Alles in einer Hand

Multi-Source Data-Management

Clemens Kujus, Paul Kristian Picht, Lukas Siegert, Cornelius Liepelt, Julius Fäustel

Inhaltsverzeichnis

1. Einleitung

In der Produktion fallen viele sogenannte Qualitätsdaten an, diese sind logisch stark mit Produktionsdaten in ERP-Systemen verknüpft. Sie sind logistisch, kostenrechnerisch oder buchhalterisch relevant. Neben der hohen Datenmenge erzeugen die vielen Sensoren auch einen sehr heterogenen Datenbestand. Durch Veränderungsprozesse im Unternehmen kommt noch eine gewisse Dynamik hinzu, diese entsteht durch Änderungen im Messinstrumentenbestand. Alle diese Daten müssen persistiert werden, auch wenn nur ein Teil davon zunächst relevant erscheint, wäre es fatal, wenn Informationen, die später nützlich sein könnten, einfach verloren gehen würden. Somit benötigt es einen flexiblen, verlässlichen und performanten Datenspeicher.

Da es viele Anbieter auf dem Datenbankmarkt gibt, war es Ziel dieses Projektes vier verschiedene Lösungsansätze an einem Praxisbeispiel zu evaluieren. Dabei sollte eine proprietäre Datenstruktur, eine universelle relationale Struktur, eine Schlüssel-Werte-Datenbank und eine dokumentenorientierte Datenbank zum Einsatz kommen. In allen Lösungen sollten Daten konsistent gespeichert werden. Weiterhin sollten verschiedene Analysen möglich sein und auch deren Ergebnisse dargestellt werden.

2. Problemstellung

2.1. Einführung in das Praxisbeispiel

In dem konkreten zu implementierenden Beispiel ging es um fünf an einem Carbonteil-Fertigungs-Prozess beteiligte Maschinen. Dabei werden teure Carbonfasern mit einem günstigen Trägerstoff vernäht. Ziel dabei ist es eine bessere Handhabbarkeit bei geringeren Herstellungskosten bereitzustellen. Die Maschine erzeugt einen Eingangsdatensatz mit 27 Werten und einen Ausgangsdatensatz mit 18 Werten. Dabei werden neben der Seriennummer, die spezifisch für ein Werkstück (Teil) ist, der Fertigungsauftrag, die Teilart, die Linie und die Ladungsträgernummer gespeichert. Zu einem Fertigungsauftrag gehören mehrere Teile. Weiterhin wird auch die Nummer des Ladungsträgers abgespeichert, dieser ist abhängig von der Teilart und kann für mehrere Werkstücke verwendet werden. Die restlichen Werte geben Aufschluss über die Teilqualität und den Fertigungsablauf.

Das Modell stellt etwas vereinfacht dar, wie die Datensatzerstellung mit der Fertigung auf der Maschine zusammenhängt. Sobald das Teil korrekt in die Maschine eingelegt wurde, wird der Eingangsdatensatz erstellt. Nach der eigentlichen Fertigungsarbeit auf der Maschine wird das Teil überprüft. Dabei entsteht der Ausgangsdatensatz. Falls die Prüfung nicht erfolgreich war, wird das Produkt erneut überprüft. Danach verlässt das Teil die Maschine, falls nun Probleme mit dem Werkstück gefunden wurden, wird das Werkstück erneut in die Maschine gegeben, falls nicht endet der von uns betrachtete Prozess.

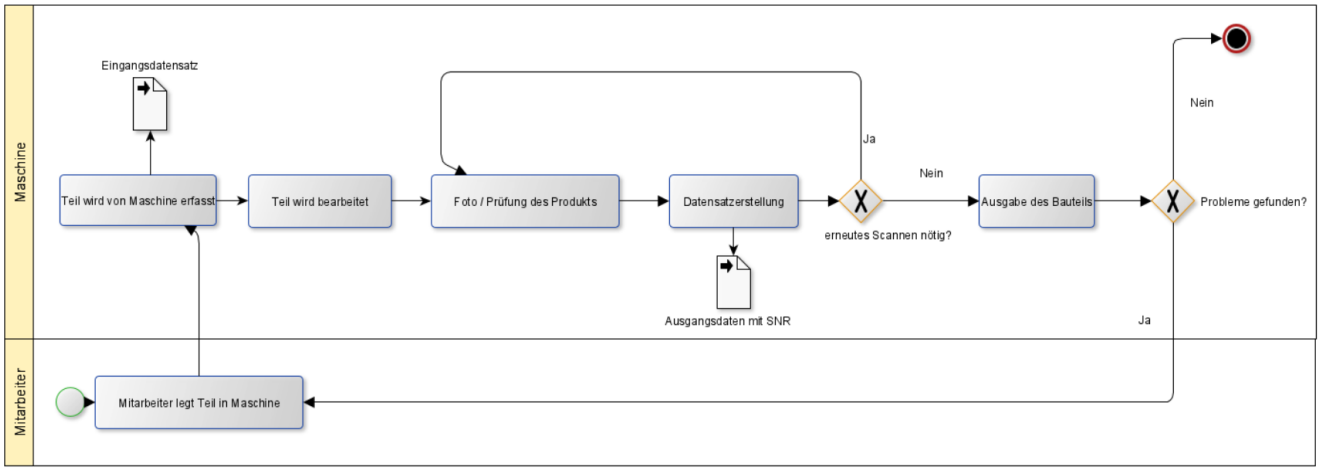

Abbildung 1. Einlesealgorithmus

2.2. Allgemeine Informationen zur Umsetzung

Da die Eingangs- und Ausgangsdatensätze in einer pro Datensatz neu angelegten CSV gespeichert werden, war es eine Anforderung an das System auf diese Events zu reagieren. Dafür wurde ein Watchdog implementiert und da dies mit Python erfolgte und sich alle Teammitglieder auf diese Programmiersprache geeinigt hatten, konnte dieser wiederverwendet werden. Dabei wurde die Bibliothek Wachtdog genutzt, diese erlaubt es mit wenigen Zeilen Veränderungen im Dateisystem festzustellen. Wie im Codebeispiel 1 zu erkennen ist, übergibt dazu einfach das zu überwachende Verzeichnis und bei Änderungen wird das jeweilige Event ausgelöst. Darunter sieht man, wie die Behandlung des genutzten Created Events aussieht. Um herauszufinden welche Datensatzart erstellt wurde, wird durch eine IF-Anweisung überprüft, ob es sich um einen Input Datensatz handelt und wenn ja, dann wird der "InputLoader" aufgerufen, falls das nicht der Fall ist der "OutputLoader". Dabei

wird der richtig aufbereitete Dateipfad mit übergeben.

Listing 1. Code 1 - Wachtdog verkürzt

```
def on_any_event(event):
        if event.event_type == 'created':
            if "input" in event.src_path:
                InputLoader.save(event.src_path.replace('\\','/'))
            else:
                OutputLoader.save(event.src_path.replace('\\','/'))
```

3. State of the Art Lösungen

Um die Lösungen des Projektes zu evaluieren war zunächst ein Blick auf die in der Industrie eingesetzten Technologien notwendig. Dafür werden im folgenden Lösungsansätze aus dem Praxiseinsatz diskutiert.

3.1. Iconics Hyper Historian

Iconics bietet Automatisierungssoftware Lösungen welche Echtzeitinformationen für jede Anwendung visualisieren, historisieren, analysieren und mobilisieren. Dabei spielt der Hyper Historian eine Schlüsselrolle in der Archivierung und Analyse historischer Daten. Dazu wurde von Iconics eine Datenbanklösung selbst entwickelt, diese basiert auf Zeitreihen und hat deshalb den Vorteil der Datenkompression. Nach außen wird ein SQL Interface bereitgestellt.

3.2. Amazon

Amazon hat mit der Amazon DynamoDB ebenfalls eine eigene NoSQL-Datenbank im Portfolio. Dabei werden Ansätze der Schlüssel-Werte und der Dokumentendatenbank kombiniert. Daneben existiert auch noch Amazon Timestream, dies ist ein Zeitreihen-Datenbankservice. Hier lässt sich schon erkennen, dass es in der Datenbankwelt keine passende Lösung für alle Probleme gibt, sondern unterschiedliche Anforderungen zu unterschiedlichen Systemen führen.

3.3. Kontron AIS GmbH

Die Kontron AIS hat je nach Projekt verschiedene Datenbanklösungen im Einsatz, dabei werden oft verschiedene Ansätze kombiniert. So ist meist auf der Auswertungsseite ein relationaler Datenbankserver wie beispielsweise von Oracle oder Microsoft. Da der Aufbau von den Relationen, aber beim Speichern Zeit benötigt, wird mit einem sogenannten Fast Layer gearbeitet. Um diesen zu implementieren werden verschiedene Lösungen genutzt. Zu einem das Big Data System Hadoop. Es ermöglicht eine hoch performante Speicherung in einem redundanten und parallelisierten Dateisystem. Ebenfalls werden als Zwischenspeicher die auch im Folgenden diskutierten Ansätze wie Schlüssel-Werte-Datenbank und dokumentenorientierte Datenbank genutzt, dabei werden diese nicht nur als Datensammlung, sondern auch als Zwischenspeicher für Auswertungen genutzt. Auffallend dabei ist, dass die Lösungen von Projekt zu Projekt variieren und das oft auch nur eine Kombination eine adäquate Problemlösung schafft.

3.4. Rückschlüsse auf das Projekt

Da die Datenlast mit fünf Maschinen und damit 10 Datensätzen pro 5 Minuten recht gering ist, wäre ein Big Data Softwarestack wie Hadoop eine zu übertriebene Lösung, da diese Systeme für mehrere tausend Datensätze pro Sekunde ausgelegt sind. Ein Zeitreihendatenbankservice wäre keine Alternative die Infrage kommt, da diese für das beständige Aufzeichnen von Datenpunkten optimiert sind. Mit dieser Voreingrenzung ist es wahrscheinlich, dass die im Projekt diskutierten Ansätze auch für ein reales Projekt infrage kommen.

4. Analysen

4.1. Analyse 1

Die erste Analyse beschäftigt sich mit der Taktung pro Artikel. Dabei wurde die Differenz zwischen Eingangs- und dem letzten Ausgangsdatensatz einer Seriennummer gemessen. Danach wurde pro Fertigungsauftrag gruppiert und das Minimum, das Maximum und der Durchschnitt ermittelt. Die Ergebnisse wurden auf eine Stunde beschnitten, um nicht zu sehr von Ausreißern beeinflusst zu werden. Die Alternative wäre gewesen nur die Zeiten zu nutzen, die in der Fertigungszeit des Unternehmens liegen, da aber die Auswertungen nur als Grundlage zum Vergleich der Datenbanklösungen dienen sollte, wurde das nicht implementiert.

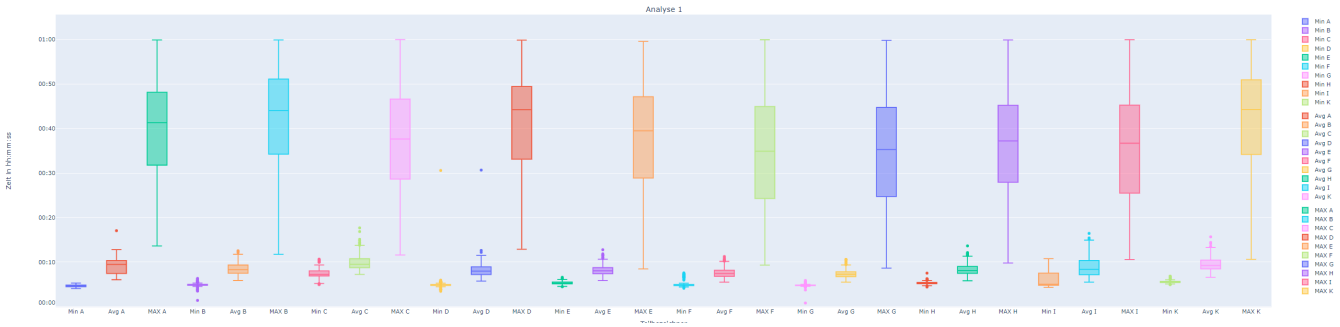

Abbildung 2. Analyse 1

4.2. Analyse 2

Diese Analyse beschäftigt sich nur mit Ausschussartikeln, also Werkstücke die mehrmals in die Maschine eingelegt worden sind, also deren SNR n In-Datensätze besitzt. Dabei wurde zunächst die Differenz zwischen Ende des fehlerhaften Vorgangs und Beginn des neuen Vorgangs gemessen. Auch hier wurde wieder Minimum, Maximum und der Durchschnitt berechnet. Aber über die Gruppierung Teil. Weiterhin wurde der Anteil an fehlerhaften Bearbeitungen an der Gesamtmenge ermittelt.

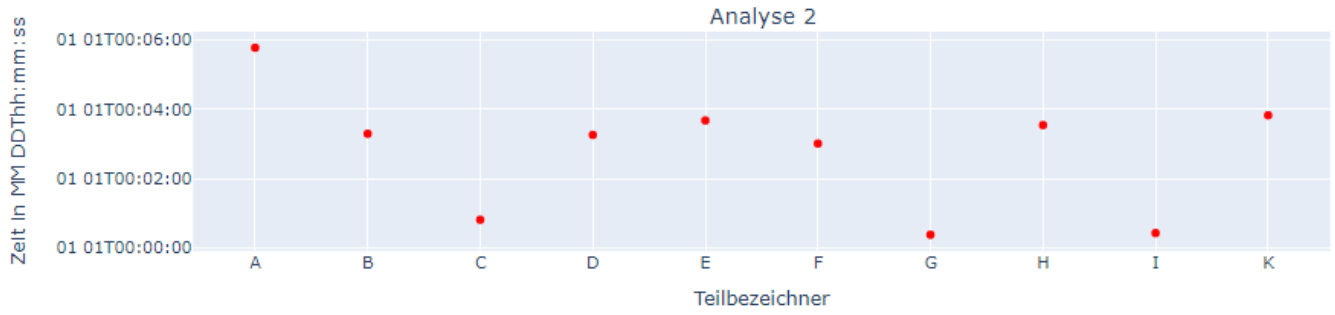

Abbildung 3. Minimum

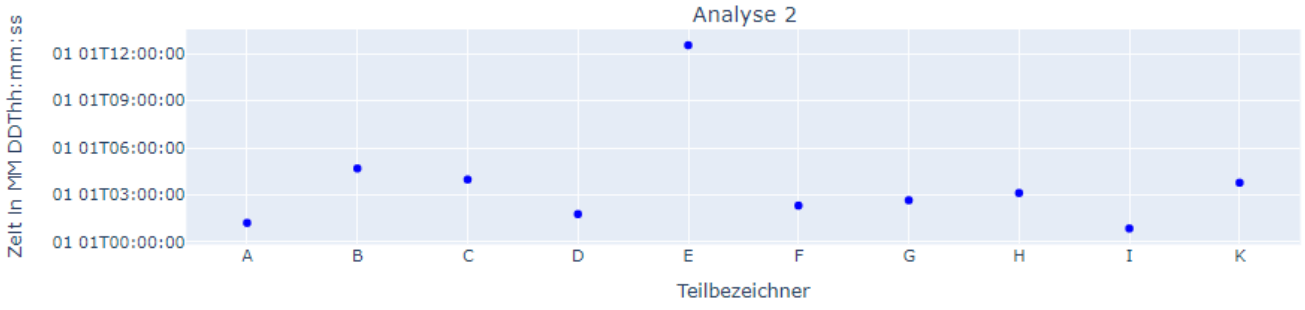

Abbildung 4. Durschschnitt

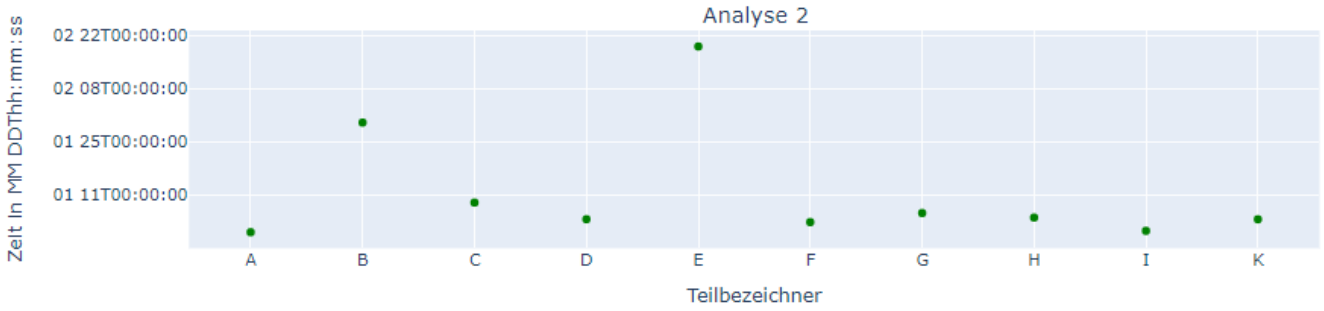

Abbildung 5. Maximum

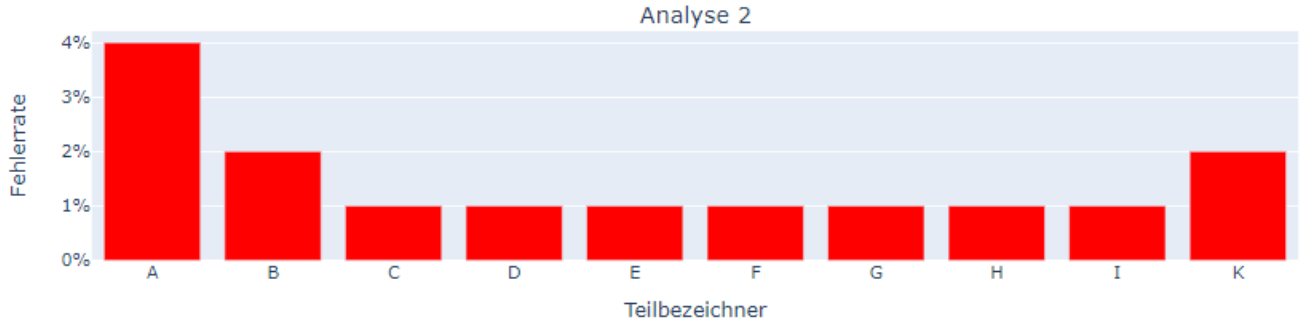

Abbildung 6. Fehlerrate

4.3. Analyse 3

Die dritte Analyse nutzt die gleichen Daten wie Analyse 1., nur mit veränderter Darstellungsform. Ziel war es hierbei Ausreißer zu finden, deshalb wird nun nicht mit Boxplots gearbeitet, sondern mit einem Streudiagramm. Damit ist es besser ersichtlich, wo entfernte Ausreißer zu finden sind.

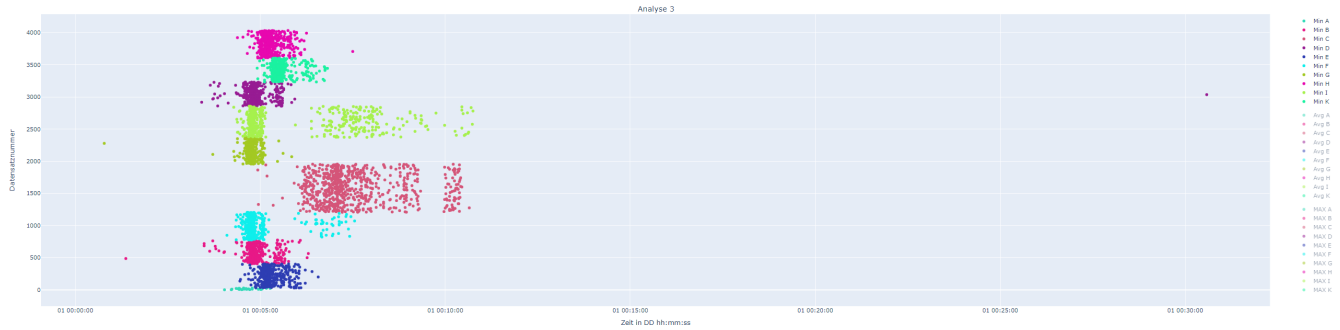

Abbildung 7. Minimum

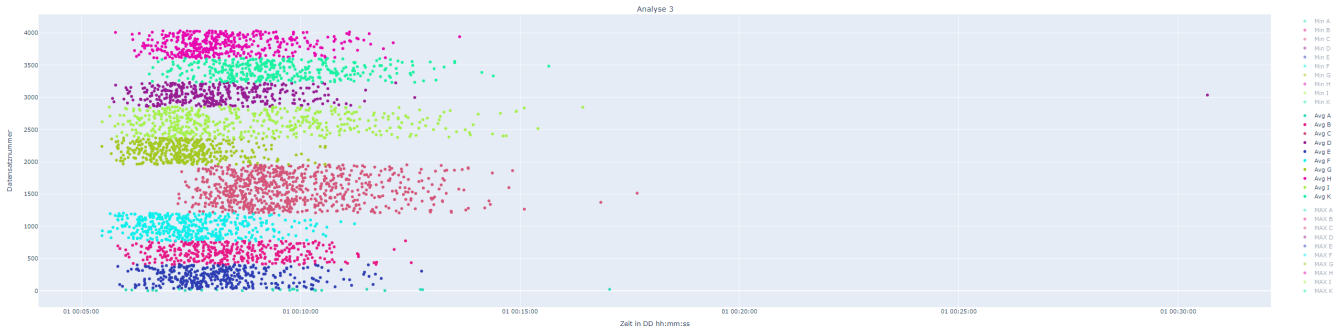

Abbildung 8. Durschschnitt

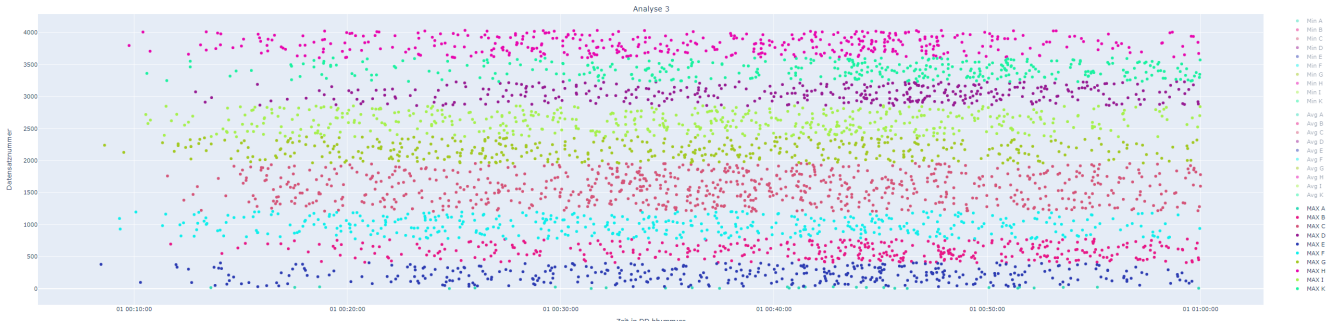

Abbildung 9. Maximum

4.4. Analyse 4

Die vierte Analyse zeigt die Nutzungszeit der Ladungsträger. Dabei ist die Nummer an der Y-Achse irrelevant und stellt jeglich die Datensatznummer dar, denn in den Ladungsträgernummern gab es eine zu große Streuung, um diese adäquat darzustellen.

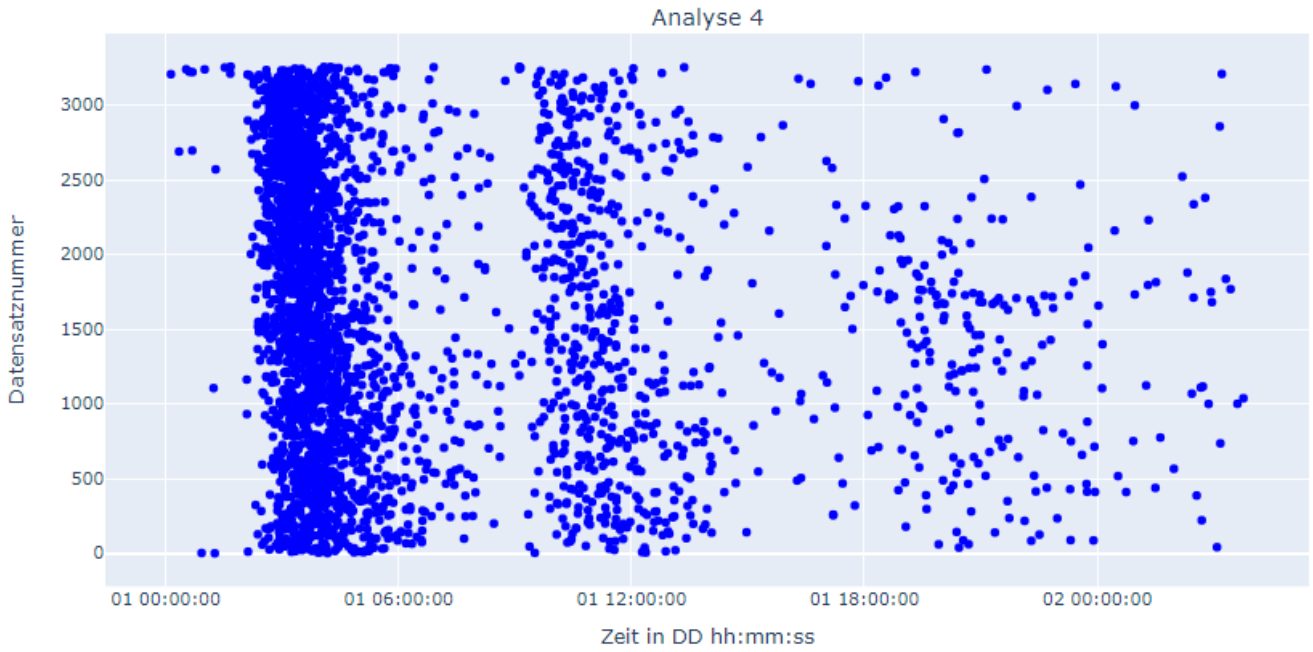

Abbildung 10. Analyse 4

4.5. Analyse 5

Die fünfte Analyse ist analog zur ersten Analyse, nur das dieses Mal die Zwischenaggregation nicht pro Fertigungsauftrag , sondern pro Ladungsträger erfolgte. Danach wurden die Daten wieder nach Teil sortiert und jeweils ein Boxplot für Minimum, Maximum und den Durchschnitt gezeichnet.

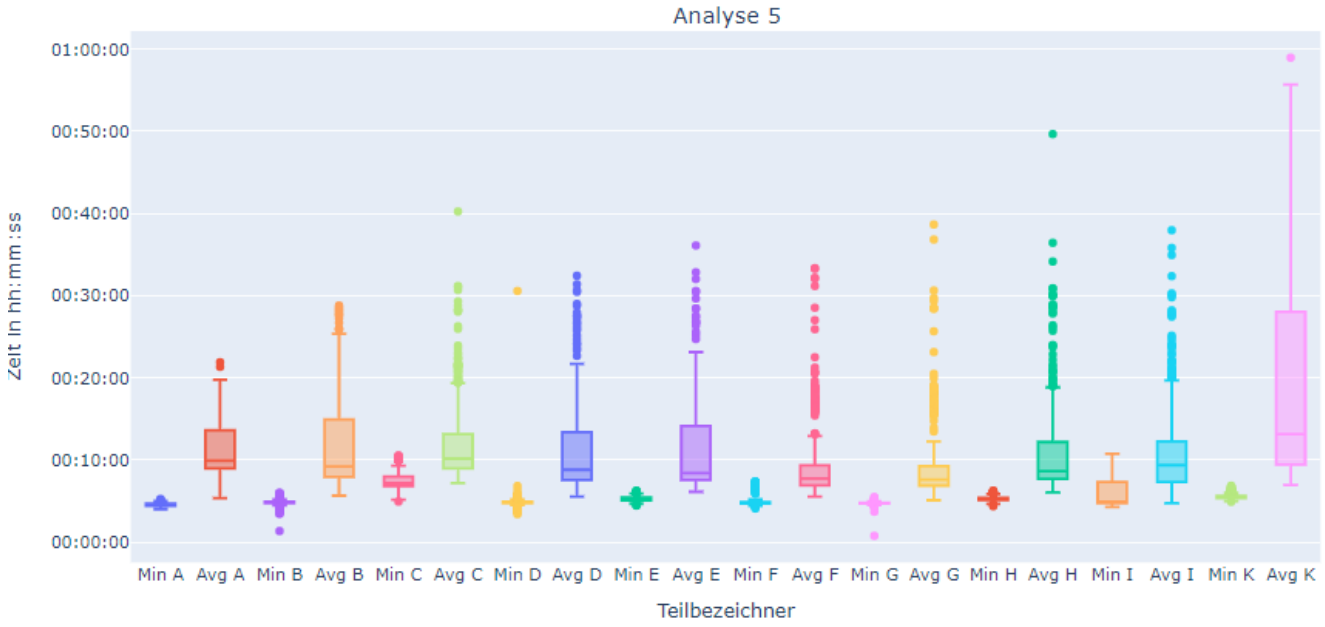

Abbildung 11. Minimum und Durchschnitt

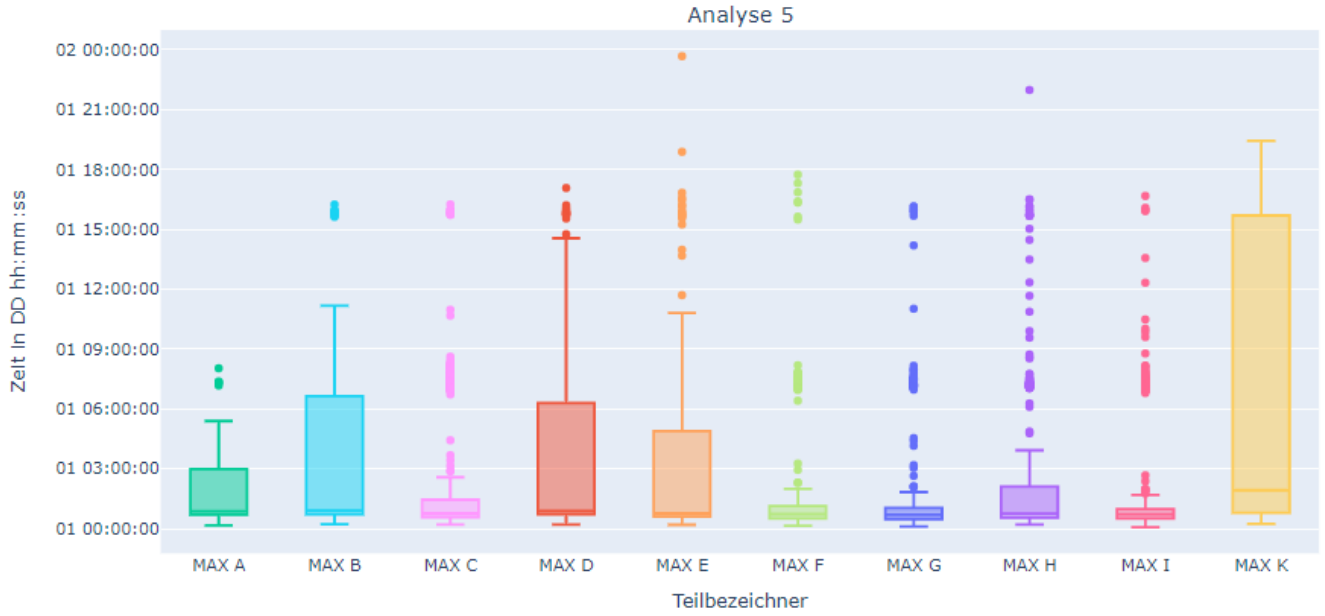

Abbildung 12. Maximum

4.6. Analyse 6

Die letzte Analyse zeigt die Umrüstzeiten zwischen den Teilen. An der Y-Achse stehen die Teile, von denen auf die Teile an der X-Achse umgerüstet wurde. Je heller die Farbe ist, desto höher ist der Wert. Hier wurde Minimum, Maximum und der Durchschnitt pro Umrüstvorgang bestimmt.

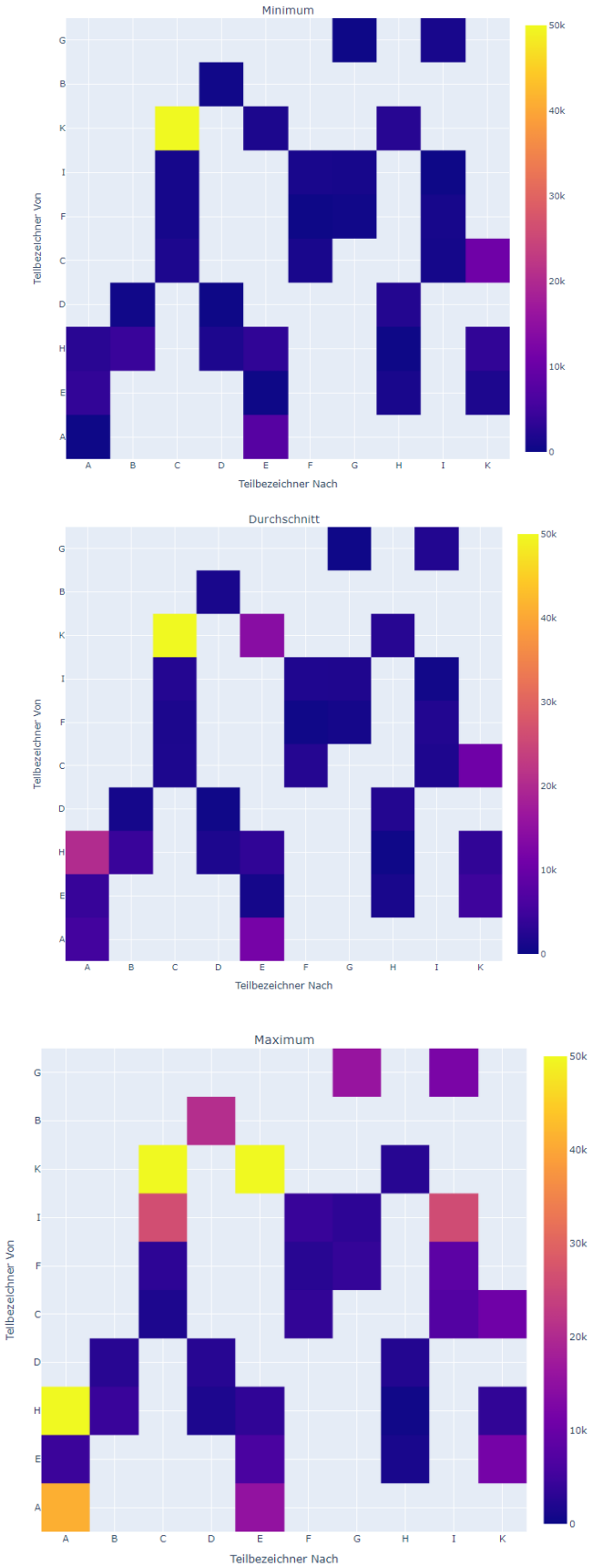

Abbildung 13. Maximum

5. Vorstellung der Lösungen

5.1. Proprietäre Datenstruktur

5.1.1. Einführung in Datenbankstruktur

Datenbankvorstellung

Die proprietäre Datenstruktur basiert auf einem relationalen Datenmodell, welches als Standard in der Datenbankentwicklung genutzt wird. Zugrunde liegen diesem vier Elemente: Tabellen, Attribute, Beziehungen und die Grundlage der relationalen Algebra. Diese ist auch die Grundlage für die Datenbanksprache SQL.
Das relationale Datenbankenmodell besteht aus einer Ansammlung von Tabellen, die durch Beziehungen miteinander verknüpft sind. In jeder Tabelle werden die Datensätze als Tupel (Zeilen) gespeichert. Jedes Tupel besteht aus mehreren Attributen, welche als Spalten in einer Tabelle bekannt sind.

Listing 2. Code 1 - Beispiel für die Erstellung einer Tabelle

```
CREATE TABLE `Test1` (
    `id` int NOT NULL AUTO_INCREMENT,
    `vorname` varchar(255) NOT NULL,
    `nachname` varchar(255) NOT NULL,
    PRIMARY KEY ('id')
)
```

Unter der proprietären Datenstruktur versteht man ein herstellerspezifisches Modell, welches zur Verwaltung von nutzerspezifischen Daten entwickelt und dessen Weiterentwicklung eingeschränkt wird. Um dies zu realisieren wird die Data Definition Language (DDL) verwendet. Sie macht eine herstellerspezifische Entwicklung möglich, allerdings schließt diese gleichzeitig eine aufwendige Weiterentwicklung mit ein.

Listing 3. Code 2 - Beispiel für die Weiterentwicklung

```
ALTER TABLE `Test1`
ADD COLUMN `alter` int,
DEFAULT NULL;
```

Der Vorteil der proprietären Datenstruktur besteht in einer spezifischen Entwicklung eines Datenmodells, welche einen geringen Entwicklungsaufwand bedarf. Dies belegt das Codebeispiel 1, währenddessen erkennt man im Codebeispiel 2, dass eine Weiterentwicklung der Datenstruktur viele Systemtest mit sich zieht und damit viel Zeit kostet.

5.1.2. Entwurf

Konzeptioneller Entwurf

Der konzeptionelle Datenbankentwurf umfasst die Datenstruktur, die Semantik sowie die Beziehungen und Integritätsbedingung in einem Datenbankmodell. Er spiegelt die Ergebnisse der

Anforderungsanalyse unseres Auftraggebers wider.
Um dies abstrakt darzustellen, wurde ein Entity-Relationship-Model (ERM) modelliert. (Abbildung 14)

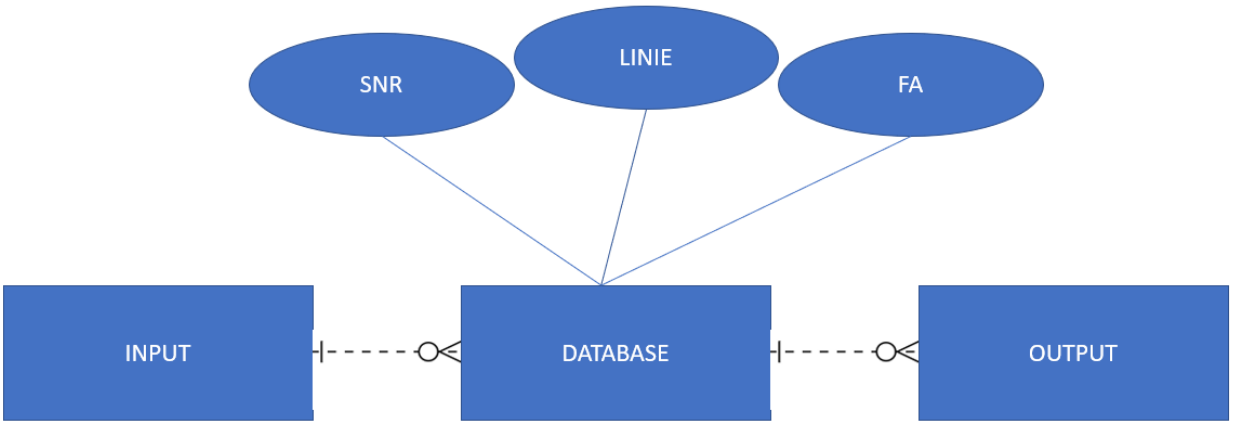

Abbildung 14. Entity-Relationship-Model

Abbildung 14 zeigt ein ER-Modell, bestehend aus grundlegenden Modellelementen.
Die Entitäten werden durch die blauen Rechtecke dargestellt, in diesem Beispiel Input, Output sowie Database. Attribute sind an Entitäten verknüpft und werden hier als blaue Ovale dargestellt. Diese klassifizieren, identifizieren und charakterisieren Entitäten, bei Database sind es hier zum Beispiel SNR (Seriennummer), Begintime und Endtime.
Zwischen den Entitäten sind Beziehungen zu erkennen, in diesem Fall erkennt man 1:m Beziehungen. Eine Seriennummer, welche ein Tupel in Database darstellt, kann mehrere In- bzw. Outputs besitzen. Jedoch kann ein Tupel in In- bzw. Output nur ein Tupel in Database besitzen.

Logischer Entwurf

Der konzeptionelle Entwurf ist die Basis für ein relationales Datenmodell, in einem logischen Entwurf.

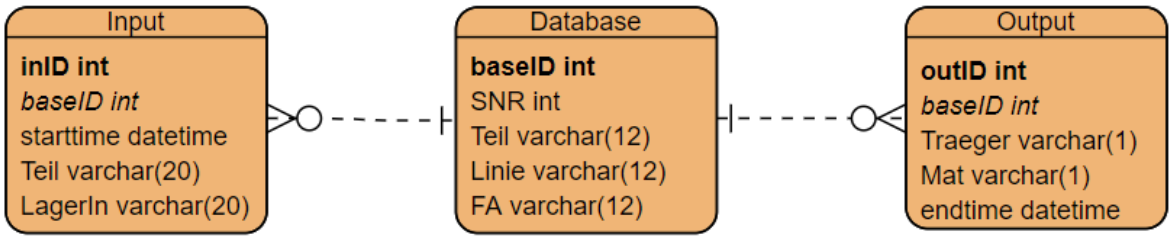

Abbildung 15. relationales Datenmodell

Das relationale Datenmodell, in Abbildung 15, basiert auf der Grundlage des ER-Modells, in Abbildung 14. Es spiegelt die logische Datenstruktur, Datentabellen, Ansichten und Indizes wider.
Die Input- sowie Output-Tabelle enthält in jedem Tupel eine ID als Primary Key, baseID als Foreign Key und andere Input-/Output Informationen. Jedes Attribut (jede Information) befindet sich in einer eigenen Spalte, wobei jedem Tupel eine eindeutige ID zugeordnet werden kann. Die Database-Tabelle enthält in jedem Tupel eine ID als Primary Key, eine SNR (Seriennummer), welches das Teil eindeutig identifiziert, sowie andere Attribute. Spezielle Informationen zum Input- bzw. Output-Datensatz werden nicht in der Database-Tabelle gespeichert, sondern direkt in der Input-/ Output-Tabelle.
Diese Tabellen sind mit dem Attribut BaseID miteinander verknüpft. Aufgrund diesem kann eine Beziehung zwischen den Tabellen hergestellt werden. Durch eine Abfrage der Seriennummer (SNR) in

der Database-Tabelle erhält man die passende BaseID. Durch diese kann man die Input- und Output-Informationen aus den anderen Tabellen abrufen.

5.1.3. Implementierung

Systemvoraussetzungen

Alle Implementierungen wurden unter den folgenden Voraussetzungen vorgenommen:

- Windows 10 Home 64-bit
- Intel Core i7-1065G7
- 16 GB RAM

Datenbank

Allgemein

Über ein relationales Datenbankmanagementsystem (RDBMS) kann die proprietäre Datenstruktur realisiert werden. Es wurde über XAMPP, einem kostenlosen Programmpaket von freier Software, welche meiste für ein Apache Webserver verwendet wird, ein MySQL Server 8.0 installiert und konfiguriert.

Abbildung 16. XAMPP

Um das relationale Datenmodell einfach zu realisieren, wurde zusätzlich HeidiSQL installiert, ein freier Client für das Datenbanksystem MySQL. Es wird hierbei die Structured Query Language (SQL) genutzt.

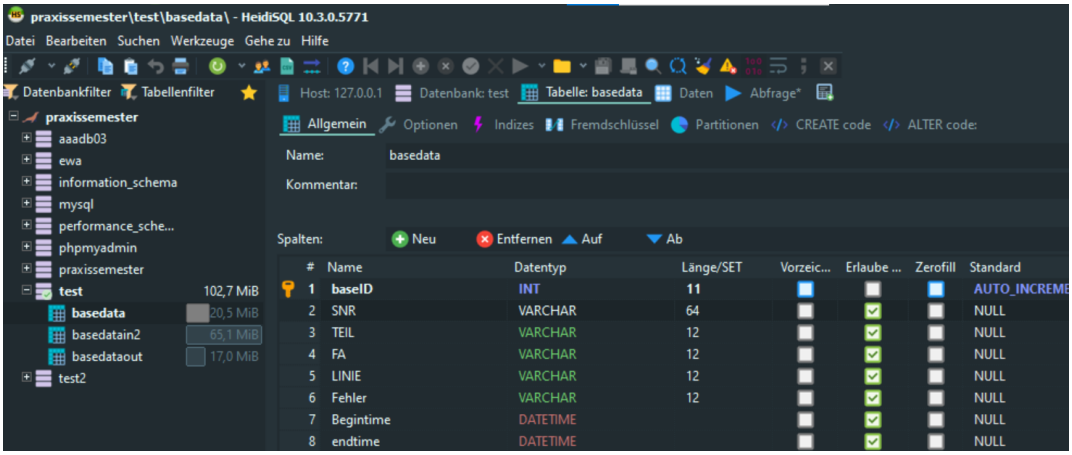

Abbildung 17. HeidiSQL

Um dies zu verdeutlichen folgen Quelltextbeispiele. Diese zeigen wie eine Tabelle angelegt, Daten

eingetragen, sowie Fremdschlüsselbeziehung erstellt werden.

Listing 4. Code 3 - Anlegen der Tabelle Database

```
CREATE TABLE `Database`
(
  baseID    int NOT NULL AUTO_INCREMENT,
  SNR       varchar(18) NOT NULL,
  FA        varchar(12),
  TEIL      varchar(12),
  LINIE     varchar(12),
  Fehler    varchar(12),
  Begintime datetime,
  Endtime   datetime,
  PRIMARY KEY (baseID)
);
```

Listing 5. Code 4 - Daten in Database eintragen

```
INSERT INTO `Database` (`SNR`,`FA`,`TEIL`,`LINIE`,`Fehler`,`Begintime`,`Endtime`)
    VALUES ('1923219423129', '009606', 'A', '1', '0', '2018-01-02T05:47:45', '2018-01-
02T05:48:45');
```

Listing 6. Code 5 - Fremdschlüsselbeziehung anlegen

```
ALTER TABLE `Database` ADD CONSTRAINT FKDatabase FOREIGN KEY (`baseID`) REFERENCES
`Input` (`baseID`);
```

Um gezielte Abfragen zu tätigen, die später für die Analyse benötigt werden, bedarf es ein SELECT-STATEMENT mit einer WHERE-CLAUSEL. Dadurch wird eine gute Performance gesichert.

Listing 7. Code 6 - SELECT-Statement inkl. WHERE-Clausel

```
SELECT * FROM `Database` WHERE `baseID` = `2`
```

Messung der Ausführungszeit

Um einen Vergleich der Abfragezeit zu gewährleisten, ist es nötig die Zeit zu messen welche die Datenbank braucht, um eine Abfrage durchzuführen. Dies ist über den folgenden Befehl zu realisieren.

Listing 8. Code 7 - Abfragezeit messen

```
SET STATISTICS TIME ON
```

Anwendungen

Programmiersprache

Die Implementierung der Anwendungen wurde mit der Programmiersprache Python vorgenommen. Als Entwicklungsumgebung, innerhalb des Projektes, wurde Notepad ++ genutzt. Für die

Implementierung wurde die Python-Version 3.7.3 genutzt.
Die Ausführung des Python-Programmes wurde über die Kommandozentrale (CMD), von Windows 10, vorgenommen.
Dies wird in im Codebeispiel 8 verdeutlicht.

Listing 9. Code 8 - Ausführung des Python-Programmes über CMD

```
>python analyse.py
```

Kommunikation zwischen Anwendung und Datenbank

Durch die Python-Bibliothek mysql-connector-python wurde eine Verbindung zwischen Anwendung und MySQL-Datenbankserver hergestellt.
Dies kann man über die Kommandozentrale installieren, wie im Codebeispiel 9 beschrieben.

Listing 10. Code 9 - Installation Python-Bibliothek über CMD

```
>pip install mysql-connector-python
```

Die installierte Bibliothek wird nun in die Anwendung eingebunden, um eine Verbindung zur Datenbank herstellen zu können. Dazu müssen folgende Parameter gesetzt werden um einen Zugriff zu erhalten: host, database, user und password.

Listing 11. Code 10 - Herstellung der Verbindung zwischen Anwender und Datenbank

```
import mysql.connector

connection = mysql.connector.connect(host = "127.0.0.1", user = "root", password =
"1234", database = "test")
```

Nach dem Aufbau der Verbindung zur Datenbank, wird ein Cursor gesetzt. Diesem wird per Cursor-Funktion execute eine SELECT-Abfrage übergeben. Zusätzlich dazu bietet der Cursor Varianten an, um die Ergebnismenge bereitzustellen. Zum Beispiel:

- *fetchall()*, welcher die gesamte Ergebnismenge wiedergibt
- *fetchone()*, welcher nur die erste Zeile der Ergebnismenge wiedergibt

Listing 12. Code 11 - Erstellung eines Cursor's sowie einer SELECT-Abfrage

```
cursor = connection.cursor()
cursor.execute("SELECT * FROM `Database`")
result = cursor.fetchall()
```

Um die execute auszuführen, bedarf es ein Commit, welches die Anweisung ausführt. Nach dem Ausführen, wird der Cursor, sowie die Connection per close() Funktion geschlossen.

Listing 13. Code 12 - Commit und Close der Connection

```
connection.commit()

connection.close()
cursor.close()
```

Datenloader

Die Voraussetzung für den Datenloader ist eine Textdatei, welche alle Informationen für den Input bzw. Output enthält. Das Vorgehen für die Input- und Output-Sätze ist im Allgemeinen gleich. Der Unterschied liegt in der Überprüfung, des zeitigen Starttermins (Begintime) und des spätesten Endtermins (Endtime). Dies wird in Abbildung 18 virtualisiert.

Die Textdateien werden über den Watchdog, welcher bereits erläutert wurde, zur Verfügung gestellt. Das Einlesen der Dateien wird über den folgenden Pythoncode dargestellt. In diesem wird das Dokument geöffnet und in einen String verpackt. Dies ermöglicht es die erhalten Informationen über ein Insert-Statement in die Datenbank zu speichern.

Listing 14. Code 13 - Daten auslesen

```
datei   = open(filename, 'r')
val     = dati.read()
dat     = val.split(';')
```

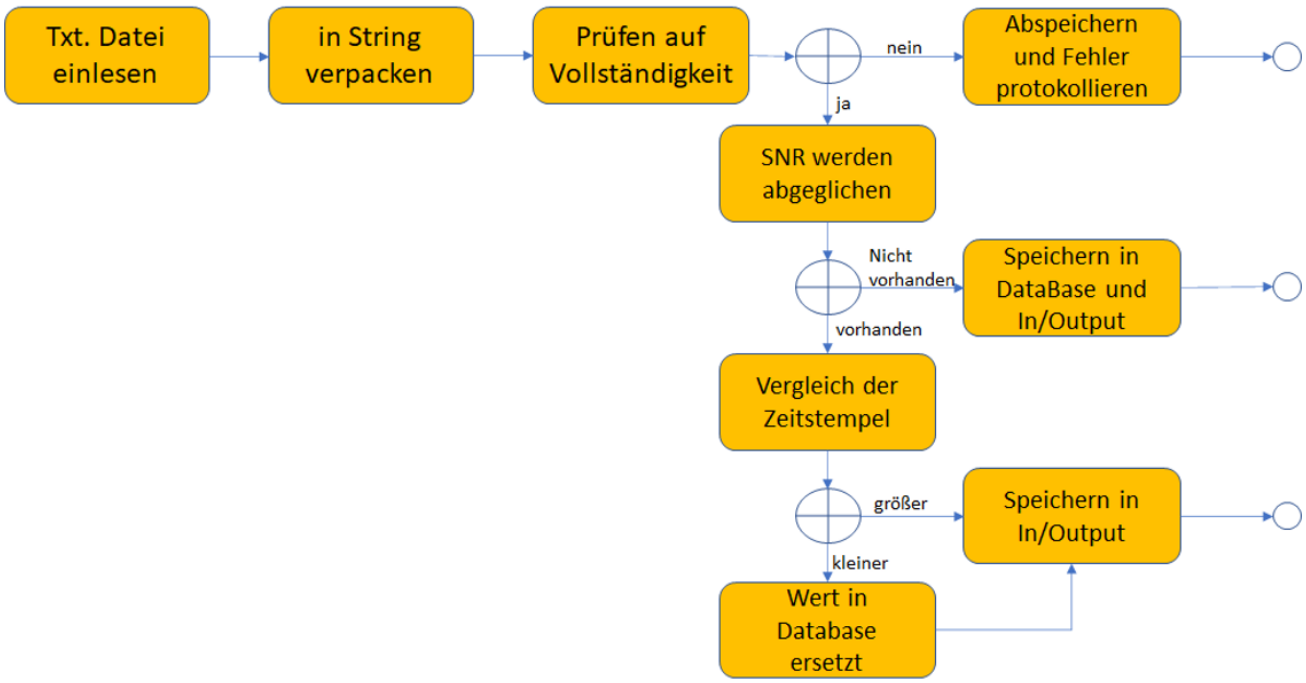

Abbildung 18. Einlese-Algorithmus

Zunächst wird eine Textdatei eingelesen, in einen String verpackt und auf Vollständigkeit geprüft. Wenn die Vollständigkeit der Daten nicht gewährt ist, wird der Datensatz mit einem Fehlercode vermerkt und in der Input- bzw. Output- Tabelle gespeichert. Nachdem die Überprüfung positiv abgelaufen ist, vergleicht der Algorithmus nun ob die Seriennummer (SNR) in der Database-Tabelle vorhanden ist.

Nachdem keine SNR gefunden wurde, wird der Datensatz in der Database- sowie Input- oder Output-Tabelle gespeichert, die baseID stellt dabei die Verknüpfung der Tabellen her. Der andere Fall wäre, dass eine SNR in der Database-Tabelle schon vorhanden ist. In dem Zusammenhang wird der Zeitstempel überprüft, im Fall des Inputdatensatzes der Startzeitpunkt. Wenn dieser kleiner ist als das bisherige Datum, wird er ersetzt. Der restliche Datensatz wird ebenso in der Input- bzw. Output-Tabelle gespeichert und miteinander verknüpft.

Analysen

Allgemein

Zunächst wurde über die Möglichkeiten der Implementierung der Analysen diskutiert. Dafür gab es im Allgemeinen drei Möglichkeiten.

Die erste Möglichkeit sah vor, alle Analysen in SQL zu schreiben und über HeidiSQL auszuführen. Die gewonnen Daten hätten man in MS Excel importieren können, um diese darin auszuwerten und grafisch darzustellen. Diese Möglichkeit wurde zunächst in Betracht gezogen, eine zuverlässige Umsetzung wurde mit dieser Methode generiert. Allerdings war diese auch durch den Wechsel zwischen HeidiSQL und MS Excel zeitaufwendig.
Um dieses Problem zu lösen, wurde Möglichkeit zwei eingeführt. Mit Devart, einem freiverfügbaren Data Integration Tool, konnte eine Abfrage direkt in Excel ausgeführt werden. Mit der Methode konnte eine Zeitersparnis festgestellt werden, die durch das Wegfallen von HeidiSQL zustande gekommen ist. Das Problem hierbei war die Datenintegration mit den anderen Datenstrukturen.
Möglichkeit drei ist die Abfrage der Daten mit Python, wie es in der Anwendung beschrieben ist. Die erfolgreiche Analyse wurde in einer CSV-Datei gespeichert. Mit dieser konnte die Analyse erfolgreich durchgeführt und mit den anderen Datenstrukturen verglichen werden.

Im Allgemeinen hat die Vorarbeit mit Möglichkeit eins und zwei die Entwicklung der finalen Lösung beschleunigt. Um dies zu verdeutlichen befinden sich im Nachfolgenden die Analyseabfragen mit Pseudocode.

Analyse 1

Listing 15. Code 14 - Taktung pro Artikel

```
SELECT teil, fa, COUNT(baseid) AS COUNT,
  MIN(UNIX_TIMESTAMP(endtime)-UNIX_TIMESTAMP(begintime)) AS MIN,
  MAX(UNIX_TIMESTAMP(endtime)-UNIX_TIMESTAMP(begintime)) AS MAX,
  AVG(UNIX_TIMESTAMP(endtime)-UNIX_TIMESTAMP(begintime))AS avg
  FROM `database`
  WHERE endtime is not null
  GROUP BY FA
  ORDER BY teil, fa
```

Analyse 2

Listing 16. Code 15 - Auftrennung

```
SELECT `TEIL`,
  COUNT(snrid) AS Anzahl,
  min(unix_timestamp(endtime) - unix_timestamp(Begintime)) AS MinFertigungszeit,
  max(unix_timestamp(endtime) - unix_timestamp(Begintime)) AS MaxFertigungszeit,
  avg(unix_timestamp(endtime) - unix_timestamp(Begintime)) AS AVGFertigungszeit
  FROM `database`
  WHERE endtime IS NOT NULL
  GROUP BY `TEIL`
```

Analyse 4

Listing 17. Code 16 - Analyse 4

```
SELECT a.Lagerin,
  max(unix_timestamp(b.endtime))-MIN(unix_timestamp(a.Begintime)) AS Dauer,
  min(a.Begintime) AS Start,
  max(b.endtime) AS Ende,
  COUNT(b.baseID)
  FROM basedatain2 AS a JOIN basedata AS b ON a.baseID = b.baseID
  where a.lagerout > 0
  GROUP BY a.Lagerin
```

Analyse 5

Listing 18. Code 17 - Analyse 5

```
SELECT a.TEIL, a.Lagerin,
  count(b.baseid),
  (min(unix_timestamp(b.endtime)-unix_timestamp(b.begintime))) AS MinFertigungszeit,
  (max(unix_timestamp(b.endtime)-unix_timestamp(b.begintime))) AS MaxFertigungszeit,
  (avg(unix_timestamp(b.endtime)-unix_timestamp(b.begintime))) AS AVGFertigungszeit
  FROM `basedatain2` as a Join basedata as b on a.baseid=b.baseid
  WHERE b.endtime IS NOT NULL  AND a.Lagerout > 0
  GROUP BY a.lagerin
  ORDER BY a.TEIL
```

Analyse 6

Listing 19. Code 18 - Analyse 6 - Pseudocode

```
Select `LINIE` from basedatain2 group by LINIE ORDER BY linie asc

Linies in row umwandeln

SELECT fa, teil, min(unix_timestamp(begintime)),
Max(unix_timestamp(begintime))
from basedatain2 WHERE LINIE = '"+row[0]+"' group by FA order by begintime asc

diff = dauer
if str(diff) >= str(0):
    if index == -1:
          * aktuelle Länge der Ergebnisliste berechnen
          * neues Element am Ende der Ergebnisliste einfügen
    else:
          * vorhandene Werte aus Liste auslesen
          * Prüfen, ob min/max verändert werden müssen
          * Dauer zur AVG Berechnung hinzufügen
          * Dauer zurück in Liste schreiben
else:
    break
```

Messung der Ausführungszeit

Um die Messung der Ausführungszeit der Analysen in Python zu realisieren, wurde die Zeit vor und nach der Ausführung gemessen. Die Differenz zwischen Start und Ende ergibt die Ausführungszeit und wird in Sekunden zurückgegeben.

Listing 20. Code 19 - Ausführungszeit messen

```
import time

start = time.time()
#Ausführen der Analyse
ende = time.time()

print('{:5.3f}s'.format(ende-start))
```

Auswertung

Allgemein

Nach der Auswertung der Analysezeiten (Prozesszeiten) bzw. der Antwortzeiten von MySQL, wurde in diesem Zusammenhang eine Grafik in MS Excel erstellt. Diese zeigt deutlich, dass die Antwortzeiten in MySQL kürzer sind als die Prozesszeiten in Python. Grund dafür ist Datenverarbeitung nach der Abfrage in Python, dies kostet zusätzlich mehr Zeit.

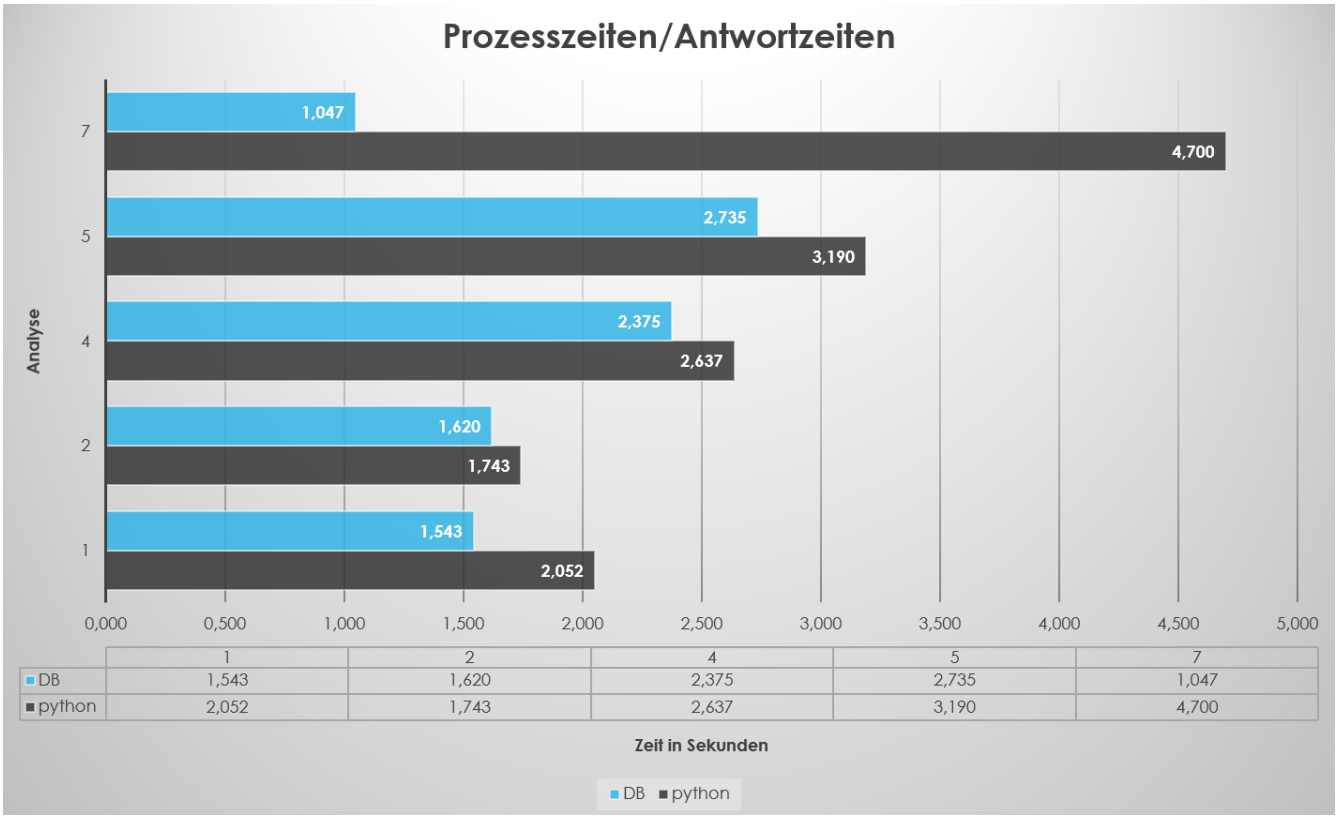

Abbildung 19. Prozesszeiten

Lessons Learned

Schlussendlich erhält man mit der proprietären Datenstruktur eine schnelle und zuverlässige Auswertung der eingelesenen Daten. Jedoch erkennt man bei dieser Struktur, dass eine Weiterentwicklung nur mit mehr Zeitaufwand möglich ist.

Für eine individuelle und schnelle Lösung ist dieses Modell gut geeignet.

5.2. Universelle relationale Struktur

5.2.1. Einführung in Datenbankstruktur

Datenbankvorstellung

Grundlage dieser Datenstruktur ist, wie bei der proprietären Struktur, ein relationales Datenmodell. Dieses Datenmodell soll die speziellen Daten beliebiger Struktur aufnehmen.
Jedoch ist im Vergleich zur proprietären Struktur, ein weiteres Ziel, dass die Struktur beliebig, weitere spezielle Daten aufnehmen kann, ohne dass diese erweitert werden muss. Genauer betrachtet, bedeutet dies, dass zur Aufnahme neuer Daten lediglich Data Manipulation Language (DML), anstatt Data Definition Language (DDL), verwendet werden soll.

Listing 21. Code 1 - Beispiel für Data Manipulation Language

```
INSERT INTO Table1 VALUES('Test', 123);
```

Listing 22. Code 2 - Beispiel für Data Definition Language

```
ALTER TABLE Table1 ADD COLUMN zipCode CHAR(5);
```

Um bereits jetzt einen Grund für die Umsetzung und gleichzeitig einen Vorteil dieser Struktur zu nennen, ist es wichtig zu verstehen, dass mit Codebeispiel 1 ausschließlich Tupel in eine Relation geschrieben werden, währenddessen im Codebeispiel 2 die Datenstruktur verändert wird, was viele Systemtest und damit einen Aufwand mit sich bringt.

5.2.2. Entwurf

Konzeptioneller Entwurf

Der konzeptionelle Entwurf basiert auf der Anforderungsanalyse und beschreibt den abzubildenden Weltausschnitt (T. Kudraß, 2015, S.46). "Die Beschreibung erfolgt unabhängig von der Realisierung in einem konkreten Datenbankmanagementsystem (DBMS) und unabhängig von konkreten Anwendungen, um eine stabile Basis für die weiteren Entwurfsphasen zu besitzen." (T. Kudraß, 2015, S.46)
Eine abstrakte Modellierung wurde mittels eines Entity-Relationship-Models (ERM) umgesetzt. (siehe Abbildung 1)

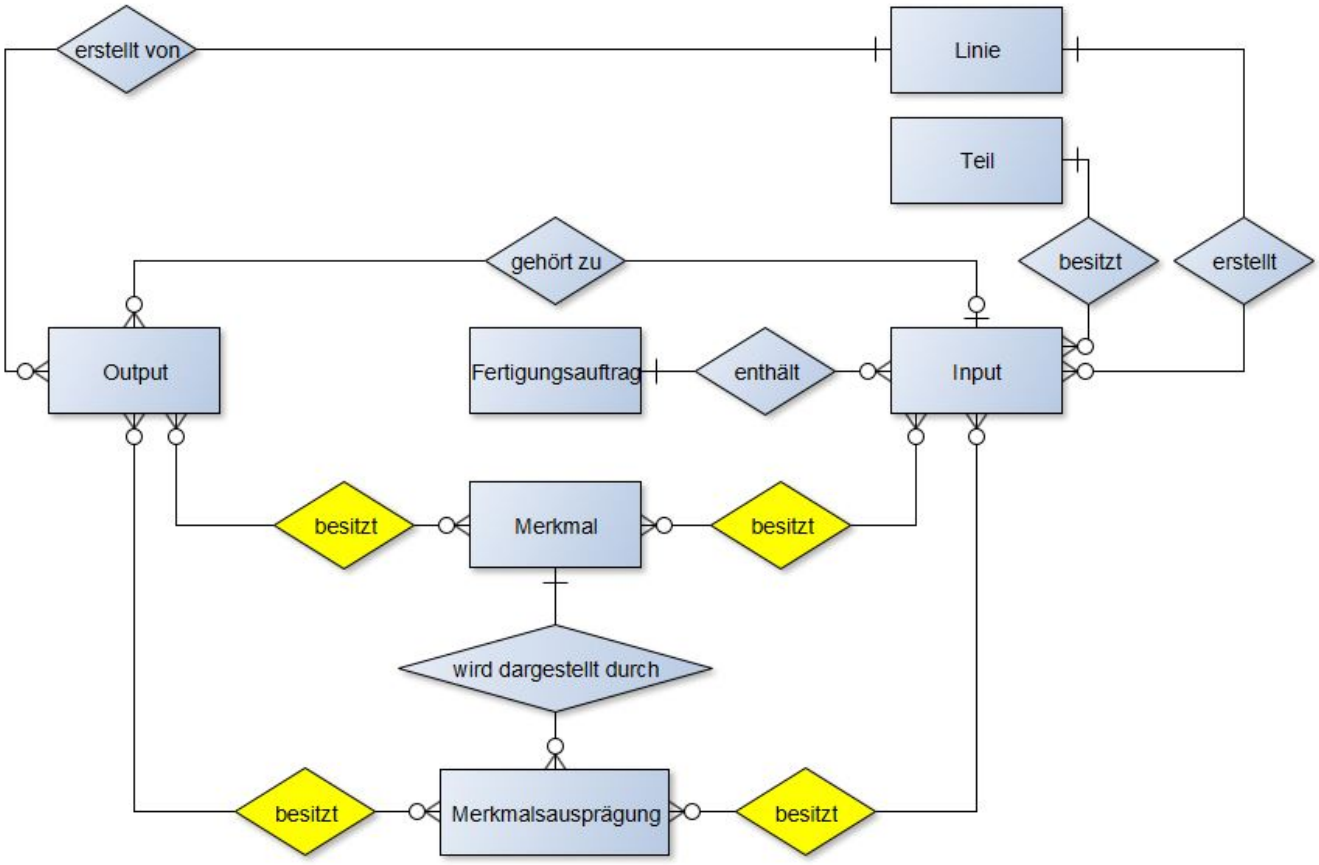

Abbildung 20. Entity-Relationship-Model

In Abbildung 1 kann man drei grundlegende Modellelemente eines ERM erkennen.
Als blaue Rechtecke sind Entitäten (auch Entity) dargestellt, welche im Allgemeinen ein abgrenzbares Objekt der Realität darstellen. Im Fallbeispiel wären dies Input, Output, Fertigungsauftrag, Linie und Teil. Diese Entitäten besitzen normalerweise Attribute bzw. spezielle Eigenschaften, welche die Entitäten charakterisieren, identifizieren und klassifizieren. Um jedoch eine universelle relationale

Struktur zu entwickeln, ist es notwendig die Eigenschaften, welche in keinem Verhältnis zu einer anderen Entität stehen, als eigene Entität zu betrachten. Da Attribute im Gegensatz zu Entitäten Wertausprägungen haben, ist es erforderlich, die Ausprägungen der Entität Merkmal in der Entität Merkmalsausprägung darzustellen.
"Zwischen Entities werden Beziehungen oder Relationships definiert. Eine Beziehung ist die logische Verknüpfung von zwei oder mehreren Entities." (T. Kudraß, 2015, S.51) Diese Beziehung sind in Abbildung 1 als Rauten dargestellt. Die schwarzen Verbindungen zwischen Entitäten und Beziehungen beschreiben die Komplexitätsgrade.
Im Fallbeispiel besitzen nur die Entitäten Input und Output spezielle Eigenschaften. Deshalb haben nur diese Entitäten eine Beziehung (gelb markiert) zu Merkmal und Merkmalsausprägung, aber es kann auch jede andere Entität diese Beziehungen besitzen.

Logischer Entwurf

Ziel des logischen Entwurfs ist es, das konzeptionelle Datenmodell in ein relationales Datenmodell zu überführen.

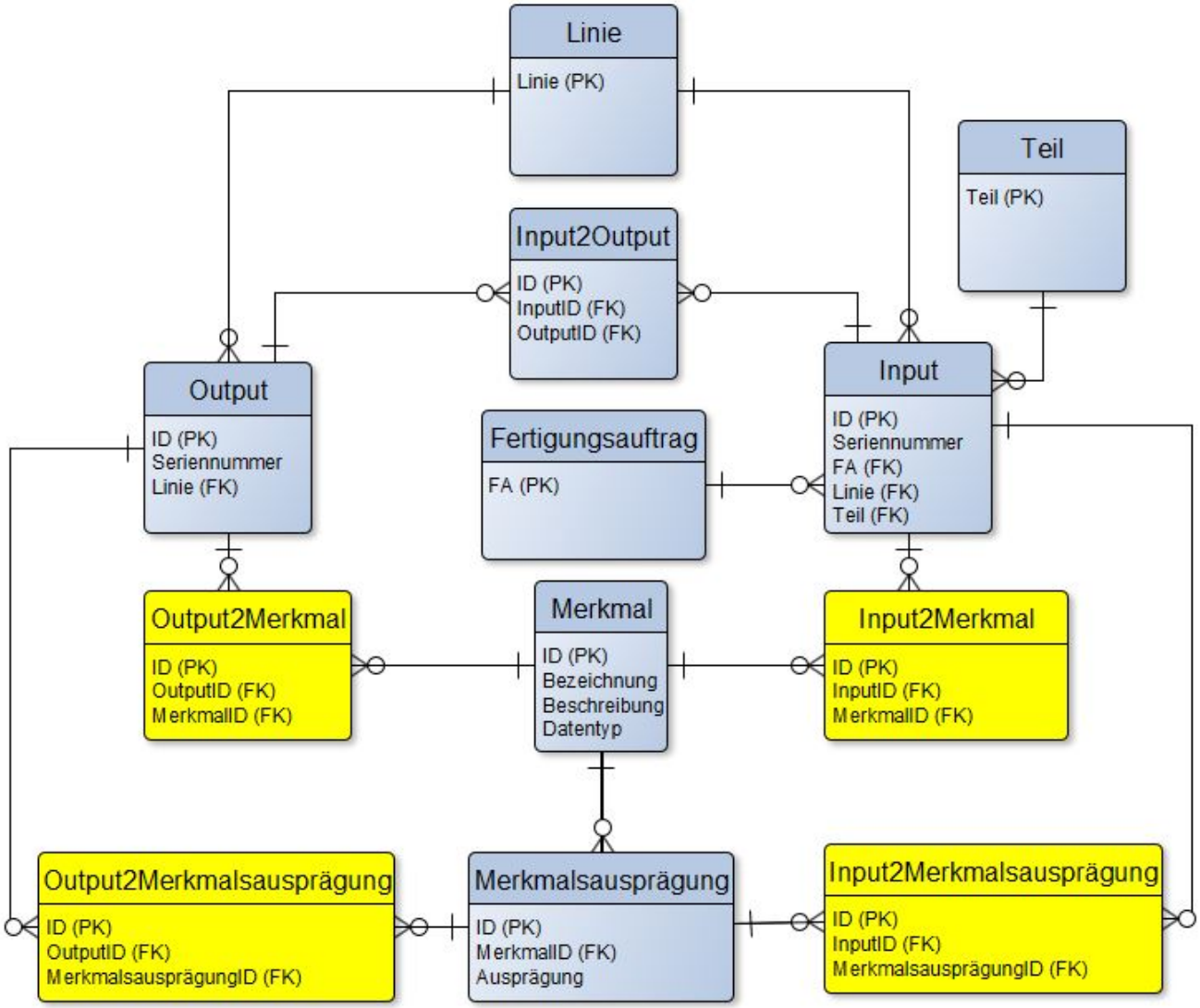

Abbildung 21. Relationenmodell

Die Abbildung 2 zeigt, dass abgeleitete Relationenmodell. Die Beziehungen, welche im konzeptionellen Entwurf noch bestanden, wurden gemäß den Ableitungsregeln aufgelöst. Wichtig für die universelle relationale Struktur sind insbesondere die gelb markierten Verbindungsrelationen, die auf Grund einer mc:mc Verbindung im konzeptionellen Entwurf entstanden. Diese

Verbindungsrelationen definieren die Merkmale eines Tupels und dessen Ausprägungen und bestehen aus einer eindeutigen ID und zwei Fremdschlüsseln. Da jedoch mit jeder Relation, die spezielle Merkmale hat, zwei neue Verbindungsrelationen entstehen, steigt die Komplexität schnell an. Um die Komplexität zu kontrollieren, kann die Ähnlichkeit der Verbindungsrelationen zur Relation Merkmal und die Ähnlichkeit der Verbindungsrelationen zur Relation Merkmalsausprägung genutzt werden, um diese zu vereinen.
In Abbildung 2 wären das Output2Merkmal und Input2Merkmal bezüglich der Relation Merkmal und Output2Merkmalsausprägung und Input2Merkmalsauprägung bezüglich der Relation Merkmalsausprägung, welche zusammengefasst werden können. Beim Zusammenfassen entsteht jedoch das Problem, dass die neu entstandene Relation sich in einem Attribut auf zwei Relationen, in unserem Fall beispielhaft auf Input und Output, bezieht, wodurch ein Tupel beiden Relationen zugeordnet werden kann.
Um dieses Problem zu lösen und dem Ansatz der geringeren Komplexität zu folgen, ist es notwendig die referentielle Integrität aufzulösen.
Daraufhin ergibt sich das Problem, dass ein Tupel der Verbindungsrelation nun nicht mehr eindeutig einer Relation, auf die es sich bezieht, zugeordnet werden kann. Zur eindeutigen Zuordnung wird deshalb ein Diskriminator verwendet.
Ein Diskriminator ist ein Attribut einer Relation, das festlegt, welcher Relation ein Tupel der Verbindungsrelation angehört. Dieses Attribut ist ein Fremdschlüssel, welcher sich in der Diskriminatortabelle definiert. Diese Tabelle bzw. Relation enthält alle definierten Objekttypen der Domäne.
Das angepasste Relationenmodell für die universelle relationale Relationenmodell ist in Abbildung 3 dargestellt.

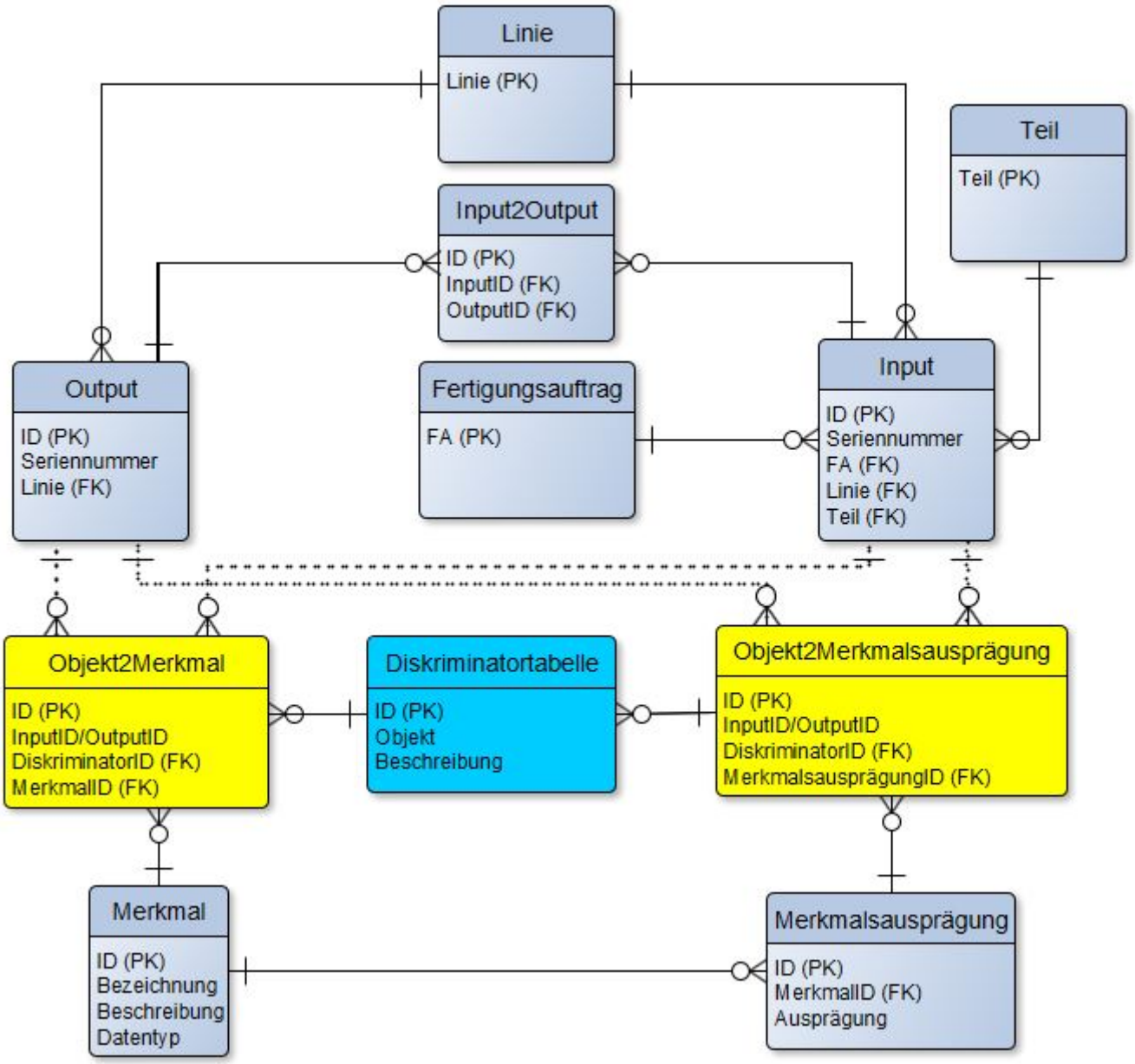

Abbildung 22. universelle relationale Struktur im Relationenmodell

Ein wichtiger Punkt im Relationenmodell ist, dass die Relation Merkmal das Attribut Datentyp besitzt. Grund für dieses Attribut ist, dass in der Relation Merkmalsausprägung alle Ausprägungen als *varchar* gespeichert werden, da diese alle in einem Attribut stehen. Um später den Datentyp einer Ausprägung einfach abfragen zu können, wird er mit dem Merkmal gespeichert.

5.2.3. Implementierung

Systemvoraussetzungen

Alle Implementierungen wurden unter den folgenden Voraussetzungen vorgenommen:

- Windows 10 Pro 64-bit
- Intel Core i5-8265U
- 16 GB RAM

Datenbank

Allgemein

Die universelle relationale Struktur kann einfach über ein relationales DBMS (RDBMS) realisiert werden.
Es wurden der MySQL Server 8.0, als Open-Source Variante, und der kostenlose SQL Server Express 2017 (MSSQL-Server) von Microsoft genutzt.
Beide Systeme können einfach durch die im Internet bereitgestellten Installer der Hersteller installiert und mit Hilfe eines Assistenten konfiguriert und danach genutzt werden.
Für das Anlegen der Datenstruktur ist es wichtig zu wissen, dass die Hersteller unterschiedliche Dialekte der Structured Query Language (SQL) nutzen.
Die folgenden beispielhaften Quelltexte beziehen sich, sofern nicht anders angegeben, auf den MySQL-Dialekt.

Listing 23. Code 3 - Anlegen der Tabelle Input mittels Data Definition Language

```
CREATE TABLE Input (
  ID    int(10) NOT NULL AUTO_INCREMENT,
  SNR   char(18) NULL,
  FA    char(20) NULL,
  TEIL  char(1) NULL,
  LINIE char(1) NULL,
  PRIMARY KEY (ID));
```

Listing 24. Code 4 - Anlegen eines Fremdschlüssels zwischen den Tabellen Input und Teil

```
ALTER TABLE Input ADD CONSTRAINT FKInput_TEIL FOREIGN KEY (TEIL) REFERENCES TEIL (
TEIL);
```

Da bei den Analysen, deren Implementierung später noch beschrieben wird, große Datenmengen der Tabellen abgefragt und teilweise über JOINS miteinander verbunden werden, spielen Indizes eine wichtige Rolle, um die Datenabfragen performant zu gestalten. Durch Indizes wird, vereinfacht beschrieben, das vollständige Durchsuchen einer Menge von Tupeln (FullTableScan) vermieden, da mittels Index eine neue Datenstruktur (häufig B*-Bäume) erstellt wird, welche auf Tabellendaten verweist, was die Laufzeit verringert.

Listing 25. Code 5 - Anlegen von Indizes für die Tabelle Input

```
CREATE INDEX INDEX_TBL_Input ON Input (SNR);
CREATE INDEX INDEX_TBL_Input2 ON Input (TEIL, SNR);
CREATE INDEX INDEX_TBL_Input3 ON Input (FA, SNR);
CREATE INDEX INDEX_TBL_Input4 ON Input (LINIE);
```

Die Auswahl der Attribute, welche in einen Index aufgenommen werden, steht in starkem Zusammenhang, mit den abgefragten Spalten in den Analysen. Beispielsweise bezieht sich Analyse 1 auf die Teilart, weshalb *INDEX_TBL_Input2* angelegt wurde.
An dieser Stelle ist es wichtig zu erwähnen, dass Microsoft mit seinem SQL-Dialekt dem Nutzer mehr Möglichkeiten gibt einen Index zielgenauer zu definieren.

Listing 26. Code 6 - Anlegen von Indizes mittels Microsoft SQL-Dialekt

```
CREATE INDEX INDEX_TBL_Input2 ON Input (TEIL) INCLUDE (FA);
CREATE INDEX INDEX_TBL_MA ON Merkmalsauspraegung (MerkmalID) INCLUDE (Auspraegung)
WHERE MerkmalID = 21;
```

Über die INCLUDE-Klausel ist es möglich Werte zu speichern und schnell abzufragen, welche nicht im Indexschlüssel stehen.
Über die WHERE- Klausel ist es möglich einen Index nur für gefilterte Werte zu erstellen.

Die meisten Tupel der Struktur werden dynamisch über die Datenloader erstellt, sofern ein neues Tupel eines Objekts angelegt wird. Doch bevor dieses Laden geschehen kann, müssen die Merkmale der speziellen Daten in der Tabelle Merkmal und die Objekttypen definiert sein. Dabei müssen der Tabelle Merkmal Werte für Bezeichnung, Beschreibung und Datentyp übergeben werden (Code 7). Das Attribut ID muss im MySQL-Dialekt nicht mittels *NULL* übergeben werden.

Listing 27. Code 7 - Anlegen der Merkmale mittels DML

```
INSERT INTO Merkmal VALUES ('DateIn', 'Zeitstempel der Prüfdaten', 'timestamp');
INSERT INTO Merkmal VALUES ('NR', 'Eingangszähler', 'int');
INSERT INTO Merkmal VALUES ('E', 'GreiferID', 'string');
```

Listing 28. Code 8 - Anlegen der Objettypen mittels DML

```
INSERT INTO ObjektTyp VALUES ('Input', 'Input Datensätze');
INSERT INTO ObjektTyp VALUES ('Output', 'Output Datensätze');
```

Abfrage von Daten

Im folgenden Absatz und Codebeispiel 9 ist das einfache Abfragen von Daten für ein Merkmal dargestellt.

Listing 29. Code 9 - einfache Datenabfrage für ein Merkmal

```
SELECT MA.Auspraegung FROM Input
  -- JOIN 1
  JOIN Objekt2Merkmalsauspraegung AS O2MA ON (O2MA.ObjektID = Input.ID AND O2MA
.DiskriminatorID = 1)
  -- JOIN 2
  JOIN Merkmalsauspraegung AS MA ON MA.ID = O2MA.MerkmalsauspraegungID
  -- JOIN 3
  JOIN Objekt2Merkmal AS O2M ON (O2M.ObjektID = Input.ID AND O2M.DiskriminatorID = 1)
  -- JOIN 4
  JOIN Merkmal AS M ON (M.ID = O2M.MerkmalID AND M.ID = MA.MerkmalID)

  WHERE M.Bezeichnung = "DateIn" AND Input.SNR = "3790034478914";
```

Ziel dieser Abfrage ist es, alle Inputzeiten für ein bestimmtes Werkstück zu erhalten.
Über die WHERE-Klausel wird eingegrenzt, dass das Abfrageergebnis das Merkmal "DateIn", also den Input-Zeitstempel, für alle Datensätze, die die Seriennummer 3790034478914 haben. darstellt.

Um das korrekte Ergebnis zu erhalten, ist es notwendig die Verbindungsrelationen mit den weiteren Relationen über JOINS zu verbinden.

Ausgehend von der Tabelle Input wird mittels dem JOIN 3 die Tabelle Objekt2Merkmal verbunden und im JOIN 4 durch den Ausdruck *M.ID = O2M.MerkmalID* die Tabelle Merkmal herangezogen. Wichtig im dritten JOIN ist der Ausdruck *O2M.DiskriminatorID = 1*, denn dadurch bezieht sich die erste Ergebnismenge nur auf den Objekttypen 1, in diesem Fall Input-Objekten und alle Tupel mit anderen Objekttypen werden nicht beachtet.
Die erste Ergebnismenge (Menge 1) würde nun die Tupel des Objekttyps Input-Datensatz mit der Seriennummer 3790034478914 enthalten, für die das Merkmal "DateIn" definiert ist.

Durch den zweiten Ausdruck im JOIN 4 (*M.ID = MA.MerkmalID*) sind die Tabellen Merkmal und Merkmalsausprägung verbunden und es werden nur die Merkmalsausprägungen angezeigt, die für das angegebene Merkmal ("DateIn") definiert sind. Durch die Verbindung von Merkmalsausprägung zur Verbindungsrelation Objekt2Merkmalsausprägung im JOIN 2, enthält die Ergebnismenge (Menge 2) nur die Tupel, die eine dieser Ausprägungen enthalten.
Durch den JOIN 1 werden nun die Ergebnismenge 1 und 2 verbunden, wodurch die Menge ensteht, die das korrekte Abfrageergebnis darstellt. Auch hier ist die Angabe des Objekttypen in der JOIN-Bedingung des JOINS 1 elementar, um nur Tupel für den richtigen Objekttypen zu erhalten.

Das Vorgehen mit den Mengen 1 und 2 entspricht nicht dem Vorgehen der Datenbank, sondern soll nur der Veranschaulichung dienen.

Messung der Datenbankausführungszeiten der Analysen

MySQL

Der MySQL-Server stellt standardmäßig die Status der 100 zuletzt ausgeführten Queries in der Systemtabelle *INFORMATION_SCHEMA.PROFILING* mit bestimmten Merkmalen bereit, sofern das Profiling aktiviert wurde (siehe Code 10).

Listing 30. Code 10 - Aktivieren des Profilings im MySQL-Server

```
SET @@profiling = 1;
```

Normalerweise verfügt diese Variante über die Möglichkeit, die Größe der Historie (auch *profiling_history_size*) zu bestimmen (siehe Code 11). Jedoch funktionierte dies im Projekt unzuverlässig, weshalb immer der Standardwert von 100 genutzt wurde, um die Zeiten zuverlässig zu messen und die ausgeführten Queries zu zählen (siehe Code 12).

Listing 31. Code 11 - Setzen der Query-Historie auf 500

```
SET @@profiling_history_size = 500;
```

Listing 32. Code 12 - Messen der Ausführungszeiten und Zählen der ausgeführten Queries

```
SELECT SUM(DURATION) FROM INFORMATION_SCHEMA.PROFILING;
SELECT COUNT(Query_ID) FROM INFORMATION_SCHEMA.PROFILING WHERE STATE = 'end';
```

Das Zurücksetzen der Historie kann einfach über die folgende Befehlsfolge im Codeabschnitt 13 erfolgen.

Listing 33. Code 13 - Initialisieren der Systemtabelle

```
SET @@profiling = 0;
SET @@profiling_history_size = 0;
SET @@profiling_history_size = 100;
SET @@profiling = 1;
```

MSSQL

Für die Nutzung des SQL Server Express 2017 wurde das Microsoft SQL Server Management Studio 17 genutzt. Diese Software ermöglicht eine einfache Administration des Datenbankservers.
Über den integrierten *XEventProfiler* können, ab Aufruf des Profilers, alle Events und Queries des Datenbankservers bzw. einer Datenbank, welche in diesem Zeitraum stattfinden, getrackt werden.
Da Systemevents während der Ausführung auftreten, muss nach dem Stoppen des Datenfeed die Ergebnismenge nach dem *client_app_name* gruppiert werden, um nur die gewünschten Ereignisse auszuwerten. Nach der Gruppierung ist noch eine Aggregation zur Summe des Feldes *duration* möglich, um die Ausführungszeit direkt abzulesen.

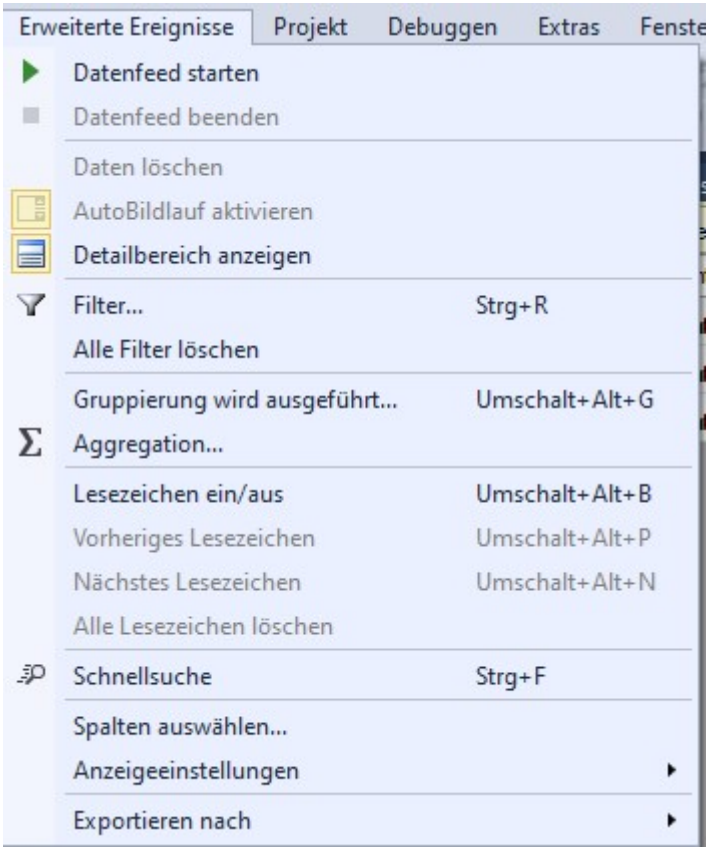

Abbildung 23. Menü zur Gruppierung und Aggregation der getrackten Queries im Microsoft SQL Server Management Studio 17

Lessons Learned

Nachdem in beiden Systemen dieselbe Struktur mit gleichen Indizes (auf Basis MySQL) erstellt wurde und erste Analysen gefahren wurden, zeigte sich, dass drei der fünf Analysen auf dem MSSQL-Server langsamer liefen. Nach der Fehlersuche stellte sich heraus, dass der Buffer des MSSQL-Servers, mit 1.4 GB, sehr schnell aufgebraucht ist. Da es sich um eine kostenlose Variante von Microsoft handelt, besteht keine Möglichkeit diesen Buffer zu erhöhen.
Da beim MySQL-Server der Buffer auch noch nicht betrachtet wurde, wurde hier die Größe überprüft

(800 MB voreingestellt) und auf 6 GB erhöht. Die Erhöhung ist möglich, indem in der Datei */ProgramData/MySQL/MySQLServer8.0/my.ini* die Variable *innodb_buffer_pool_size* auf *6G* gesetzt wurde. Wichtig ist dabei, dass die Datei mit Rechten des Administrators geändert werden muss.
Durch diese Veränderung ließ sich eine starke Senkung in den Ausführungszeiten der Analysen erreichen (siehe Tabelle 1 und Bild).

Tabelle 1. Ausführungszeiten MySQL-DB in Abhängigkeit von der Puffergröße

Analyse	Ausführungszeit 800MB Puffer	Ausführungszeit 6GB Puffer	Senkung
001	19 min : 19 sek	06 min : 12 sek	67.9 %
002	00 min : 56 sek	00 min : 07 sek	86.6 %
004	23 min : 20 sek	06 min : 24 sek	72.6 %
005	31 min : 49 sek	07 min : 28 sek	76.5 %
007	07 min : 31 sek	00 min : 43 sek	90.4 %

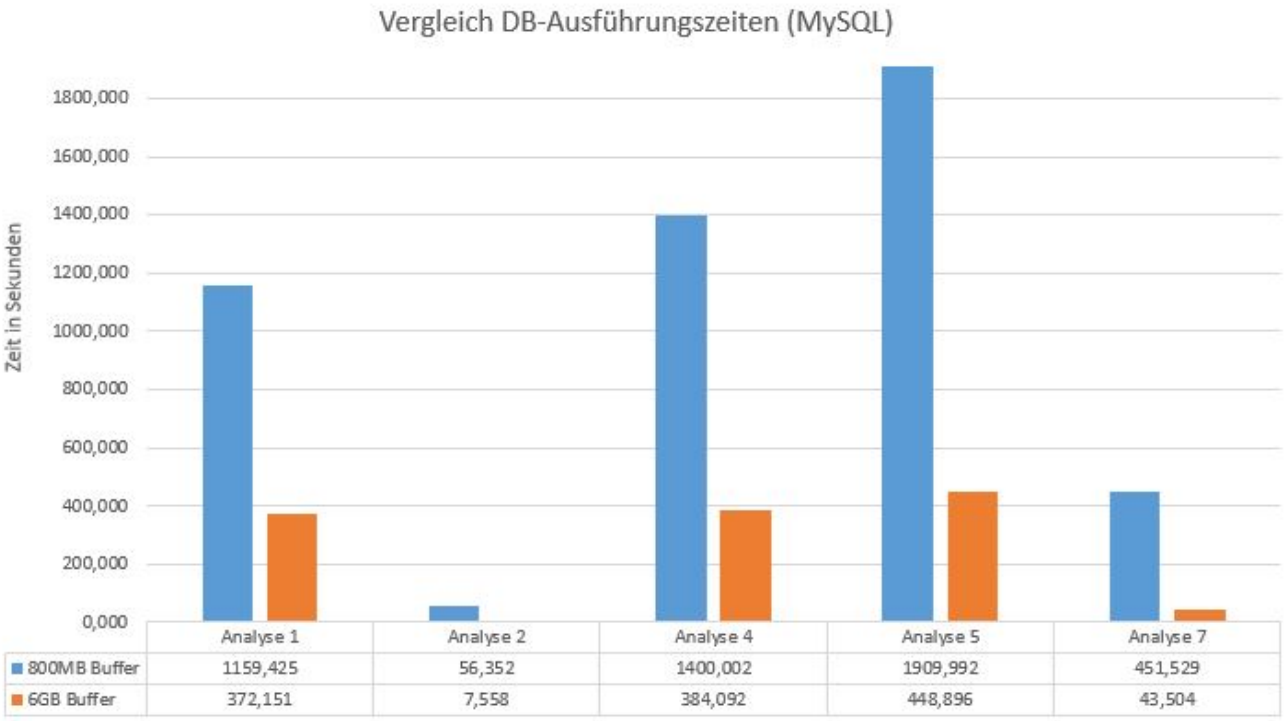

Abbildung 24. Ausführungszeiten MySQL-DB in Abhängigkeit von der Puffergröße

Anwendungen

Programmiersprache

Zur Implementierung der Anwendungen wurde die Programmiersprache Python verwendet.
Im Projekt wurde Visual Studio Code als Entwicklungsumgebung (IDE) genutzt, welche es ermöglicht, einfach die Python-Extension herunterzuladen und zu nutzen.
Für die Implementierung wurde die Python-Version 3.7.3 genutzt.

Verwendete Bibliotheken zur Kommunikation zwischen Anwendung und Datenbank

MySQL

Zur Verbindung zwischen Anwendung und MySQL-Datenbankserver wurde die Python-Bibliothek *mysql-connector-python* genutzt. Diese kann in Visual Studio Code über die Konsole durch den im Beispiel 14 dargestellten Code installiert werden.

Listing 34. Code 14 - Installieren der MySQL-Bibliothek für Python

```
pip install mysql-connector-python
```

Damit die Anwendung eine Verbindung zur Datenbank herstellt, muss die Bibliothek eingebunden und die Parameter *user*, *password*, *host* und *database* übergeben werden. Um Operationen ausführen zu können, muss ein Cursor genutzt werden. (siehe Code 15)

Listing 35. Code 15 - Herstellen der Verbindung und Erstellen eines Cursors

```
import mysql.connector

connection = mysql.connector.connect(user = "root", password = "demo", host =
"127.0.0.1",  database = "project")
cursor = connection.cursor()
```

Für SELECT-Abfragen muss nun lediglich ein Statement der Cursor-Funktion *execute* übergeben werden, damit die Abfrage ausgeführt wird. Der Cursor bietet drei Methoden, um zu definieren, welche Menge der Ergebnismenge bereitgestellt wird:

- *fetchall()* für die komplette Ergebnismenge
- *fetchone()* für die erste Zeile der Ergebnismenge
- *fetchmany(size = x)* für die ersten x Zeilen der Ergebnismenge

Listing 36. Code 16 - Ausführen einer Abfrage und Fetch der kompletten Ergebnismenge

```
statement = "SELECT Input.FA FROM Input WHERE TEIL = 'A' GROUP BY Input.FA ORDER BY
Input.FA;"
  cursor.execute(statement)
  FA_List = cursor.fetchall()
```

Sofern ein Insert, Update oder Delete durchgeführt wurde, muss nach der Ausführung mittels *execute()* ein Commit erfolgen, um die Änderungen zu übernehmen. (siehe Code 17)

Listing 37. Code 17 - Verbindungscommit nach Insert-Anweisung

```
statement = "INSERT INTO LINIE VALUES (1);"
  cursor.execute(statement)
  connection.commit()
```

Am Ende der Anwendung können der Cursor und die Verbindung einfach über die Funktion *close()* geschlossen werden. (siehe Code 18)

Listing 38. Code 18 - Schließen des Cursors und Abbau der Verbindung

```
cursor.close()
connection.close()
```

Genauere Ausführungen und weitere Informationen sind in der MySQL-Dokumentation verfügbar.

MSSQL

Zur Verbindung zwischen Anwendung und MSSQL-Datenbankserver wurde die Python-Bibliothek *pyodbc* genutzt. Diese kann in Visual Studio Code über die Konsole durch den im Beispiel 19 dargestellten Code installiert werden. Außerdem muss der "Microsoft ODBC Driver for SQL Server", welcher in der Microsoft Dokumentation zu finden ist (ODBC Driver), installiert werden.

Listing 39. Code 19 - Installieren der pyodbc-Bibliothek für Python

```
pip install pyodbc
```

Im Unterschied zu MySQL muss zum Verbindungsaufbau noch der weitere Parameter *DRIVER* übergeben werden. Um Operationen ausführen zu können, muss auch hier ein Cursor genutzt werden. (siehe Code 20)

Listing 40. Code 20 - Herstellen der Verbindung und Erstellen eines Cursors

```
import pyodbc

connection = pyodbc.connect(driver = '{ODBC Driver 17 for SQL Server}', server =
'Desktop\\SQLEXPRESS' , database = 'project', UID = 'root', PWD = 'demo')
cursor = connection.cursor()
```

Alle weiteren im MySQL-Teil ausgeführten Befehle gelten unter pyodbc ebenfalls in der gleichen Form.

Messung der Skriptausführungszeiten

Zur Messung der Skriptausführungszeiten wurde von der Python-Bibliothek *time* die Methode *process_time_ns()* geladen, mit der die Summe der System- und Benutzer-CPU-Zeit des aktuellen Prozesses in Nanosekunden berechnet werden kann. Diese Methode schließt die während des Ruhezustands verstrichene Zeit nicht ein.

Listing 41. Code 21 - Messen der Skriptausführungszeit

```
from time import process_time_ns()

start = process_time_ns()
# Code der auszuführen ist
stop = process_time_ns()

duration = stop - start
```

Datenloader

Die Datenloader, über die Datensätze in die Struktur geladen werden, unterscheiden sich auf Grund der unterschiedlichen SQL-Dialekte. Jedoch ist das allgemeine Vorgehen, welches hier erläutert wird, gleich. Ein kleiner Unterschied liegt nur in der Verknüpfung des Outputs mit dem Input, was später erläutert wird.
Voraussetzung, bevor Datensätze eingelesen werden können, ist wie bereits erwähnt, dass Merkmale und Objekttypen bereits in der Struktur definiert wurden.

Aus dem bereits erläuterten Watchdog, erhält die Anwendung den Pfad des Textdokuments, welches ausgelesen werden muss. Im Codebeispiel 22 ist dargestellt, wie eine Datei mit Leserechten geöffnet wird, der Inhalt mittels *read()* ausgelesen und als String gespeichert wird und dieser String aufgearbeitet wird, dass alle Elemente, die durch ein Semikolon getrennt sind, ein Element in einer Liste werden.

Listing 42. Code 22 - Auslesen der vorhandenen Datei

```
def insert (file):
  datei = open(file,'r')
  values = datei.read()
  data = values.split(';')
```

Die Verfahren zum Einlesen der Input- und Output-Datensätze sind sehr ähnlich. Deshalb wurden die Verfahren zusammen in den Bildern 6 und 7 dargestellt.

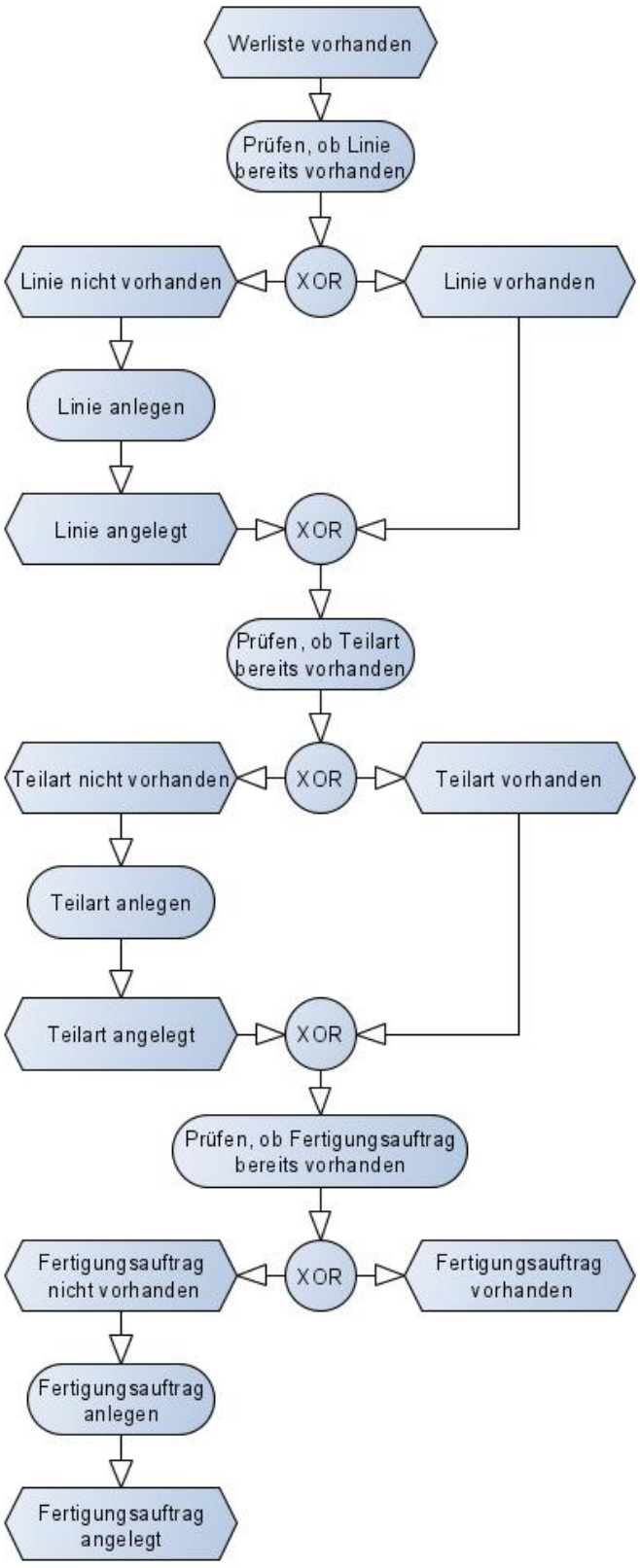

Abbildung 25. ereignisgesteuerte Prozesskette zur Darstellung des Einlesens von Werten anderer Objekttypen

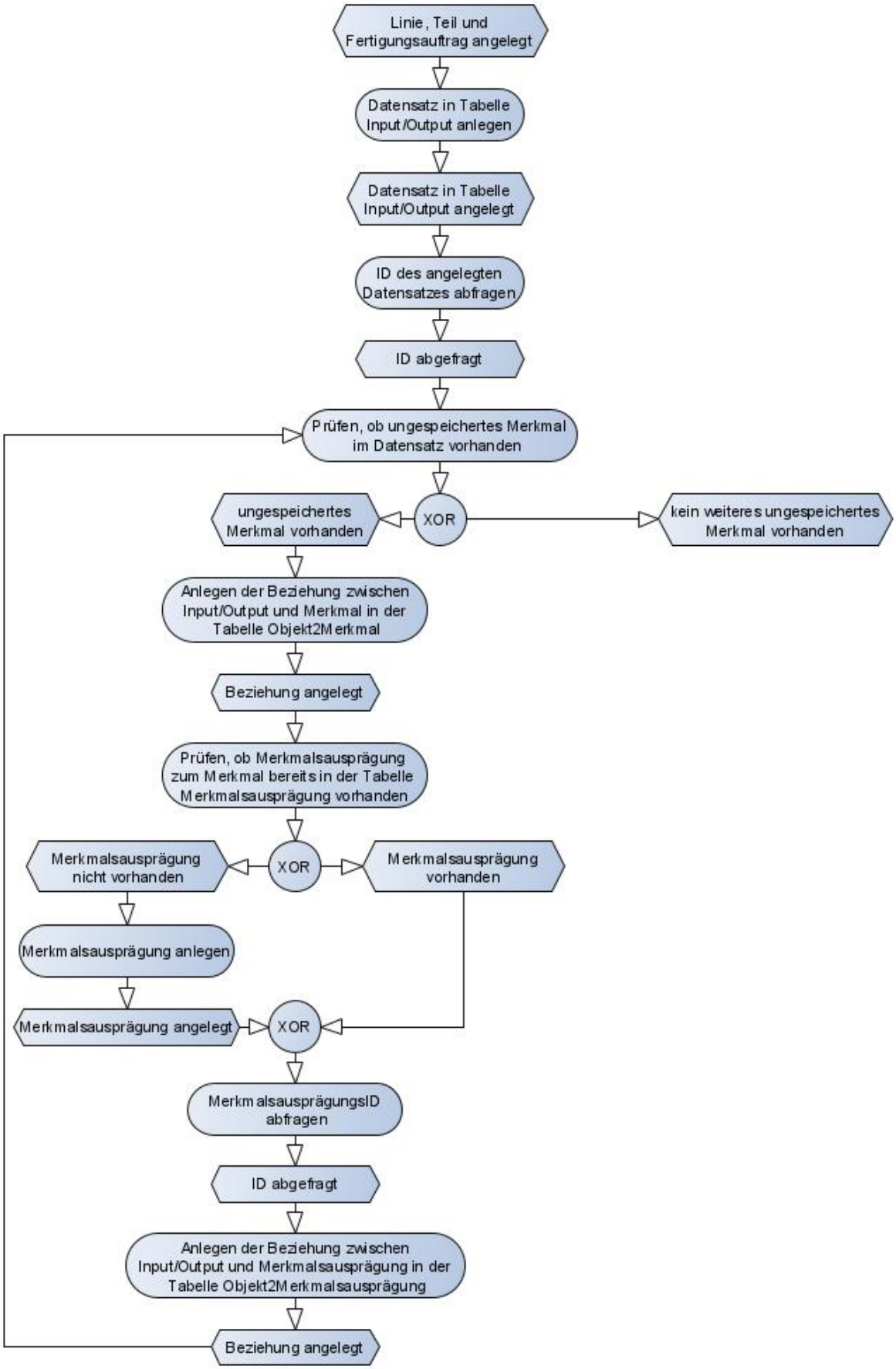

Abbildung 26. ereignisgesteuerte Prozesskette zur Darstellung des Einlesens spezieller Merkmale sind

Nachdem ein Output-Datensatz in der Tabelle Output angelegt worden ist (Bild 7, Ereignis 2), muss dieser noch, sofern möglich, mit einem Input-Datensatz verknüpft werden. Dies passiert über die Zeitstempel der Datensätze. Sofern es für die Seriennummer nur einen Input-Datensatz gibt, erfolgt

eine direkte Verknüpfung, außer die Zeitdifferenz zwischen Output und Input ist negativ. Sollten jedoch mehrere Input-Datensätze zu einer Seriennummer vorhanden sein, muss die Zeitdifferenz zwischen Output und jedem passenden Input berechnet werden. Dabei wird der Output mit dem Input verknüpft zu dem die kleinste nicht negative Differenz besteht.

Analysen

Allgemein

Bevor mit der Implementierung der vorgegebenen Analysen begonnen wurde, wurde über Möglichkeiten der Realisierung in Python nachgedacht. Grundsätzlich lassen sich drei Varianten realisieren, welche mit ihren Vor- und Nachteilen, die sich auch auf LessonsLearned des Projekts zurückführen lassen, in der folgenden Tabelle dargestellt sind.

Tabelle 2. Realisierungsmöglichkeiten der Analysen

	(1) kleine Abfragen mit genauen WHERE-Klauseln (bspw. je SNR)	**(2) mittlere Abfragen mit Mengen in WHERE-Klauseln (bspw. je FA)**	**(3) große Abfragen ohne Selektion in SQL**
Vorteile	• gesamtes Vorgehen einfach nachvollziehbar • geringer Aufwand in Programmiersprache • verständlichere SQL-Abfragen	• geringere Netzwerklast als bei kleinen häufigen Abfragen • Verteilung der Komplexität in Abfragen und Programmiersprache	• einmalige Netzwerklast
Nachteile	• Netzwerklast dauerhaft • in Summe höhere Abfragezeiten auf der Datenbank	• dauerhafte Netzwerklast größer als bei großen Abfragen	• Vorgehen schwerer nachvollziehbar • höherer Aufwand in Programmiersprache • Gruppierungen, die bereits einfach mit SQL gelöst werden können, müssen in der Programmiersprache erledigt werden

Zum Test wurde versucht über jede Variante eine vordefinierte Datenmenge aus der Datenbank abzufragen. Da sich die Zeitergebnisse für diese Datenmenge nur gering unterschieden, wurde entschieden, um die unbekannten Analysen vorerst in kleinen logischen Schritten zu lösen, dass die Variante 1 umgesetzt wird.
Nach Fertigstellung der Variante 1 für jede Analyse wurde zum Vergleich Variante 2 für die Analysen 1, 4 und 5 umgesetzt, da dort relativ lange Zeiten auftraten.

In den folgenden Absätzen werden kurz selbstdefinierte Funktionen gezeigt und das Vorgehen in den Analysen für die verschiedenen Varianten als Pseudocode, zur einfachen Verständlichkeit erläutert.

Für die Realisierung der Variante 2 wurde die Python-Bibliothek *pandas* genutzt, welche einfache und flexible Möglichkeiten der Datenanalyse und -manipulation bietet.

Listing 43. Code 23 - Installieren der pandas-Bibliothek für Python

```
pip install pandas
```

Eigene Funktionen

Zur Umsetzung der Implementierungen wurden zwei selbstdefinierte Funktionen genutzt.
Zum einen eine Funktion, um Datumswerte, welche in der Struktur als *VARCHAR* gespeichert sind, in Sekunden für die Zeitdifferenzberechnung umzuwandeln. (siehe Code 24)

Listing 44. Code 24 - Umwandeln eines Datumsstrings in Sekunden

```
import datetime, time

def convert_from_datestring( TimeString ):
  Date = datetime.datetime.strptime(TimeString, "%Y-%m-%dT%H:%M:%S.%f")
  Second = time.mktime(Date.timetuple())
  return Second
```

Zum anderen wurde eine Funktion zur Umwandlung der Zeitdifferenzen in Sekunden verwendet, um diesen Wert in einen einfach menschlichen lesbaren String bestehend aus Tagen, Stunden, Minuten und Sekunden umzurechnen. (siehe Code 25)

Listing 45. Code 25 - Umwandeln eines Sekundenwerts in einen einfach lesbaren String

```
def convert_from_s( seconds ):
  minutes, seconds = divmod(seconds, 60)
  hours, minutes = divmod(minutes, 60)
  days, hours = divmod(hours, 24)
  string = str(int(days))+"T:"+str(int(hours))+"h:"+str(int(minutes))+"m:"+str(int
(seconds))+ "s"
  return string
```

Analyse 1 - Taktung pro Artikel

Variante 1 - kleine Abfragen

Listing 46. Code 26 - Pseudocode Analyse 1.1

```
Abfrage aller Teilarten;

FOR EACH Teilart der Teilarten {
  Anzahl gefertigter Teile pro Teiltyp ermitteln;
  Fertigungsaufträge des Teiltyps abfragen;

  FOR EACH Fertigungsauftrag in Fertigungsaufträgen {
    Anzahl gefertigter Teile pro Fertigungsauftrag ermitteln;
    Alle Seriennummern abfragen, die mehr als einen Input in diesem Fertigungsauftrag
haben (Auschuss);

    FOR EACH Seriennummer im Ausschuss {
      Anzahl Inputs für Seriennummer finden;
    }

    Minimum, Maximum, Durchschnitt des Ausschusses bestimmen;
    InputID's abfragen, die einen Output haben, zum Fertigungsauftrag gehören und eine
Seriennummer haben;

    FOR EACH InputID in InputID's {
      Input-Zeit abfragen und konvertieren;
      Alle OutputID's für Input ID abfragen;

      FOR EACH OutputID in OutputID's {
        Output-Zeit abfragen, konvertieren und Differenz zu Input-Zeit berechnen;
      }

      Maximum der Differenzen bestimmen;
    }

    Minimum, Maximum, Durchschnitt aller Differenzen pro Fertigungsauftrag bestimmen;
    Ausgabe pro Fertigungsauftrag;
  }
}
```

Variante 2 - mittlere Abfragen

Listing 47. Code 27 - Pseudocode Analyse 1.2

```
Abfrage aller Teilarten;

FOR EACH Teilart der Teilarten {
  Anzahl gefertigter Teile pro Teiltyp ermitteln;
  Fertigungsaufträge des Teiltyps abfragen;

  FOR EACH Fertigungsauftrag in Fertigungsaufträgen {
    Anzahl gefertigter Teile pro Fertigungsauftrag ermitteln;
    Alle InputID's mit ihren Input-Zeitstempeln abfragen;
    Für alle InputID's den maximalen Output-Zeitstempel (über erstellte Verknüpfung)
ermitteln;

    FOR EACH InputID in InputID's {
      Suche des passenden Outputs in Outputs;
      Zeitstempel konvertieren und Differenz berechnen;
    }

    Minimum, Maximum, Durchschnitt aller Differenzen pro Fertigungsauftrag bestimmen;
    Alle Seriennummern abfragen, die mehr als einen Input in diesem Fertigungsauftrag
haben (Auschuss);
    Anzahl des Ausschusses pro Seriennummer für alle Seriennummern abfragen;
    Minimum, Maximum, Durchschnitt des Ausschusses bestimmen;
    Ausgabe pro Fertigungsauftrag;
  }
}
```

Analyse 2 - Auftrennung

Listing 48. Code 28 - Pseudocode Analyse 2

```
Abfrage aller Teilarten;

FOR EACH Teilart der Teilarten {
  Anzahl gefertigter Teile pro Teiltyp ermitteln;
  Fertigungsaufträge des Teiltyps abfragen;

  FOR EACH Fertigungsauftrag in Fertigungsaufträgen {
    Anzahl gefertigter Teile pro Fertigungsauftrag ermitteln;
    Alle Seriennummern abfragen, die mehr als einen Input in diesem Fertigungsauftrag
haben (Auschuss);

    FOR EACH Seriennummer im Ausschuss {
      InputID's und die Zeitstempel ermitteln;

      FOR EACH InputID in InputID's {
        Output-Zeitstempel der InputID abfragen;

        IF kein Output-Zeitstempel vorhanden {
          nächste InputID;
        }

        IF aktuelle InputID nicht die Letzte {
          Output-Zeitstempel der InputID konvertieren;
          Input-Zeitstempel der nächsten InputID konvertieren;
          Differenz berechnen;
        }
      }
    }
  }
  Minimum, Maximum, Durchschnitt aller Differenzen pro Teilart bestimmen;
  Ausgabe pro Teilart;
}
```

Analyse 4

Variante 1 - kleine Abfragen

Listing 49. Code 29 - Pseudocode Analyse 4.1

```
Abfrage aller LadungsträgerIn;

FOR EACH Ladungsträger der LadungsträgerIn {
  Anzahl gefertigter Teile pro Ladungsträger ermitteln;
  InputID's des aktuelle Ladungsträgers abfragen;

  FOR EACH InputID in InputID's {
    Input-Zeitstempel abfragen und konvertieren;
    OutputID's zur aktuellen InputID ermittlen;

    FOR EACH OutputID in OutputID's {
      Output-Zeitstempel abfragen und konvertieren;
    }
  }

  Minimum Input-Zeitstempel bestimmen;
  Maximum Output-Zeitstempel bestimmen;
  Differenz berechnen;
  Ausgabe pro Ladungsträger;
}
```

Variante 2 - mittlere Abfragen

Listing 50. Code 30 - Pseudocode Analyse 4.2

```
Abfrage aller LadungsträgerIn;

FOR EACH Ladungsträger der LadungsträgerIn {
  Anzahl gefertigter Teile pro Ladungsträger ermitteln;
  InputID's des aktuelle Ladungsträgers abfragen;
  minimalen Input-Zeitstempel der InputID's abfragen;
  maximalen Output-Zeitstempel der mit den InputID's verknüpften Outputs ermitteln;
  Differenz berechnen;
  Ausgabe pro Ladungsträger;
}
```

Analyse 5

Variante 1 - kleine Abfragen

Listing 51. Code 31 - Pseudocode Analyse 5.1

```
Abfrage aller Teilarten;

FOR EACH Teilart der Teilarten {
  genutzte LadungsträgerIn für den Teiltyp abfragen;

  FOR EACH Ladungsträger in LadungsträgerIn {
    Anzahl gefertigter Stücke pro Ladungsträger des Teiltyps ermitteln;
    InputID's des Teiltypen abrufen, die auf dem aktuellen Ladungsträger gefertigt
wurden;

    FOR EACH InputID in InputID's {
      Input-Zeitstempel abfragen und konvertieren;
      OutputID's zur InputID abfragen;

      FOR EACH OutputID in OutputID's {
        Output-Zeitstempel abfragen und konvertieren;
        Differenz zwischen Output und Input berechnen;
      }
      Maximum der Differenzen bestimmen;
    }
    Minimum, Maximum, Durchschnitt aller Differenzen pro Ladungsträger bestimmen;
    Ausgabe pro Ladungsträger;
  }
}
```

Variante 2 - mittlere Abfragen

Listing 52. Code 32 - Pseudocode Analyse 5.2

```
Abfrage aller Teilarten;

FOR EACH Teilart der Teilarten {
  genutzte LadungsträgerIn für den Teiltyp abfragen;

  FOR EACH Ladungsträger in LadungsträgerIn {
    Anzahl gefertigter Teile pro Ladungsträger des Teiltyps ermitteln;
    InputID's des Teiltypen abrufen, die auf dem aktuellen Ladungsträger gefertigt
wurden;
    alle InputID's mit ihren Input-Zeitstempeln abfragen;
    maximale Output-Zeitstempel der mit den InputID's verknüpften Outputs ermitteln;
    Differenzen berechnen zwischen zusammengehörigen Outputs und Inputs;
    Minimum, Maximum, Durchschnitt ermitteln;
    Ausgabe pro Ladungsträger;
  }
}
```

Analyse 6

Listing 53. Code 33 - Pseudocode Analyse 6

```
Abfrage aller Linien;

FOR EACH Linie in Linien {
  Fertigungsaufträge der Linie abfragen;

  FOR EACH Fertigungsauftrag in Fertigungsaufträge {
    alle Input-Zeitstempel und Teilart des Fertigungsauftrags abfragen;
    minimalen und maximalen Input-Zeitstempel mit Teilart als ein Element in einer
Liste speichern;
  }
  Liste nach minimaler Input-Zeit sortieren;

  FOR EACH Element der Liste {
    maximalen Input-Zeitstempel des aktuellen Elements konvertieren;
    minimalen Input-Zeitstempel des nächsten Elements konvertieren;
    Differenz zwischen maximalen Input-Zeitstempel des aktuellen Elements und
minimalen Input-Zeitstempel des nächsten Elements bilden;
    IF Differenz positiv {
      Wechsel der Teilart mit Differenzzeit notieren;
    }
  }

  Minimum, Maximum, Durchschnitt der Wechselzeiten pro Linie berechnen;
}
```

Auswertung

Nach Messung aller Ausführungszeiten ergab sich eine deutliche Senkung der Datenbankausführungszeiten durch die Umstellung der Analysevorgehen. (siehe Bild 8)

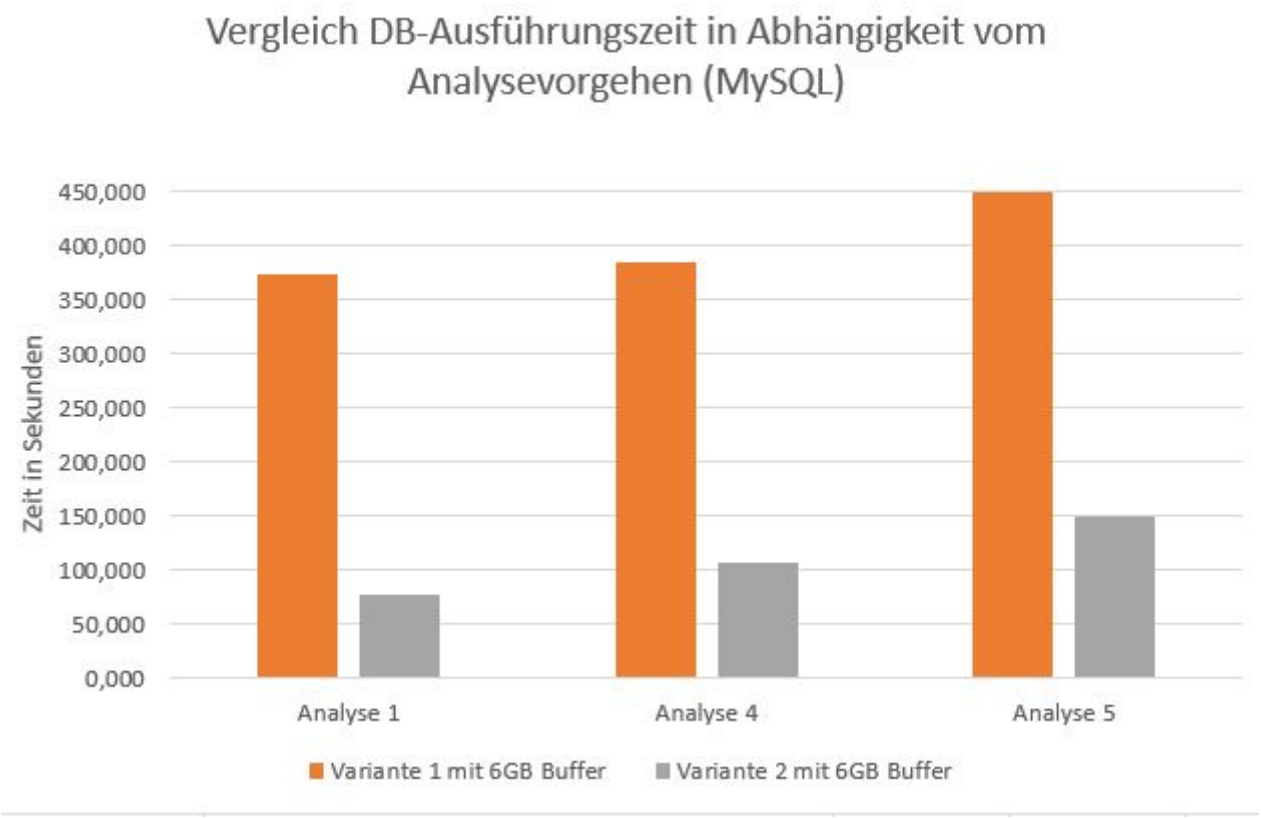

Abbildung 27. Vergleich der Analysevorgehen bezüglich der Datenbankausführungszeit

Jedoch zeigte sich auch in der Ausführungszeit der Skripte ein deutliche Zeitverbesserung. (siehe Bild 9)

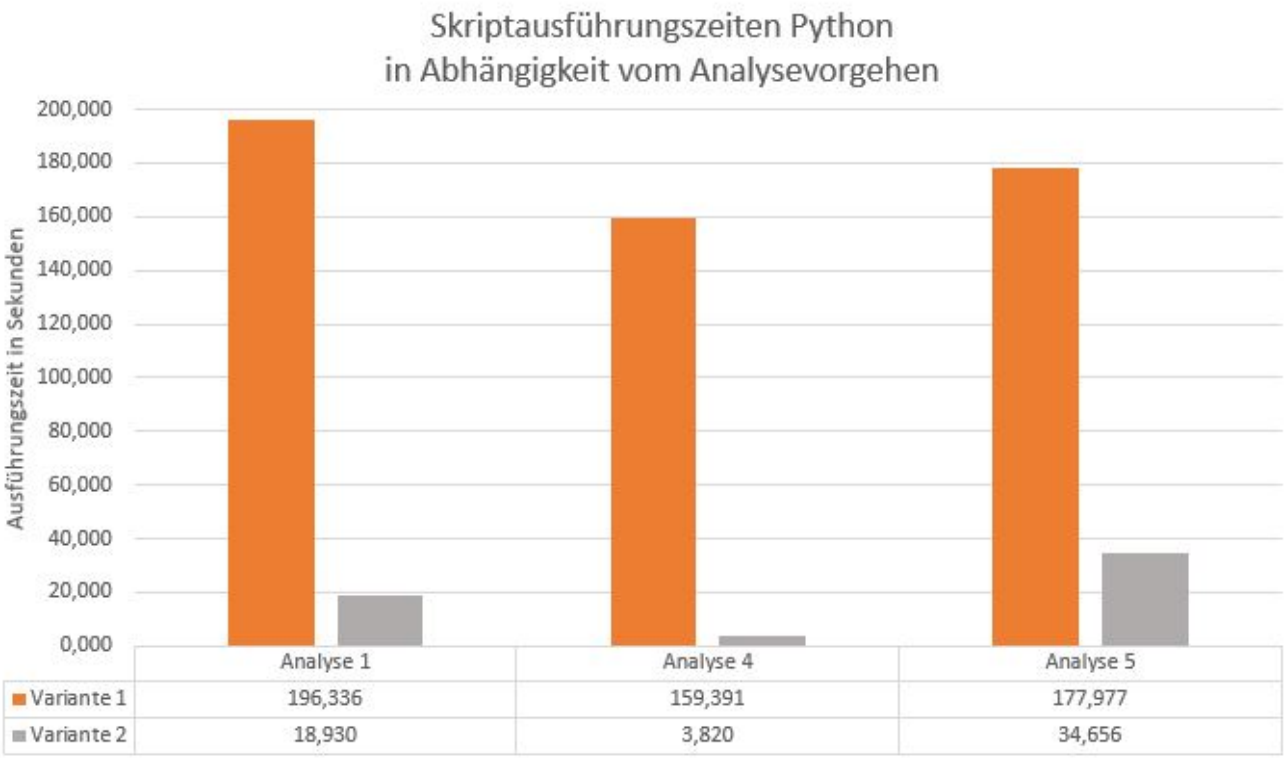

Abbildung 28. Vergleich der Analysevorgehen bezüglich der Skriptausführungszeit

Lessons Learned

Durch eine starke Verschachtelung in FOR-Schleifen der Variante 1 aller Analysen ist es möglich sehr genaue SQL-Statements zu entwerfen und so nur einen kleinen Teil der gebrauchten Daten zu

manipulieren, was den Manipulationsaufwand in Python verringert. Jedoch entstehen dadurch sehr großen Analysen mit sehr vielen Abfragen, welche der Datenbank gestellt werden müssen.

Mit Variante 2 sind weniger Abfragen nötig, jedoch müssen die Daten aufwendiger mittels Python manipuliert werden. Demgegenüber zeigte sich aber, dass dieser Mehraufwand sich deutlich in den Datenbank- und Skriptausführungszeiten widerspiegelt.

Interessant wäre noch ein Vergleich mit Variante 3 gewesen, wofür aber leider die Zeit fehlte.

5.3. Dokumentenorientierte Datenbank

5.3.1. Einführung in Datenbankstruktur

Datenbankvorstellung

Dokumentenorientierte Datenbank

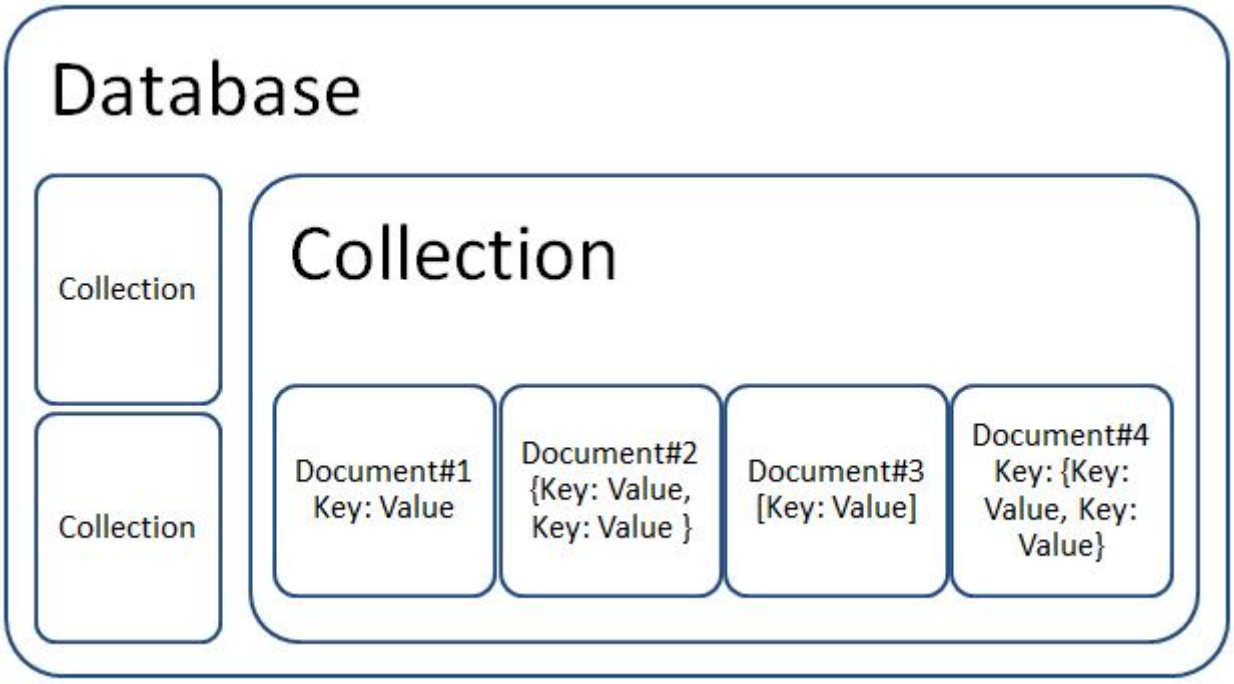

Abbildung 29. Aufbau einer Dokumentenorientierten Datenbank

Dokumentenorientierte Datenbanken zählen zu den No-SQL Datenbanken das bedeutet, dass sie im Gegensatz zu SQL Datenbanken keinen relationalen Ansatz verfolgen. In einer Dokumentenorientierten Datenbank liegen die Daten in einzelnen "Dokumenten", wobei ein Dokument am besten mit einer Tabellenzeile in einer herkömmlichen Datenbank zu vergleichen ist. Jedes Dokument verfügt dabei über einen eindeutigen Identifikator, was eine gewisse ähnlichkeit zu Schlüssel-Wert Datenbanken hervorruft. Mehrere Dokumente werden wiederum in einer "Collection", welche vergleichbar mit einer Tabelle in relationalen Datenbanken ist, zussamengefasst. Im Gegensatz zu Tabellen in herkömmlichen relationalen Datenbanken, in welchen die Daten strukturiert vorliegen, liegen die Daten in "Collections" unstrukturiert. Lediglich die "Dokumente" besitzen innerhalb ihrer Daten eine Struktur. Dies bedeutet ,dass "Dokumente" innerhalb einer "Collection" sich beliebig in Anzahl und Nomenklatur ihrer Attribute unterscheiden können. Es hierbei aber zu beachten, dass wenn man auf eine "Collection" Abfragen fahren möchte, die Notwendigkeit besteht, dass gleichartige "Dokumente" mit gleichen Attributen denselben "Keys" für ihre Attribute wählen.

MongoDB als Datenbankmanagementsystem

Bei MongoDB handelt es sich um eines der größten und verbreitetesten Datenbankmanagementsysteme für Dokumentenorientierte Datenbanken. Ich möchte einmal darauf eingehen warum ich mich für MongoDB und nicht z.B. für CouchDB entschieden habe.

MongoDB erlaubt es "Queries" im sogenannten BSON Format, welches eine Variante von JSOn ist, bei der Daten im Binär-Format gespeichert werden ist, zu speichern. MongoDB über seine eigene "Querie-Language", während bei CouchDB Abfragen über eine RESTful-API laufen. Außerdem hat MongoDB die Nase im Bereich Performance bei großen Datenmengen klar vorn. Ein weiterer wichtiger Faktor war für mich Die Menge an Informationsmaterial (z.B. Dokumentation) im Internet und da MongoDB über eine weitaus größere Userbase als andere Dokumentenorientierte Datenbankmanagementsysteme verfügt wodurch man im Internet leichter Informationen bzw. Hilfe findet.

5.3.2. Entwurf

Daten-Schema

Wie bereits erwähnt verzichten Dokumentenorientierte Datenbanken auf ein ausgiebiges Schema. Dies hat sowohl Vor- als auch Nachteile. Ein großer Vorteil ist das sich die Strucktur der einzulesenden Daten ändern kann, ohne das man änderung an der Datenbank durchführen muss. Es ist so z.B. der Datenbank völlig egal ob sich nun der Datentyp eines Feldes in der Datenbank von dem eines neu zu speichernden Datensatzes unterscheidet. Es kann auch ohne Probleme ein weiteres Attribut in den neuen Daten hinzukommen welches es in den alten Daten noch nicht gibt. Dies ist gleichzeitig aber auch ein Problem denn wo z.B. ein relationales Datenbankmanagementssystem das Speichern gar nicht zulassen würde akzeptiert ein Dokumentenorientiertes dies erst einmal. Es besteht aber die Möglichkeit dies zu verhindern und Datenüberprüfungen festzulegen. Dies ist aber vergleichsweise aufwendiger und schränkt den größten Vorteil der Dokumentenorientierten Datenbank ein. Ich habe die Datentypen und Attributsbezeichnung innerhalb des Programmes festgelegt welche die Daten in die Datenbank schreibt, um so eine gewisse Konsistenz zu gewährleisten, ohne die Komplexität unnötig aufzublähen aber dazu später mehr.

Auch in Dokumentenorientierten Datenbanken besteht die Möglichkeit Relationen zwischen "Dokumenten" herzustellen. Es gibt dabei 3 Varianten, welche man verfolgen kann.

One-To-One Embedded

```
{
    "_id": {
        "$oid": "5ffc7e265cb0d6fc06feceb1"
    },
    "DATE": {},
    "FA": "5689",
    "SNR": "4072932452082",
    "out": [{
        "SNR": "4072932452082",
        "LINIE": "4",
        "TRÄGER": true,
        "MAT": true,
        "NAHT": true,
        "AngleGrad": -999999.999999,
        "LengthMM": 1094.48769498026,
        "LengthdiffMM": 0.572394223372209,
        "AngleDiffGrad": -0.473530139018093,
        "AxisDistMMX": -0.42886893965259704,
        "AxisDistMMY": -0.9622981273473179,
        "AxisDistMM": 1.05353986791905,
        "RTotalNominal": 5,
        "RTotalCurrent": 5,
        "RCount": 5,
        "Date": {}
    }]
}
```


Abbildung 30. Beispiel eines "Dokuments" mit einem "Embedded Document"

Hierbei speichert man zusammengehörige "Dokumente" in einem. Das bedeutet man legt ein weiteres Attribut an welches ein weiteres Dokument enthällt (Siehe Bild). Diese Methode eignet sich vorallem wenn ein Dokument immer nur in Relation zu einem anderen steht also bei 1-1 Beziehungen. Es besteht dadurch der Vorteil das keine Joins durchgeführt werden müssen da alle Daten an einer Stelle liegen.

One-To-Many Embedded


```
{
    "_id": {
        "$oid": "5ffc7e265cb0d6fc06feceb0"
    },
    "DATE": {},
    "FA": "5689",
    "SNR": "4072932452084",
    "out": [{
        "SNR": "4072932452084",
        "LINIE": "4",
        "Date": {}
    }, {
        "SNR": "4072932452084",
        "LINIE": "4",
        "Date": {}
    }]
}
```


Abbildung 31. Beispiel eines Dokuments mit einem Array aus "Embedded Documents"

Es handelt sich bei dieser Methode um eine Erweiterung der vorhergegangenen. Man speichert Dabei nicht nur ein "Dokument" in ein anderes, sondern ein Array aus "Dokumenten". Diese Methode macht vorallem bei 1-n Beziehungen Sinn. Es handelt sich hierbei auch um das für meine Anwendung geeigneste Schema. Da eine Menge von Ausgangsdatensätzen immer eindeutig einem Eingangsdatensatz zugeordnet werden kann. Einzelne Elemente des Arrays lassen sich bei Abfragen sogar über den Index ansteuern.

Many-To-Many Reference

Dieses Schema möchte ich hierbei nur am Rande erwähnen. Es handelt sich hierbei um eine Methode

welche sehr der Foreign Key Methode in relationalen Datenbanken entspricht. Hierbei muss es in beiden "Dokumenten" in verschiedenen "Collections" ein Attribut mit dem selben Schlüssel geben. Bei einer Abfrage kann dann soweit auch die Ausprägung des Attributes übereinstimmt eine Verknüpfung duchgeführt werden. Dies dauert aber länger bei Abfragen als die anderen Methoden da erst ein Join durchgeführt werden muss und eignet sich daher wirklich nur, wenn eine m-n Relation vorliegt.

5.3.3. Implementierung

Entwicklungsumgebung

Verwendete Python-Bibliotheken

Python-Bibliothek	Verwendung / Nutzen
pandas	Datenstrukturen wie Datenframes zur leichteren Manipulierung großer Datenmengen
pymongo	Verbindung des Python Programmes mit der Datenbank zum Speichern und Abfragen von Daten
time	Hier speziell die Funktion process_time um die Performance meines Python-Programmes zu messen.

Verbindung mit der Datenbank

Listing 54. Code 1 - Verbinden mit der Datenbank

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["projekttest"]

mycol = mydb["out_data_embedded"]
```

Wie in Codebeispiel 1 sichtbar legt man verschiedene Objekte an. Ein Objekt für den Datenbankserver (hier MongoClient). Weitergehen legt man fest welche Datenbank man verwendet (mydb). Zum Schluss besteht die Möglichkeit optional auch direkt die Collection festzulegen (mycol), dies ist aber nicht zwangsläufig nötig.

Implementierung der Struktur

Festlegen von Datentypen

Listing 55. Code 2 - Datentypfestlegung

```
    df['DATE'] = pd.to_datetime(df['DATE'])
    df['Begin'] = pd.to_datetime(df['Begin'])
    df['FA'] = df['FA'].astype(str)
    df['SNR'] = df['SNR'].astype(str)
    df['LINIE'] = df['LINIE'].astype(str)
    df['E'] = df['E'].astype(str)
    df['ScanE'] = df['ScanE'].astype(bool)
    df['MessageE'] = df['MessageE'].astype(bool)
    df['V2'] = df['V2'].astype(np.float64)
    df['V1'] = df['V1'].astype(np.float64)
    df['UseM3'] = df['UseM3'].astype(np.float64)
    df['UseM2'] = df['UseM2'].astype(np.float64)
    df['UseM1'] = df['UseM1'].astype(np.float64)
    df['Delta'] = df['Delta'].astype(np.float64)
    df['Fehler'] = df['Fehler'].astype(int)
    df['Span'] = df['Span'].astype(int)
    df['ChargeM1'] = df['ChargeM1'].astype(str)
    df['ChargeM2'] = df['ChargeM2'].astype(str)
    df['ChargeM3'] = df['ChargeM3'].astype(str)
    df['ScanA'] = df['ScanA'].astype(bool)
    df['MessungA'] = df['MessungA'].astype(bool)
    df['LagerIn'] = df['LagerIn'].astype(str)
    df['LagerOut'] = df['LagerOut'].astype(str)
```

Wie bereits Anfangs erwähnt besitzen Dokumentenorientierte Datenbanken kein eigentliches Schema in diesem Sinne. So richten sich die initialen Datentypen der Attribute in den "Dokumenten" nach dem Datentyp den sie beim einlesen besessen haben. Da aber eine gewisse Konsistenz wichtig ist, gerade da später auch Abfragen auf die Daten gefahren werden sollen ist es nötig, dass die Datentypen einheitlich sind. Dies verhindert das es später zu unerwünschten Typkonvertierungen oder gar dem Absturz des Programmes kommt.

Dies löse ich sehr einfach wie in Codebeispiel 2 zu sehen in den ich die Daten, nachdem ich sie aus der Textdatei in einen Datenframe eingelesen habe, innerhalb dieses Datenframes bearbeite. Also egal welchen Datentyp die Daten in der Textdatei hatten beim Prozess des Einlesens findet immer eine Typkonvertierung statt und wenn sie dann in die Datenbank geschrieben werden stimmt ihr Typ solange man den Code nicht anfasst immer überein.

Herstellen der Relation

Listing 56. Code 3 - Herstellung der Relation

```
if (snr == "nan"):
        mycol_null.insert_many(df.to_dict('records'))
    else:
        x = mydb.in_data_embedded.aggregate([
        {'$match': {'SNR': snr}},
        {'$project': {'_id':1,
                      'SNR': 1,
                      'DATE': 1,
                      'difference':{'$subtract':[date,'$DATE']}}}
        ])
        return_df = pd.DataFrame()
        for data in x:
            data_x = pd.DataFrame(data, index=[data.get('_id')])
            return_df = return_df.append(data_x)
        if (not return_df.empty):
            return_df = return_df.drop(return_df[return_df.difference < 0].index)
            if (not return_df.empty):
                obj_id = return_df['difference'].idxmin()
                mydb.in_data_embedded.update_one({'_id': obj_id}, {"$addToSet": {
"out": dictionary}})
```

Es ist für mich besonders wichtig das die Relation direkt beim speichern der Daten korrekt durchgeführt wird, denn ich kann im Nachhinein nicht einfach meine Abfrage welche die Verknüpfung durchführt, einfach noch einmal anpassen. Die Daten müssen von Anfang an Korrekt sein da eine spätere Anpassung mit einem immensen Aufwand verbunden wäre.

Bei der Herstellung der Relation spricht der Code in Codebeispiel 3 nicht für sich selbst, daher wird er hier etwas näher erklärt. Zuerst überprüfe ich, ob die "SNR" des eingelesenen Ausgangsdatensatzes existiert. Sollte die "SNR" nicht existieren ist das "Dokument" für mich unbrauchbar und ich speichere es der Vollständigkeit halber in einer extra "Collection". Gibt es die "SNR" suche ich in der Datenbank nach Eingangsdatensätzen und speichere diese zusammen mit ihrer zeitlichen Differenz zum Ausgangsdatensatz in einem Datenframe. Da Eingangsdaten immer vor den Ausgangsdaten erstellt werden kann dieser Datenframe eigentlich nie leer sein aber um zu verhindern das mein Programm bei fehlerhaften Daten abstürzt prüfe ich ob der Datenframe leer ist. Ist dies nicht der Fall entferne ich Datensätze bei denen die zeitliche Differenz kleiner null ist, da diese Datensätze nach dem betrachteten Output in die Datenbank eingespeist wurden. Nun prüfe ich erneut ob der Datenframe leer ist. ist dies nach wie vor nicht der Fall lasse ich mir die eindeutige Objekt-ID des Datensatzes mit der geringsten zeitlichen Differenz geben und speichere den Ausgangsdatensatz in den Attributsschlüssel "out" des Eingangsdatensatzes.

Die Suche innerhalb der Datenbank nach einem passenden Input zum aktuellen Output stellt den einzige Flaschenhals beim Schreiben der Daten in die Datenbank dar. Ich habe hierfür einmal einen Vergleich durchgeführt zwischen dem suchen mit einem Index für die "SNR" und einmal ohne Index. Einen aufsteigenden Index kann man in Python mithilfe von Pymongo durch den Code in Codebeispiel 4 erstellen.

Listing 57. Code 4 - Anlegen eines Indexes

```
mydb.in_data_embedded.create_index([ ("SNR", 1) ])
```

Um zu vergleichen wie lange die Suche mit Index und ohne Index dauert, habe ich den MongoDB Profiler verwendet. Hierfür muss man nur über die MongoShell wie in Codebeispiel 4 eine bestimmte "Collection" erstellen. Und einstellen welche Aktionen alles gespeichert werden sollen. Profiling-Level 2 sorgt dafür dass jede aktion gespeichert wird.

Listing 58. Code 5 - Festlegen des Profilings

```
use projekt
db.createCollection( "system.profile", { capped: true, size:4000000 })
db.setProfilingLevel(2)
```

Durch diesen Vergleich bin ich zu folgenden Ergebnissen gekommen.

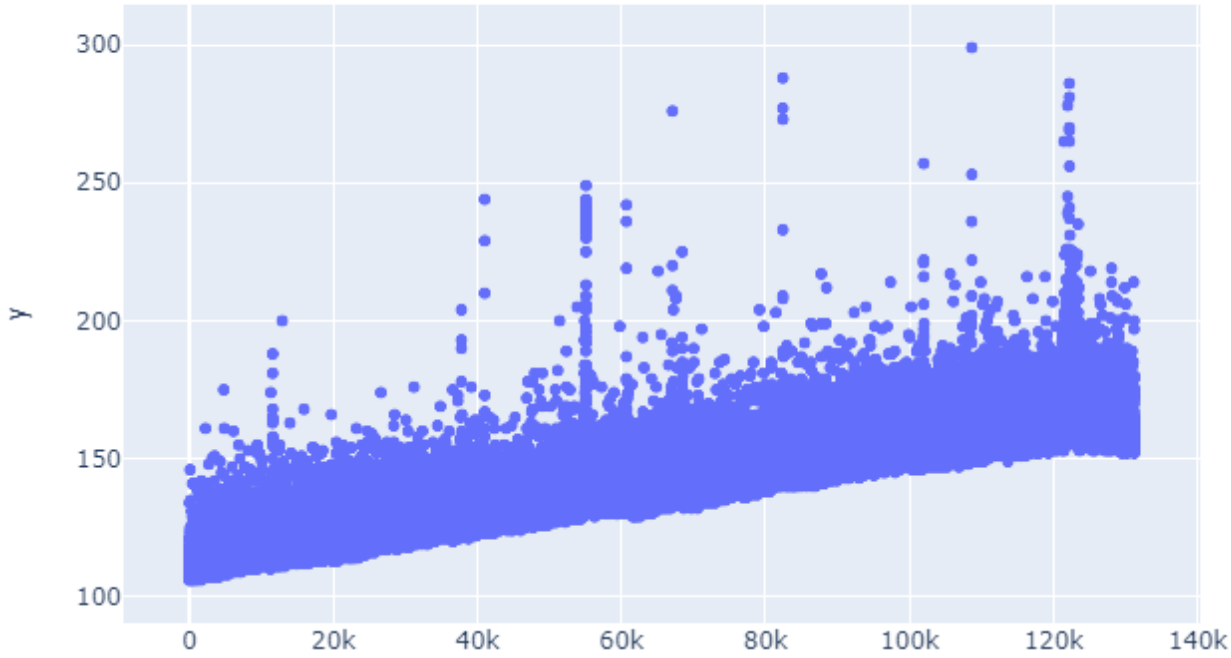

Abbildung 32. Ohne Index auf "SNR", x-Achse=Anzahl Eingangsdatensätze in der "Collection", y-Achse=Verarbeitungszeit in Millisekunden

Wie man auf Abbildung 4 ganz klar erkennen kann, zeichnet sich hier ein sehr unschöner Trend ab. Je mehr Eingangsdatensätze sich in der "Collection" befinden desto länger dauert die Suche nach einer "SNR" und dies fast in einem konstant linearen Anstieg. Bei der Menge an Datensätzen, die wir haben mag dies aktuell vll. noch kein großes Problem darstellen aber das kann sich in Zukunft schnell ändern.

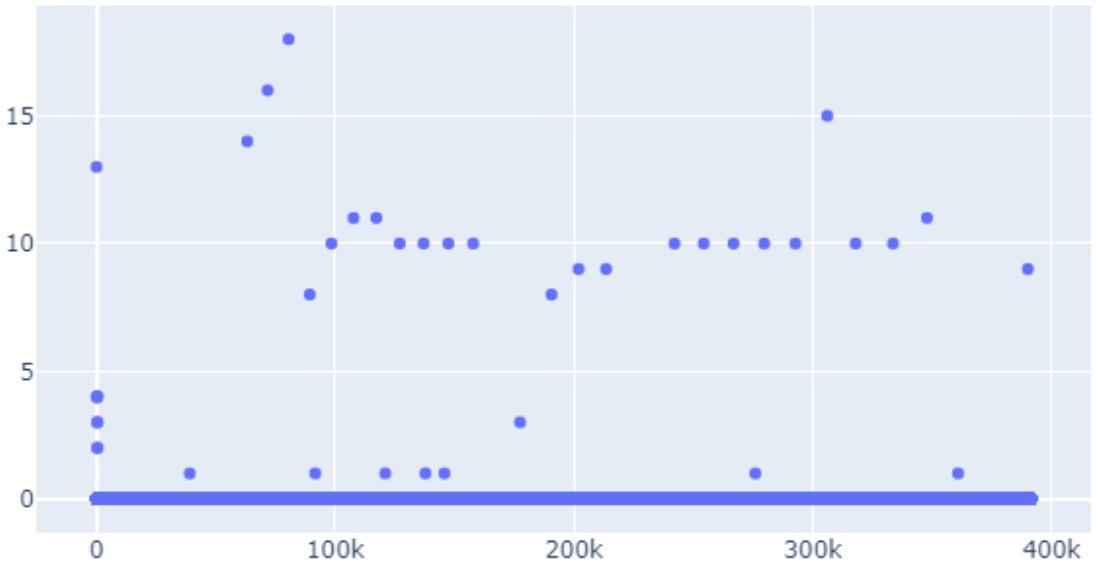

Abbildung 33. Mit Index auf "SNR", x-Achse=Anzahl Eingangsdatensätze, y-Achse=Verarbeitungsdauer in Millisekunden

Auf der Abbildung 5 wird eindeutig, welche Wirkung das Anlegen eines Indexes hatte. Zum einen fällt auf das es zu keinem linearen Anstieg der Verarbeitungszeit mehr kommt, zum anderen hat sich die Verarbeitungszeit insgesamt sehr stark reduziert. Vorher starteten die Zeiten bei 100 Millisekunden, jetzt hat selbst das Maximum bei einem vorhandenen Index nur einen Wert von ca. 17 Millisekunden.

Implementierung der Datenloader

Zu den Datenloadern lässt sich bei mir nicht viel weiteres erwähnen. Ich bekomme durch den Watchdog einen Pfad zu einer Datei über den Pfad wird dabei auch identifiziert ob es sich um einen Eingangs- oder Ausgangsdatensatz handelt und dann die zugehörige Funktion mit diesem Pfad als Übergabeparameter aufgerufen.

Implementierung der Analysen

Für die Analysen habe ich erneut Python als Programmiersprache genutzt. Ich möchte in diesem Teil vorallem auf die Abfragen in MongoDBs Abfragesprache eingehen.

Analyse 1

Listing 59. Code 6 - Query 1 zu Analyse 1

```
x = mydb.in_data_embedded.aggregate(
[{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "output": {"$arrayElemAt": ["$out", -1]}}},
 {"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "difference":{"$subtract":["$output.Date","$Begin"]}}},
 {"$match": {
            "difference": {"$lt": 3600000},
            "SNR": { "$ne": "nan" },
            "TEIL": teil}},
 {"$group" : {
            "_id":{
                "teil":"$TEIL",
                "fa":"$FA"},
            "teile_count": {"$sum":1},
            "maxFert": {"$max": "$difference"},
            "minFert": {"$min": "$difference"},
            "avgFert": {"$avg": "$difference"}}}])
```

Mithilfe von der Abfrage im Codebeispiel 6 lass ich mir zu allen Fertigungsaufträgen eines einzelnen Teils die Menge an Produkten die gefertigt wurden, sowie die maximale minimale und durchschnittliche Fertigungsdauer die dafür benötigt wurde. In der ersten *$project* Phase wähle ich die Attribute welche ich betrachten möchte. Die eingebaute Funktion *$arrayElemAt* kann ich mir das Element eines Arrays über den Index geben lassen. An dieser Stelle lasse ich mir durch -1 das letzte Element in dem Array der Ausgangsdatensätze eines Eingangsdatensatzes geben. Innerhalb der zweiten *$project* Phase rechne ich durch *$subtract* die Differenz zwischen dem Zeitstempel des Eingangsdatensatzes und dem Ausgangsdatensatzes aus. In der *$match* Phase lege ich fest das ich nur Daten möchte welche speziellen Bedingungen entsprechen. Ich lege fest das die Differenz durch *$lt* kleiner als 3600000 Millisekunden sein soll was einer Stunde entspricht. Außerdem sage ich das ich nur Datensätze möchte welche eine SNR haben und als Teil dem Wert der Variable *teil* entsprechen. In der letzten Phase der *$group* Phase führe ich ein Group By durch nach Fertigungsauftrag durch und berechne durch *$sum* die Menge an Produkten sowie die maximale, minimale und durchschnittliche Fertigungsdauer. Durch die Variable *teil* kann ich durch ein Array aus den Teilen iterieren und dies für jedes Teil wiederholen.

Listing 60. Code 7 - Query 2 zu Analyse 1

```
y = mydb.in_data_embedded.aggregate(
[{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "output": {"$arrayElemAt": ["$out", -1]}}},
{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "difference":{'$subtract':['$output.Date','$Begin']}}},
{"$match": {
            "difference": {"$lt": 3600000},
            "SNR": { "$ne": "nan" },
            "FA": fa }},
{"$group": {
            "_id": {
                "SNR":"$SNR",
                "TEIL": "$TEIL",
                "FA": "$FA"},
            "count": {"$sum":1}}},
{"$group": {
            "_id": {
                "teil":"$_id.TEIL",
                "fa":"$_id.FA"},
            "max_o":{"$max": "$count"},
            "min_o":{"$min": "$count"},
            "avg_o":{"$avg": "$count"}}},
{"$sort": {"_id.fa":1}}])
```

Aus der Abfrage im Codebeispiel 6 bekomme ich eine Menge an Fertigungsaufträgen, mit Daten zu diesen nun möchte ich noch wissen wie viel Ausschuss bei diesen Fertigungsaufträgen entstanden ist. Dafür sind die ersten 3 Phasen gleich wie in der vorrangegangenen Abfrage aus Codebeispiel 5. Nur in der *$match* Phase wie in Codebeispiel 7 zusehen ändert sich etwas, wir möchten nun Datensätze welche einem gewissen Fertigungsauftrag entsprechen. In der ersten *$group* Phase gruppieren wir nach "SNR", "TEIL" und "FA". Durch das gruppieren nach SNR entfernen wie Dopplungen gleichzeitig zählen wir aber auch wie oft eine "SNR" aufgetaucht ist. Dieser gezählte Wert wiederrum spiegelt dann wenn er >1 ist die Menge an Ausschuss wieder. in der letzten *$group* Phase gruppiere ich nach Teil und Fertigungsauftrag und ermittle für die Fertigungsaufträge das Maximum an Ausschuss sowie das Minimum und den Durchschnitt. Sortiert wird das Ergebnis aufsteigend nach Fertigungsauftrag. Da bei dieser Abfrage mehrere Abfragen in einer Schleife durchgeführt werden, dauert sie relativ lange, hier kann durch Verbesserungen/Anpassungen sicher noch Performance gut machen.

Analyse 2

Listing 61. Code 8 - Query 1 Analyse 2

```
z = mydb.in_data_embedded.aggregate([
{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "output": {"$arrayElemAt": ["$out", -1]}}},
{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1, "difference":{"$subtract":["$output.Date","$Begin"]}}},
{"$match": {
            "difference": {"$lt": 3600000},
            "SNR": { "$ne": "nan" },
            "TEIL": teil}},
{"$group" : {
            "_id":{
                "teil":"$TEIL",
                "fa":"$FA"},
            "teile_count": {"$sum":1}}},
{"$group":{
            "_id": "$_id.teil",
            "count": {"$sum":"$teile_count"}}}])
```

Nachdem nun durch die Erklärung der ersten Analyse klar sein sollte wie eine MongoDB Abfrage funktioniert möchte ich aufgrund des Umganges der Abfragen nicht übermäßig ins Detail gehen. In der Abfrage aus Codebeispiel 8 lassen wir uns die Gesamtfertigungsmenge jedes einzelnen Teils ausgeben, dabei berücksichtigen wir nur Datensätze mit einer Fertigungsdauer unter einer Stunde und einer vorhandenen "SNR". Diese Gesamtfertigungsmenge brauchen wir um später die Fehlerrate auszurechnen.

Listing 62. Code 9 - Query 2 Analyse 2

```
y = mydb.in_data_embedded.aggregate([
{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "out": {"$ifNull": [ "$out", [{"Date":"undefined"}]]}}},
{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "out":{"$arrayElemAt": ["$out", -1]}}},
{"$project": {
            "_id":1,
            "TEIL":1,
            "FA":1,
            "Begin":1,
            "SNR":1,
            "output_date":"$out.Date"}},
{"$match": {
            "SNR": { "$ne": "nan" },
            "TEIL": teil}},
{"$group" : {
            "_id": "$SNR",
            "count": {"$sum":1},
            "starts":{
                "$push":{
                    "Begin":"$Begin",
                    "Out":"$output_date"}}}},
{"$match": {"count":{"$gt":1}}}])
```

Bei der Abfrage in Codebeispiel 9 gibt es einige Besonderheiten auf die ich gerne eingehen möchte. Zum einen ändern wir die erste *$project* Phase um sicherzugehen das wir auch Datensätze bekommen die keinen Ausgangsdatensatz haben. Dies ist wichtig da wir um die Zeiten bei der Auftrennung zu berechnen den Zeitstempel eines Ausgangsdatensatzes von dem Zeitstempel des nachfolgenden Eingangsdatensatzes mit derselben "SNR" subtrahieren und hierfür ist es wichtig das auch Eingangsdatensätze zu denen noch kein Ausgangsdatensatz existiert berücksichtigt werden. An der zweiten und dritten *project* Phase sowie der *$match* Phase ändert sich nichts. Interessant wird es in der *$group* Phase hier gruppieren wir nach "SNR" dabei wird zusätzlich die Menge gezählt wie oft eine "SNR" aufgetaucht ist, aber das eigentlich wichtige ist das in dieser Phase während gruppiert wird jeweils zu jedem Datensatz der Zeitstempel des Eingangsdatensatzes und des Ausgangsdatensatzes in ein Array gespeichert wird. Schlussendlich legen wir noch fest ,dass wir nur Datensätze wollen bei denen auch Ausschuss entstanden ist.

Da diese Abfrage in Codebeispiel 9 zu den Komplexesten zählt, möchte ich an dieser Stelle auch auf

den Python Code eingehen der die Abfrage schließlich verarbeitet.

Listing 63. Code 10 - Pythoncode Analyse 2

```
    for data in y:
        x = 1
        amount += data.get("count")-1
        differences = []
        data_sorted = sorted(data.get('starts'), key = lambda i: i['Begin'])
        while x < len(data.get('starts')):
            value_1 = data_sorted[x].get('Begin')
            value_2 = data_sorted[x-1].get('Out')
            if(value_2 != 'undefined'):
                value = value_1 - value_2
                if (value > datetime.timedelta()):
                    value = value.total_seconds()
                    differences.append(value)
                    avg_val.append(value)
            x += 1
        if len(differences)>0:
            max_val.append(max(differences))
            min_val.append(min(differences))
    maximum = max(max_val)
    minimum = min(min_val)
    avg = sum(avg_val)/len(avg_val)
```

Wir iterieren in Codebeispiel 10 durch den Coursor welchen wir durch die MongoDB Abfrage aus Codebeispiel 9 bekommen haben. Wir errechnen den Ausschuss in dem wir den Wert der Datenbank minus eins rechnen da es sich bei einem Datensatz ja um einen erfolgreichen handelt. Wir legen ein Array an, um die Zeitdifferenzen abzuspeichern. Außerdem sortieren wir das Array welches wir zu jeder "SNR" bekommen haben und welches die Zeitstempel enthält. Wir sortieren aufsteigend nach dem Zeitstempel des Eingangsdatensatzes. Zum verständniss es handelt sich um ein Array aus Objekten, wobei jedes Objekt zwei Attribute enthält den Zeitstempel des Eingangsdatensatzes sowie den Zeitstempel des dazugehörigen Ausgangsdatensatzes. Durch dieses Array gehen wir nun in einer Schleife hindurch und subtrahieren den Zeitstempel des Ausgangsdatensatzes vom Zeitstempel des nachfolgenden Eingangsdatensatzes. Dabei überprüfen wir ob auch alle Werte existieren und schließlich ob die errechnete Differenz > null ist. Trifft beides zu wandeln wir die errechnete Differenz in Sekunden um und speichern sie in einem Array. Es gibt hierbei zwei Arrays eins enthält alle Zeitdiffferenzen zu einer "SNR" aus diesem ermitteln wir später den maximalen und minimalen Wert und speichern diesen wiederum in einem Array. Das zweite Array benötigen wir um später den Durschnitt an Zeitdifferenzen über alle "SNRs" aus zu rechnen. Dieses zweite Array leert sich nicht für jede "SNR" wieder sondern enthällt alle Zeitdiffferenzen.

Alle obigen Aktionen werden in einer Schleife für jedes "TEIL" durchgeführt. Auch hier kann man Performance verbesserungen durchführen in dem man die Anzahl an Abfragen auf die Datenbank reduziert.

Analyse 4

Listing 64. Code 11 Query 1 Analyse 4

```
x = mydb.in_data_embedded.aggregate([
{"$project": {
            "_id":1,
            "LagerIn":1,
            "Begin":1,
            "SNR":1,
            "output": {"$arrayElemAt": ["$out", -1]}}},
{"$project": {
            "_id":1,
            "LagerIn":1,
            "Begin":1,
            "SNR":1,
            "end":"$output.Date"}},
{"$match": {
            "SNR": { "$ne": "nan" }}},
{"$group" : {
            "_id":{
                "SNR":"$SNR",
                "LagerIn":"$LagerIn"},
            "start": {"$min": "$Begin"},
            "end": {"$max": "$end"}}},
{"$group": {"_id":"$_id.LagerIn",
            "anz":{"$sum":1},
            "start":{"$min": "$start"},
            "end":{"$max": "$end"}}},
{"$project":{
            "_id":1,
            "anz":1,
            "start":1,
            "end":1,
            "duration":{'$subtract':['$end','$start']}}},
{"$sort":{"_id": 1}}])
```

Bei Analyse 4 handelt es sich um eine der simpleren Analysen hier kann alles mit einer einzelnen Query gelöst werden. Die ersten drei Phasen sind wieder die Auswahl der Felder sowie das aussortieren von Datensätzen ohne "SNR". In der ersten *$group* Phase gruppieren wir nach "SNR" und "LagerIn" um Dopplungen bei den "SNR" loszuwerden. An dieser Stelle wird außerdem der minimale Beginn bei mehreren gleichen "SNRs" festgestellt sowie das späteste Ende. Weiterführend gruppieren wir in der zweiten *$group* Phase gruppieren wir nach "LagerIn" und zählen die gefertigten Produkte. Wir ermitteln das minale Startdatum der Nutzung eines Ladungsträgers und das späteste Enddatum. Schlussendlich berechnen wir aus den beiden Zeitstempeln in der *$project* Phase die Nutzungsdauer eines Ladungträgers und sortieren dann unsere Ergebnisse aufsteigend nach Ladungsträger.

Analyse 5

Listing 65. Code 12 Query 1 Analyse 5

```
y = mydb.in_data_embedded.aggregate([
{"$project": {
            "_id":1,
            "TEIL":1,
            "LagerIn":1,
            "Begin":1,
            "SNR":1,
            "output": {"$arrayElemAt": ["$out", -1]}}},
{"$project": {
            "_id":1,
            "TEIL":1,
            "LagerIn":1,
            "Begin":1,
            "SNR":1,
            "difference":{'$subtract':['$output.Date','$Begin']}}},
{'$match': {"SNR": { "$ne": "nan" }}},
{"$group": {
            "_id":{
                "LagerIn":"$LagerIn",
                "Teil":"$TEIL"},
            "anz":{"$sum":1},
            "min":{"$min": "$difference"},
            "max":{"$max": "$difference"},
            "avg":{"$avg":"$difference"}}},
{"$sort":{"_id": 1}}])
```

Die 5. Analyse gleicht fast vollständig der ersten (zu sehen in Codebeispiel 6 und 7), nur das bei hier nach "LagerIn" also Ladungsträger gruppiert wird und nicht nach Fertigungsauftrag.

Analyse 6

Listing 66. Code 13 Query 1 Analyse 6

```
x = mydb.in_data_embedded.aggregate([
{"$project": {
            "_id":1,
            "LINIE":1,
            "FA":1,
            "TEIL":1,
            "Begin":1,
            "SNR":1,
            "output": {"$arrayElemAt": ["$out", -1]}}},
{"$project": {
            "_id":1,
            "LINIE":1,
            "FA":1,
            "TEIL":1,
            "Begin":1,
            "SNR":1,
            "end":"$output.Date"}},
{"$match": {
            "SNR": { "$ne": "nan" },
            "LINIE":line}},
{"$group": {
            "_id":{
                "FA": "$FA",
                "TEIL": "$TEIL"},
            "start":{"$min": "$Begin"},
            "end":{"$max": "$Begin"}}},
{"$sort":{"start": 1}}])
```

In Analyse 6 mache ich innerhalb einer Schleife Abfragen zu jeder Linie, dabei wähle ich mir wie in Codebeispiel 13 zu sehen in den ersten beiden *$project* Phasen die Attribute aus die ich betrachten möchte und lasse mir den Zeitstempel des Outputs geben. Weiterführend wähle ich der *$match* Phase nur Elemente welche eine "SNR" haben und welche als "LINIE" den Wert der Variablen Linie haben. Zum Schluss gruppiere ich nach Fertigungsauftrag und Teil. Außerdem lasse ich mir zu jeder Gruppe den frühesten Startzeitpunkt und spätesten Endzeitpunkt geben. Eine aufsteigende Sortierung nach den Startzeitpunkten bringt die Datensätze schließlich in die richtige Reihenfolge.

Listing 67. Code 14 Pythoncode 1 Analyse 6

```
    teil_value = pd.DataFrame()
    i=0
    list = []
    for data in x:
        if i == 0:
            teil_1 = data.get("_id").get('TEIL')
            time_1 = data.get("end")
            i=1

        if i==2:
            teil_2 = data.get("_id").get('TEIL')
            time_2 = data.get("start")
            end = data.get("end")
            difference = time_2 - time_1
            seconds = difference.total_seconds()
            if seconds > 0:
                data = {"FROM_TO": teil_1 + " zu " + teil_2, "Dauer": seconds }
                list.append(data)
            else:
                print(teil_1,teil_2)
            teil_1 = teil_2
            time_1 = end
            i=2
        else:
            i += 1
    teil_values = pd.DataFrame(list)
    distinct_values = teil_values["FROM_TO"].unique()
    for value in distinct_values:
        helper_list = teil_values.loc[teil_values['FROM_TO'] == value]["Dauer"]
        maximum = helper_list.max()
        minimum = helper_list.min()
        avg = sum(helper_list)/len(helper_list)
```

Ich gehe nun wie in Codebeispiel 14 in einer Schleife durch die Ergebnisse meiner Query aus Codebeispiel 12, dabei betrachte ich immer ein paar aus Datensätzen Für den ersten Datensatz speichere ich das Teil um das es sich handelt sowie den Zeitstempel des Endes. Für den zweiten Datensatz speichere ich auch das Teil um welches es sich handelt sowie den Zeitstempel des Starts und den des Endes. Schließlich berechne die ich die Differenz der Zeitstempel also den Start des zweiten subtrahiert vom Ende des ersten. Diese Zeitdifferenz wandel ich in Sekunden um und Speichere sie als Dictionary in einer Liste zusammen mit einem String der den Wechsel der Teile repräsentiert. Ich sätze nun sowohl die Zeit als auch das Teil vom ersten Teil auf die Werte des zweiten und lasse die Schleife erneut laufen. Wenn die Schleife fertig ist wandel ich die Liste in einen Datenframe um um die pandas eigene Funktion *.unique()* anzuwenden welche mir jeden Wert welcher in der Spalte FROM_TO steht und die Teil wechsel repräsentiert zurückgibt und Dopplungen ignoriert. Zum Schluss iteriere ich durch diese eindeutigen Werte durch und lasse mir immer eine Hilfliste erstellen in welcher alle Zeitdifferenzen zu einem Teil-Wechsel stehen, aus dieser List kann ich nun wiederrum das Maximum, Minimum und den Durschnitt ermitteln.

Performance-Messung der Analysen

Die Performance meiner Abfragen messe ich an zwei Stellen zum einen mit dem Datenbank-Profiler von MongoDB und zum anderen in dem ich die Prozesszeit des Python-Programmes messe.

Listing 68. Codebeispiel 15 - Prozesszeiten-Messung

```
start = process_time()

# eigentliches Programm

end = process_time()

print(end - start)
```

Wie in Codebeispiel 15 zu sehen nutze ich dafür die Funktion *process_time()* und erstelle mir zu Beginn des Programmes sowie zum Ende des Programmes einen Zeitstempel, welche ich dann voneinander subtrahiere.

Dadurch komme ich zu folgenden Performance Ergebnissen bei denen Prozesszeiten und Datenbankzeit kummuliert sind.

Analyse	Performance
Analyse 1	18.87 Sekunden
Analyse 2	20.51 Sekunden
Analyse 4	4.99 Sekunden
Analyse 5	3.24 Sekunden
Analyse 6	6.91 Sekunden

5.4. Schlüssel-Werte-Datenbank

5.4.1. Einführung in Datenbankstruktur

Quellen

http://wi-wiki.de/doku.php?id=bigdata:keyvaluedb https://www.ionos.de/digitalguide/hosting/hosting-technik/key-value-store/ https://en.wikipedia.org/wiki/Key%E2%80%93value_database https://db-engines.com/en/ranking/key-value+store

Datenbankvorstellung

Allgemein

Die Schlüssel-Werte-Datenbank, englisch Key-Value-Store, ist ein auf dem Schlüssel-Werte-Datenmodell basierendes Datenbankmodell. Sie gehört zu den ältesten NoSQL Datenbanken, verfolgt also einen nicht-relationalen Ansatz. Das Prinzip einer solchen Datenbank stellt die Verknüpfung eines Schlüssels mit einem Wert dar, welche einen Datensatz ergibt. Die einzige Restriktion besteht darin, dass jeder Schlüssel eindeutig sein muss. Der Wert ist aus Sicht des Datenbankmodells lediglich eine

Bitfolge.

Aus dieser einfachen Struktur ergeben sich in der Anwendung diverse Vorteile. Zum einen entsteht ein Vorteil dadurch, dass man direkt über den Schlüssel auf den Wert zugreift. Dies ermöglicht sehr schnelle Zugriffe. Ein weiterer Vorteil ist, dass alleine die Eindeutigkeit des Schlüssels als Restriktion für einen Datensatz vorliegt. Das bedeutet, dass die Schlüssel und Werte beliebige Strukturen aufweisen können. So kann ich einem Schlüssel als Wert eine einfache 0, einem anderen Schlüssel beispielsweise eine Liste aus 100.000 Elementen zuweisen. Dies führt dazu, dass man sich nicht an strukturelle Regeln halten muss und somit auch im laufenden Betrieb beliebig bestehende Strukturen anpassen und Neue einführen kann. Diese einzige Limitierung führt außerdem dazu, dass die Datensätze auf dem Datenträger einfach hintereinander weg geschrieben werden können.

Die einfache Struktur des Datenbankmodells führt allerdings auch zu Herausforderungen, welchen begegnet werden muss. Dabei ist vor allem hervorzuheben, dass Werte nicht durchsucht werden können. Dies führt insbesondere dazu, dass komplexe Abfragen weitestgehend selbst ermöglicht werden müssen - die Datenbank übernimmt nur Abfragen über den Schlüssel. Außerdem ist es nicht möglich den Inhalt des Wertes zu filtern, da dieser aus Sicht des Modells einfach nur eine Bitfolge ist. So wird bei einem read immer der ganze Wert zurückgegeben und bei einem update der ganze Wert neu geschrieben.

Redis

In unserem Anwendungsfall haben wir uns für Redis als Implementierung einer Schlüssel-Werte-Datenbank entschieden. Gründe für die Entscheidung waren, dass Redis open-source ist und als am weitesten verbreitete Schlüssel-Werte-Datenbank und Datenbankmanagementsystem gut dokumentiert ist. Von der Community erstellte Bibliotheken bieten Schnittstellen für die Kommunikation zwischen Programm und Datenbank, wodurch wir alle für uns relevanten Datenbankoperationen einfach in Python abbilden konnten. Redis arbeitet standardmäßig In-Memory, wodurch Zugriffe auf andere Datenträger wegfallen. Persistenz wird dabei auch ermöglicht und ist individuell anpassbar. Unterstützt werden in Redis als Datentypen nicht nur Strings, sondern auch z.B. Lists, Sets, Hashes und Bitmaps.

Eine Herausforderung bezüglich Redis bestand für uns darin, dass Windows weder offiziell noch durch Implementierungen der Community unterstützt wird. Wir haben zwei Lösungsansätze probiert, zum einen die Installation auf dem offiziellen Linux Subsystem von Windows und zum anderen die Installation auf einer virtuellen Maschine. Dabei haben die ersten Implementierungen bereits gezeigt dass auf dem Linux Subsystem die Performance von Datenbankoperationen deutlich niedriger ist als auf der virtuellen Maschine. Die Implementierungen für unseren Anwendungsfall erfolgten deshalb im "VMware Workstation 16 Player" mit Ubuntu 20.04 LTS 64 bit.

5.4.2. Entwurf

Erstes Datenmodell

Das erste Datenmodell war ein einfaches Modell, welches sich die Vorteile einer Schlüssel-Werte-Datenbank eher weniger zunutze gemacht hat. Das bedeutet insbesondere, dass die Abwesenheit der Möglichkeit komplexer Abfragen in Redis zur Verschiebung dieser Aufgabe in die eigene Python-Programmierung geführt hat. Außerdem wurden lediglich Listen und Hashes als Datentypen verwendet, welche zwar ausreichend, für die Performance aber negativ zuträglich waren. Analysen haben aufgrund der sehr großen Anzahl an Abfragen an die Datenbank und Vergleichsoperationen zwischen Listen in diesem Modell zwischen 4 und 8 Minuten gedauert.

Finales Datenmodell

Das neue Datenmodell wurde aufgrund der schlechten Performance der Implementierung des ersten Entwurfs initiiert. Die Anforderungen an das neue Modell waren zum einen die Reduzierung der Anzahl der Abfragen an die Datenbank insgesamt und zum anderen schneller die richtigen Datensätze zu laden und auszuwerten, also insbesondere Datensätze mit gleichen Eigenschaften (z.B. Maschine 2 und Teil B) performant zu finden und auszuwerten. Die Reduzierung der Abfragen wurde dadurch gelöst, in die Schlüssel der Verknüpfungen direkt zugehörige, für die Analysen notwendige Informationen, also Anfangs-/Enddatum und Seriennummer, zu speichern. So muss nicht mehr noch eine ebene tiefer in die Rohdaten gegangen werden. Das performante Auffinden der richtigen Datensätze wird durch Bitmaps realisiert. Dadurch können Übereinstimmungen direkt serverseitig festgestellt werden und es ist nicht mehr nötig Listen mit einer bis zu sechsstelligen Anzahl an Elementen zu übertragen und selbst auszuwerten. Daneben wurde noch die Wahl der Datentypen angepasst um eine höhere Performance zu erreichen.

Der Kern des Datenmodells ist in folgender Abbildung dargestellt. Im Zentrum stehen die Verknüpfungen, welche für die Analysen wichtige Informationen enthalten. Sie verweisen auf die Rohdatensätze, die diesen Informationen zugrunde liegen. Die Eigenschaften geben mit den Bitmaps an, in welchen Verknüpfungen sie vorkommen.

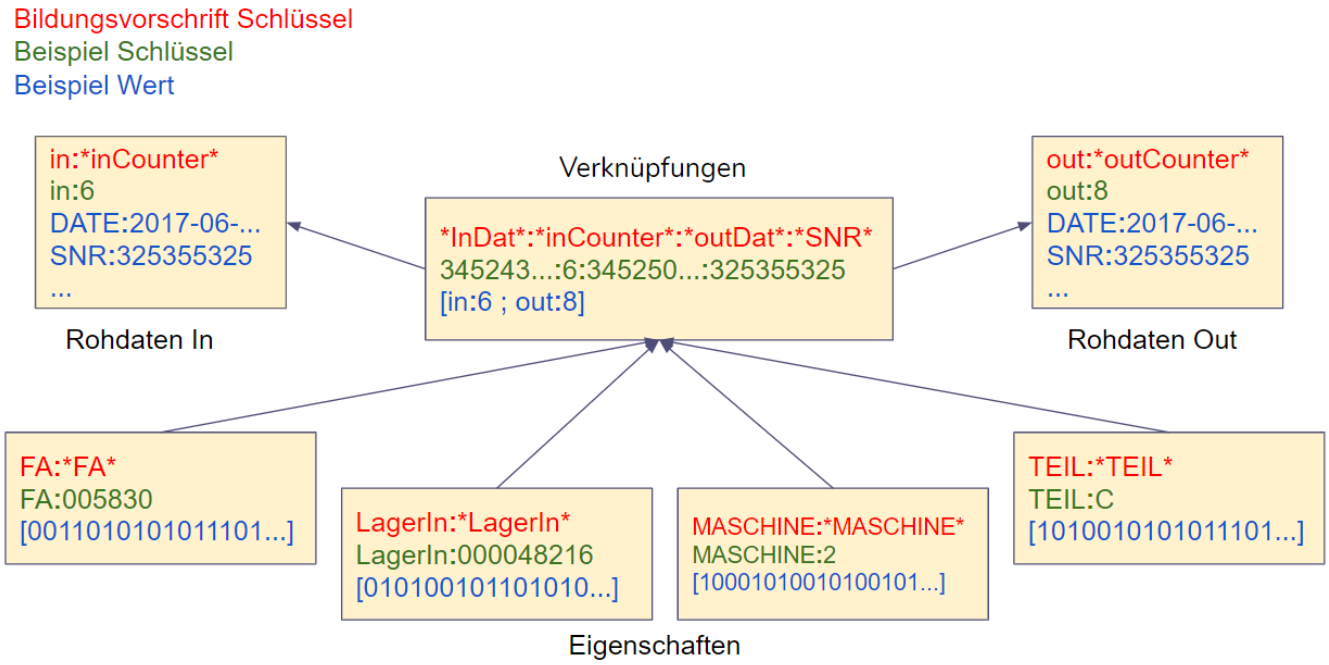

Abbildung 34. Datenmodell

Zusätzlich zu den dargestellten ergibt sich das Datenmodell aus mehreren Komponenten, welche genutzt werden um neue Daten einzufügen und aus den vorhandenen Daten ordentlich zu lesen. In der Bildungsvorschrift *kursiv* geschriebene Begriffe sind dynamisch, variieren also je nach Datensatz dieser Komponente. Die von uns vergebenen Begrifflichkeiten sind in der Legende im letzten Anstrich erläutert:

- Rohdaten
  - Abbildung der Datensätze wie sie kommen
  - ¬Datentyp Hash ¬ Werte Sind Field-Value Paare (Strings)
  - Bildungsvorschrift Schlüssel: "in:*inCounter* bzw. out:*outCounter*"
  - Beispiel:

Key: **in:1**

Type: **Hash**

Field	Value
DATE	2017-12-22T10:08:51.0000000
FA	005689

Abbildung 35. Rohdatensatz

- Erklärung: Alle Rohdatensätze müssen gespeichert werden. Die Speicherung als Hash ermöglicht es, nur bestimmte Felder abzufragen. Für die im Projekt geforderten Analysen haben die abgespeicherten Rohdatensätze keine Bedeutung mehr.

- Zähler
  - Abbildung des aktuellen Standes von inCounter bzw. outCounter
  - ¬Datentyp String ¬ Inkrementiert für jeden neuen In- bzw. Out-Datensatz
  - Bildungsvorschrift Schlüssel: "inCounter" bzw. "outCounter"
  - Beispiel:

Key: **inCounter**

Type: **String**

100

Abbildung 36. Zähler

  - Erklärung: Um die Eindeutigkeit der Schlüssel zu wahren, werden inCounter und outCounter in der Datenbank selbst persistiert, um für den nächsten Rohdatensatz den richtigen Zähler bereitstellen zu können.

- Seriennummern
  - Abbildung der zu einer Seriennummer gehörenden Input-Datensätze
  - ¬Datentyp Liste ¬ Minimal 1 Element
  - Bildungsvorschrift Schlüssel: "*SNR*"
  - Beispiel:

Key: **8743032488871**

Type: **List** (2 Items)

#	Value
0	in:60
1	in:72

Abbildung 37. Seriennummern

  - Erklärung: Ein Output-Datensatz gehört zu einem Input-Datensatz. Als zugehörige Input-Datensätze kommen nur solche mit entsprechender Seriennummer in Frage. Um den richtigen Input-Datensatz zu finden, muss nur die Liste mit entsprechender Seriennummer

durchgegangen werden.

- Zeitverknüpfung
  - Abbildung des Zeitstempels des Input-Datensatzes und des höchsten Zeitstempels der mit dem Input verknüpften Out-Datensätze
  - ¬Datentyp Liste ¬ Minimal 1, maximal 2 Elemente
  - Bildungsvorschrift Schlüssel: "*inCounter*"
  - Beispiel:

Key: **42**

Type: **List** (2 Items)

#	Value
0	1514869942.0
1	1514870324.0

Abbildung 38. Zeitverknüpfung

  - Erklärung: Da in einer Schlüssel-Werte-Datenbank nicht über Werte gesucht werden kann und die Verknüpfungen im Schlüssel Zeitstempelinformationen enthalten (die vom Input und die des spätesten Output), braucht es zum Zugriff auf den Schlüssel der entsprechenden Verknüpfung die richtigen Zeitstempel. Diese können hier geladen und eingetragen werden.

- Verknüpfungen
  - Abbildung der Verknüpfungen eines Input-Datensatzes mit zugehörigen Output-Datensätzen
  - ¬Datentyp Liste ¬ Erstes Element ist die Referenz auf den Input-Datensatz, ab zweitem Element Referenz auf Output-Datensatz
  - Bildungsvorschrift Schlüssel: "*inDat*:*inCounter*:*outDat*:*SNR*"
  - Beispiel:

Key: **1514868597.0:15:1514869172.0:8743032489524**

Type: **List** (2 Items)

#	Value
0	in:15
1	out:16

Abbildung 39. Verknüpfungen

  - Erklärung: Verknüpfungen sind das zentrale Element des Modells. Dabei ist vor allem der Schlüssel mit den Informationen von Zeitstempel (der vom Input und des spätesten Output) und Seriennummer wichtig. Auf die Elemente der Listen wird in den geforderten Analysen nicht zugegriffen.

- Verknüpfungsliste
  - Abbildung aller Verknüpfungen
  - ¬Datentyp Liste ¬ Elemente sind Schlüssel der Verknüpfungen

- Bildungsvorschrift Schlüssel: "con"
- Beispiel:

Key: **con**

Type: **List**

#	Value
0	1513933731.0:1:1514868283.0:4072932452084
1	1513933812.0:2:1514868465.0:4072932452082
2	1514867795.0:3:1514868545.0:4072932452087

Abbildung 40. Verknüpfungsliste

- Erklärung: Diese Liste enthält alle Verknüpfungen. Die 1-Positionen, die aus der AND-Verknüpfungen der Bitmaps der Eigenschaften entstehen, können als Indices dieser Liste betrachtet werden.

• Eigenschaften
  - Abbildung der Verknüpfungen, welche diese Eigenschaft besitzen
  - ¬Datentyp Bitmap ¬ Folge von 0 und 1
  - Bildungsvorschrift Schlüssel: "*Eigenschaftsname*:*Eigenschaftsausprägung*"
  - Beispiel:

Key: **FA:005689**

Type: **String (Binary)**

```
00000000  70 EF BF BD EF BF BD 6A  p ï ¿ ½ ï ¿ ½ j
00000008  EF BF BD 44 EF BF BD 00  ï ¿ ½D ï ¿ ½ .
00000010  12 49 EF BF BD 00 40     . I ï ¿ ½ . @
```

Abbildung 41. Eigenschaften

  - Erklärung: Für jede Verknüpfung wird im Wert angegeben, ob diese die Eigenschaft besitzt (1) oder nicht (0).

• Ausprägungen der Eigenschaft
  - Abbildung der Ausprägungen der Eigenschaften, welche in den Datensätzen vorgekommen sind
  - ¬Datentyp Set ¬ Unsortierte Menge der Ausprägungen
  - Bildungsvorschrift Schlüssel: "*Eigenschaftsname*"
  - Beispiel:

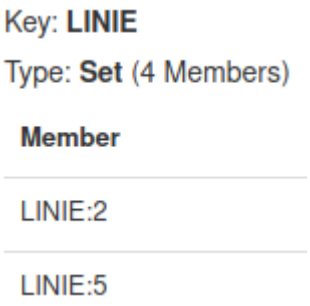

Abbildung 42. Ausprägungen der Eigenschaften

- Erklärung: Die Ausprägungen der Eigenschaften werden gespeichert, damit bei Abfragen über alle Ausprägungen alle entsprechenden Schlüssel der Eigenschaften (Name+Ausprägung) bekannt sind.

- Rohdatensätze ohne Seriennummer
  - Abbildung der Rohdatensätze ohne Seriennummer
  - ¬Datentyp Hash ¬ Werte Sind Field-Value Paare (Strings)
  - Bildungsvorschrift Schlüssel: "defect:raw:in:*inCounter*" bzw. "defect:raw:out:*outCounter*"
  - Beispiel:

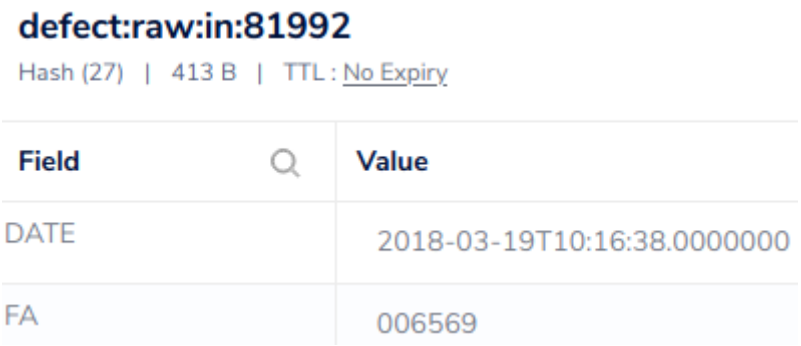

Abbildung 43. Rohdatensatz ohne Seriennummer

  - Erklärung: Datensätze ohne Seriennummer besitzen für die Analysen keine Relevanz. Aufgrund der Anforderungen werden sie trotzdem gespeichert.

- Liste der Rohdatensätze ohne Seriennummer
  - Abbildung aller Rohdatensätze ohne Seriennummer
  - ¬Datentyp Set ¬ Schneller Zugriff auf alle Elemente
  - Bildungsvorschrift Schlüssel: "defect:list:in" bzw. "defect:list:out"
  - Beispiel:

Abbildung 44. Liste Rohdatensätze ohne Seriennummer

  - Erklärung: Damit alle Datensätze ohne Seriennummer schnell gefunden werden können, werden deren Schlüssel in diesem Set gespeichert.

- Bitpositionen
  - Abbildung des Ergebnisses der bitweisen AND Operation
  - Bildungsvorschrift Schlüssel: "opCon"
  - Beispiel:

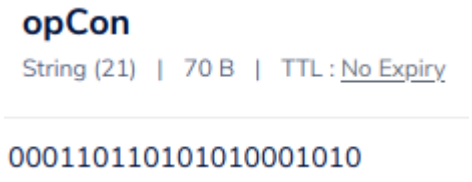

Abbildung 45. Bitpositionen AND-Operation

  - Das Ergebnis einer bitweisen AND Operation kann nicht direkt zurückgegeben werden, sondern wird selbst in der Datenbank gespeichert.

- Legende:
  - inCounter: Zähler, der für jeden eingelesenen Input-Datensatz inkrementiert
  - outCounter: Zähler, der für jeden eingelesenen Output-Datensatz inkrementiert
  - inDat: Zeitstempel des Input-Datensatz
  - outDat: Zeitstempel des Output-Datensatz
  - SNR: Seriennummer
  - Eigenschaftsname: Titel der Eigenschaft, z.B. "LINIE" oder "TEIL"
  - Eigenschaftsausprägung: Ausprägung der Eigenschaft mit dem zugehörigen Eigenschaftstitel, z.B. "005757" für den Fertigungsauftrag Nr. 005757

5.4.3. Implementierung

Verwendete Tools

Für die Implementierung wurden mehrere Tools verwendet.

- VMware Workstation 16 Player mit Ubuntu 20.04 LTS 64 bit

- Redis 5.0.7
- Python 3.8.5
- redis-py (Python-Bibliothek für Redis)
- redis-commander (Web-UI für Redis Server)
- RedisInsight (Web-UI für Redis Server)

Implementierung der Struktur

Aufgrund der Eigenschaften einer Schlüssel-Werte-Datenbank, insbesondere der Flexibilität von Datenstrukturen, wird die Struktur mit dem Laden der Daten aufgebaut. Damit ist sichergestellt, dass nur tatsächlich benötigte Strukturen gespeichert werden. Die Implementierung der Struktur erfolgt somit während der Laufzeit im Datenloader, die Grundstruktur ist eine leere Redis-Instanz.

Implementierung der Datenloader

Es gibt 2 verschiedene Datenloader, einen für Input- und einen für Output-Datensätze. Diese bekommen vom Watchdog den Pfad mit der Datei, in dem der jeweiligen Datensatz steht, im Funktionsaufruf übergeben. Anschließend wird die Datei als csv-Datei mit Trennzeichen ";" behandelt. Besitzt der Datensatz keine Seriennummer, so wird der Datensatz als "defekter" Datensatz gespeichert (siehe Entwurf "Rohdatensätze ohne Seriennummer" und "Liste aller Rohdatensätze ohne Seriennummer"). Besitzt er eine Seriennummer, so wird je nach Typ (Input/Output) unterschieden. In den folgenden Vorgehensweisen wird von ordentlichen Rohdatensätzen ausgegangen.

Input-Datensatz

Zuerst wird der Rohdatensatz als Hash mit entsprechenden Feld-Werte-Paaren abgespeichert. Besitzt er keine Seriennummer, so wird er als defekter Datensatz abgelegt. Anschließend wird der Schlüssel dieses Hash an die Liste der zugehörigen SNR angehängt, um den Datensatz für den Output verknüpfbar zu machen. Danach wird die Verknüpfung für diesen In-Datensatz als Liste angelegt, wobei als Zeitstempel 2-mal das Datum in Sekunden genommen wird und der Wert der Schlüssel des Rohdatensatzes ist. Diese Verknüpfung wird an die Liste aller Verknüpfungen angehangen. Nun wird jede vorgegebene Eigenschaft durchgegangen. Falls die Eigenschaft noch nicht oder noch nicht in dieser Ausprägung vorhanden ist, wird sie angelegt. Das 1-Bit wird in den entsprechenden Datensätzen an der entsprechenden Position angelegt. Der Zeitstempel des Rohdatensatz wird in die Zeitverknüpfung als erstes Element der Liste eingetragen und der Zähler (inCounter) inkrementiert.

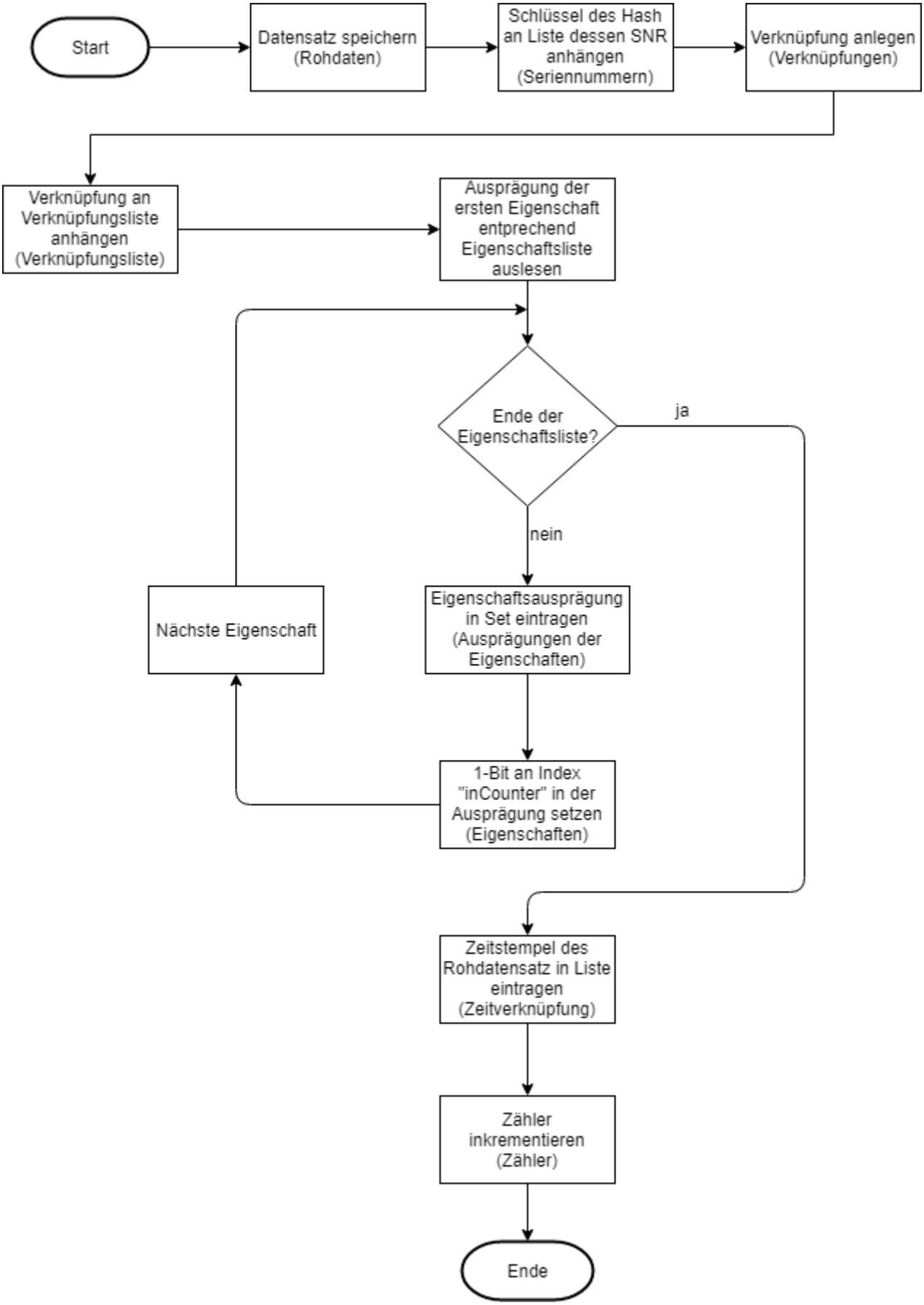

Abbildung 46. Einlesealgorithmus Input-Datensatz

Output-Datensatz

Der Rohdatensatz wird analog zu den Input-Datensätzen als Hash abgespeichert. Im nächsten Schritt werden alle Input-Datensätze aus der Liste der Seriennummer des Out-Datensatzes durchgegangen und

die Zeitdifferenzen des Input und Output berechnet. Der Output-Datensatz wird dem Input mit der kleinsten positiven Zeitdifferenz zugeordnet. Im Anschluss wird die Zeitverknüpfung dieses Input-Datensatzes als Liste geladen. Gibt es in dieser Liste nur 1 Element, so wurde dem Input noch kein Output zugeordnet und der Zeitstempel des Output-Datensatzes wird angehangen. Ist der Zeitstempel des Outputs größer als der des bisherig größten Outputs in der Zeitverknüpfung, so wird dieser Zeitstempel mit dem des neuen ersetzt. Entsprechend wird der Schlüssel der Verknüpfung mit diesem neuen Zeitstempel versehen. Anschließend wird der Schlüssel des Output-Rohdatensatzes als Element in die Verknüpfung aufgenommen und der Schlüssel der Verknüpfung in der Liste aller Verknüpfungen entsprechend mit dem aktuellen Schlüssel ersetzt.

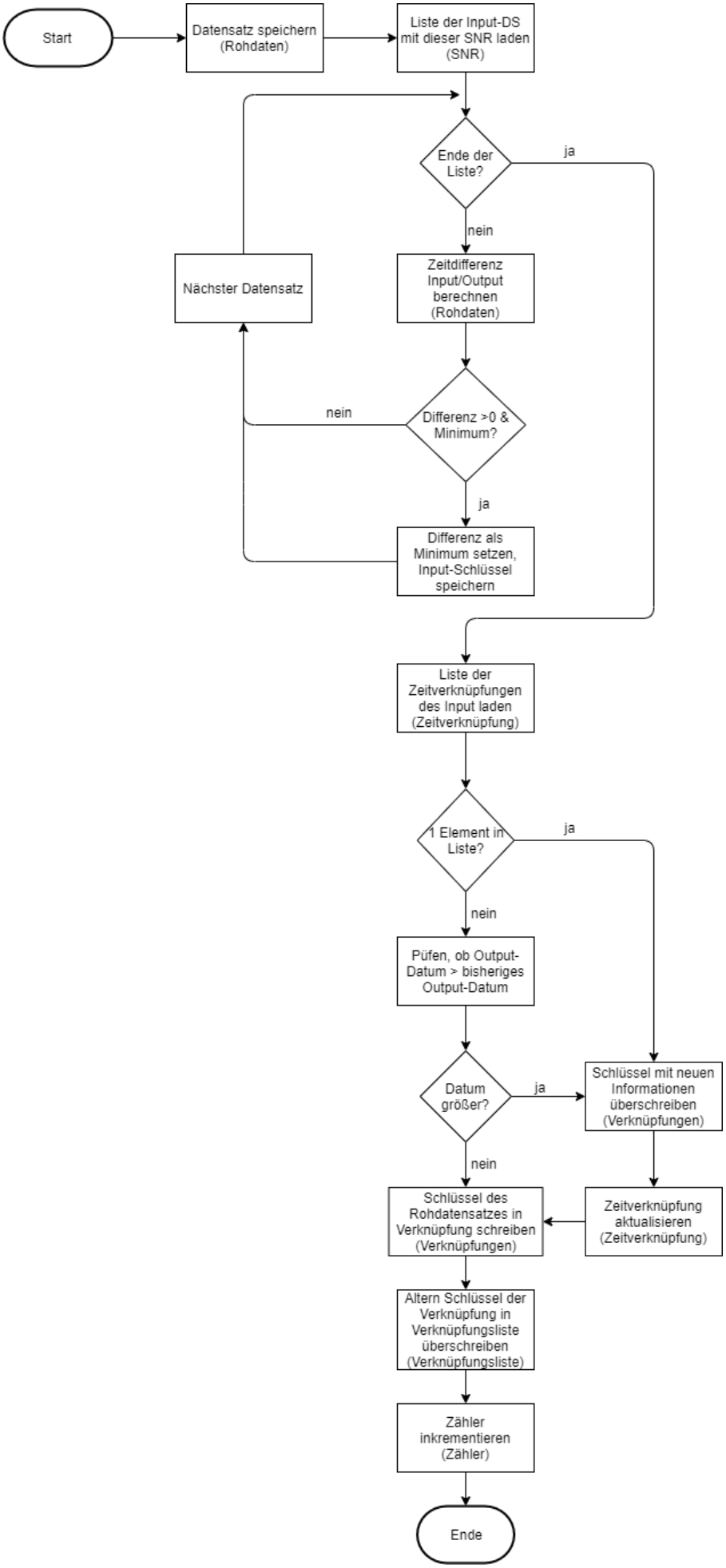

Abbildung 47. Einlesealgorithmus Output-Datensatz

Implementierung der Analysen

Vorgehensweise

Die Analysen folgen einem Grundgerüst, welches sich zwar in der Ausprägung, nicht aber in der Struktur unterscheidet. Am Anfang werden die Listen mit den benötigten Daten geladen. Diese sind zum einen die Liste mit allen Verknüpfungen, die Verknüpfungsliste, und die Sets der gefragten Eigenschaften mit den Ausprägungen als Elemente dieser.

Listing 69. Laden der benötigten Daten

```
import redis
r = redis.Redis(decode_responses=True)

allCons = r.lrange("con",0,-1)
allConLen = len(allCons)
allTeil = r.smembers("TEIL")
allFa = r.smembers("FA")
```

Danach wird die Ausgabedatei vorbereitet.

Listing 70. Ausgabedatei vorbereiten

```
writer = ""
diffFile = open("takt1.txt", "w")
diffFile.write("TEIL;FA;COUNT;MIN;MAX;AVG\n")
```

Anschließend wird ein Lua-Skript vorbereitet und in Redis eingebunden. Das Skript ist notwendig, da es in Redis keine Funktionalität dafür gibt die 1-Positionen einer Bitmap sinnvoll auszulesen. In unserem Fall brauchen wir die 1-Positionen der Bitmap "opCon", welche das Ergebnis der bitweisen AND Operation speichert. Redis ermöglicht es Lua-Skripte einzubinden und diese serverseitig auszuführen. Das Skript ist in den Quelltexten der Analysen aufgeführt, im folgenden Beispiel durch drei Punkte ersetzt.

Listing 71. Lua-Skript einbinden

```
lua = """ ... """
myLua = r.register_script(lua)
```

Nun erfolgt die eigentliche Analyse. Dabei wird immer mit den Bitmaps der Eigenschaften gearbeitet, da diese angeben welche Verknüpfungen diese Eigenschaft besitzen. Zur Veranschaulichung dient folgendes Beispiel:

- Gesucht: Alle Verknüpfungen mit den Eigenschaften "TEIL:B" und "FA:005830"
- ¬Bitmap "TEIL:B" ¬ [0101010]
- ¬Bitmap "FA:005830" ¬ [1100110]

Die 1-Bits einer Bitmap bedeuten, dass die Verknüpfung mit der Nummer "Index des 1-Bit" diese Eigenschaft besitzt (Erster Index einer Bitmap ist Index Nr. 1). In dem Beispiel besitzen also die Verknüpfungen 2,4 und 6 die Eigenschaft "TEIL:B" und Verknüpfungen 1,2,5,6 die Eigenschaft

"FA:005830". Um nun diejenigen Verknüpfungen zu finden, welche beide Eigenschaften besitzen, müssen nur beide Bitmaps bitweise AND verknüpft werden. Das Ergebnis dieser Operation ist wieder eine Bitmap, welche unter dem Schlüssel "opCon" gespeichert wird.

- ¬Bitmap "opCon" ¬ [0100010]

Es besitzen also Verknüpfung Nummer 2 und 6 beide Eigenschaften. Das eingebundene Lua-Skript gibt nun alle 1-Positionen von "opCon" als Liste zurück, in dem Fall also "(2,6)". Diese Liste kann nun auf die Liste aller Verknüpfungen angewendet werden, wodurch nur Verknüpfungen mit diesen bestimmten Eigenschaften betrachtet werden. Im Folgenden noch das Codebeispiel.

Listing 72. Bitmap Operationen

```
#Bitweise AND für dieses Teil und diesen Fertigungsauftrag
r.bitop("AND","opCon",teil,fa)
#Lua-Skript auf 'opCon' anwenden
#So viele Bits auf 1en prüfen wie die Liste aller Verknüpfungen lang ist
result = myLua(keys=['opCon'],args=[1,allConLen])
#Für jede 1-Position die passende Verknüpfung aus Liste holen
for res in result:
    con = allCons[res-1]
    #...
```

Am Ende jeder Analyse wird das Ergebnis in einer entsprechenden Ausgabedatei festgehalten.

Listing 73. Ausgabedatei schreiben

```
writer = teilSplit+";"+faSplit+";"+str(menge)+";"+str(minDiff)
            +";"+str(maxDiff)+";"+str(round(avgGesTime,2))+"\n"
diffFile.write(writer)
```

Darstellung der Analysen in Pseudo-Code

Da Redis aufgrund der Eigenschaften einer Schlüssel-Werte-Datenbank selbst keine Möglichkeit für komplexe Abfragen bietet, muss die Abfragelogik zum großen Teil in Python geschehen. Das führt dazu, dass der Quellcode für die Analysen recht lang ist. Daher ist im Folgenden nur Pseudo-Code dargestellt, welcher die wichtigsten Schritte jeder Analyse vorgibt.

Legende:

- FA: Fertigungsauftrag
- SNR: Seriennummer
- LA: Ladungsträger

Analyse 1:

```
FOR EACH Teil in alleTeile {
    FOR EACH FA in alleFA {
        Verknüpfungen ermitteln, welche dieses Teil und diesen FA haben

        IF Verknüpfung(en) exisieren {
            FOR EACH Verknüpfung {
                IF Verknüpfung besitzt Ouput {
                    Zeitdifferenz Input/Output berechnen
                    IF Differenz <= 1 Stunde {
                        Differenz auf Minimum/Maximum prüfen
                        Differenz auf Gesamtzeit addieren
                        Ausschuss für diese SNR inkrementieren
                    }
                }
            }

            Menge an SNR ermitteln
            Maximum, Minimum und Durchschnitt Ausschuss berechnen
            Maximum, Minimum und Durchschnitt Zeiten berechnen
            In Ausgabedatei schreiben
        }
    }
}
```

Analyse 2:

```
FOR EACH Teil in alleTeile {
    Verknüpfungen ermitteln, welche dieses Teil haben

    FOR EACH Verknüpfung {
        Verbinde die Input/Output Zeitstempel mit der jeweiligen SNR
    }

    FOR EACH SNR in SNR-Zeitstempel-Verbindungen {
        Ausschuss berechnen
        Verbindungen nach Input-Zeitstempel sortieren

        IF Anzahl Verbindung > 1 {
            FOR EACH Verbindung {
                Berechne Zeitdifferenz letzter Output bis aktueller Input
                Differenz auf Maximum / Minimum prüfen
                Differenz auf Gesamtzeit addieren
            }
        }
    }

    Durchschnitte berechnen
    In Ausgabedatei schreiben
}
```

Analyse 4:

```
FOR EACH LA in alleLA {
    Verknüpfungen ermitteln, welche diesen LA haben

    FOR EACH Verknüpfung {
        Zeitstempel auf Maximum / Minimum prüfen
    }

    Differenz von Maximum und Minimum berechnen
    In Ausgabedatei schreiben
}
```

Analyse 5:

```
FOR EACH Teil in alleTeile {
    FOR EACH LA in alleLA {
        Verknüpfungen ermitteln, welche dieses Teil und diesen FA haben

        IF Verknüpfung(en) existieren {
            FOR EACH Verknüpfung {
                SNR in Set aller SNR hinzufügen

                IF Verknüpfung besitzt Ouput {
                    Zeitdifferenz Input/Output berechnen
                    Differenz auf Gesamtzeit addieren
                    Differenz auf Minimum/Maximum prüfen
                }
            }

            Menge an SNR ermitteln
            Maximum, Minimum und Durchschnitt Zeiten berechnen
            In Ausgabedatei schreiben
        }
    }
}
```

Analyse 6:

```
FOR EACH Linie in alleLinien {
    FOR EACH FA in alleFA {
        Verknüpfungen ermitteln, welche diese Linie und diesen FA haben

        FOR EACH Verknüpfung {
            Zeitstempel auf Maximum / Minimum prüfen

            IF Maximum {
                FA Maximum zuordnen
            }
            IF Minimum {
                FA Minimum zuordnen
            }
        }
    }

    FA-Zeit-Verbindungen nach Zeitstempel des Input aufsteigend sortieren

    FOR EACH FA-Zeit-Verbindung {
        Zeitdifferenz letzter Input / aktueller Input berechnen
        Differenz auf Maximum / Minimum für diese Teilkombination prüfen
    }

    In Ausgabedatei schreiben
}
```

Messung der Performance

Für die Performancemessung kommen 2 Kennzahlen zum Einsatz, zum einen die Zeitdauer der Ausführung des Python-Skripts und zum anderen die Zeitdauer der Ausführungen der Operationen in Redis.

Für Python wurde die Zeit in Nanosekunden mithilfe der Funktion process_time_ns() des Moduls time gemessen. Diese misst nur die reine Prozesszeit des Programms. Der Timer startet direkt nach dem Einbinden der Bibliotheken und endet nach der letzten für die Analyse relevanten Operation.

Listing 74. Messung der Prozesszeit des Python Programms

```
from time import process_time_ns
start = process_time_ns()
#...
stop = process_time_ns()
print(str((stop-start)/10**9))
```

Die Ausführungszeiten der Redis Operationen wurden mit der built-in Funktionalität SLOWLOG gemessen. Dieses System ermöglicht es, die Ausführungszeiten von Operationen zu messen welche eine bestimmte Dauer überschreiten. Um alle Operationen zu messen wurde die Schranke auf 0 Mikrosekunden gesetzt ("slowlog-log-slower-than 0"). Da sich in der Praxis allerdings gezeigt hat, dass dieser Log für eine große Anzahl an Operationen nicht zuverlässig funktioniert (ab mehreren tausend

Einträgen wurden alle Einträge gelöscht), wurden die Operationszeiten für jeden Durchlauf (z.B. ein Input-Datensatz wird vollständig eingelesen) gemessen und der SLOWLOG im Anschluss mittels "SLOWLOG RESET" bereinigt. Im Folgenden ist ein Beispiel für die Messung der Operationszeit beim einlesen eines Input-Datensatzes aufgeführt. Dabei ist anzumerken, dass "SLOWLOG RESET" und "SLOWLOG LEN" (Anzahl Einträge im Log) selbst gemessen werden und damit das Ergebnis verfälschen würden, weshalb diese in der if-Anweisung ausgeklammert werden.

Listing 75. Messung der Prozesszeit von Redis

```
timeLog = r.slowlog_get(r.slowlog_len())
for time in timeLog:
    if time['command'] != 'SLOWLOG RESET' and time['command'] != 'SLOWLOG LEN':
        timeFile.write(str(time['command'])+str(time['duration'])+'\n')
r.slowlog_reset()
```

6. Lösungsvergleich

6.1. Übersicht

Um die Lösungen zu evaluieren, wurden verschiedene Kriterien genutzt, diese sind in der Tabelle zu finden. Dabei wurden die Abstufungen sehr positiv (++) bis sehr negativ (--) genutzt. Es wurde keine absolute Skale verwendet, sondern jeglich die Lösungen untereinander verglichen. Natürlich ist dieser Vergleich subjektiv, allein schon von der Auswahl der Vergleichskriterien.

Das erste Kriterium vergleicht den Programmieraufwand der einzelnen Lösungen. Dabei wird die Komplexität und Größe der Abfragen sowie der Verwaltungsaufwand der Datenbank berücksichtigt. Das nächste Kriterium handelt von dem Aufwand das Datenmodell zu erstellen und dessen Verständlichkeit. Ein weiteres Kriterium für den Vergleich war der Entwicklungsaufwand für das Hinzufügen eines neuen Attributes, also wenn eine neue Werteart gespeichert werden muss. Die Schreib-Performance umfasst die benötigte Zeit, um neue Datensätze in die Datenbank zu bringen, die dazugehörigen Grafiken sind unter Einleseperformance zu finden. Die Lese-Performance beschäftigt sich im Besonderen mit der Dauer der Analysen, die Diagramme sind unter der Analyseperformance zu finden. Für das nächste Kriterium wurde der von der Datenbank genutzte Speicher gemessen, damit auch der benötigte Speicherbedarf für Indices. Im letzten Kriterium wurde das Vorhandensein von Entwicklerdokumentation im Zusammenhang mit einer aktiven Nutzergemeinschaft verglichen.

Tabelle 3. Lösungsvergleich

Kriterium/Datenbank	Proprietäre Datenstruktur	Universelle relationale Struktur	Schlüssel-Werte-Datenbank	Dokumentenorientierte Datenbanklösung
geringer Entwicklungs-aufwand	++	- -	- -	+
geringe Komplexität des Datenmodells	++	-	-	++
Hinzufügen eines Attribut	- -	++	++	++
Schreib-Performance	-	- -	+	++
Lese-Performance	++	- -	+	+
geringer Speicher-aufwand	++	- -	+	++
Entwickler Dokumentation	++	++	-	+
kumuliertes Ergebnis	7	-3	1	11

6.2. Performance

6.2.1. Einleseperformance

Die Diagramme zeigen die Speicherdauer für einen Datensatz in Millisekunden. Dabei war die Einlesezeit für die Input Datensätze der Dokumentendatenbank nicht messbar, da diese kleiner als eine Millisekunde waren und Mongo DB kein Profiling in diesem bereich unterstützt.

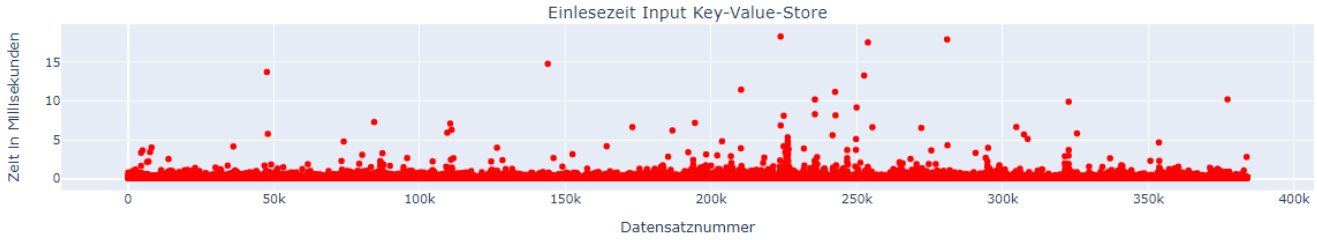

Abbildung 48. Einlesezeit Input

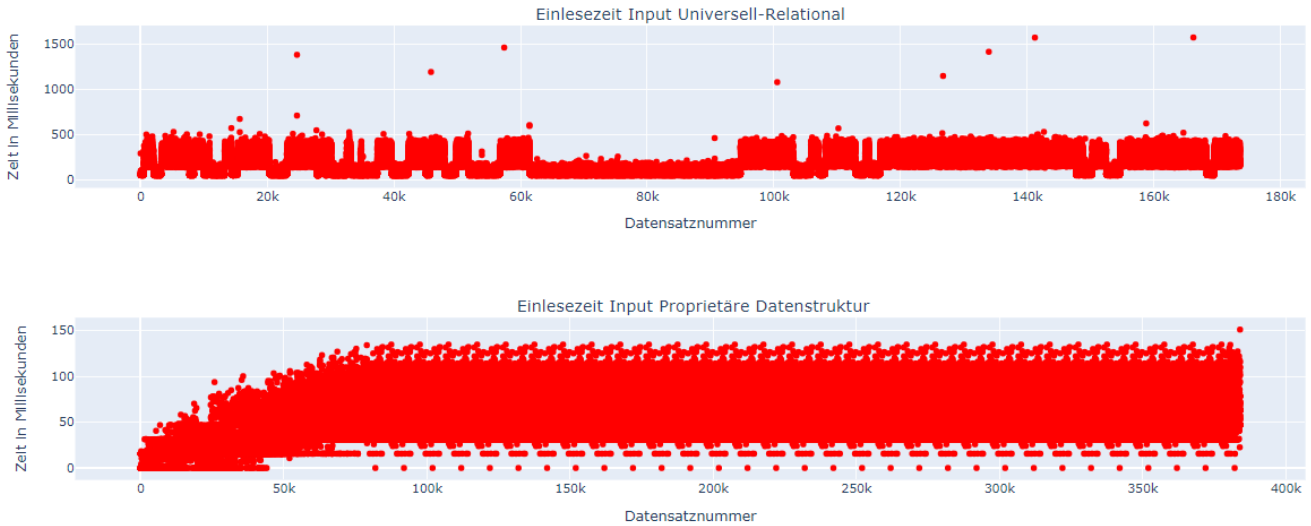

Abbildung 49. Einlesezeit Input

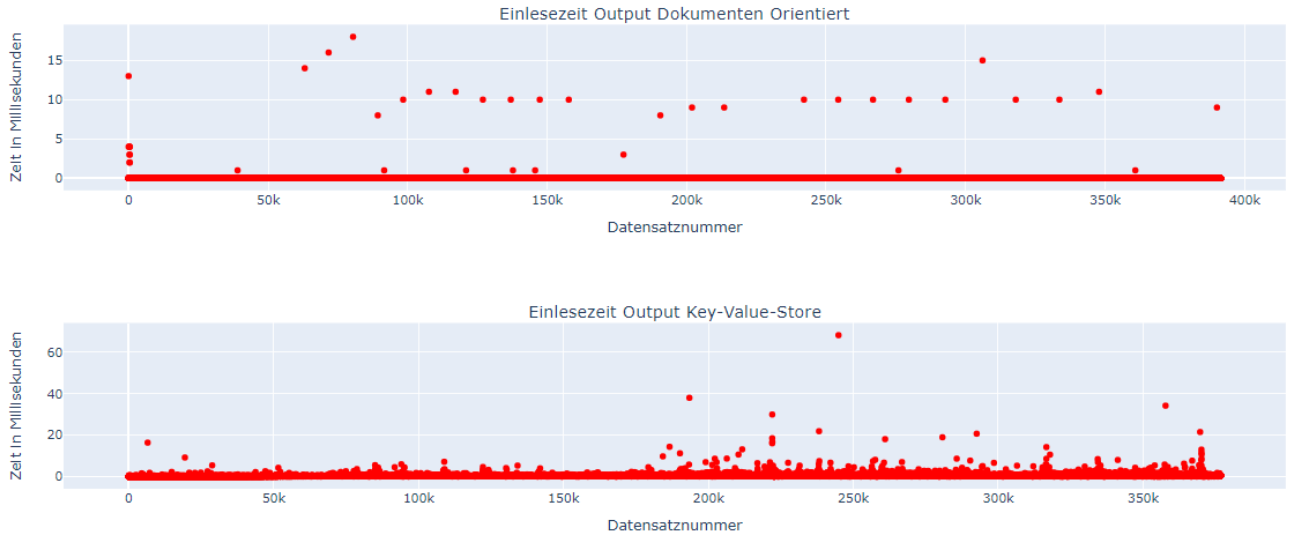

Abbildung 50. Einlesezeit Output

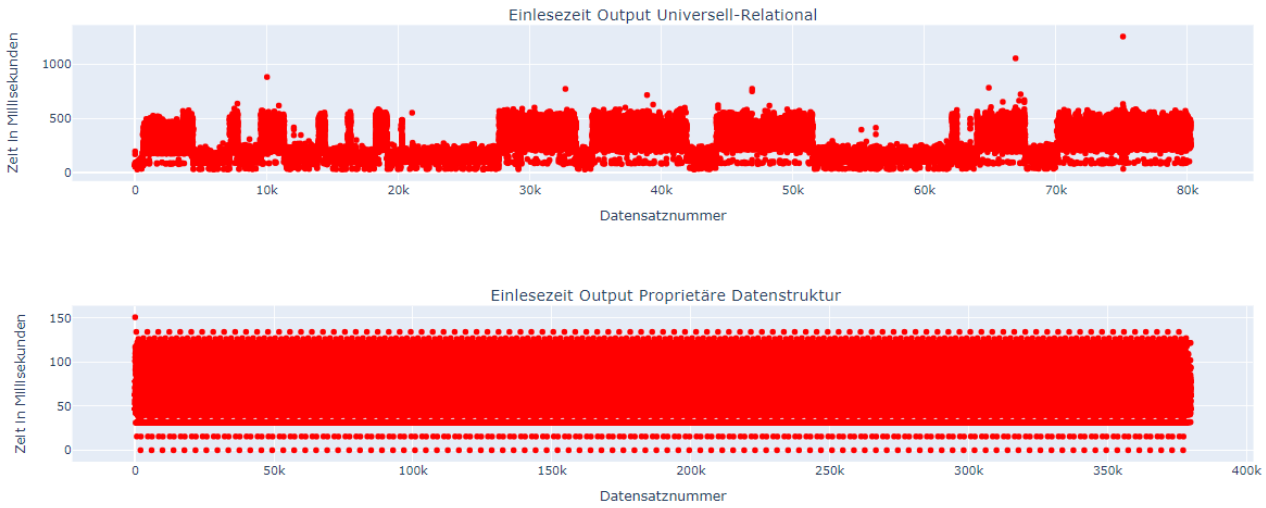

Abbildung 51. Einlesezeit Output

6.2.2. Analysenperformance

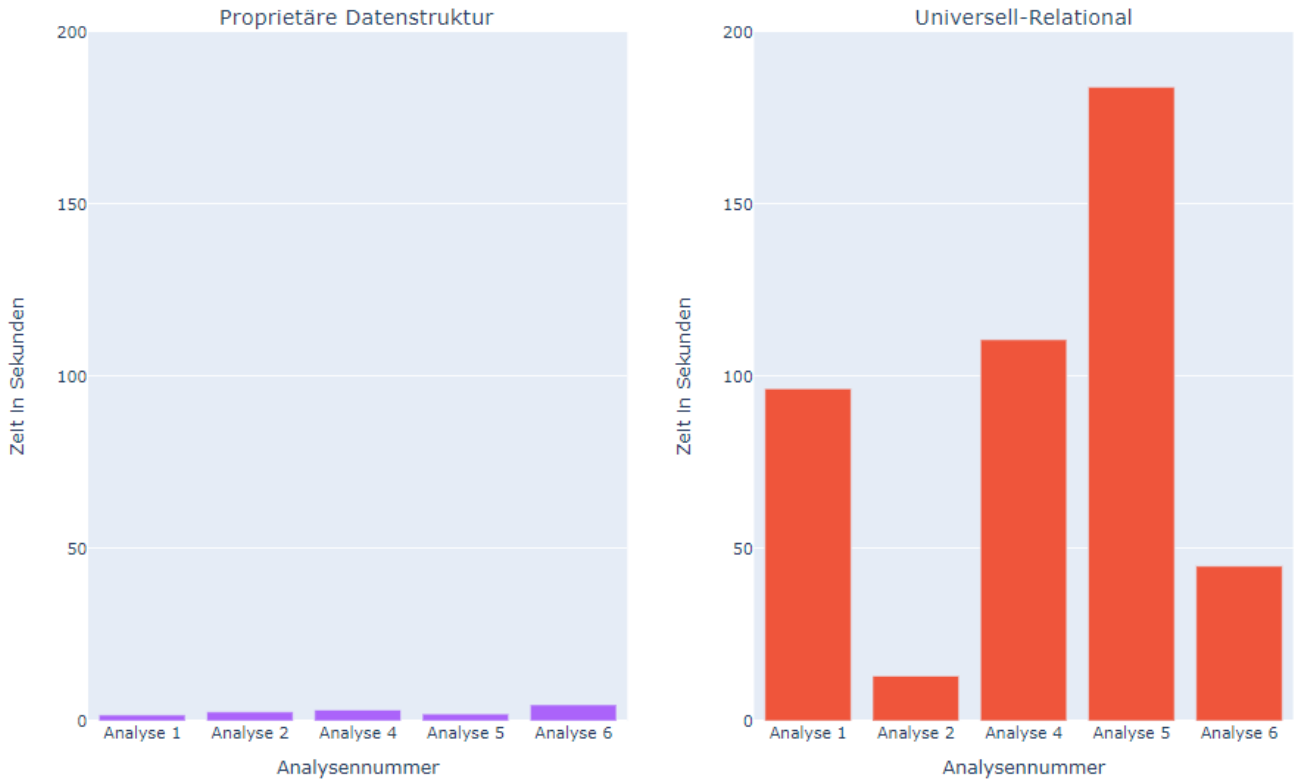

Abbildung 52. Analysedauer in den SQL basierten Lösungen

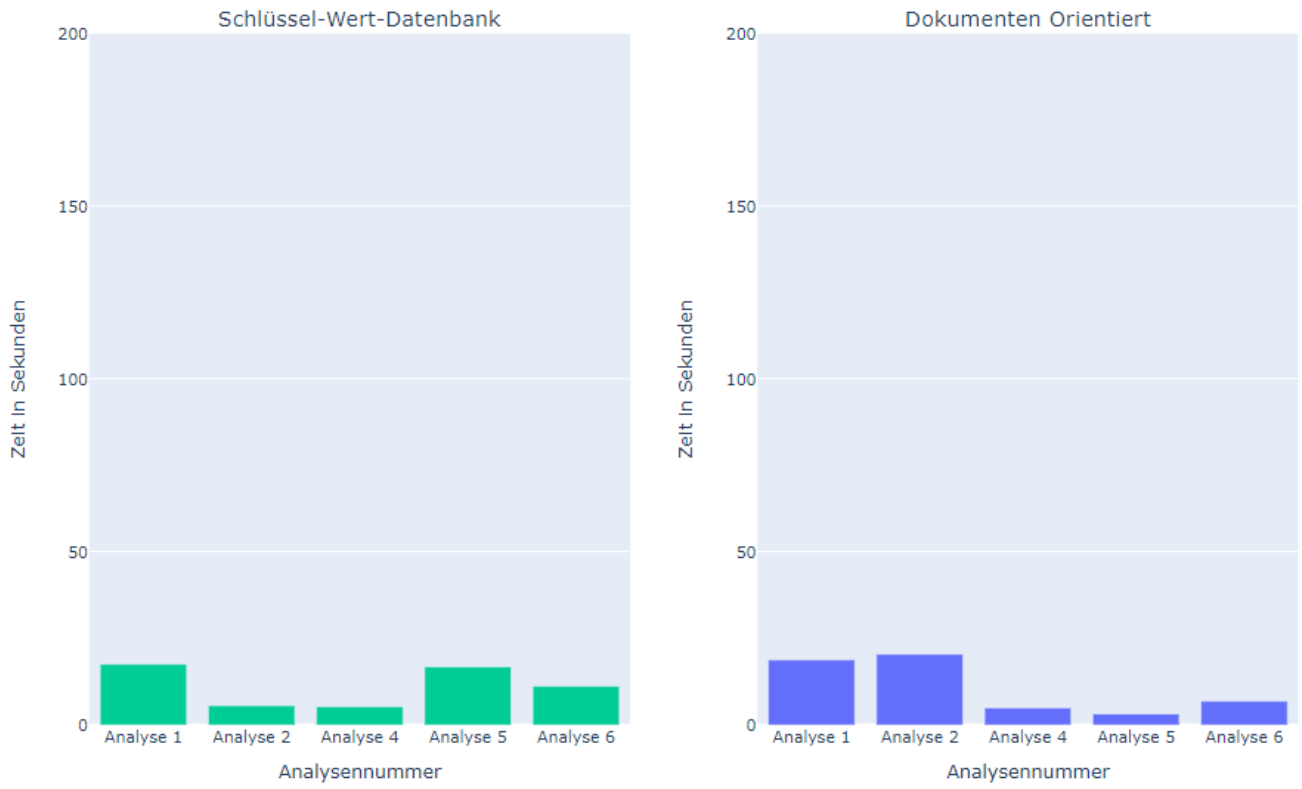

Abbildung 53. Analysedauer in den NOSQL Datenbanken

6.3. Vor- und Nachteile der einzelnen Lösungen

6.3.1. Proprietäre Datenstruktur

Der wohl klassischste Lösungsansatz besticht im Besonderen mit einer hohen Abfrageperformance, diese können auch mit einfach bedienbaren Werkzeugen wie Excel erstellt werden. Die Speicherperformance könnte im Besonderen bei komplexeren Datenmodellen mit verschiedenen Foreign-Key-Constraints problematisch werden, da diese Lösung in der Schreibperformance nur Platz drei belegt. Auch kann der Entwicklungsaufwand verbunden mit Änderungen im Sensorbestand ein Ausschlusskriterium für manche Anwendungsfälle sein.

6.3.2. Universelle relationale Struktur

Das größte Problem mit der vorangegangenen Lösung wird mit dem universellen Ansatz behoben, denn mit Änderungen im Messtechnikbestand hat dieses Modell kein Problem. Dabei bleibt aber der Vorteil bestehen, dass SQL basierte Lösungen auf ein potentes Umfeld aus Datenbankservern und Dokumentationen zurückgreifen können. Die Flexibilität hat dabei aber leider den Preis eines sehr komplexen Datenmodells, was auch zu der mit Abstand schlechtesten Lese- und Schreibperformance führt.

6.3.3. Schlüssel-Werte-Datenbank

Diese Performanzprobleme hat der dritte Lösungsansatz nicht, besonders bei Einzelabfragen hat diese Lösung ihre Stärke. Um aber Datenzusammenhänge darzustellen, ist einiges an Aufwand nötig, der für Entwickler aus dem SQL Umfeld zunächst irritierend wirken kann. Weiterhin sind Operationen wie Joins und Gruppierungen nicht ohne weitere Implementierung möglich, was Abfragen komplizierter macht.

6.3.4. Dokumentenorientierte Datenbanklösung

Diese Abfragekomplexität wird bei der dokumentenorientierten Datenbanklösung, durch ein SQL ähnliches Abfrageraster entgegengewirkt. Da dieses Raster erst bei der Abfrage über die Datenbank gelegt wird, hat die Datenbank auch keine Performanzprobleme und eignet sich somit auch für den Einsatz in einem Fast Layer. Damit hat diese Lösung eine sehr ausgeglichene Leistung über all unsere Kritieren, deshalb hat sie auch das größte Lösungspotential der besprochenen Lösungen.

7. Zusammenfassung und Ausblick

7.1. Lessons Learned

Die wichtigste Entscheidung ist welche Datenzusammenhänge wie gespeichert werden. Wenn diese Zusammenhänge durch die Datenbankstruktur vorgegeben werden sind Abfragen einfach und performant. Wenn diese Beziehungen aber nicht gespeichert werden, dann müssen diese Informationen durch die Abfragen wiederhergestellt werden. So sind Abfragen in der proprietären Struktur mit einfachen SQL-Abfragen möglich, während bei der universellen Struktur Abfrageergebnisse zum Teil zwischengespeichert werden müssen, um die richtige Analyse zu ermöglichen. Das sollte man berücksichtigen, im Besonderen, da Analysen besonders wertvoll sind, wenn diese durch Mitarbeiter der Fachdomäne erstellt werden können. Mit der Datenstruktur ist natürlich die Abfragestruktur eng verbunden, auch hier hat man eine Wahl Möglichkeit. Man kann entweder große komplexe Abfragen nutzen oder mehrere kleine. Damit kann die Logik der Abfragen zwischen Datenbankserver und Client verschoben werden. Die Datenbank nahe Lösung ist meist performanter, aber auch schwerer verständlich oder auch nicht ohne weiteres realisierbar.

Weiterhin hat gerade die universell relationale Struktur gezeigt, dass Laufzeit und Speicher sich gegenseitig ersetzten können. So hat das Setzen von Indices die Performance erhöht, aber auch die Datenbank um einiges vergrößert.

7.2. Fazit

Das Projekt hat gezeigt, dass es bei der Datenbank und Datenmodellauswahl wie bei den meisten Architekturentscheidungen keine allgemeingültige Antwort auf die Frage "Was ist die beste Lösung?" gibt. Dafür ist die Variabilität der Anforderungen zwischen den verschiedenen Anwendungsgebieten zu hoch. Man muss somit von Projekt zu Projekt unterscheiden, was die am besten umsetzbare Lösung ist. In realen Projekten kommen natürlich auch neben technischen Anforderungen, einschränkende Faktoren wie Kosten, bestehende Datenbankserver und Fähigkeiten des Entwicklerteams hinzu.

8. Quellen

Laurenz Wuttke (29 Apr. 2020) Hadoop Einfach Erklärt: Was Ist Hadoop? Was Kann Hadoop?, URL: https://datasolut.com/apache-hadoop-einfuehrung/#Unterschied-zwischen-Hadoop-relationalen-Datenbank [Letzter Besuch: 5 Feb. 2021].
Übersicht Über AWS IoT Analytics Amazon Web Services, URL: https://aws.amazon.com/de/iot-analytics/?c=i&sec=srv [Letzter Besuch: 8 Feb. 2021].
Kudraß, Thomas (2015): Taschenbuch Datenbanken, München, Deutschland: Carl Hanser Verlag.