2.2 beschrieben, durch die lokale Datenhaltung ausreichend sind.

Im Kapitel 4.2.3 wurden die Anforderungen an die Kompatibilität der Datenhaltung definiert. Generell bietet sich durch den Einsatz einer relationalen Datenbank die Verwendung der Abfragesprache SQL an, was wie beschrieben zu einer besseren Plattformunabhängigkeit beiträgt. Darüber hinaus trägt die Abfragesprache SQL zur Nachhaltigkeit bei, da sie den Umstieg auf andere relationale Datenbanken erleichtert, falls z.B. SQLite nicht mehr unterstützt wird. Darüber hinaus verfügen beide Datenbankmodelle über eine ausreichende Menge an Datentypen, wobei MySQL über erweiterte Datentypen verfügt, die jedoch für die Informationsanforderungen dieser Datenhaltung nicht benötigt werden.

Ein weiterer Aspekt der nicht-funktionalen Anforderungen ist die Zuverlässigkeit. MySQL unterstützt verschiedene Verfügbarkeitsmechanismen wie z.B. Replikation, um Ausfallsicherheit zu gewährleisten. Es unterstützt auch Backup- und Wiederherstellungsoptionen, um Datenverluste zu minimieren. SQLite hingegen bietet keine integrierten Mechanismen zur Gewährleistung der Verfügbarkeit, da es sich jedoch um eine dateibasierte Datenbank handelt, hängt die Verfügbarkeit direkt von der Verfügbarkeit der Datei selbst ab.

Aus Gründen der Benutzerfreundlichkeit und Abhängigkeitsreduktion sollte, wie im Kapitel 4.2.5 beschrieben, auf die Installation weiterer neuer Softwarekomponenten zur Verwaltung der Datenbank verzichtet werden. Eine Schnittstelle zur SQLite-Datenbank ist bereits in der Standard-Python-Bibliothek vorhanden und kann daher ohne weitere Installation genutzt werden. Für die Schnittstelle zur MySQL-Datenbank müssen hingegen weitere Python-Pakete installiert werden, so dass in diesem Punkt die SQLite-Datenbank die bessere Wahl darstellt.

Zusammenfassend bietet somit die SQLite-Datenbank aus Sicht der Anforderungsanalyse die beste Lösung für den vorliegenden Anwendungsfall und wird für diesen verwendet. Das folgende Datenbankdesign basiert daher auf einer relationalen SQL-Datenbank auf Basis von SQLite.

5.2. Datenbankentwurf

Durch die Analyse der vorgestellten Datenbankmodelle und der Entschluss über die Eingliederung der Datenbank in die JUBE Ordnerstruktur, kann im folgenden der Datenbankentwurf erläutert werden.

5.2.1. Konzeptuelles Datenmodell

Ausgehend von der oben ausgewählten Datenbankoption und den in der Datenbank zu speichernden Informationen wird mit Hilfe geeigneter Modellierungstechniken ein konzeptuelles Datenmodell entwickelt. Dazu werden die Objekte, ihre Attribute und Beziehungen definiert. Die am häufigsten verwendete Modellierungstechnik für relationale Datenbanken ist das Entity-Relationship-Modell, kurz ERM. Dieses wird im Kapitel 2.3 näher erläutert und wird in diesem Unterkapitel verwendet, um das konzeptuelle Datenmodell zu erstellen.

Im Kapitel 4.1.1 wurden bereits die notwendigen Informationen beschrieben, die in der Datenbank gespeichert werden sollen. Darauf aufbauend soll das entsprechende

ERM Schritt für Schritt aufgebaut werden. Da das resultierende ERM zu umfangreich ist, werden in diesem Abschnitt nur Ausschnitte des Modells gezeigt. Ein vollständiges Diagramm der Entitätstypen ohne die zugehörigen Attribute findet sich im Anhang unter Abbildung A.1.

Zuerst wird der Teil des ERM hergeleitet, der die workflow-spezifischen Informationen speichert. Die Darstellung der Workflow-Entität und die Beziehung zu anderen Entitäten ist in Abbildung 5.1 dargestellt.

Da die Daten pro Workflow gespeichert werden, gibt es pro Datenhaltung genau eine Workflow- bzw. Benchmark-Entität, wie sie in diesem Modell genannt wird. Diese hat eine ID, die als Primärschlüssel verwendet werden kann, und das *name*-, *comment*-, *outpath*-, *version*- und *file_path_ref*-Attribut. Die verwendeten JUBE-Tags sollten in einem eigenen Entitätstyp gespeichert werden, da die Speicherung von Listen in einer Datenbank nicht der ersten Normalform entspricht.[1] Dieser Tag-Entitätstyp hat das *value*-Attribut, das den Inhalt des Tags beschreibt, und es muss eine Primärschlüssel-ID hinzugefügt werden.

Zudem kann ein Workflow eine beliebige Anzahl von Parameter-, Pattern-, File- und Substitutesets besitzen. Aufgrund der Eindeutigkeit des Namens können alle vier Sets diesen als Primärschlüssel verwenden. Das Parameterset speichert zusätzlich das *duplicate*-Attribut. Auf die *iofiles*- und *sub*-Tags des Substitutesets wird weiter unten eingegangen und an dieser Stelle nicht näher beschrieben.

Zusätzlich zu den Sets besitzt eine Workflow-Entität eine beliebige Anzahl von Ausführungsschritten (Steps). Der Name des Ausführungsschrittes kann aufgrund seiner Eindeutigkeit ebenfalls als Primärschlüssel verwendet werden. Alle in Kapitel 4.1.1 beschriebenen Attribute der Ausführungsschritte können als Attribute der Entität übernommen werden, mit Ausnahme des *use*-Attributs. Dieses beschreibt die Verwendung und damit die Beziehung zu verschiedenen Sets. Dieses Attribut kann daher von der Datenbank bzw. im ERM als Beziehung dargestellt werden. Somit ist auch für diese Entität die erste Normalform erfüllt. Diese Verwendung von *use* als Beziehung wird weiter unten behandelt und in diesen Ausschnitt des ERM (Abbildung 5.1) noch nicht abgebildet.

Schließlich hat ein Workflow eine beliebige Anzahl von Analyser- und Result-Entitäten. Diese haben wiederum einen eindeutigen Namen, der als Primärschlüssel verwendet werden kann. Zusätzlich speichert der Analyse-Entitätstyp das *reduce*-Attribut. Das *use*-Attribut des Analyse-Entitätstyp stellt die Verbindung zu den Patternsets her. Um die erste Normalform erfüllen zu können, wird diese durch eine Relation dargestellt werden, auf die später in diesem Kapitel noch eingegangen wird. Die Result-Entität speichert das *result_dir*-Attribut und eine ID als Primärschlüssel. Das *use*-Attribut der Result-Entität kann ebenfalls als Relation zum Analyse-Entitätstyp dargestellt werden und wird in einem späteren Abschnitt näher beschrieben.

In Abbildung 5.1 ist ein Ausschnitt des ERM dargestellt, der den Workflow-Entitätstyp und die direkt verknüpften Entitätstypen zeigt.[2] Dabei ist zu beachten, dass für die Workflow-Entität die historische Bezeichnung *Benchmark* verwendet wurde, da diese Bezeichnung im Programmcode von JUBE weiterhin verwendet wird und somit ein

[1] Eine Erläuterung der Normalformen findet sich im Kapitel 2

[2] Es ist zu beachten, dass normalerweise für jede Beziehung eine Raute mit der Beschreibung vorhanden sein sollte. In diesem Diagramm wurde aus Platzgründen nur eine verwendet, da alle Entitätstypen die gleiche 1-zu-N-Beziehung zum Referenzentitätstyp haben. Außerdem werden in der folgenden Abbildung zur besseren Übersicht nicht mehr alle Attribute der bereits ausgeführten Entität angezeigt.

einheitliches Bild bei der Implementierung der Datenbank ergibt.

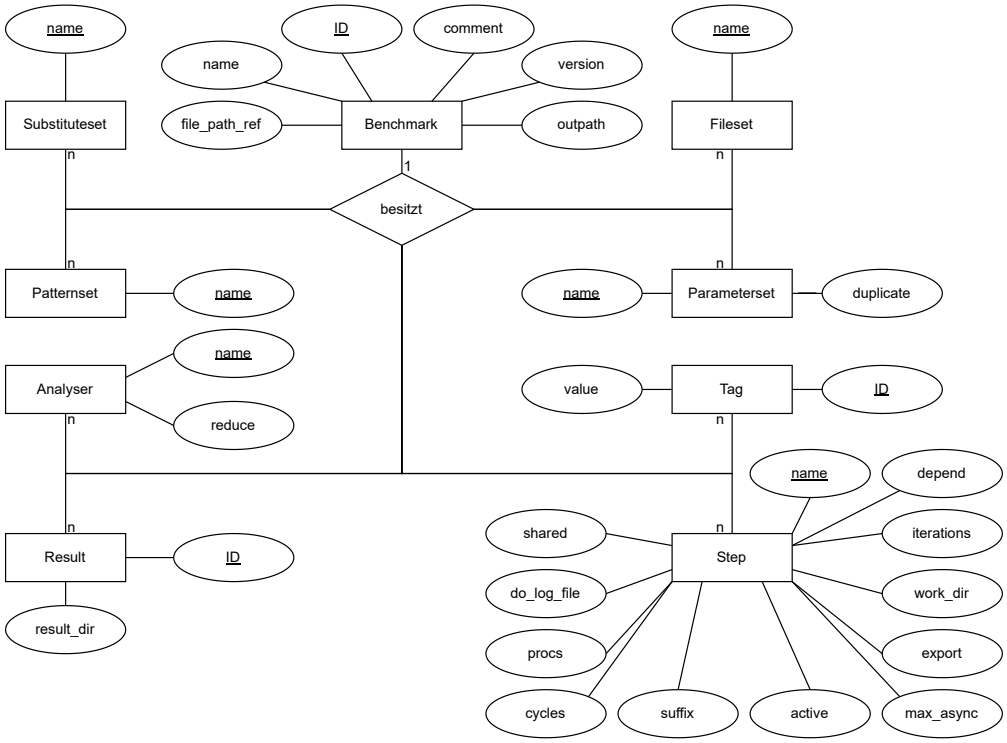

Abbildung 5.1.: Teilauszug des ERM der Datenbank für die Benchmark-Entität

Im nächsten Schritt wird der in Abbildung 5.2 dargestellte Ausschnitt des ERM erläutert. Er spiegelt die Informationen der verschiedenen Sets wider.

Der Fileset-Entitätstyp besitzt eine beliebige Anzahl von Files, die jedoch eine Zugehörigkeit zu ausschließlich einem Fileset haben. Diese besitzen derzeit keinen eindeutigen Primärschlüssel, so dass der File-Entität eine Primärschlüssel-ID hinzugefügt werden muss. Außerdem muss der File-Entität das *type*-Attribut hinzugefügt werden, das speichert, ob es sich um ein *Copy*, ein *Link* oder ein *Prepare* handelt. Die übrigen in Kapitel 4.1.1 beschriebenen Attribute können für diesen Entitätstyp übernommen werden. Die 1-zu-N-Beziehung der Files zum Fileset ist auch auf die Parameter und Pattern und deren Sets übertragbar. Der Name des Parameters bzw. des Patterns ist innerhalb eines Workflows eindeutig, sodass dieser als Primärschlüssel genutzt werden kann. Die restlichen Attribute die in Kapitel 4.1.1 beschrieben sind, werden ebenfalls für die beiden Entitäten übernommen.

Das Substituteset speichert derzeit die Informationen über die Substitutionsdatei und die Substitution in einer Tupel- bzw. Dictionary-Liste. Da diese Darstellungsmöglichkeit nach der ersten Normalform in einer Datenbank nicht anwendbar ist, werden die Substitutefile- und Substitute-Entitätstypen zusätzlich hinzugefügt. Dabei ist zu beachten, dass ein Substituteset beliebig viele dieser Entitätstypen besitzen kann, die Entitätstypen jedoch immer nur zu einem Substituteset gehören. Beide neuen Typen erhalten eine Primärschlüssel-ID. Der Substitutefile-Entitätstyp besitzt das *in*-, *out*- und *out_mode*-Attribut und der Substitute-Entitätstyp besitzt das *source*- und *dest*-Attribut, die angeben, inwieweit welcher Ausdruck ersetzt werden soll.

Abbildung 5.2 zeigt den Ausschnitt des ERM der Sets und der zugehörigen Entitäts-

typen. [1]

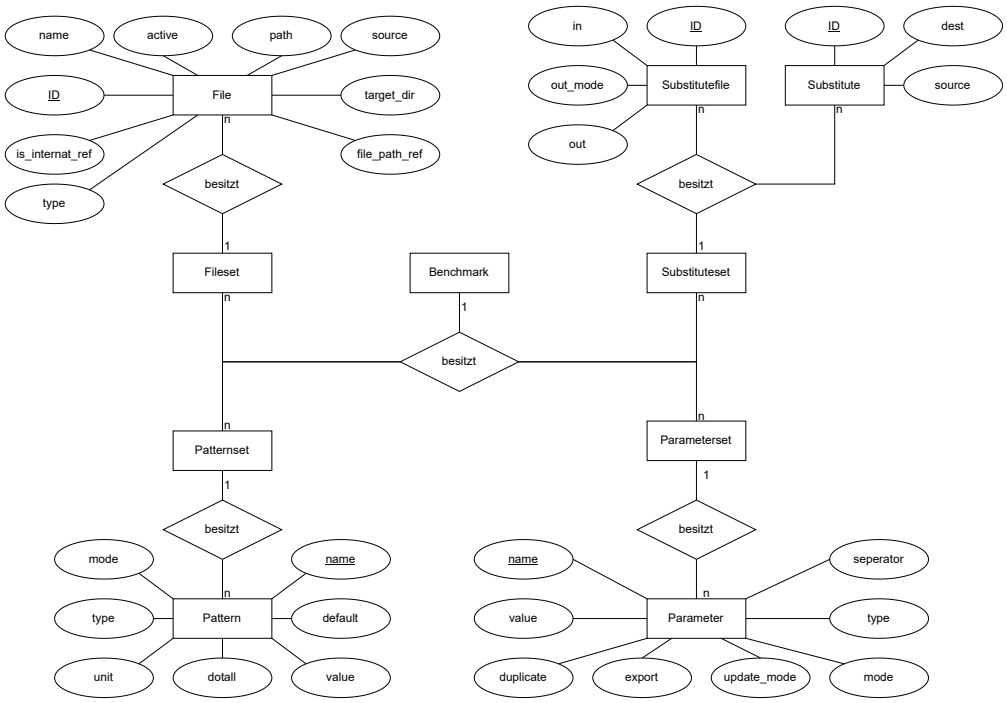

Abbildung 5.2.: Teilauszug des ERM der Datenbank für die Set-Entitäten

Im nächsten Schritt werden die Daten zu den Ausführungsschritten und deren Operationen und Sets näher erläutert. Die Darstellung der Ausführungsschritt- bzw. Step-Entität und ihre Beziehung zu anderen Entitäten ist in Abbildung 5.3 dargestellt. In Abbildung 5.1 war bereits zu sehen, dass alle Attribute bis auf *use* in der Step-Entität gespeichert werden. In *use* wurden bisher die Namen der benötigten Sets gespeichert. Dies kann nun durch Relationen im ERM abgebildet werden. So kann ein Ausführungsschritt beliebig viele Parameter-, File- und Substitutesets verwenden und diese Sets können von verschiedenen Ausführungsschritten verwendet werden. Es entsteht eine N-zu-N-Beziehung.

Zusätzlich zu den verwendeten Sets besitzt ein Ausführungsschritt verschiedene Operationen. Eine Operation gehört zu einem eindeutigen Ausführungsschritt und besitzt alle in Kapitel 4.1.1 beschriebenen Attribute. Da keines dieser Attribute eindeutig ist, wird eine Primärschlüssel-ID hinzugefügt.

[1] Bei der Benennung dieser Entitäten ist besondere Vorsicht geboten, da es in SQLite vordefinierte Wörter, sogenannte Keywords, gibt, die nicht zur Benennung von Tabellen oder Spalten verwendet werden dürfen. Dazu gehören unter anderem die Keywords *IN*, *OUT* und *DEFAULT*. Daher müssen die Spalten der Entitäten Substitutefile und Pattern entsprechend angepasst werden. Die angepassten Spaltennamen lauten dann *in_file*, *out_file* und *pattern_default*.

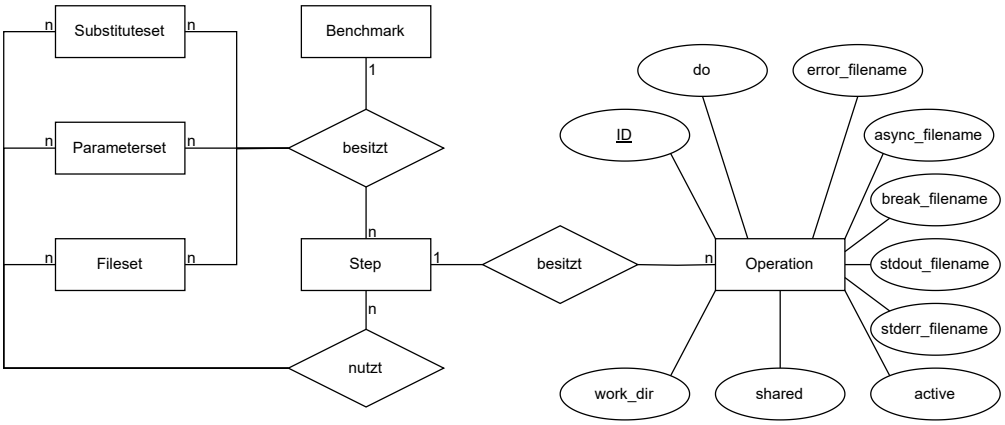

Abbildung 5.3.: Teilauszug des ERM der Datenbank für die Ausführungsschritt-Entität

Abschließend werden die Entitätstypen der Arbeitspakete, auch Workpakages genannt, beschrieben, die in Abbildung 5.4 dargestellt sind.

Ein Ausführungsschritt wird durch verschiedene Arbeitspakete beschrieben, die wiederum zu einem eindeutigen Ausführungsschritt gehören und eine ID als Primärschlüssel besitzen. Die bisher als Liste gespeicherten *iteration_sibling*- und *parents*-IDs (siehe Kapitel 3.3), die die Verbindung zu anderen Arbeitspaketen speichern, können im ERM als Relationen dargestellt werden.

Wie im Kapitel 3.3 näher erläutert, verwendet ein Arbeitspaket eine mögliche Parameterkombination aus den verwendeten Parametersets des Ausführungsschrittes. Dies kann in der Datenbank entweder durch zwei Attribute in der Beziehung zwischen Arbeitspaket und Parameter oder bspw. durch eine Auswahl-Entität gespeichert werden. Diese beiden Attribute bzw. die Entität speichern wie bisher den Index der ausgewählten Parameterkombination und deren Wert. Die Modellierung über eine zusätzliche Entität würde zwar der dritten Normalform entsprechen, jedoch zu komplexeren Abfragen in der Datenhaltung führen. Aus Komplexitätsgründen wird daher auf die zusätzliche Entität und die damit verbundene dritte Normalform verzichtet und stattdessen die beiden Attribute in der Beziehung verwendet, um die Effizienz der Datenbank zu erhöhen.

Darüber hinaus besitzt ein Arbeitspaket eine beliebige Anzahl von Umgebungsvariablen, die als Environment-Entität gespeichert werden und von verschiedenen Arbeitspaketen verwendet werden. Daraus ergibt sich eine N-zu-N-Beziehung zwischen der Workpackage- und der Environment-Entitäten. Der Environment-Entitätstyp speichert das *name*-, *value*- und *env*-Attribut, wobei *name* eindeutig ist und daher als Primärschlüssel verwendet wird.

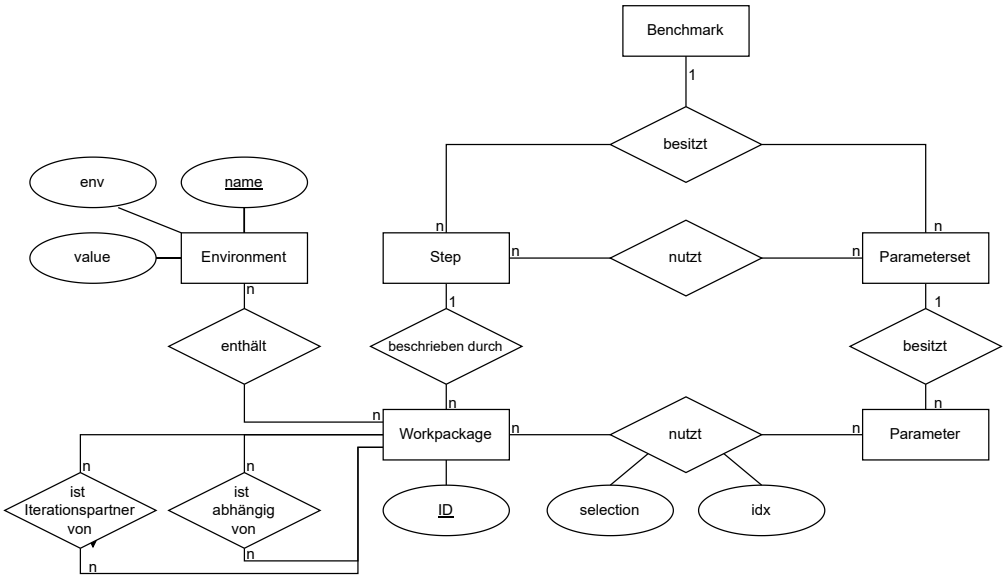

Abbildung 5.4.: Teilauszug des ERM der Datenbank für die Workpackage-Entität

Zudem besitzt eine Workflow-Entität eine beliebige Anzahl von Analyse- und Ergebnis-Entitäten, die jeweils zu genau einer Workflow- bzw. Benchmark-Entität gehören. Diese beiden Entitäten besitzen wiederum verschiedene Analyse-Dateien und Ergebnistypen. Abbildung 5.5 zeigt die Analyse- und Result-Entitätstypen und deren verknüpfte Entitäten.

Bisher wurde über das <*analyse*>-Tag des Analysers die Verbindung zum zugehörigen Ausführungsschritt und der zu analysierenden Datei hergestellt. Dies kann durch eine Beziehung zu dem AnalyseFile-Entitätstyp ersetzt werden. Zusätzlich wurde durch das *use*-Attribut des Analysers das verwendete Patternset angegeben. Dies kann durch eine N-zu-N-Beziehung dargestellt werden, da ein Analyser verschiedene Patternsets verwenden kann und ein Patternset wiederum in verschiedenen Analysern verwendet werden kann. Der AnalyseFile-Entitätstyp besitzt kein eindeutiges Attribut und wird daher zusätzlich zum *path*-Attribut durch eine Primärschlüssel-ID ergänzt. Das *step*-Attribut des AnalyseFiles beschreibt den verwendeten Ausführungsschritt, was wiederum durch eine 1-zu-N-Beziehung dargestellt werden kann, da hier ein AnalyseFile einen Ausführungsschritt verwendet und die Ausführungsschritte von mehreren AnalyseFiles verwendet werden können. Zusätzlich kann durch das *use*-Attribut ein weiteres Patternset angegeben werden, dass spezifisch für diesen Ausführungsschritt benutzt werden soll. Dies ist ebenfalls eine 1-zu-N-Beziehung, da ein AnalyseFile genau ein weiteres Patternset benutzen kann, die Patternsets aber von mehreren AnalyseFiles genutzt werden können.

Wie bereits oben erwähnt, ist es auch bei dem Result-Entitätstypen nicht notwendig, das *use*-Attribut zu speichern, das bisher angab, welche Analyse verwendet wurde. Dies kann nun durch eine N-zu-N-Beziehung dargestellt werden, wobei ein Result mehrere Analyser verwenden kann und eine Analyser von mehreren Results verwendet werden kann.

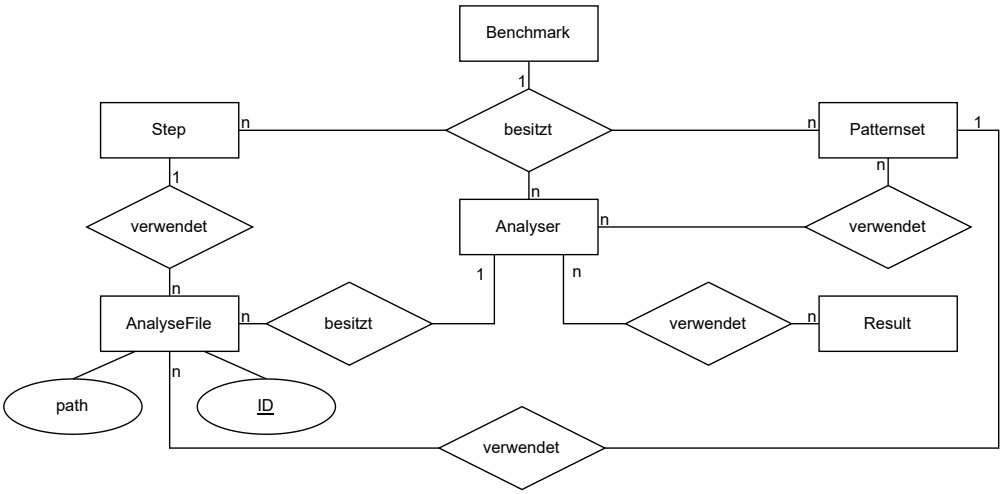

Abbildung 5.5.: Teilauszug des ERM der Datenbank für die Analyse-Entität

Schließlich sind für die Darstellung der Ergebnisse, wie bereits beschrieben, verschiedene Ausgabearten möglich. So gehören zu jedem Ergebnis eine oder mehrere Table-, Syslog- oder Database-Entitäten. Jede dieser Entitäten gehört zu genau einer Ergebnis-Entität. Die Entitäten besitzen die in Abbildung 5.6 aufgelisteten Attribute und den eindeutigen Namen als Primärschlüssel.

Tabellen bestehen aus mehreren Spalten, den Columns. Jede Spalte besitzt das *value-*, *format-*, *title-* und *colw*-Attribut und eine Primärschlüssel-ID. Jede Tabelle kann beliebig viele Columns haben, aber jede Column gehört zu genau einer Tabelle.

Die Datenbank- und Syslog-Ausgaben bestehen aus verschiedenen Keys. Jeder Key gehört zu genau einer der beiden Ausgaben und speichert das *format-* und *name*-Attribut. Zusätzlich wird ein Primärschlüssel benötigt, der mit Hilfe einer ID realisiert wird. Der Datenbank-Key speichert zusätzlich das *primekeys*-Attribut das derzeit eine Liste der Namen der Keys ist, die als Primärschlüssel der zu erstellenden Ergebnis-Datenbank definiert sind. In dem ERM wird dies durch eine *is_primary*-Attribut in der Entität der Keys definiert, der einen Wahrheitswert besitzt, ob dieser Key als Primärschlüssel definiert ist oder nicht. [1]

Abbildung 5.6 zeigt den Ausschnitt des ERM, der die Result-Entität beschreibt.

[1] Bei der Benennung dieser Entitäten ist auch auf die SQLite-Keywords zu achten. Die Namen *TABLE* und *DATABASE* gehören zu diesen Schlüsselwörtern, so dass die Entitäten für die Ausgabe einer Tabelle bzw. einer Datenbank nicht “Table“ und “Database“ heißen können. Aus diesem Grund erhalten diese Entitätstypen und alle Entitätstypen, die der Result-Entität zugeordnet sind, das Präfix “Result“.

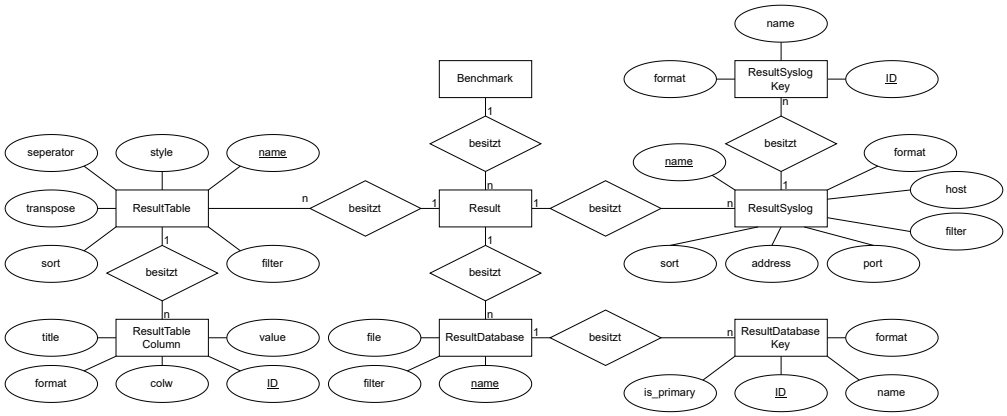

Abbildung 5.6.: Teilauszug des ERM der Datenbank für die Result-Entität

5.2.2. Relationales Datenmodell

Auf der Basis des konzeptionellen Datenmodells kann im Folgenden eine relationale Datenbank abgeleitet werden. Dazu wird die Struktur der Datenbank auf einer höheren Abstraktionsebene mit Hilfe des Relationenmodells, welches in Kapitel 2 erklärt wird, dargestellt. Dabei werden Entitäten, Attribute und Relationen in Form von Tabellen, Spalten und Schlüsseln abgebildet.

Darüber hinaus werden die Datentypen der Attribute definiert. Diese ergeben sich aus den Beschreibungen in Kapitel 3 und Abschnitt 4.1.1. Das DBMS SQLite verfügt über die Datentypen *integer* für Ganzzahlen, *real* für Fließkommazahlen und *test* für Zeichenketten. SQLite besitzt hingegen keinen Datentyp für Wahrheitswerte, jedoch können diese durch Integer als 0 für *false* und 1 für *true* gespeichert werden. Obwohl seit SQLite-Version 3.23.0 sogar die Keywords *true* und *false* unterstützt werden, werden 0 und 1 aufgrund der Rückwärtskompabilität in der Datenbank genutzt.

Im Folgenden wird das Relationsmodell analog zum ERM Schritt für Schritt aufgebaut, wobei zunächst Ausschnitte aus dem vollständigen Relationsmodell dargestellt werden, das im Anhang unter A.2 zu finden ist.

Der Teilausschnitt des ERM in Abbildung 5.1 beschreibt den Workflow-Entitätstyp und die direkt verbundenen Entitätstypen. Diese Entitätstypen können als Relation dargestellt und die zugehörigen Attribute als Spalten hinzugefügt werden. Die 1-zu-N-Beziehungen aller Entitätstypen zur Workflow-Entität können durch eine weitere Spalte mit der jeweiligen Workflow-ID als Fremdschlüssel dargestellt werden.

Das daraus resultierende Teildiagramm des Relationsmodell ist in Abbildung 5.7 zu sehen.

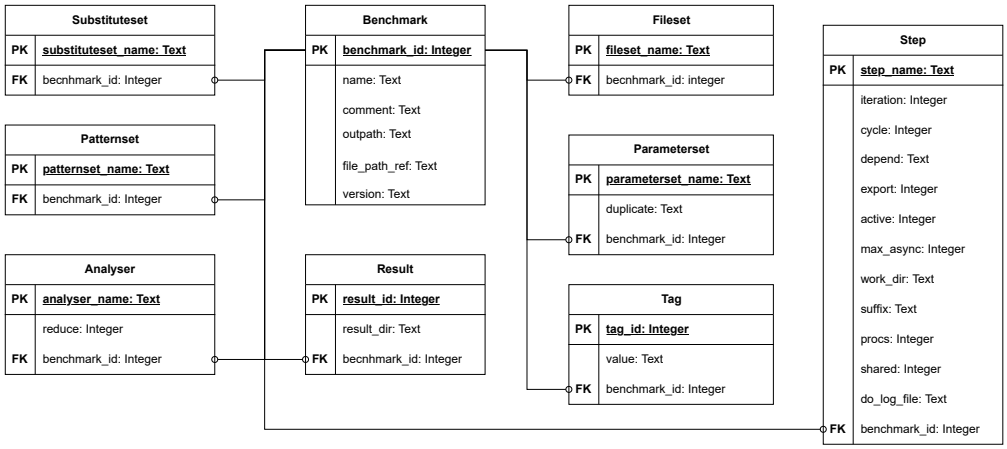

Abbildung 5.7.: Teilauszug des RM der Datenbank für die Benchmark-Entität

Die in Abbildung 5.2 dargestellten Entitätstypen der Sets können ebenfalls in Relationen umgewandelt werden. Auch hier werden alle Attribute als eigene Spalten eingefügt und die Files, Pattern, Parameter, Substitutefiles und Substitutes erhalten als Fremdschlüssel(FK) den Namen des jeweiligen Sets. Der Teilausschnitt des Relationsmodells ist in Abbildung 5.8 zu sehen.

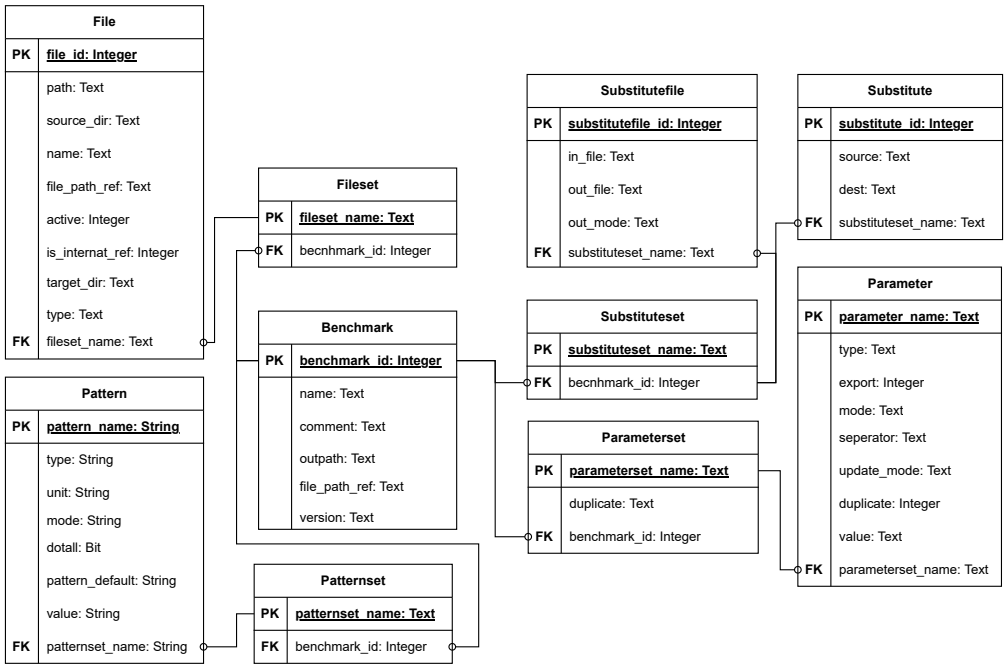

Abbildung 5.8.: Teilauszug des RM der Datenbank für die Sets-Entitäten

Die Entitäten des ERM zu Steps und Sets (Abbildung 5.3) können ebenfalls als Relation übernommen werden. Die N-zu-N-Beziehung von Steps und Sets kann über Beziehungstabellen abgebildet werden, die die jeweiligen Primärschlüssel der beiden Relationen als Fremdschlüssel enthalten. Die Kombination der beiden Fremdschlüssel bildet den Primärschlüssel der Relation. Das resultierende Teildiagramm des Relationenmodells ist in Abbildung 5.9 zu sehen.

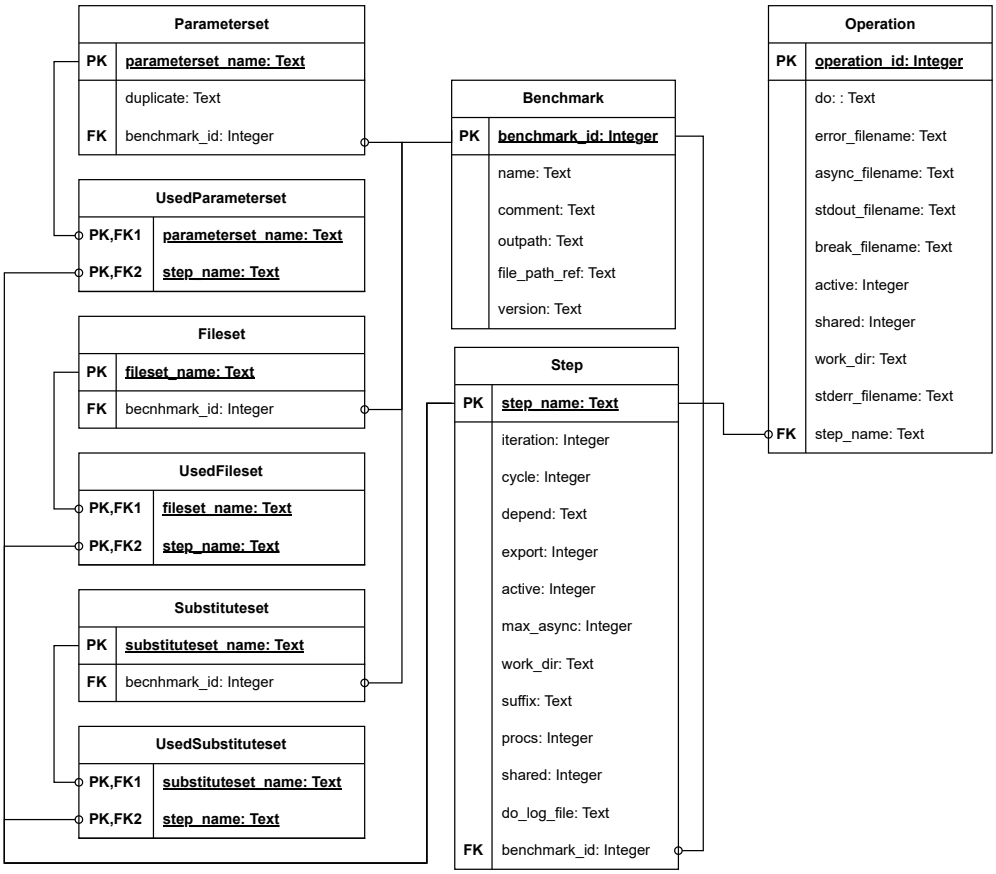

Abbildung 5.9.: Teilauszug des RM der Datenbank für die Step-und Set-Entitäten

Die Workpackage-Entität mit den zugehörigen Environment, Steps, Parameterset und Parametern kann durch Relationen mit den Spalten der jeweiligen Attribute erzeugt werden. In der Workpackage-Relation wird zusätzlich der Step-Name als Fremdschlüssel hinzugefügt, um den zugehörigen Ausführungsschritt zu kennzeichnen. Für die Eltern- und Geschwisterbeziehungen wird jeweils eine neue Tabelle mit den jeweiligen IDs der Workpackages als Fremdschlüssel verwendet. Dieses Verfahren wird auch für die Beziehung zwischen Workpackages und Environment verwendet.

Die gewählte Parameteroption eines Workpackages wird durch eine weitere Tabelle mit den Primärschlüsseln der Workpackage- und der Parameterbeziehung als Fremdschlüssel repräsentiert. Zusätzlich enthält diese Tabelle die Spalten *selected* und *idx*. Aus diesen Angaben ergibt sich das in Abbildung 5.10 dargestellte Teildiagramm des Relationenmodells.

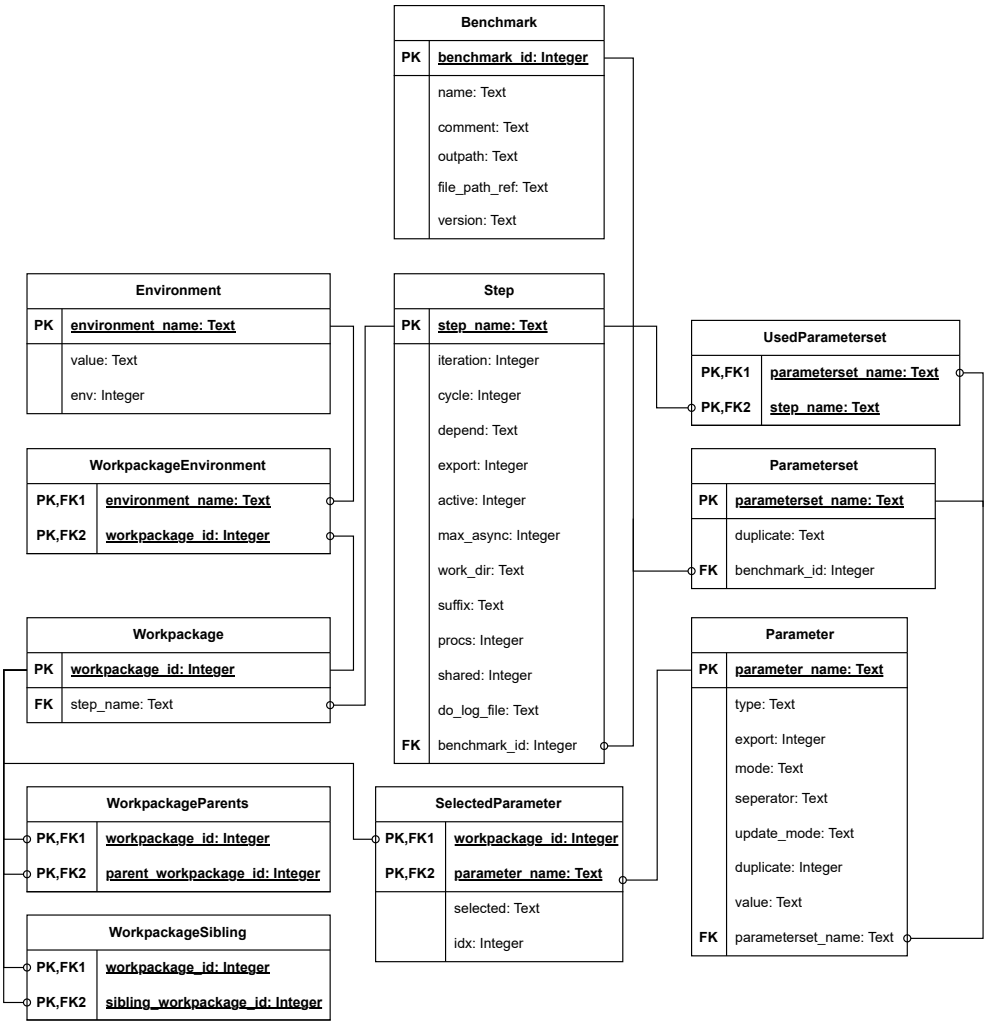

Abbildung 5.10.: Teilauszug des RM der Datenbank für die Workpackage-Entitäten

Die Entitäten und Attribute des ERM zum Analyser (Abbildung 5.5) können auch als Relationen mit den entsprechenden Spalten übernommen werden. Dem Analyse-File wird der Name des Analysers als Fremdschlüssel hinzugefügt und die N-zu-N-Beziehungen vom Analyser zum Ergebnis und zum Patternset werden in einer weiteren Tabelle festgehalten. Gleiches gilt für die N-zu-N-Beziehung zwischen Step und AnalyseFile.

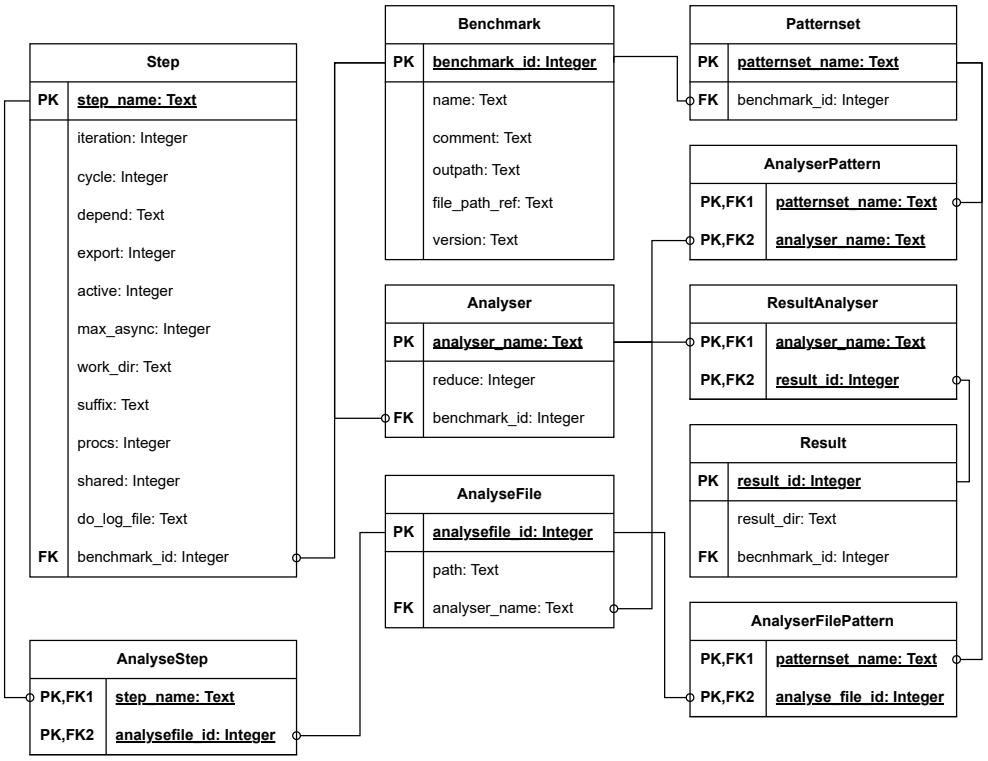

Abbildung 5.11.: Teilauszug des RM der Datenbank für die Analyse-Entitäten

Die verschiedenen Ergebnistypen des ERM zum Result (Abbildung 5.6) können auch als Relationen (Abbildung 5.12) mit den entsprechenden Spalten übernommen werden. Die Table-, Database- und Syslog-Relationen werden mit dem Namen der entsprechenden Result-Relation als Fremdschlüssel versehen, um die zugehörige Result-Relation zu identifizieren. Gleiches gilt für die Column-Relation mit dem zugehörigen Tabellennamen, um kenntlich zu machen, welche Spalten zu welcher Tabelle gehören. Ebenso werden die Syslog-Key- und Database-Key-Relationen mit dem Fremdschlüssel der zugehörigen Syslog- bzw. Database-Relation hinzugefügt.

In Abbildung 5.12 ist der Teilausschnitt des Relationenmodells für die Ergebnisrelation dargestellt.

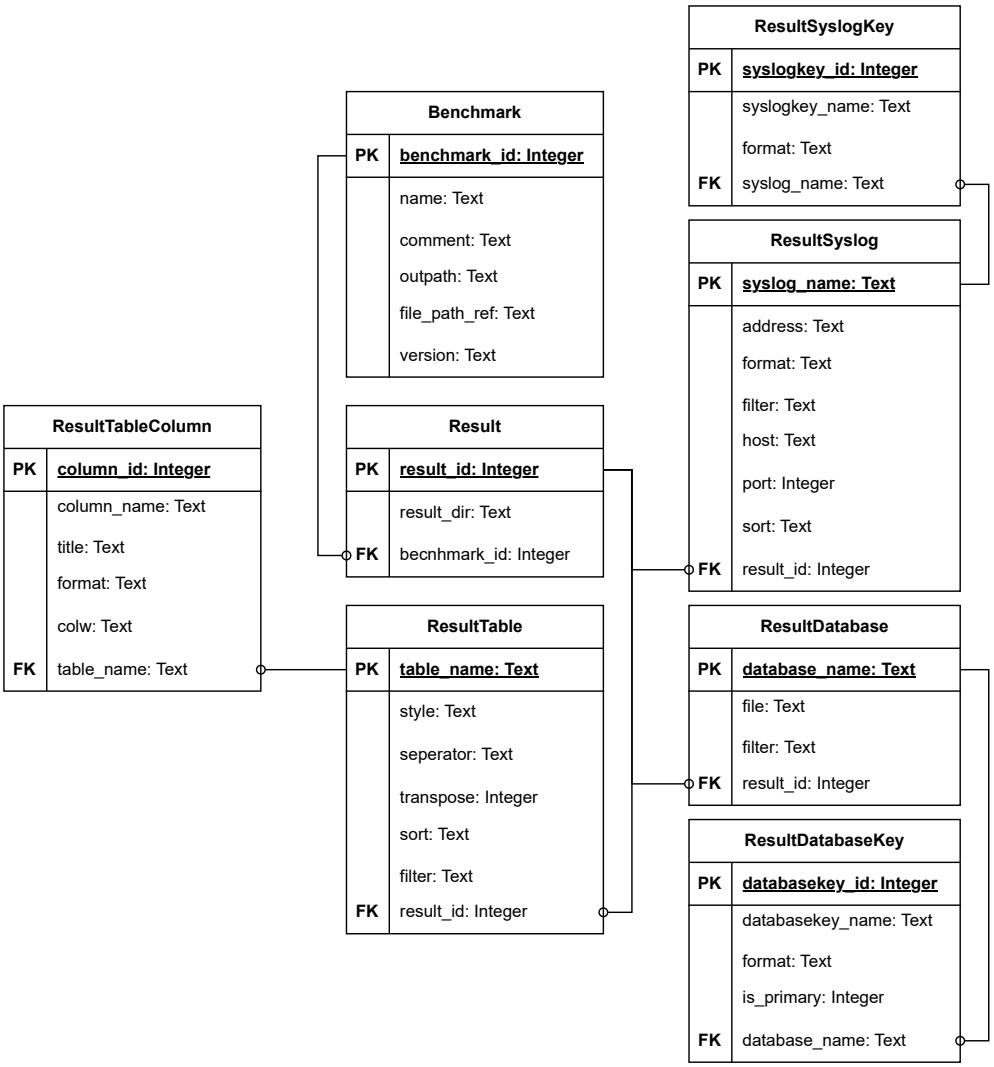

Abbildung 5.12.: Teilauszug des RM der Datenbank für die Result-Entitäten

5.2.3. Datenbankentwurfssprache

Das erstellte Relationenmodell definiert die Tabellen, Schlüssel, Spalten und Beziehungen. Um es in eine Datenbank umzuwandeln, wird die im Kapitel 5.1 ausgewählte Datenbankentwurfssprache verwendet, also SQL unter Verwendung von SQLite. Im Folgenden soll gezeigt werden, wie mit Hilfe von SQL die Datenbank erstellt und die Daten manipuliert werden können. Dazu werden die einzelnen für diese Arbeit relevanten Befehle anhand einiger repräsentativer Beispiele kurz vorgestellt. [6, S. 62]

Die Tabellen werden mit dem SQL-Befehl *CREATE TABLE* erstellt. Dabei werden der Name, die Spaltennamen und die jeweiligen Datentypen angegeben, wie am Beispiel der Parametertabelle in Listing 5.1 gezeigt.

```
CREATE TABLE IF NOT EXISTS Parameter
(
   parameter_name text NOT NULL PRIMARY KEY,
   type text NOT NULL,
   export integer,
   mode text NOT NULL,
```

```
    separator text NOT NULL,
    update_mode text,
    duplicate integer NOT NULL,
    value text NOT NULL,
    parameterset_name text NOT NULL,
    FOREIGN KEY (parameterset_name) REFERENCES Parameterset(parameterset_name)
);
```

Listing 5.1: Erstellung der Parametertabelle mit Hilfe von SQL

Dieser Befehl erzeugt eine Tabelle mit dem Namen *Parameter*, die mehrere Spalten mit unterschiedlichen Datentypen enthält. Die Tabelle beinhaltet zunächst die Spalte *parameter_name*, die als Text deklariert wird. Außerdem ist durch die Schlüsselwörter *PRIMARY KEY* und *NOT NULL* festgelegt, dass dies der Primärschlüssel ist und der Inhalt dieser Spalte nicht leer sein darf. Zusätzlich wurden die Spalten *type*, *export*, *mode*, *seperator*, *update_mode*, *duplicate* und *value* definiert, die teilweise ebenfalls nicht leer sein dürfen. Da es in SQL keinen Wahrheitswert gibt, wird für die Spalte *export* der Datentyp *integer* verwendet, der einen ganzzahligen Wert speichert. 0 steht in diesem Anwendungsfall für *false* und jede andere Zahl für *true*. Zuletzt wird der Fremdschlüssel *parameterset_name* der Tabelle Parameterset definiert. Dieser ist ebenfalls ein Text. [6, S. 62]

Für die Erstellung der Beziehungstabellen wird ebenfalls der Befehl *CREATE TABLE* verwendet. Beispielhaft für eine Beziehungstabelle wird in Listing 5.2 die Erstellung der *SelectedParameter*-Tabelle gezeigt. Wie in Listing 5.1 werden die beiden Fremdschlüssel definiert und schließlich die Kombination aus beiden als Primärschlüssel definiert.

```
CREATE TABLE IF NOT EXISTS SelectedParameter
(
    parameter_name text NOT NULL,
    workpackage_id integer NOT NULL,
    selected text NOT NULL,
    idx integer NOT NULL,
    PRIMARY KEY (parameter_name, workpackage_id),
    FOREIGN KEY (parameter_name) REFERENCES Parameter(parameter_name),
    FOREIGN KEY (workpackage_id) REFERENCES Workpackage(workpackage_id)
);
```

Listing 5.2: Erstellung der SelectedParameter-Tabelle mit Hilfe von SQL

Die Erstellung der Beziehungstabelle zwischen Workpackage und dem ausgewählten Parameter wird durch die zwei Fremdschlüssel *workpackage_id* und *parameter_name* und die anschließende Erstellung des Primärschlüssels durch die beiden Fremdschlüssel definiert. Zusätzlich wurden die Spalten *selected* und *idx* definiert.

Mit dem SQL-Befehl *CREATE TABLE* können die Tabellen und ihre Beziehungen untereinander erstellt werden. Außerdem sollen Daten in die Tabellen eingefügt, aus ihnen gelesen, aktualisiert und gelöscht werden können. Hierbei handelt es sich um sogenannte CRUD-Befehle, die bereits in der Anforderungsanalyse (Kapitel 4) erläutert wurden.

Mit dem *INSERT* Befehl werden Daten in die Tabellen der Datenbank eingefügt, wie beispielhaft in Listing 5.3 gezeigt.

```
INSERT INTO Benchmark (name, comment, outpath, file_path_ref, version)
    VALUES ('Beispiel', 'Beispiel Benchmark', 'bench_run', 'path/to/ref',
```

```
    "2.5.2");
```

Listing 5.3: Einfügen von Daten in die Benchmark-Tabelle mit Hilfe von SQL

Dieser Befehl fügt die aufgelisteten Werte in die angegebenen Spalten der Benchmark-Tabelle ein. Wenn für alle Spalten ein Wert eingefügt wird, kann der Spaltenname weggelassen werden. Wie zu sehen ist, wird keine ID angegeben, die den Primärschlüssel eines Workflows definiert. Dies liegt daran, dass SQLite bei Angabe einer ID, also einem *integer* als Primärschlüssel, die ID automatisch generiert und diese nicht manuell eingegeben werden muss. [6, S. 63]

Das Auslesen der Daten funktioniert mit Hilfe des SQL-Befehls *SELECT*, wie beispielhaft in Listing 5.4 gezeigt.

```
SELECT name, comment
FROM Benchmark;
```

Listing 5.4: Auslesen von Daten aus der Benchmark-Tabelle mit Hilfe von SQL

Das Beispiel gibt den Namen und den Kommentar aller Datensätze in der Benchmark-Tabelle zurück. Anstelle des Spaltennamens kann auch das Zeichen * verwendet werden, um alle Spalten der Tabelle auszugeben. [6, S. 63]

Zum aktualisieren der Daten, wird der SQL-Befehl *UPDATE* genutzt, wie beispielhaft in Listing 5.5 gezeigt.

```
UPDATE Benchmark
SET comment = 'Neuer Beispiel Kommentar'
WHERE name = 'Beispiel';
```

Listing 5.5: Aktualisieren von Daten in die Benchmark-Tabelle mit Hilfe von SQL

Mit Hilfe des *WHERE*-Ausdrucks innerhalb des SQL-Befehls *UPDATE* können Daten gefiltert werden, sodass nur die gewünschten Datensätze aktualisiert werden. Somit aktualisiert das obige Beispiel den Kommentar des Workflow-Datensatzes, wenn der Name des Workflows 'Beispiel' entspricht. [6, S. 63]

Zum Löschen von Daten wird der SQL-Befehl *DELETE* verwendet, der ebenfalls mit Hilfe des *WHERE*-Ausdrucks nach den gesuchten Daten filtert und beispielhaft in Listing 5.5 gezeigt ist.

```
DELETE FROM Benchmark WHERE name = 'Beispiel';
```

Listing 5.6: Löschen von Daten in die Benchmark-Tabelle mit Hilfe von SQL

Der angegebene Befehl löscht den Datensatz mit dem Namen 'Beispiel' in der Benchmark-Tabelle. [6, S. 63]

Nachdem die wichtigsten Befehle zur Erstellung und Bearbeitung von Tabellen in der ausgewählten Datenbankentwurfssprache erläutert wurden, wird im nächsten Abschnitt die Umsetzung des Datenbankmodells in eine Datenbankschnittstelle mit Hilfe von Python beschrieben.

5.2.4. Physisches Datenmodell

Die im letzten Unterkapitel beschriebenen SQL-Befehle werden benötigt, um Daten in die Datenbank einzugeben, auszulesen, zu aktualisieren und zu löschen. Für diese Funktionen wird JUBE um eine neue Klasse erweitert, die die in diesem Kapitel modellierte Datenbankschnittstelle abbildet. Durch die Verwendung einer eigenen Klasse wird die

Datenbankzugriffslogik von anderen Teilen des Codes entkoppelt, wodurch der Code modularer und leichter wartbar wird. Außerdem kann die Datenbankschnittstelle in anderen Teilen des Codes wiederverwendet werden, und die Testbarkeit wird erleichtert.

In den folgenden Unterkapiteln wird näher erläutert, wie die Datenbankschnittstelle implementiert und die zugehörigen Tests durchgeführt werden.

5.2.4.1. Datenbankschnittstelle

Die Datenbankschnittstelle wird, wie der restliche JUBE-Code, in Python implementiert. Das *sqlite3*-Python-Paket ermöglicht die Ausführung von SQL-Befehlen durch SQLite. Es wird daher in der zu implementierenden Datenbankschnittstelle verwendet, um die gewünschten Befehle auf der Datenbank auszuführen, die ebenfalls von der Schnittstelle erstellt wird. Eine Modellierung dieser Datenbankschnittstelle durch ein UML-Klassendiagramm ist in Abbildung 5.13 dargestellt.

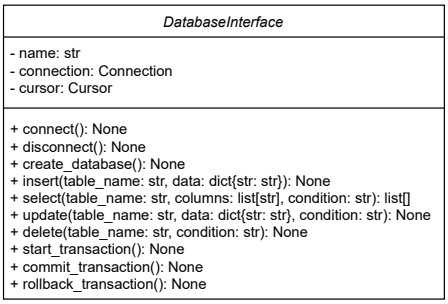

Abbildung 5.13.: UML-Klassendiagramm für die Datenbankschnittstelle

Bei der Erstellung eines Objekts der Datenbankschnittstelle kann ein Name angegeben werden, der im *name*-Attribut als Zeichenkette gespeichert wird. Falls kein Name angegeben wird, soll der Standardwert *'database.db'* genutzt werden.

Mit Hilfe der *connect()*-Methode und den passenden Funktionen des sqlite3-Pakets und dem Namen der Datenbank kann die Verbindung zur Datenbank hergestellt werden. Diese Verbindung wird durch das *connection*-Attribut gespeichert. Zudem wird innerhalb der Methode ein sogenannter Cursor auf die Datenbank erstellt, auf dem die SQL-Befehle ausgeführt werden. Dieser wird durch das *cursor*-Attribut gespeichert.

Außerdem kann die Verbindung zur Datenbank mit der *disconnect()*-Methode beendet werden, indem die *close()*-Funktion des *sqlite3*-Python-Pakets auf den gespeicherten Cursor angewendet wird. Sowohl die *disconnect()*-Methode als auch die *connect()*-Methode haben keinen Rückgabewert, was in Abbildung 5.13 durch das Schlüsselwort *None* angezeigt wird.

Für die Erstellung der Datenbanktabellen ist die *create_database()*-Methode zu verwenden. Diese erstellt, falls die Datenbanktabellen noch nicht existieren, die benötigten Tabellen mit dem *CREATE TABLE*-SQL-Befehl(Listing 5.1). Da es sich, wie in den Kapiteln 5.2.1 und 5.2.2 gezeigt, um eine Vielzahl von Tabellen handelt, sollten die *CREATE*-Befehle in eine eigene Datei ausgelagert und mit der *executescript(filename)*-Funktion des *sqlite3*-Python-Pakets, die mit dem entsprechenden Dateinamen auf dem Cursor aufgerufen wird, ausgeführt werden. Auch die *create_database()*-Methode liefert keinen Rückgabewert.

Weiterhin sollen die CRUD-Abfragen mit Hilfe der *insert(table_name, data)*-, *select(table_name, columns, condition)*-, *update(table_name, data, condition)*- und *delete(table_name, condition)*-Methoden implementiert werden. Diese Methoden nutzen den *execute()*-Funktion des *sqlite3*-Python-Paket, um die gewünschten SQL-Abfrage auf der Datenbank auszuführen.

Der *insert(table_name, data)*-Methode wird der Tabellennamen als Zeichenkette und der einzufügende Datensatz als Dictionary übergeben. Dieses Dictionary enthält als Schlüssel die Namen der Spalten und die zugehörigen Werte zum Speichern in der Datenbanktabelle im jeweilig nötigen Datentypen. Die Methode erstellt aus diesen Daten und dem Tabellennamen den passenden SQL-Befehl und führt diesen aus.

Zum Auslesen der Datensätze wird der *select(table_name, columns, condition)*-Methode der Tabellennamen als String, die auszulesenden Spalten als Liste von Strings und die Bedingung als String, um nach dem entsprechenden Datensatz suchen zu können, übergeben. Diese Methode erzeugt auch das erforderliche SQL-Statement, führt es aus und gibt den oder die gefundenen Datensätze als Liste zurück.

Die Aktualisierung von Datensätzen mit der *update(table_name, data, condition)*-Methode erfordert ebenfalls den Tabellennamen, die Namen der benötigten Spalten und die neuen Daten durch *data* sowie die Bedingung, um den oder die zu aktualisierenden Datensätze herausfiltern zu können. Diese Methode generiert auch das erforderliche SQL-Statement und führt es aus.

Die *delete(table_name, condition)*-Methode ermöglicht das Löschen eines oder mehrerer Datensätze anhand des Tabellennamens und einer Bedingung. Sie generiert den entsprechenden SQL-Befehl und führt ihn aus.

Zusätzlich zur Grundfunktionalität sollten die Abfragen transaktionsorientiert sein, um die Daten in der Datenbank konsistent zu halten. Für transaktionsorientierte Abfragen stehen die *start_transaction()*-Methode zum Starten, die *commit_transaction()*-Methode zum Beenden und die *rollback_ transaction()*-Methode zum Abbrechen zur Verfügung.

5.2.4.2. Test der Datenbankschnittstelle

Um sicherzugehen dass die Datenbankschnittstelle einwandfrei funktioniert, sollten Tests eingebaut werden, die das Verhalten der Datenbank durch Aufruf der zuvor erläuterten Funktionen überprüft.

Zum Testen von Python-Klassen wird das *unittest*-Python-Paket verwendet. Dieses Paket ermöglicht die Entwicklung automatisierter Testklassen. Eine Modellierung dieser Testklassen durch ein UML-Klassendiagramm ist in Abbildung 5.14 zu sehen.

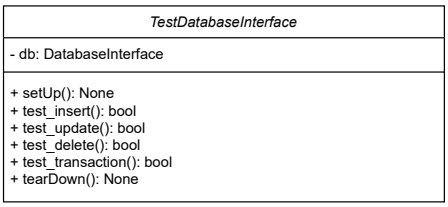

Abbildung 5.14.: UML-Klassendiagramm für die Test-Klasse der Datenbankschnittstelle

Die *setUp()*- und *tearDown()*-Methoden werden bei der Erstellung eines Objekts der Testklasse bzw. nach der Fertigstellung aller Tests ausgeführt und können ange-

passte Konfigurationen für die Testklasse enthalten. Im Falle dieser Testklasse für die Datenbank enthält die *setUp()*-Methode die Erstellung des Datenbankschnittstellenobjekts, die Verbindung zur Datenbank und die Erstellung der Tabellen. Die *tearDown()*-Methode beendet die Verbindung zur Datenbank und löscht die erstellte Datenbankdatei.

Die *test_insert()*-, *test_update()*- und *test_delete()*-Methoden enthalten den spezifischen Code zum Testen der jeweiligen Methoden der Datenbankschnittstellen. Die *test_insert()*-Methode fügt mit der *insert(table_name, data)*-Methode der Datenbankschnittstelle bestimmte Datensätze in die Datenbank ein und prüft mit der *select(table_name, columns, condition)*-Methode, ob die Daten in der Datenbank enthalten sind. Die *test_update()*-Methode prüft ebenfalls mit den *update(table_name, data, condition)*- und *select(table_name, columns, condition)*-Methoden der Datenbankschnittstelle, ob die Aktualisierung der Daten funktioniert. Die *test_delete()*-Methode löscht die Datensätze in der Datenbank mit der *delete(table_name, condition)*-Methode und prüft anschließend mit der *select(table_name, columns, condition)*-Methode, ob die Datenbank leer ist. Alle diese Methoden geben einen Wahrheitswert zurück, der angibt, ob der Test korrekt funktioniert hat (*true*) oder ob ein Fehler aufgetreten ist (*false*).

Zusätzlich zu den Tests der CRUD-Operationen gibt es die *test_transaction()*-Methode, die prüft, ob eine Transaktion das gewünschte Verhalten zeigt. Dazu wird zunächst eine Transaktion mit der *start_transaction()*-Methode der Datenbankschnittstelle gestartet und verschiedene *insert*- und *update*-Operationen korrekt ausgeführt. Nachdem diese Operationen ausgeführt wurden und keine Fehler während der Ausführung aufgetreten sind, wird die Transaktion mit *commit_transaction()* beendet und die Änderungen werden übertragen. Dies wird durch die *select(table_name, columns, condition)*-Methode überprüft. Tritt ein Fehler auf, ist der Test fehlgeschlagen und gibt *false* zurück. Wenn die Änderungen über die Datenbankschnittstelle bestätigt wurden, wird eine weitere Transaktion gestartet, die eine absichtlich falsche *insert*-Operation auslöst. Tritt der entsprechende Fehler auf, wird die *rollback_transaction()*-Methode ausgeführt und anschließend geprüft, ob die Daten unverändert geblieben sind. Ist dies nicht der Fall oder wurde der Fehler nicht erkannt, schlägt der Test ebenfalls fehl. Ansonsten verlief der Test wie erwartet und liefert den Wahrheitswert *true* zurück.

Die Testklasse wird automatisch ausgeführt, wenn Code-Änderungen in JUBE getätigt werden, um zu überprüfen, ob diese zu Fehlern in der Datenbankschnittstelle führen. Auf diese Weise kann sichergestellt werden, dass der Zugriff auf die Datenbank jederzeit korrekt möglich ist.

5.3. Konzept zur Implementierung in JUBE

Die vorhandenen Methoden der Datenbankschnittstelle wurden bereits im Kapitel 5.2.4.1 erläutert, während dieses Unterkapitel sich mit dem Aufruf dieser Methoden innerhalb des JUBE-Codes befasst. Dafür müssen die Codestellen an denen die bisherige Methoden zur Ausschreibung der Daten (*etree_repr()* und andere Verwaltungsmethoden der bisherigen Datenhaltung an die neue Schnittstelle angepasst werden.

In Kapitel 3.4 wurde das Erstellen, Lesen und Aktualisieren der Informationsdatei anhand der Programmaufrufe erläutert. Im Folgenden werden die Anpassungen anhand der gleichen Struktur beschrieben.

Der *run*-Befehl dient zur erstmaligen Erstellung der aktuellen Datenhaltung. Die Extraktion der Daten aus der Eingabedatei, das Ableiten der zugehörigen Klassen-Objekte und die Erstellung der Ordnerstruktur können zunächst erhalten bleiben.

Bei der Erstellung des spezifischen Workflow-Ordners wird derzeit auch die Konfigurationsdatei mit der *write_benchmark_configuration(filename, outpath=None)*-Methode erstellt. Diese Methode kann vollständig durch eine neue Methode ersetzt werden, die zuerst die Datenbank mit Hilfe der Datenbankschnittstelle erzeugt und dann die erforderlichen Daten einfügt. Diese Methode heißt *add_benchmark_configuration_to_database()*. Das erzeugte Datenbankobjekt wird in der Benchmark-Klasse als Attribut gespeichert, so dass bei Änderungen auch andere Klassenmethoden auf die Datenbank zugreifen können und nicht erneut eine Verbindung aufbauen müssen.

Die Workflow-spezifischen Daten, die zum Speichern in der Datenbank benötigt werden, sind die Daten des Workflows und der im Workflow definierten Sets, Ausführungsschritte, Analyser und Ergebnisausgaben. Die Daten des Workflows sind in der Benchmark-Klasse enthalten, in der die Methode aufgerufen wird und somit ein direkter Zugriff besteht. Bevor diese Daten in die Datenbank eingefügt werden, sollte eine Transaktion gestartet werden, um sicherzugehen dass die Datenbank in einem konsistenten Zustand ist und bleibt. Nach dem Speicherung der Daten ohne Fehlerauftritt, können die Daten in die Datenbank übertragen werden.

Nachdem die Workflow-spezifischen Daten gespeichert wurden, müssen die Daten der Sets, Ausführungsschritte, Analyser und Ergebnisausgaben in die Datenbank ausgeschrieben werden. Um diese Daten in die XML-Datei auszuschreiben wurde bisher die Klassen-spezifische *etree_repr()*-Methode genutzt. Diese kann nun durch eine Methode namens *write_information_to_database()* ersetzt werden. Die Benchmark-Klasse ruft dementsprechend für jedes Sets, Ausführungsschritte, Analyser und Ergebnisausgaben die jeweilige *write_information_to_database()*-Methode auf, in der jeweils eine Transaktion gestartet wird und die objektspezifischen Daten mit Hilfe der übergebenen Datenbankinstanz und der *insert(table_name, data)*-Methode eingetragen werden. Nach der Eintragung aller Sets, Ausführungsschritte, Analyser und Ergebnisausgaben wird die Verbindung zur Datenbank geschlossen.

Im nächsten Schritt wird bisher ein Zähler für die Workpackage ID auf 0 gesetzt. Dieser Zähler wird nicht mehr benötigt, da die ID eines Workpackages automatisch von der Datenbankverwaltung gesetzt wird. Die anschließende Erstellung eines Python-Dictionaries mit einer Liste der Workpackages zu den jeweiligen Ausführungsschritten und die Erstellung und Speicherung der Workpackage-Objekte in diesem Dictionary durch die Step-Klasse kann zunächst beibehalten werden.

Die *write_workpackage_information(filename)*-Methode der Benchmark-Klasse, die die Informationen der Workpackages innerhalb der aktuellen Datenhaltung ausschreibt, kann vollständig durch eine Methode zum Einfügen der Daten in die Datenbank ersetzt werden. Die neue Methode heißt *add_workpackage_information_to_database()*.

Um die Daten der Workpackages und deren zugehörigen Klassen auszuschreiben wurde bisher die *etree_repr()*-Methode genutzt. Diese Methode kann in allen Klassen, die innerhalb der aktuellen *write_workpackage_information(filename)*-Methode genutzt wird, durch eine Methode namens *write_information_to_database()* ersetzt werden, die mit Hilfe der Datenbankschnittstelle die Informationen des aktuellen Objektes anhand von Transaktionen in die Datenbank einfügt.

Nach der Erstellung durch den ersten Teil der Ausführung des *run*-Befehls werden die Benchmarks ausgeführt. Wie in Kapitel 3.4 beschrieben, kann es dabei zu

Datenänderungen kommen, die zu einem erneuten Aufruf der *write_workpackage_-information(filename)*-Methode führen. Da die neue Datenhaltung jedoch eine Update-Funktionalität besitzt, kann anstelle der *write_workpackage_information(filename)*-Methode eine neue *update_information_in_database()*-Methode aufgerufen werden, in der die Änderungen der Klassen und der Bedingung in das entsprechende Datenformat formatiert werden und die *update(table_name, condition)*-Methode der Datenbankschnittstelle aufgerufen wird.

Beim Ausführen des *continue*-Befehls werden aktuell die Daten der Konfigurationsdatei durch die *benchmarks_from_xml()*-Parser-Methode ausgelesen. Um die Versionkompaktibilität zu gewährleisten und somit bei alten Läufen die alte Datenhaltung durch die neu geplante zu ersetzen, soll weiterhin geprüft werden, ob die Konfigurationsdatei im XML-Format existiert. Falls dies der Fall sein sollte, wird weiterhin die Parser-Methode *benchmarks_from_xml()* aufgerufen, um die Daten der Konfigurationsdatei zu extrahieren. Falls jedoch bereits eine Datenbank angelegt wurde, so werden mit Hilfe der *select(table_name, data)*-Methode der Datenbankschnittstelle die passenden Daten aus der Datenbank extrahiert und die entsprechenden Klassen erstellt. Dies geschieht in der neuen *read_benchmark_data()*-Methode, die sowohl den Aufruf der *benchmarks_from_xml()*-Methode, als auch das Extrahieren der Datenbanksätze beinhaltet.

Das anschließende Wiederherstellen der Workpackage-Informationen soll ebenfalls die Auswahlmöglichkeit für die alte Datenhaltung als auch die neue Datenhaltung besitzen. Die neue *read_workpackage_data()*-Methode beinhaltet somit den Aufruf der *workpackage_from_xml()*- und den Aufruf der *select(table_name, columns, condition)*-Methode der Datenbankschnittstelle.

Auch für den *continue*-Befehl kann es bei der Ausführung der Workflows zu Änderungen der Daten kommen, die mit Hilfe der entsprechenden *update_information_-in_database()*-Methode in die Datenbank eingefügt werden. Dabei ist zu beachten, dass die Datenbank ggf. noch nicht existiert und diese noch angelegt werden muss. Dies sollte abgefragt werden, um anstelle des Updates die Daten in die erneuerte Datenhaltung übertragen zu können und die alte Datenhaltung zu entfernen.

Das Wiederherstellen der ehemaligen Benchmark- und Workpackage-Informationen bzw. die erneuerte Abfrage durch die Datenbank kann für den *analyse*- und *result*-Befehl übernommen werden und wird im Folgenden nicht mehr wiederholt.

Bei der Ausführung des *analyse*- und *result*-Befehls kann durch die Kommandozeilenoption *-u* eine neue bzw. modifizierte Eingabedatei mitgegeben werden, die die aktuellen Daten, die bei der Ausführung des *run*-Befehls übergeben wurden, überschreibt. Die Erneuerung der Benchmark- und Workpackage-Informationen wird in der neuen Datenhaltung durch die *update_information_in_database()*-Methode geschehen. Auch hier sollte wie oben beschrieben auf die ggf. stattfindende Übertragung der Daten in die neue Datenhaltung abgefragt werden.

Für den *remove*-Befehl kann das Löschen des Benchmarks wie bisher über den Ordnernamen beibehalten bleiben. Das Löschen der Workpackage-Daten innerhalb der *remove_workpackage()*-Methode kann mit Hilfe der *delete(table_name, condition)*-Methode der Datenbankschnittstelle auf die erneuerte Datenhaltung angepasst werden. Durch die Angabe der Workpackage-ID kann auf das Extrahieren der Daten aus der Datenbank verzichtet werden und die Workpackage-Daten direkt durch die Angabe der ID gelöscht

werden.

Auch beim Ausführen des *info*- und *status*-Befehls kann mit Hilfe der angegebenen ID die benötigten Daten direkt über die Datenbankschnittstelle abgefragt werden.

Da der *log*-Befehl die Protokolldaten ausgibt und somit keinerlei Auswirkungen auf die Datenhaltung hat, gibt es an diesem Ablauf keine Änderungen.

6. Bewertung

Die Bewertung des entwickelten Entwurfs zur datenbankgestützten Datenverwaltung in der internen Ablaufsteuerung des Workflow-Tools JUBE soll die Möglichkeit bieten, die angestrebten Ziele der Arbeit zu überprüfen und die Auswirkungen der neuen Datenverwaltung auf verschiedene Aspekte zu untersuchen.

Auf Basis der prototypischen Entwicklung erfolgt zunächst die Bewertung des Entwurfs. Dazu wird zu Beginn des Kapitels diese prototypische Entwicklung hinsichtlich des Umfangs und der Änderungen gegenüber dem Implementierungsentwurf betrachtet.

Im weiteren Verlauf dieses Kapitels werden die verschiedenen Tests zur Überprüfung der Einhaltung der in dieser Arbeit definierten Anforderungen erläutert und angewendet. Dazu werden unter anderem Tests zur funktionalen Äquivalenz, zum Speicherbedarf und zur Leistung definiert, durchgeführt und die Ergebnisse erläutert. Anhand dieser Ergebnisse kann abgeschätzt werden, ob der Prototyp bereits die gewünschten Verbesserungen aufweist.

Auf der Grundlage der durch die Testläufe des Prototyps gewonnenen Erkenntnisse soll eine abschließende Bewertung des gesamten Implementierungsentwurfs vorgenommen werden. Dazu werden die derzeitigen Stärken und Schwächen des Prototyps und deren Bedeutung für den gesamten Implementierungsentwurf herausgearbeitet.

6.1. Prototypische Entwicklung

Um den in Kapitel 5 erarbeiteten Implementierungsentwurf bewerten zu können, wurde dieser prototypisch umgesetzt. Im Folgenden soll die prototypische Umsetzung zunächst hinsichtlich ihres Umfangs und ihrer Abweichungen vom Entwurf beschrieben werden, um in den folgenden Kapiteln eine Bewertung vornehmen zu können.

Im Unterkapitel 5.2 wurde zunächst der Entwurf der Datenbank dargestellt. Die aus dem Abschnitt 5.2.2 ableitbaren Datenbanktabellen wurden mit den herausgearbeiteten Attributen, Primär- und Fremdschlüsseln und Datentypen wie beschrieben in die prototypische Entwicklung übernommen.

Darüber hinaus wurde die im Unterkapitel 5.2.4 beschriebene Datenbankschnittstelle und die zugehörige Testklasse mit Hilfe der im Unterkapitel 5.2.3 erläuterten SQL-Befehle und des *sqlite3*-Python-Moduls vollständig in die prototypische Entwicklung übernommen. Die Erstellung der Datenbank erfolgt mittels der in der Datei *sql_create_queries.sql* enthaltenen *CREATE TABLE*-Befehle, die von der *create_database()*-Methode ausgeführt werden. Bei der Umsetzung des Konzepts in JUBE, auf die später noch eingegangen wird, stellte sich jedoch heraus, dass es bei der Eingabe von Datensätzen mit Primärschlüssel-ID über die *insert(table_name, data)*-Methode von Vorteil ist, wenn die selbst generierte ID zurückgegeben wird, um später auf diesen Datensatz verweisen zu können. Beispielsweise kann die ID des eingefügten Workflow-Datensatzes bei der Erstellung der zugehörigen Sets verwendet werden. Die *select(table_name, columns, condition)*-Methode liefert daher nicht *None*, sondern einen ganzzahligen Wert, also ein *Integer*, zurück. Alle anderen Methoden der Datenbankschnittstelle und ihrer Testklasse wurden wie beschrieben übernommen.

Nach der Erstellung der Datenbankschnittstelle konnte das Konzept für die Implementierung in JUBE, das in Kapitel 5.3 erarbeitet wurde, in die prototypische Entwicklung einfließen. Die beschriebenen Änderungen wurden bis auf wenige Ausnahmen übernommen, so dass diese Entwicklung nur als Prototyp zu betrachten ist. Diese Ausnahmen werden im Folgenden näher erläutert.

Für die erstmalige Erstellung der Datenbank durch den *run*-Befehl wurde zunächst der Zähler für die Workpackage-ID weiter verwendet und die Arbeitspakete mit der jeweiligen ID in die Datenbank eingefügt.

Bei der Implementierung der geplanten *update_information_in_database()*-Methode, die als Teil des *run*-Befehls ausgeführt werden sollte, um die Daten in der Datenbank zu aktualisieren, wurde festgestellt, dass nur die Daten der Arbeitspakete geändert werden, so dass diese Methode in *update_workpackage_information_in_database()* umbenannt wurde und nur die Daten der Arbeitspakete ändert.

Im Gegensatz dazu werden bei der Aktualisierung durch die Kommandozeilenoption *--update* bei der Ausführung des *analyse*- und *result*-Befehls oder bei der Änderung des Kommentars durch den *comment*-Befehl nur die Daten des Workflows geändert, so dass der Entwurf um eine *update_benchmark_configuration_in_database()*-Methode erweitert wurde, die nur diese Daten aktualisiert.

Zur Aktualisierung der Arbeitspaket- bzw. Workflowdaten durch die *update_workpackage_information_in_database()*- und die *update_benchmark_configuration_in_database()*-Methode ist generell zu sagen, dass derzeit alle möglichen Tabellen mit allen verfügbaren Attributen, die sich geändert haben könnten, am Ende jeder Ausführung eines Arbeitspaketes aktualisiert werden. Im Prototyp werden also nicht nur die Änderungen gespeichert und nur diese aktualisiert, sondern der Einfachheit halber wird für alle Tabellen, die sich geändert haben könnten, die *update(table_name, data, condition)*-Methode der Datenbankschnittstelle aufgerufen.

Außerdem wurde die Erstellung der XML-Dateien im Prototyp beibehalten, um die Äquivalenz mit den Daten der Datenbank, wie in Kapitel 6.2 beschrieben, zu testen. Auch der Aufbau der internen Speicherstruktur erfolgt vorzugsweise durch Extraktion der XML-Datenhaltung, was ebenfalls beibehalten werden soll, um die Versionskompatibilität zu gewährleisten. Der Aspekt, der in der prototypischen Entwicklung fehlt, ist das Nicht-Erstellen bzw. das Löschen der XML-Dateien, wenn die Datenbank erstellt wurde.

Abschließend ist anzumerken, dass nur die *run*- und *continue*-Befehle unter der Bedingung, dass kein <*result*>-Tag angegeben wird, voll funktionsfähig sind und bei den anderen Befehlen noch gelegentlich Fehler auftreten, die aber aus Zeitgründen nicht genau identifiziert und damit behoben werden konnten.

Zusammenfassend kann gesagt werden, dass die Prototypentwicklung einen Ansatz zur Überprüfung der funktionalen und nicht-funktionalen Anforderungen bietet, der eine Einschätzung der Bewertung des Implementierungsentwurfs ermöglicht. Es gibt jedoch noch einige relevante Unterschiede zwischen dem Prototypen und dem Entwurf, die für die endgültige Überführung in JUBE noch bearbeitet werden müssen.

6.2. Validierung der Anforderungen

Mit Hilfe der im vorherigen Kapitel beschriebenen prototypischen Entwicklung können die funktionalen und nicht-funktionalen Anforderungen bereits überprüft werden, auch

wenn es sich noch nicht um eine vollständige Umsetzung des Entwurfs handelt. Der Schwerpunkt dieses Unterkapitels liegt auf dem Test der funktionalen Äquivalenz und den Unterschieden in Speicherbedarf und Leistung. Zusätzlich wird die Realisierung der nicht-funktionalen Anforderungen argumentiert.

6.2.1. Funktionale Äquivalenz

Da es sich um eine Änderung des internen Datenhaltungsformats handelt und somit keine funktionale Änderung der Anwendung erfolgen soll, ist zunächst die funktionale Äquivalenz zu prüfen. Funktionale Äquivalenz bedeutet, dass die programminterne Struktur, die aus der Datenhaltung aufgebaut wird, sowohl in der XML-Datenhaltung als auch in der Datenbank gleich bleibt. Da die programminterne Struktur in JUBE durch Klassenobjekte realisiert wird, können diese Objekte auf Äquivalenz getestet werden. Um diese Objekte auf Gleichheit zu testen, wird in allen Klassen eine Methode implementiert, die die Attribute zweier Klassenobjekte miteinander vergleicht und zurück gibt, ob diese gleich sind. [1]

Da in der derzeitigen prototypischen Entwicklung bei der Generierung der Datenhaltung durch den *run*-Befehl noch beide Datenhaltungen erzeugt werden, können z.B. bei einem Aufruf des *continue*-Befehls, bei dem die Daten aus der Datenhaltung geladen werden und die programminterne Struktur durch die Klassenobjekte erzeugt wird, die Inhalte der resultierenden Klassenobjekte beider Datenhaltungsformen auf Äquivalenz getestet werden.

Als Testfälle dienen zum einen einige Eingabedateien der Beispiele aus dem JUBE-Tutorial[2] die im Anhang unter Listing A.2 bis A.5 zu finden sind. Dabei ist zu erwähnen, dass es sich bei den Beispielen des JUBE-Tutorials nur um Abbildungen bestimmter Basisfunktionalitäten handelt und keine anwendungsnahen Beispiele sind. Die Beispiele zeigen die Basisfunktionalitäten des importieren von externen Daten (*include*), der Skriptparameter (*scripting_parameter*) und der JUBE-Tags (*tagging*).

Zum anderen dienen einige selbst erstellte Eingabedateien, die im Anhang unter A.6 bis A.8 zu finden sind als Testfälle. Im Anhang befindet sich auch die zugehörige Beschreibung der Eingabedateien, die aufgrund ihrer Funktionalität mit der prototypischen Entwicklung und der repräsentativen Darstellung der Nutzung des Workflow-Tools JUBE ausgewählt wurden. Diese Beispiele zeigen die Nutzung der Iterationen (*hello_world_iter*), der Abhängigkeiten zwischen Ausführungsschritten (*dependencies*) und die Ausführung eines Benchmarks (*stream_benchmark*).

Dieser und alle folgenden Testfälle werden auf einem Lenovo ThinkPad P14s Gen 2a mit einem AMD Ryzen 7 pro 5850u, 16 Kernen und 32 GiB Arbeitsspeicher ausgeführt und verglichen.

Die Überprüfung der angegebenen Testfälle hat ergeben, dass sowohl bei der Erstellung als auch beim Auslesen aus der Datenhaltung die gleichen Daten erzeugt bzw. ausgelesen werden, da die Inhalte der erzeugten Klassenobjekte identisch sind. Damit ist die funktionale Äquivalenz der prototypischen Entwicklung mit der originalen JUBE-Version gegeben.

[1] Dazu wird die *__eq__*-Methode verwendet (https://docs.python.org/3/reference/datamodel.html#basic-customization)

[2] https://apps.fz-juelich.de/jsc/jube/jube2/docu/tutorial.html

Der interne Test der Objektübereinstimmung kann bis zur endgültigen Entwicklung der neuen Datenhaltung im Programmcode beibehalten werden, um die funktionale Äquivalenz weiterhin sicherzustellen.

6.2.2. Speicherplatzbedarf

Neben der funktionalen Äquivalenz werden auch einige der Hauptgründe für den Wechsel des Datenhaltungsformats getestet, nämlich der Speicherplatzbedarf und die Leistung. Um den Speicherplatzbedarf der Datenbank im Vergleich zur XML-Datenhaltung zu testen, kann zunächst die Größe der Datenbankdatei mit der Größe der XML-Dateien nach Ausführung des *run*-Befehls, der die Dateien erzeugt, verglichen werden. Da die prototypische Entwicklung derzeit beide Datenhaltungen erzeugt, kann sie für diesen Test unverändert verwendet werden.

Auch für diesen Test können als Testfälle einerseits die Eingabedateien der Beispiele aus dem JUBE-Tutorial und andererseits die selbst erstellten Eingabedateien, die im Anhang unter A.2 bis A.8 zu finden sind, verwendet werden.

Diese Testfälle werden mit Hilfe des *run*-Befehls ausgeführt und anschließend der Speicherplatzbedarf der XML-Dateien im Vergleich zur Datenbank verglichen. Dabei ist zu beachten, dass die Summe des Speicherplatzbedarf der beiden XML-Dateien verwendet wird.

Wird der Speicherplatz der Datenhaltungen verglichen, so ist zu erwarten, dass die Datenbank bei kleineren Eingabedateien bzw. kleinen Workflow-Konfigurationen und damit weniger Datensätzen einen deutlich höheren Speicherplatzbedarf hat als die XML-Dateien. Dies liegt daran, dass die Tabellenstruktur der Datenbank einen Grundspeicherbedarf hat, den die XML-Dateien nicht haben. Im Gegensatz dazu wird erwartet, dass bei großen Beispielen mit vielen Datensätzen die Datenbank weniger Speicherplatz benötigt als die XML-Dateien.

In Tabelle 6.1 sind die Ergebnisse des Tests auf den Speicherplatzbedarf der beiden Datenhaltungen gerundet auf volle KiloByte [kB] dargestellt. Dabei wird der Speicherplatzbedarf der Konfigurations- und Workpackagedatei zusammengefasst.

Eingabedatei	Speicherplatzbedarf XML-Dateien	Speicherplatzbedarf Datenbank
include	2 [kB]	254 [kB]
scripting_parameter	7 [kB]	254 [kB]
tagging	1 [kB]	254 [kB]
hello_world_iter	128 [kB]	279 [kB]
dependencies	185 [kB]	300 [kB]
stream_benchmark	2.406 [kB]	918 [kB]

Tabelle 6.1.: Speicherplatzbedarf der beiden Datenhaltungen in KiloByte [kB].

Die Ergebnisse der in der Tabelle 6.1 aufgeführten Testfälle zeigen, dass die oben beschriebenen Erwartungen zutreffen. Bei den kleineren Beispielen des JUBE-Tutorials liegt der Speicherbedarf der XML-Dateien bei wenigen Bytes, während der Speicherbedarf der Datenbank konstant bei 254 [kB] bleibt. Dies ist auf den oben erwähnten Grundspeicherbedarf der Datenbank und die sehr kleinen Beispiele zurückzuführen. Im

Vergleich zu diesen Tutorial-Beispielen sind die XML-Dateien der selbst erstellten Testfälle deutlich größer, da diese aus einer größeren Anzahl von Arbeitspaketen bestehen. Während der Speicherplatzbedarf der XML-Datenhaltung im Vergleich zu den Tutorial-Beispielen auf mehr als das 2.000-fache, genauer auf 2.406 [kB], ansteigt, erhöht sich der Speicherplatzbedarf der Datenbank nur auf weniger als das 4-fache, nämlich auf 918 [kB].

So ist zu erkennen, dass sich zwar der Speicherplatzbedarf für kleine Beispiele verschlechtert, aber z.B. für den STREAM-Benchmark, der eher dem normalen Anwendungsfall von JUBE entspricht, eine deutliche Verbesserung des Speicherplatzbedarfs zu erkennen ist. Die prototypische Entwicklung zeigt also bereits die gewünschte Verbesserung.

6.2.3. Leistung

Um die Leistung der Datenhaltungen zu vergleichen, soll die Zeit für das Erstellen, das Auslesen und das Aktualisieren der jeweiligen Datenhaltung gemessen werden. Für diesen Test muss eine Änderung an der aktuellen prototypischen Entwicklung vorgenommen werden, um das Zeitverhalten der Datenbank mit der XML-Datenhaltung vergleichen zu können.

Dazu wird die prototypische Entwicklung mit Hilfe des *time*-Python-Moduls[1] so ergänzt, dass die Zeit vor und nach dem Erstellen, Lesen und Aktualisieren der jeweiligen Datenhaltung (XML oder Datenbank) gemessen wird. Diese Zeiten werden addiert und am Ende ausgegeben, so dass die tatsächliche Verarbeitungszeit der Datenhaltungen verglichen werden kann.

Auch für diesen Test können als Testfälle einerseits einige Eingabedateien der Beispiele aus dem JUBE-Tutorial und andererseits die selbst erstellten Eingabedateien, die im Anhang unter A.2 bis A.8 zu finden sind, verwendet werden.

Diese Testfälle werden ebenfalls jeweils mit Hilfe des *run*- und des *continue*-Befehls ausgeführt und anschließend das Zeitverhalten der Erstellung, Auslesung und Aktualisierung der XML-Dateien im Vergleich zur Datenbank in Sekunden verglichen.

Im Rahmen des Leistungsvergleichs wird bereits bei der prototypischen Entwicklung von einer Verbesserung des Zeitverhältnisses ausgegangen, da zum einen der komplette Neuaufbau der XML-Struktur entfällt und zum anderen die Aktualisierung punktuell erfolgen kann, ohne die gesamte Datei neu zu schreiben.

In der Tabelle 6.2 sind die Laufzeitmessungen für die Erstellung, das Einfügen und das Aktualisieren der Daten der beiden Datenhaltungen für den *run*-Befehl in Sekunden [s] dargestellt. Um eine aussagekräftige Zeitmessung gewährleisten zu können, wird der *run*-Befehle insgesamt fünf mal ausgeführt und die durchschnittliche Zeitmessung für die Datenhaltung gerundet auf die vierte Nachkommastelle betrachtet.

Zunächst zeigen die in der Tabelle 6.2 dargestellten Ergebnisse des *run*-Befehls, dass die Laufzeiten der XML-Datenhaltung für die Tutorial-Beispiele deutlich schneller sind als die der Datenbank, genauer gesagt um das fünf- bis 30-fache schneller. Dies könnte an der Vielzahl von Tabellen der Datenbank liegen, die erst angelegt werden müssen und somit viel Laufzeit kosten können. Um die Laufzeit zu verbessern, könnte in der weiteren Entwicklung die Datenbank als Vorlage im JUBE-Paket gespeichert werden und

[1] https://docs.python.org/3/library/time.html

Eingabedatei	Zeitmessung XML-Dateien	Zeitmessung Datenbank
include	0,0050 [s]	0,0679 [s]
scripting_parameter	0,0134 [s]	0,0763 [s]
tagging	0,0019 [s]	0,0617 [s]
hello_world_iter	16,9280 [s]	10,7989 [s]
dependencies	50,6030 [s]	44,9279 [s]
stream_benchmark	1016,7354 [s]	385,6499 [s]

Tabelle 6.2.: Zeitmessung der beiden Datenhaltungen bei Ausführung des *run*-Befehls in Sekunden [s].

bei der Ausführung des *run*-Befehls nur kopiert werden, anstatt sie neu zu erstellen. Für die anwendungsnahen Beispiele ist jedoch zu sehen, dass die Laufzeit der Datenbank-Version schneller ist, was auf eine schnellere Aktualisierungsfunktionalität hinweisen könnte.

In der Tabelle 6.3 sind die Laufzeitmessungen für die Erstellung, das Einfügen und das Aktualisieren der Daten der beiden Datenhaltungen für den *continue*-Befehl in Sekunden [s] dargestellt. Um eine aussagekräftige Zeitmessung gewährleisten zu können, werden auch hier der *continue*-Befehl insgesamt fünf mal ausgeführt und die durchschnittliche Zeitmessung für die Datenhaltung gerundet auf die vierte Nachkommastelle betrachtet.

Eingabedatei	Zeitmessung XML-Dateien	Zeitmessung Datenbank
include	0,0109 [s]	0,0104 [s]
scripting_parameter	0,0272 [s]	0,0244 [s]
tagging	0,0043 [s]	0,0004 [s]
hello_world_iter	14,5909 [s]	11,6174 [s]
dependencies	54,9885 [s]	47,0704 [s]
stream_benchmark	624,4697 [s]	145,1249 [s]

Tabelle 6.3.: Zeitmessung der beiden Datenhaltungen bei Ausführung des *continue*-Befehls in Sekunden [s].

Aus den Ergebnissen des *continue*-Befehls, die in der Tabelle 6.3 zu finden sind, lassen sich weitere Schlüsse ziehen. Zunächst sind die Laufzeiten der Tutorial-Beispiele für beide Datenhaltungen sehr ähnlich. Dies bestätigt die Aussage, dass die Erstellung der Datenbank mehr Zeit in Anspruch nimmt, da beim *continue*-Kommando die Datenbank nicht wie beim *run*-Kommando erstellt, sondern nur aktualisiert werden muss. Die Datenbanklaufzeiten der anwendungsnahen Beispiele sind wiederum deutlich schneller als die der XML-Datenhaltung, was ebenfalls für eine deutlich schnellere Aktualisierung durch die Datenbank spricht. Vor allem beim größten Beispiel (*stream_benchmark*) zeigt der Vergleich zwischen ca. 10,5 Minuten für die XML-Datenhaltung und ca. 2,5 Minuten für die Datenbank einen enormen Unterschied und damit eine deutliche Leistungssteigerung der Datenbank für anwendungsnahe Beispiele.

Als Fazit zur Leistung kann somit festgehalten werden, dass die gewünschte Verbesserung der Laufzeit bei großen Beispielen bereits vorhanden ist, die Datenbankerstellung aber vermutlich noch weiter verbessert werden könnte. Um die aufgestellten

Vermutung zur Leistungsfähigkeit der Datenbank zu bestätigen, sind vor der Weiterentwicklung weitere Leistungstests angedacht, die die einzelnen Zeiten für die Erstellung, Auslesung und Aktualisierung der Datenhaltungen vergleichen. Zusammenfassend kann gesagt werden, dass die prototypische Entwicklung mit einigen Verbesserungen bei der Erstellung der Datenbank als Grundlage für die Weiterentwicklung dienen kann und eine Verbesserung für repräsentative JUBE-Beispiele zeigt, jedoch eine Verschlechterung für kleine Beispiele, die jedoch nur zur Demonstration der JUBE-Funktionalitäten dienen und keine anwendungsnahen Beispiele sind. Darüber hinaus bewegt sich diese Verschlechterung der Laufzeit im Bereich von weniger als einer Sekunde und stellt somit keine Einschränkung für den praktischen Einsatz dar.

6.2.4. Nicht-funktionale Anforderungen

Abschließend soll argumentiert werden, ob auch die im Kapitel 4 näher erläuterten nicht-funktionalen Anforderungen an die Datenhaltung erfüllt wurden. Der Schwerpunkt liegt dabei nicht auf der Überprüfung durch Tests, sondern auf der Argumentation anhand des Entwurfs und der prototypischen Entwicklung.

Die Leistungskomponente der nicht-funktionale Anforderung wurde bereits durch den in Kapitel 6.2.3 aufgeführten Test für die erneuerte Datenhaltung validiert.

Eine weitere nicht-funktionale Anforderung war die Anforderung an die Sicherheit der neuen Datenbank. Wie bereits im Unterkapitel 4.2.2 erläutert, konnten diese Aspekte durch lokale Datenhaltung und Transaktionen erfüllt werden. Diese beiden Punkte wurden in der prototypischen Entwicklung umgesetzt. Weitere Sicherheitsaspekte waren die Verschlüsselung und der Passwortschutz der Daten. Diese beiden Aspekte wurden in der prototypischen Entwicklung nicht implementiert, da diese sich auf die wichtigsten funktionalen Aspekte beschränkt, die Verschlüsselung der Daten zu einem erhöhten Zeitaufwand beim Schreiben und Lesen führen könnte und die beiden Punkte das Teilen der Daten und das gemeinsame Arbeiten an Daten erschweren würden. Aus diesen Gründen werden sie auch nicht für die endgültige Entwicklung übernommen.

Darüber hinaus konnte der Aspekt der Kompatibilität der neuen Datenhaltung durch die Wahl geeigneter Datentypen, einer portablen Datenhaltung und einer Versionskompatibilität erfüllt werden. Die Datenypen wurden bereits im Datenbankentwurf im Unterkapitel 5.2.2 näher beschrieben und im prototypischen Entwurf entsprechend angelegt. Die Portabilität der Datenhaltung war einer der Hauptgründe für die Wahl der Datenhaltung mittels der portablen Abfragesprache SQL und die Versionskompatibilität wurde, wie bereits im Implementierungsentwurf beschrieben, in den prototypischen Entwurf integriert.

Zusätzlich kann die Zuverlässigkeit durch den Einsatz einer lokalen Datenhaltung mit entsprechender Fehlerbehandlung und Transaktionen gewährleistet werden. Wie bereits beschrieben, wurde die lokale Datenhaltung mit Transaktionen im prototypischen Design umgesetzt und eine entsprechende Fehlerbehandlung implementiert, um Transaktionen rückgängig machen zu können. Auch bei der Erzeugung der Datenbankinstanz wurde eine Fehlerabfrage und -behandlung implementiert, die im Fehlerfall das Programm geordnet abbricht. Weitere Fehlerbehandlungen, z.B. bei fehlgeschlagenem Verbindungsaufbau, werden für die vollständige Entwicklung implementiert.

Eine weitere nicht-funktionale Anforderung ist die Benutzerfreundlichkeit. Aspekte, die diese sicherstellen, sind eine Dokumentation der neuen Funktionalitäten, Kommen-

tare im Programmcode, keine zusätzlichen Installationen durch weitere Abhaengigkeiten für den Benutzer sowie Protokollierung von Datenbankabfragen und Versionshinweise. Durch die Entscheidung für eine relationale Datenbank mit Hilfe von SQLite musste lediglich das zusätzliche *sqlite3*-Python-Paket verwendet werden, das bereits in der Standardbibliothek vorhanden ist und somit keine weitere Installation erfordert. Die weiteren Punkte sind in der prototypischen Entwicklung noch nicht umgesetzt, allerdings für die Weiterentwicklung vorgesehen.

Die im Unterkapitel 4.2.6 näher beschriebene Wartbarkeit kann durch eine dokumentierte und kommentierte Schnittstelle mit zugehörigen Tests erfüllt werden. Wie bereits in Kapitel 6.1 beschrieben, wurde diese Schnittstelle mit den zugehörigen Tests nach dem Design des Implementierungsentwurfs in der prototypischen Entwicklung implementiert, jedoch noch nicht ausreichend dokumentiert.

Abschließend war die Übertragbarkeit der Datenhaltung eine nicht-funktionalen Anforderungen, die durch eine portable Datenhaltung mit einer geeigneten Datenbankschnittstelle und der Verwendung von Standardtechnologien gewährleistet werden kann. Die genutzte relationale Datenbank ist eine portable Datenbank mit einer darauf zugeschnittenen Datenbankschnittstelle und der portablen Abfragesprache SQL. Somit ist die Datenhaltung unabhängig von der Plattform und leicht zu modifizieren.

Insgesamt ist die Bewertung der nicht-funktionalen Anforderungen ein wertvoller Beitrag zur Einschätzung der Gesamtqualität der prototypischen Entwicklung. So kann abgeschätzt werden, wie viele Anforderungen im Prototyp bereits erfüllt sind und wie viele in der weiteren Entwicklung noch erfüllt werden müssen. Diese Bewertung hat gezeigt, dass zwar einige Aspekte noch nicht umgesetzt sind, dass es sich dabei aber um eine überschaubare und realisierbare Menge handelt.

6.3. Gesamtfazit

Anhand der Erläuterung der prototypischen Entwicklung, der Validierung der Anforderungen und die in Abschnitt 3.5 aufgeführten Probleme der XML-Datenhaltung soll abschließend ein Fazit gezogen werden, welche Stärken bzw. Schwächen der erstellte Entwurf und dessen prototypische Entwicklung aufweisen und ob die in Kapitel 3.5 beschriebenen Probleme der XML-Datenhaltung durch die Ablösung durch eine Datenbank gelöst werden konnten.

Zunächst haben die Tests der funktionalen Äquivalenz, des Speicherplatzbedarfs und der Leistung sowie die Bewertung der nicht-funktionalen Anforderungen gezeigt, dass die prototypische Entwicklung insgesamt eine solide Basis für die Weiterentwicklung darstellt, die als vollwertiger Ersatz für die XML-Datenhaltung eingesetzt werden kann. Da der entwickelte Entwurf somit umsetzbar und einsatzfähig ist, ist dies eine wesentliche Stärke des Entwurfs und der prototypischen Entwicklung.

Dies zeigt auch der Vergleich mit den Problemen der XML-Datenhaltung, die im Abschnitt 3.5 näher erläutert werden. Es handelt sich dabei um doppelte Datensätze durch die Speicherung in zwei XML-Dateien, fehlende Datentypen und die allgemeine XML-Struktur, die zu zusätzlichen Formatierungen führen, und die vereinfachte Manipulation. Diese Probleme wurden bereits in der prototypischen Entwicklung beseitigt bzw. sind durch das erarbeitete Konzept für die Weiterentwicklung vorgesehen.

Eine weitere Stärke der prototypischen Entwicklung ist der im Unterkapitel 6.2.2 getestete Speicherplatzbedarf. Damit ist einer der Hauptgründe für die Ablösung der XML-Datenhaltung erfüllt und der Speicherplatzbedarf für realistische Anwendungsfälle drastisch reduziert. Dabei ist zu beachten, dass aufgrund des Grundspeicherplatzbedarfs der Datenbank und der Testergebnisse mit zunehmender Komplexität der Workflow- und Arbeitspaketdaten ein steigender Gewinn an Speicherplatzbedarf zu verzeichnen ist, so dass JUBE durch die neue Datenhaltung für noch komplexere Workflows geeignet ist.

Neben dem Speicherplatzbedarf wurde auch die Leistung der Datenhaltung getestet, die sich ebenfalls als Stärke des Entwurfs herausstellte. Dies ist auch ein Hauptgrund für die Ablösung der XML-Datenhaltung, die die Laufzeiten für anwendungsnahe Beispiele drastisch reduzieren kann und mit Hilfe weiterer Konzepte zur Erstellung und Aktualisierung der Datenhaltung noch bessere Leistungsergebnisse erreichen kann.

Zudem ist ein weiterer Vorteil der Datenbank gegenüber der XML-Datenhaltung und damit eine Stärke des Entwurfs, die Konsistenz der Daten. Da die Datenbank transaktional arbeitet und mit diesen in JUBE gearbeitet wird, ist jederzeit sichergestellt, dass die gespeicherten Informationen korrekt und gültig sind.

Eine weitere Stärke, die im prototypischen Entwurf zwar noch nicht vollständig umgesetzt ist, aber anhand des Implementierungsentwurfs abgeschätzt werden kann, ist die Persistenz der Datenhaltung. Das zu entwickelnde DBMS mit seinen Transaktions-, Protokollierungs- und Wiederherstellungsmechanismen ist darauf ausgelegt, Daten langfristig und zuverlässig zu speichern, d.h. Persistenz zu gewährleisten.

Trotz der erfolgreichen Umsetzung der Kernfunktionen und -qualitäten weist der Prototyp noch in Teilen Schwächen auf. Eines der Hauptprobleme betrifft die nicht vollständig optimierten Aktualisierungsfunktionalität und die Vielzahl an Datenbanktabellen die bei jedem Lauf neu generiert werden müssen. Die Aktualisierungsfunktionalität wird als nicht vollständig optimiert angesehen, da einerseits die Datenbank zu den gleichen Programmzeitpunkten erstellt und aktualisiert wird wie die bisherige XML-Datenhaltung. Dies wäre nicht mehr notwendig, da die Datenbank nicht zwingend am Ende jeder Ausführung aktualisiert werden muss, sondern direkt bei Änderungen aktualisiert werden kann. Zudem werden derzeit in der prototypischen Entwicklung alle definierten Tabellen, die Änderungen enthalten können, aktualisiert und nicht nur die, die tatsächlich Änderungen enthalten. Außerdem werden derzeit alle Datenbanktabellen generiert, unabhängig davon, ob sie benötigt werden oder nicht. Beispielsweise kann ein Benutzer auf die Ausgabe der Ergebnisse verzichten, so dass keine Result-Tabelle und keine zugehörigen Tabellen erstellt werden müssen. Auch dies ist ein Schwachpunkt, der, wenn er behoben wird, zu besseren Laufzeiten und geringerem Speicherplatzbedarf führen könnte.

Insgesamt lässt sich also sagen, dass die zentrale Frage, ob die Probleme der XML-Datenhaltung durch den in dieser Arbeit entwickelten und prototypisch implementierten Entwurf einer datenbankgestützten Datenhaltung gelöst werden konnten, mit “Ja“ beantwortet werden kann. Dies lässt sich vor allem anhand der bereits gelösten XML-Datenhaltungsprobleme bzw. des Entwurfs zur Lösung dieser sowie anhand der Validierung der Anforderungen zeigen. Einziger Kritikpunkt ist derzeit noch die nicht vollständig optimierten Aktualisierungs- und Erstellungsfunktionalität der Datenbank, die eine Weiterentwicklung des Entwurfs in Bezug auf die Erstellung und Aktualisierung der Datenbank erfordert, für die aber bereits Konzepte vorliegen.

7. Zusammenfassung und Ausblick

Im Mittelpunkt der vorliegenden Arbeit mit dem Titel *"Entwurf, Implementierung und Analyse einer datenbankgestützten Datenverwaltung in der internen Ablaufsteuerung des Workflow-Tools JUBE"* steht die zentrale Fragestellung, ob und wie die aktuelle Datenhaltung durch ein effizientes Datenhaltungssystem ersetzt werden kann, mit dem Ziel die Daten konsistent und persistent zu halten, den Speicherplatzbedarf zu reduzieren und die Leistung zu steigern. Im Folgenden werden die wesentlichen Ausarbeitungen und Schlussfolgerungen dieser Arbeit vorgestellt.

Zunächst wurde im Kapitel 1 die Relevanz von Datenhaltung und Datenmanagement in Anwendungen im Allgemeinen erläutert und eine kurze Einführung in das Workflow-Tool JUBE gegeben. Darauf aufbauend wurde im folgenden Kapitel eine ausführliche Grundlagenbehandlung der Themen JUBE, Datenhaltung und der dafür relevanten Normen und Standards vorgenommen, um die Grundlagen für die Themen der folgenden Kapitel zu schaffen. Des Weiteren wurden in diesem Kapitel die für diese Arbeit in Frage kommenden Datenbankmanagementsysteme (DBMS) grundlegend erläutert, um eine Basis für die spätere Auswahl zu schaffen.

Um eine Ablösung der bisherigen Datenhaltung zu begründen, wurde in Kapitel 3 die Ausgangssituation des Workflow-Tools JUBE erläutert und die Probleme der bisherigen Datenhaltung aufgezeigt. Dabei wurde deutlich, dass die derzeitige Datenhaltung vor allem aufgrund ineffizienter Aktualisierungsfunktionen und redundanter Speicherung nicht die optimale Lösung darstellt. Des Weiteren wurden die Daten, die in der aktuellen Datenhaltung gespeichert werden und deren derzeitige Strukturierung, näher erläutert. Aus diesen Informationen wurden die Anforderungen abgeleitet, die im Kapitel 4 in funktionale und nicht-funktionale Anforderungen unterteilt wurden. Dabei beschreiben die funktionalen Anforderungen die zu speichernden Informationen und deren Verarbeitung, während die nicht-funktionalen Anforderungen Qualitätsmerkmale wie z.B. die Leistung oder die Sicherheit der neuen Datenhaltung beschreiben.

Ausgehend von den Anforderungen an die neue Datenhaltung wurden im Kapitel 5 die in Frage kommenden Datenhaltungsoptionen evaluiert und damit die Entscheidung für das DBMS SQLite begründet. Aufbauend auf dieser Entscheidung wurde mit Hilfe der herausgearbeiteten Informationen, die die aktuelle Datenhaltung speichert, das Datenbankdesign entworfen und auf Basis der Anforderungen an die neue Datenhaltung ein Entwurf für die Datenbankschnittstelle erstellt. Abschließend wurde ein Konzept für das Workflow-Tool JUBE erstellt, das die Änderungen am bestehenden Programmcode beschreibt.

Aufbauend auf dem so erarbeiteten Implementierungskonzept wurde eine prototypische Entwicklung erstellt, die im Kapitel 6 detailliert beschrieben wurde. Um zunächst die funktionale Äquivalenz, den Speicherplatzbedarf und die Leistungsfähigkeit gegenüber der bisherigen Datenhaltung beurteilen zu können, wurden Tests durchgeführt, deren Ergebnisse in diesem Kapitel beschrieben und ausgewertet wurden. Darüber hinaus wurde die prototypische Entwicklung anhand der aufgestellten Anforderungen analysiert, um deren Erfüllung zu überprüfen. Aus diesen Ergebnissen konnte ein Gesamtfazit bezüglich der Stärken und Schwächen des Implementierungsentwurfs und der prototypischen Entwicklung gezogen werden. Dieses Fazit konnte zur Beantwortung der Kernfrage, ob die Probleme der bisherigen Datenhaltung durch diesen Entwurf gelöst

wurden, herangezogen werden.

Der Entwurf der datenbankgestützten Datenhaltung in der internen Ablaufsteuerung von JUBE hat sich als vielversprechende erste Idee erwiesen, um die redundante Speicherung und den daraus resultierenden erhöhten Speicherplatzbedarf sowie die verlängerte Laufzeit durch das erneute Schreiben der XML-Dateien des Workflow-Tools zu beheben. Die prototypische Implementierung und Bewertung hat gezeigt, dass die neue Datenhaltung die bisherige Datenhaltung erfolgreich ersetzen kann und zahlreiche Vorteile mit sich bringt. Bei der weiteren Umsetzung sind jedoch mehrere Punkte zu beachten, um die Leistungsfähigkeit bei der Datenbankerstellung und -aktualisierung noch weiter zu erhöhen.

Zudem gibt es sowohl beim Entwurf der datenbankgestützten Datenhaltung als auch beim Workflow-Tool JUBE weitere Verbesserungspotenziale, die im folgenden Ausblick näher beschrieben werden.

Um die Laufzeit weiter zu verbessern, kann einerseits die Erstellung der Datenbank verbessert werden, indem die notwendigen Tabellen nur punktuell erstellt werden, so dass nicht immer alle Tabellen, ob notwendig oder nicht, erstellt werden. Dadurch könnte auch der Speicherplatzbedarf weiter reduziert werden. Zum anderen kann eine Datenbankvorlage innerhalb des JUBE-Pakets gespeichert werden und muss bei neuen Läufen nicht mehr neu erstellt, sondern nur noch kopiert werden. Außerdem kann die derzeitige Aktualisierungsfunktion verbessert werden, indem die Änderungen gespeichert werden und diese Daten direkt bei der Abarbeitung der Arbeitspakete punktuell aktualisiert werden. Damit wäre das grundlegende Design für die Verwendung in JUBE abgeschlossen.

Darüber hinaus kann die neu konzipierte Datenhaltung in einer Weiterentwicklung um weitere Informationen erweitert werden, die aktuell bei der Durchführung des *analyse*-Befehls erzeugt werden. Dabei handelt es sich um die Analyseergebnisse, die ebenfalls im XML-Format gespeichert und bei der Ausgabe der Ergebnisse mit dem *result*-Befehl ausgelesen werden. Diese XML-Datei wurde im Entwurf dieser Arbeit nicht berücksichtigt, da sich dieser Entwurf auf die Kernelemente konzentriert und die Erweiterung um die Analyseergebnisse nach erfolgreicher Implementierung der Kernelemente sehr einfach sein sollte. Zusätzlich kann dabei die Funktionalität des *analyse*-Befehls dahingehend erweitert werden, dass nicht nur die statistischen Werte des Patterns, sondern alle gefundenen Übereinstimmungen gespeichert und somit angezeigt werden können.

Eine weitere Information, die in der Datenbank gespeichert werden kann, ist der Status der Arbeitspakete. Dieser wird derzeit über die Existenz oder den Namen verschiedener Dateien abgefragt und bildet in seiner Gesamtheit den Status des Workflows, der über den *status*-Befehl abgefragt werden kann. Durch die Aufnahme dieser Information in die Datenbank kann die Erstellung von temporären Zustandsdateien vermieden werden, die keinen Inhalt haben und nur durch ihre Existenz oder ihren Dateinamen den Zustand speichern.

Schließlich gibt es noch Verbesserungsmöglichkeiten für das Implementierungskonzept in JUBE. Zum einen sind durch die Speicherung der Attribute als String in der XML-Datei bisher viele Attribute auch in der internen Ablaufsteuerung als String statt im passenden Datentyp gespeichert. Da die neue Datenhaltung die Attribute im passenden Datentyp speichern kann, können auch die Datentypen der internen Ablaufsteuerung angepasst werden. Außerdem kann der Programmteil, der für die Erzeugung der Ergebnisausgabe als Datenbank zuständig ist, an die neue Datenbankschnittstelle an-

gebunden werden, so dass auch hier die implementierten Methoden weiter verwendet werden können. Diese Anpassungen würden zu einer saubereren Strukturierung des JUBE-Programmcodes führen.

Zusammenfassend eröffnet die vorliegende Masterarbeit vielversprechende Perspektiven für die zukünftige Weiterentwicklung der internen Datenhaltung und Verbesserung des allgemeinen Programmcodes von JUBE.

A.Anhang

A.1. Acronyme und Fachbegriffe

FZJ Forschungszentrum Jülich

JSC Jülich Supercomputing Centre

JUBE Jülich Benchmarking Environment

Workflow Ein Workflow beschreibt eine definierte Abfolge von Arbeitsschritten, die notwendig sind, um ein bestimmtes Ziel zu erreichen. Ein Beispiel für einen Workflow ist das Benchmarking.

Benchmarking Benchmarking beschreibt die systematische Durchführung verschiedener Kombinationen und den Vergleich der Ergebnisse mit einem festgelegten Referenzwert.

HPC High-Performance Computing

XML Extensible Markup Language

YAML YAML Ain't Markup Language

JSON JavaScript Object Notation

Datenverwaltung Datenverwaltung ist der Prozess zur Speicherung von Daten, unabhängig vom Speichermedium und bezieht sich auf den reinen Speicheraspekt, der wiederum unabhängig von der Struktur und Organisation der Daten ist. Datenverwaltung wird auch als Datenmanagement bezeichnet.

Datenhaltung Datenhaltung ist die Speicherung von Daten in einem bestimmten Format oder einer bestimmten Struktur.

Datenbank Eine organisierte Sammlung einer Menge von Daten, die je nach Art der Datenbank in strukturierter oder unstrukturierter Form gespeichert sind.

Relationale Datenbank Eine relationale Datenbank ist eine Sammlung von Daten, die mit Hilfe von Tabellen, den so genannten Relationen, nach festen Schemata organisiert sind.

NoSQL Datenbanken Not-only-SQL-Datenbanken sind schemafreie Datenbanken, die keiner festen Datenstruktur folgen.

DBMS Datenbankmanagementsysteme

SQL Structured Query Language

MySQL Eines der weltweit am weitesten verbreiteten relationalen DBMS für komplexe Anwendungen

SQLite Eines der weltweit am weitesten verbreiteten relationalen DBMS für leichtgewichtige Anwendungen

MongoDB Ein dokumentenorientiertes NoSQL-DBMS, das Daten in JSON-ähnlichen Dokumenten speichert.

BaseX Ein dokumentenorientiertes NoSQL-DBMS, das Daten in XML-Dokumenten speichert.

ISO Internationale Organisation für Normung

ERM Entity-Relationship-Modell

RM Relationenmodell

A.2. Auflistungen

A.2.1. Attribute des `<parameterset>`-Tags

- *name*: Legt den Name des Parametersets fest.
- *init_with*: Initialisierung des aktuell Parametersets mit den Parametern des gleichnamigen Paramtersets aus der in diesem Attribut angegeben Datei (Optional)
- *duplicate*: Legt die Handhabung bei Parameterdefinitionen mit demselben Namen fest (Standartwert: *replace*, weitere Möglichkeiten: *concat* oder *error*) (Optional)

A.2.2. Attribute des `<parameter>`-Tags

- *name*: Legt den Name des Parameters fest.
- *type*: Legt den Datentyp des Parameters fest (Standartwert: *string*, weitere Möglichkeiten: *int*, *float* oder *boolean*) (Optional)
- *separator*: Legt das Trennzeichen der Parameter-Inhalts fest (Standartwert: Komma(,)) (Optional)
- *mode*: Legt die Auswertung des Parameters fest (Möglichkeiten: *python*, *perl*, *shell*, *env*, *tag*) (Optional)
- *export*: Falls *true*, wird dieser Parameter als Umgebungsvariable in die Shell-Umgebung expotiert (Optional)
- *update_mode*: Steuert die Neuauswertung des Parameters (Standardwert: `never`, weitere Möglichkeiten: `use`, `step`, `cycle`, `always`) (Optional)
- *duplicate*: Legt die Handhabung bei Parameterdefinitionen mit demselben Namen fest. Überschreibt das gleichnamige Attribut des Parametersets, in dem der Parameter enthalten ist (*none* als weitere Möglichkeit und Standardwert zur Übernahme des Parameterset-Attribut) (Optional)
- *tag*: Der Parameter wird für die Ausführung berücksichtig, wenn der Nutzer die hier aufgelisteten JUBE-Tags bei der Ausführung von JUBE angibt (Optional)

A.2.3. Attribute des `<step>`-Tags

- *name*: Legt denn Namen des Ausführungsschrittes fest.
- *depend*: Definiert eine Abhängigkeit zu einem anderen Ausführungsschritt über dessen Namen (Optional)
- *work_dir*: Legt ein alternatives Arbeitsverzeichnis für diesen Ausführungsschritt fest (Optional)
- *suffix*: Erweitert den Namen des Arbeitsverzeichnisses um den Inhalt dieses Attributes (Optional)
- *shared*: Legt einen gemeinsamen Ordner für alle erstellten Arbeitspaketen des Ausführungsschrittes an (Optional)

- *active*: Mit Hilfe der Option *false* kann dieser Ausführungsschritt deaktiviert werden. Angabe eines Parameters oder eines JUBE-Tags möglich. (Optional)
- *export*: Falls *true*, werden die aktuellen Umgebungsvariablen in die abhängigen Schritte exportiert. (Standardwert *false*)
- *max_async*: Legt die Anzahl der asynchron auszuführenden Arbeitspakete fest (Optional)
- *iterations*: Anzahl der Wiederholungen des gesamten Ausführungsschrittes und somit jedes zugehörigen Arbeitspakets(Optional)
- *cycles*: Anzahl der Wiederholungen der einzelnen Shell-Befehle im selben Arbeitspaket (Optional)
- *procs*: Anzahl der Prozesse die für die parallele Ausführung der Arbeitspakte genutzt werden (Optional)
- *do_log_file*: Name oder Pfad der Logdatei, die versucht, die do´s und die Umgebung eines Arbeitspaketes des Schrittes nachzuahmen, um ein ausführbares Skript zu erzeugen. (Optional)

A.2.4. Attribute des `<do>`-Tags

- *stdout*: Name oder Pfad der Ausgabedatei (Optional)
- *stderr*: Name oder Pfad der Fehlerdatei (Optional)
- *work_dir*: Festlegung eines alternativen Arbeitsverzeichnisses (Optional)
- *active*: Mit Hilfe der Option *false* kann der Shell-Befehl deaktiviert werden. Angabe eines Parameters oder eines JUBE-Tags möglich. (Optional)
- *done_file*: Markierung eines asynchronen Befehls. Dieser Befehl wird erst fortgesetzt, wenn die angegebene Datei existiert. (Optional)
- *error_file*: Abbruch der Ausführung und ausgabe einer Fehlermeldung, falls die angegebene Datei existiert (Optional)
- *break_file*: Die Wiederholungen eines Befehls durch das Attribut *cycle* des zugehörigen `<step>`-Tags werden beendet, falls die angegebene Datei existiert. (Optional)
- *shared*: Falls *true*, wird das Arbeitsverzeichnis des Shell-Befehls im gemeinsamen Ordner abgelegt. (Standardwert *false*) (Optional)

A.2.5. Attribute des `<patternset>`-Tags

- *name*: Legt den Name des Patternsets fest.
- *init_with*: Wiederverwendung eines bereits vorhandenen Patternsets mit demselben Namen in der angegebenen Datei (Optional)

A.2.6. Attribute des `<pattern>`-Tags

- *name*: Legt den Name des Patterns fest.
- *default*: Weist dem Pattern einen Standardwert zu, wenn keine Übereinstimmung mit dem Pattern gefunden wurde. (Optional)
- *unit*: Einheit für die Anzeige in der Ergebnistabelle (Optional)
- *mode*: Legt die Auswertung des Patterns fest (Möglichkeiten: *python*, *perl*, *shell*, *env*, *tag*) (Optional)
- *type*: Legt den Datentyp des Patterns fest (Standartwert: *string*, weitere Möglichkeiten: *int*, *float* oder *boolean*) (Optional)
- *dotall*: Der Punkt (.) als Zeichensatz innerhalb eines regulären Ausdrucks entspricht jedem Zeichen außer einem Zeilenumbruch. Wenn diese Option auf *true* gesetzt wird, entspricht der Punkt auch einem Zeilenumbruch (Standartwert: *false*, weitere Möglichkeiten: *true*) (Optional)

A.2.7. Attribute des `<table>`-Tags

- *name*: Legt den Name der Tabelle fest.
- *style*: Legt die Formatierung der gesamten Tabelle fest. (Standardwert: *csv*, weitere Möglichkeiten: *pretty*, *aligned*) (Optional)
- *separator*: Angabe des Trennzeichnen bei style=*csv*. (Optional)
- *sort*: Kann eine Liste von Pattern- oder Parameternamen enthalten, die für die Sortierung der Tabellenzeilen verwendet werden. (Optional)
- *transpose*: Falls *true*, wird die Tabelle transponiert, d.h. die Spalten werden zu Zeilen und umgekehrt. (Standardwert *false*) (Optional)
- *filter*: Kann einen boolschen Ausdruck enthalten, um nur bestimmte Ergebniseinträge anzuzeigen. (Optional)

A.2.8. Attribute des `<column>`-Tags

- *colw*: Angabe der Spaltenbreite (Optional)
- *format*: Angabe des Spaltenformats (z.B. für Gleitkommazahlen format=”.2f”)
- *title*: Angabe eines alternativen Spaltentitels. Ohne Angabe entspricht dem Spaltentitel der Parameter- bzw. Patternname (Optional)

A.2.9. Attribute des `<database>`-Tags

- *name*: Legt den Name der Datenbank fest.
- *primekeys*: Liste von Parameter- oder Patternnamen, die als Primärschlüssel der Datenbanktabelle verwendet werden. (Optional)

- *file*: Angabe des Pfads und des Dateinamen zur Speicherung der Datenbank. (Optional)
- *filter*: Kann einen boolschen Ausdruck enthalten, um nur bestimmte Ergebniseinträge anzuzeigen. (Optional)

A.2.10. Attribute des `<key>`-Tags

- *format*: Angabe des Spaltenformats (z.B. für Gleitkommazahlen format=".2f")
- *title*: Angabe eines alternativen Spaltentitels. Ohne Angabe entspricht dem Spaltentitel der Parameter- bzw. Patternname (Optional)

A.2.11. Attribute des `<syslog>`-Tags

- *name*: Legt den Name des Syslogs fest.
- *address*: Legt die Adresse der Syslolg-Servers fest.
- *host*: Legt den Host des Syslog-Servers fest.
- *port*: Legt den Port des Syslog-Servers fest. (Standardwert 541)
- *sort*: Kann eine Liste von Parameter- oder Patternnamen enthalten, die für die Sortierung der Keys verwendet werden. (Optional)
- *format*: Angabe eines Python-Protokollformats (Standardwert: *jube[%(process)s]: %(message)s*)
- *filter*: Kann einen boolschen Ausdruck enthalten, um nur bestimmte Ergebniseinträge anzuzeigen. (Optional)

A.3. Abbildungen

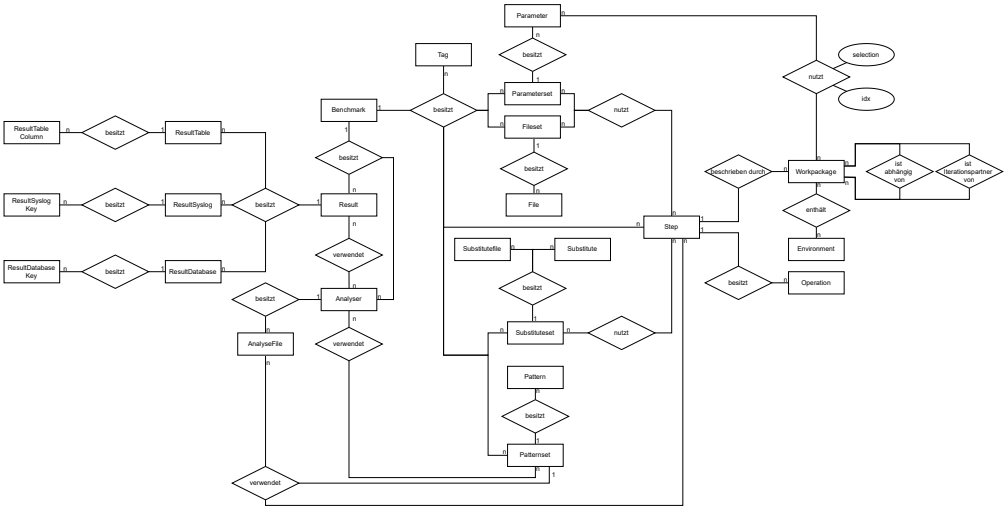

Abbildung A.1.: Vollständiges Entity-Relationship-Modell für den Datenbankentwurf (In diesem Diagramm wurden die Attribute der einzelnen Entitätstypen aus Platzgründen weggelassen.)

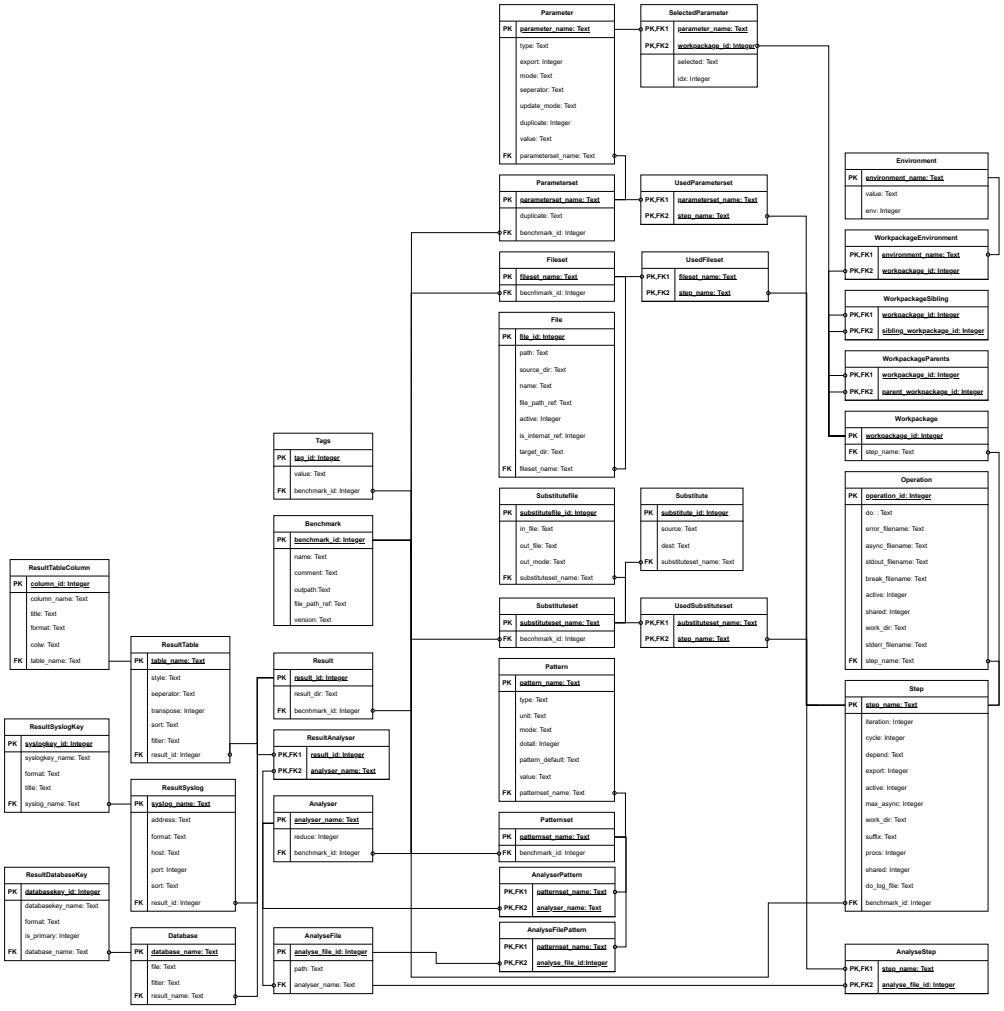

Abbildung A.2.: Vollständiges Relationsmodell für den Datenbankentwurf

A.4. Listings

```
<?xml version="1.0" encoding="UTF-8"?>
<workpackages>
   <workpackage id="0">
      <step iteration="0" cycle="0">first_step</step>
      <parameterset>
         <parameter name="number" type="int" separator="," duplicate="none"
             mode="text">
            <value>1,2</value>
            <selection idx="0">1</selection>
         </parameter>
         <parameter name="text" type="string" separator="," duplicate="none"
             mode="text">
            <value>Hello,World</value>
            <selection idx="0">Hello</selection>
         </parameter>
      </parameterset>
      <iteration_siblings>0</iteration_siblings>
      <environment/>
   </workpackage>
   <workpackage id="1">
      <step iteration="0" cycle="0">first_step</step>
      <parameterset>
         <parameter name="number" type="int" separator="," duplicate="none"
             mode="text">
            <value>1,2</value>
            <selection idx="0">1</selection>
         </parameter>
         <parameter name="text" type="string" separator="," duplicate="none"
             mode="text">
            <value>Hello,World</value>
            <selection idx="1">World</selection>
         </parameter>
      </parameterset>
      <iteration_siblings>1</iteration_siblings>
      <environment/>
      </workpackage>
   <workpackage id="2">
      <step iteration="0" cycle="0">first_step</step>
      <parameterset>
         <parameter name="number" type="int" separator="," duplicate="none"
             mode="text">
            <value>1,2</value>
            <selection idx="1">2</selection>
         </parameter>
         <parameter name="text" type="string" separator="," duplicate="none"
             mode="text">
            <value>Hello,World</value>
            <selection idx="0">Hello</selection>
         </parameter>
      </parameterset>
      <iteration_siblings>2</iteration_siblings>
      <environment/>
   </workpackage>
```

```
<workpackage id="3">
   <step iteration="0" cycle="0">first_step</step>
   <parameterset>
      <parameter name="number" type="int" separator="," duplicate="none"
          mode="text">
         <value>1,2</value>
         <selection idx="1">2</selection>
      </parameter>
      <parameter name="text" type="string" separator="," duplicate="none"
          mode="text">
         <value>Hello,World</value>
         <selection idx="1">World</selection>
      </parameter>
   </parameterset>
   <iteration_siblings>3</iteration_siblings>
   <environment/>
</workpackage>
<workpackage id="4">
   <step iteration="0" cycle="0">second_step</step>
   <parameterset>
      <parameter name="EXPORT_ME" type="string" separator=","
          duplicate="none" mode="text" export="true">
         <value>VALUE</value>
      </parameter>
   </parameterset>
   <parents>0</parents>
   <iteration_siblings>4,5</iteration_siblings>
   <environment>
      <env name="SHELL_VAR">'Hello'</env>
      <env name="EXPORT_ME">'VALUE'</env>
   </environment>
</workpackage>
<workpackage id="5">
   <step iteration="1" cycle="0">second_step</step>
   <parameterset>
      <parameter name="EXPORT_ME" type="string" separator=","
          duplicate="none" mode="text" export="true">
         <value>VALUE</value>
      </parameter>
   </parameterset>
   <parents>0</parents>
   <iteration_siblings>4,5</iteration_siblings>
   <environment>
      <env name="SHELL_VAR">'Hello'</env>
      <env name="EXPORT_ME">'VALUE'</env>
   </environment>
</workpackage>
<workpackage id="6">
   <step iteration="0" cycle="0">second_step</step>
   <parameterset>
      <parameter name="EXPORT_ME" type="string" separator=","
          duplicate="none" mode="text" export="true">
         <value>VALUE</value>
      </parameter>
   </parameterset>
```

```
      <parents>1</parents>
      <iteration_siblings>7,6</iteration_siblings>
      <environment>
        <env name="SHELL_VAR">'Hello'</env>
        <env name="EXPORT_ME">'VALUE'</env>
      </environment>
    </workpackage>
    <workpackage id="7">
      <step iteration="1" cycle="0">second_step</step>
      <parameterset>
        <parameter name="EXPORT_ME" type="string" separator=","
            duplicate="none" mode="text" export="true">
          <value>VALUE</value>
        </parameter>
      </parameterset>
      <parents>1</parents>
      <iteration_siblings>7,6</iteration_siblings>
      <environment>
        <env name="SHELL_VAR">'Hello'</env>
        <env name="EXPORT_ME">'VALUE'</env>
      </environment>
    </workpackage>
    <workpackage id="8">
      <step iteration="0" cycle="0">second_step</step>
      <parameterset>
        <parameter name="EXPORT_ME" type="string" separator=","
            duplicate="none" mode="text" export="true">
          <value>VALUE</value>
        </parameter>
      </parameterset>
      <parents>2</parents>
      <iteration_siblings>8,9</iteration_siblings>
      <environment>
        <env name="SHELL_VAR">'Hello'</env>
        <env name="EXPORT_ME">'VALUE'</env>
      </environment>
    </workpackage>
    <workpackage id="9">
      <step iteration="1" cycle="0">second_step</step>
      <parameterset>
        <parameter name="EXPORT_ME" type="string" separator=","
            duplicate="none" mode="text" export="true">
          <value>VALUE</value>
        </parameter>
      </parameterset>
      <parents>2</parents>
      <iteration_siblings>8,9</iteration_siblings>
      <environment>
        <env name="SHELL_VAR">'Hello'</env>
        <env name="EXPORT_ME">'VALUE'</env>
      </environment>
    </workpackage>
    <workpackage id="10">
      <step iteration="0" cycle="0">second_step</step>
      <parameterset>
```

```
        <parameter name="EXPORT_ME" type="string" separator=","
            duplicate="none" mode="text" export="true">
          <value>VALUE</value>
        </parameter>
      </parameterset>
      <parents>3</parents>
      <iteration_siblings>11,10</iteration_siblings>
      <environment>
        <env name="SHELL_VAR">'Hello'</env>
        <env name="EXPORT_ME">'VALUE'</env>
      </environment>
    </workpackage>
    <workpackage id="11">
      <step iteration="1" cycle="0">second_step</step>
      <parameterset>
        <parameter name="EXPORT_ME" type="string" separator=","
            duplicate="none" mode="text" export="true">
          <value>VALUE</value>
        </parameter>
      </parameterset>
      <parents>3</parents>
      <iteration_siblings>11,10</iteration_siblings>
      <environment>
        <env name="SHELL_VAR">'Hello'</env>
        <env name="EXPORT_ME">'VALUE'</env>
      </environment>
    </workpackage>
</workpackages>
```

Listing A.1: Vollständige Workpackagedatei zu Kapitel 3.3

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="include" outpath="bench_run">
      <comment>A include example</comment>

      <!-- use parameterset out of an external file and add a additional
          parameter -->
      <parameterset name="param_set" init_with="include_data.xml">
         <parameter name="foo">bar</parameter>
      </parameterset>

      <!-- Operation -->
      <step name="say_hello">
         <use>param_set</use> <!-- use existing parameterset -->
         <use from="include_data.xml">param_set2</use> <!-- out of an
             external file -->
         <do>echo $foo</do> <!-- shell command -->
         <include from="include_data.xml" path="dos/do" /> <!-- include all
             available tag -->
      </step>
   </benchmark>
</jube>
```

Listing A.2: Eingabedatei für Include-Beispiel. Bei dieser Eingabedatei handelt es sich um eine Tutorial-Eingabedatei mit insgesamt drei Arbeitspaketen.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <parameterset name="param_set">
      <parameter name="number" type="int">1,2,4</parameter>
   </parameterset>

   <parameterset name="param_set2">
      <parameter name="text">Hello</parameter>
   </parameterset>

   <dos>
      <do>echo Test</do>
      <do>echo $number</do>
   </dos>
</jube>
```

Listing A.3: Includedatei für Include-Beispiel.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="scripting_parameter" outpath="bench_run">
      <comment>A scripting parameter example</comment>

      <!-- Configuration -->
      <parameterset name="param_set">
         <!-- Normal template -->
         <parameter name="number" type="int">1,2,4</parameter>
         <!-- A template created by a scripting parameter-->
         <parameter name="additional_number" mode="python" type="int">
         ",".join(str(a*${number}) for a in [1,2])
         </parameter>
         <!-- A scripting parameter -->
         <parameter name="number_mult" mode="python" type="float">
         ${number}*${additional_number}
         </parameter>
         <!-- Reuse another parameter -->
         <parameter name="text">Number: $number</parameter>
      </parameterset>

      <!-- Operation -->
      <step name="operation">
         <use>param_set</use> <!-- use existing parameterset -->
         <!-- shell commands -->
         <do>echo "number: $number, additional_number:
             $additional_number"</do>
         <do>echo "number_mult: $number_mult, text: $text"</do>
      </step>
   </benchmark>
</jube>
```

Listing A.4: Eingabedatei für Scritpting-Parameter-Beispiel. Bei dieser Eingabedatei handelt es sich um eine Tutorial-Eingabedatei mit insgesamt sechs Arbeitspaketen.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="tagging" outpath="bench_run">
      <comment>Tags as logical combination</comment>

      <!-- Configuration -->
      <parameterset name="param_set">
         <parameter name="hello_str" tag="!deu+eng">Hello</parameter>
         <parameter name="hello_str" tag="deu|!eng">Hallo</parameter>
         <parameter name="world_str" tag="eng">World</parameter>
      </parameterset>

      <!-- Operation -->
      <step name="say_hello">
         <use>param_set</use> <!-- use existing parameterset -->
         <do>echo '$hello_str $world_str'</do> <!-- shell command -->
      </step>
   </benchmark>
</jube>
```

Listing A.5: Eingabedatei für Tagging-Beispiel. Bei dieser Eingabedatei handelt es sich um eine Tutorial-Eingabedatei mit insgesamt einem Arbeitspaket.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="hello_world" outpath="bench_run">
      <comment>A simple hello world iteration</comment>

      <!-- Configuration -->
      <parameterset name="hello_parameter">
         <parameter name="hello_str">Hello World</parameter>
         <parameter name="iteration" type="int"
             mode="python">",".join([str(x) for x in range(300)])</parameter>
      </parameterset>

      <!-- Operation -->
      <step name="say_hello">
         <use>hello_parameter</use> <!-- use existing parameterset -->
         <do>echo $hello_str</do> <!-- shell command -->
      </step>
   </benchmark>
</jube>
```

Listing A.6: Eingabedatei für Iteration-Beispiel. Bei dieser Eingabedatei handelt es sich um eine leicht abgeänderte Tutorial-Eingabedatei des *hello_world*-Beispiels mit erhöhter Arbeitspaketzahl. Diese Eingabedatei erstellt insgesamt 300 Arbeitspakete.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="dependencies" outpath="bench_run">
      <comment>A Dependency example</comment>

      <!-- Configuration -->
      <parameterset name="param_set">
         <parameter name="number" type="int" mode="python">",".join([str(x)
             for x in range(300)])</parameter>
      </parameterset>

      <!-- Operations -->
      <step name="first_step">
         <use>param_set</use> <!-- use existing parameterset -->
         <do>echo $number</do> <!-- shell command -->
      </step>

      <!-- Create a dependency between both steps -->
      <step name="second_step" depend="first_step">
         <do>cat first_step/stdout</do> <!-- shell command -->
      </step>
   </benchmark>
</jube>
```

Listing A.7: Eingabedatei für Abhängigkeits-Beispiel. Bei dieser Eingabedatei handelt es sich um eine leicht abgeänderte Tutorial-Eingabedatei des *dependencies*-Beispiels mit erhöhter Arbeitspaketzahl. Diese Eingabedatei erstellt insgesamt 600 Arbeitspakete.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
   <benchmark name="stream" outpath="stream_run">

      <parameterset name="systemParameter" init_with="platform.xml">
         <parameter name="nodes" type="int">1</parameter>
         <parameter name="taskspernode" type="int">${task}</parameter>
         <parameter name="threadspertask" type="int">${threads}</parameter>
         <parameter name="timelimit">00:10:00</parameter>
         <parameter name="executable">compile/stream.exe</parameter>
         <parameter name="account">cstao</parameter>
         <parameter name="queue">batch</parameter>
         <parameter name="preprocess">${load_modules}; </parameter>
         <parameter name="measurement">time</parameter>
      </parameterset>

      <parameterset name="modules">
         <parameter name="load_modules">ml Intel Stages/2023 GCC/11.3.0
             OpenMPI</parameter>
      </parameterset>

      <parameterset name="config">
         <parameter name="mode" type="int">0,1,2</parameter>
         <parameter name="task" type="int" mode="python">['1','4',
             ",".join([str(2**i) for i in range(3)])][$mode]</parameter>
         <parameter name="threads" type="int"
             mode="python">[",".join([str(2**i) for i in range(3)]),
             ",".join([str(2**i) for i in range(3)]), 1][$mode]</parameter>
         <parameter name="cpu_bind">threads,rank,rank_ldom,cores</parameter>
         <parameter name="node">block,cyclic</parameter>
         <parameter name="socket">cyclic,block,fcyclic</parameter>
         <parameter name="core">fcyclic,block,cyclic</parameter>
      </parameterset>

      <parameterset name="executeset" init_with="platform.xml">
         <parameter name="args_starter" separator="|">-A $account -n $tasks
             -c $threads --cpu_bind=verbose,${cpu_bind}
             --distribution=${node}:${socket}:${core}
             --disable-turbomode</parameter>
      </parameterset>

      <fileset name="sources">
         <copy>stream_mpi.c</copy>
      </fileset>

      <step name="compile">
         <use>modules</use>
         <use>sources</use>
         <!--<do>${load_modules}</do>-->
         <!--<do>mpicc -fopenmp stream_mpi.c -o stream.exe</do>-->
      </step>

      <step name="execute" depend="compile">
         <use>executeset</use>
         <use>systemParameter</use>
```

```
        <use>config</use>
        <!--<do>${load_modules}</do>-->
        <!--<do>export OMP_NUM_THREADS=$threads</do>-->
        <do>echo $args_starter</do>
        <!--<do>$starter $args_starter $executable $args_exec</do>-->
      </step>
    </benchmark>
</jube>
```

Listing A.8: Eingabedatei für STREAM-Benchmark. Bei dieser Eingabedatei handelt es sich um eine leicht abgeänderte JUBE-Eingabedatei die für die Bearbeitung der Bachelorarbeit mit dem Thema *Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien anhand von verschiedenen Benchmarks auf HPC Systemen* genutzt wurde und dort näher erläutert wird.[29] Diese Eingabedatei erstellt insgesamt 649 Arbeitspakete.

Literaturverzeichnis

[1] *BaseX Documentation*. https://docs.basex.org/wiki/Main_Page. Stand: 8. August 2023.

[2] Margit Becher. *XML: DTD, XML-Schema, XPath, XQuery, XSL-FO, SAX, DOM*. 2. Auflage. Springer Vieweg, 2021. ISBN: 978-3-658-35435-0.

[3] *Bildungsmaterialien online. Relationenmodell (RM)*. https://bildung4u.eu/relationenmodell-rm/. Stand: 8. August 2023.

[4] Freimut Bodendorf. "Daten und Wissen". In: *Daten- und Wissensmanagement*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–5. ISBN: 978-3-662-06494-8. DOI: 10.1007/978-3-662-06494-8_1. URL: https://doi.org/10.1007/978-3-662-06494-8_1.

[5] Thomas Breuer et al. "Tackling challenges in energy system research with HPC". In: ISC High Performance 2023, Hamburg (Germany), 21 May 2023 - 25 May 2023. May 21, 2023. URL: https://juser.fz-juelich.de/record/1007796.

[6] Peter Bühler, Patrick Schlaich, and Dominik Sinner. *Datenmanagement: Daten – Datenbanken – Datensicherheit*. de. Bibliothek der Mediengestaltung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019. ISBN: 978-3-662-55506-4. DOI: 10.1007/978-3-662-55507-1. URL: http://link.springer.com/10.1007/978-3-662-55507-1.

[7] *Extensible Markup Language (XML) 1.0 (Fifth Edition). 1 Introduction*. https://www.w3.org/TR/xml/#sec-intro. Stand: 5. Juli 2023.

[8] *Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation 5 April 2012*. https://www.w3.org/TR/xml/#sec-starttags. Stand: 5. Juli 2023.

[9] *ISO. Benefits of standards*. https://www.iso.org/benefits-of-standards.html. Stand: 12. Juni 2023.

[10] *ISO. ISO/IEC 25010:2011*. https://www.iso.org/standard/35733.html. Stand: 12. Juni 2023.

[11] Helmut Jarosch. *Grundkurs Datenbankentwurf: eine beispielorientierte Einführung für Studierende und Praktiker: mit 227 Abbildungen, 14 Tabellen und 5 Aufgaben mit Lösungen*. ger. 4., überarbeitete und aktualisierte Auflage. Lehrbuch. Wiesbaden: Springer Vieweg, 2016. ISBN: 978-3-8348-1682-5. DOI: 10.1007/978-3-8348-2161-4.

[12] *JUBE 2.4.1 documentation*. https://apps.fz-juelich.de/jsc/jube/jube2/docu/. Stand: 8. Juni 2023.

[13] *JUBE Benchmarking Environment*. https://www.fz-juelich.de/en/ias/jsc/services/user-support/jsc-software-tools/jube. Stand: 5. Juli 2023.

[14] Sascha Kersken. *IT-Handbuch für Fachinformatiker*. Rheinwerk Verlag, 2019. ISBN: 978-3-8362-1420-9. URL: https://openbook.rheinwerk-verlag.de/it_handbuch/.

[15] Stephan Kleuker. *Grundkurs Datenbankentwicklung: von der Anforderungsanalyse zur komplexen Datenbankanfrage*. ger. 4. Auflage. Lehrbuch. Wiesbaden: Springer Vieweg, 2016. ISBN: 978-3-658-12338-3. DOI: 10.1007/978-3-658-12338-3.

[16] *MongoDB. Build A Python Database With MongoDB*. https://www.mongodb.com/languages/python. Stand: 8. August 2023.

[17] *MySQL. Introduction to MySQL Connector/Python*. https://dev.mysql.com/doc/connector-python/en/connector-python-introduction.html. Stand: 8. August 2023.

[18] *Oracle Deutschland. Was ist Datenmanagement?* https://www.oracle.com/de/database/what-is-data-management/. Stand: 21. April 2023.

[19] *Python documentation. 5. Data Structures*. https://docs.python.org/3/tutorial/datastructures.html. Stand: 5. Juli 2023.

[20] *Python documentation. The Python Tutorial*. https://docs.python.org/3/tutorial/index.html. Stand: 8. Juni 2023.

[21] *Python documentation. xml.etree.ElementTree - The ElementTree XML API*. https://docs.python.org/3/library/xml.etree.elementtree.html. Stand: 8. Juni 2023.

[22] W. Sharples et al. "A run control framework to streamline profiling, porting, and tuning simulation runs and provenance tracking of geoscientific applications". In: *Geoscientific Model Development* 11.7 (2018), pp. 2875–2895. DOI: 10.5194/gmd-11-2875-2018. URL: https://gmd.copernicus.org/articles/11/2875/2018/.

[23] *SQLite Tutorial. SQLite Python*. https://www.sqlitetutorial.net/sqlite-python/. Stand: 8. August 2023.

[24] Ralph Steyer. *Programmierung in Python: ein kompakter Einstieg für die Praxis*. Springer Vieweg, 2018. ISBN: 978-3-658-20705-2.

[25] *Storage. Was ist ein Buffer / Pufferspeicher?* https://www.storage-insider.de/was-ist-ein-buffer-pufferspeicher-a-b0e632ec3c7ce1350948fec31f0efce3/. Stand: 8. August 2023.

[26] *W3C XML Schema Definition Language (XSD) 1.1. 3 Logical Structures*. https://www.w3.org/TR/xml/#sec-starttags. Stand: 5. Juli 2023.

[27] *W3C XML Schema Definition Language (XSD) 1.1. 2 Documents*. https://www.w3.org/TR/xml/#sec-comments. Stand: 8. August 2023.

[28] *Was ist ein Workflow?* https://www.microtech.de/erp-wiki/workflow/. Stand: 8. August 2023.

[29] Julia Wellmann. "Untersuchung der Auswirkung unterschiedlicher Prozess-Pinning Strategien anhand von verschiedenen Benchmarks auf HPC Systemen". Bachelorarbeit. FH Aachen, 2021, 59 p. URL: https://juser.fz-juelich.de/record/896756.

[30] *What is Python? Executive Summary*. https://www.python.org/doc/essays/blurb/. Stand: 5. Juli 2023.

Abbildungsverzeichnis