Thesis

Stream-Processing mit Rust
zur Analyse von Luftqualitätsdaten

Referent	:	Maxim Balsacq
Vorgelegt am	:	19.08.2021
Vorgelegt von	:	Maxim Balsacq

Abstract

The lockdowns during the COVID-19 pandemic indirectly affected the air quality worldwide. In this work, the improvements of air quality in different cities in Germany are calculated. Using stream processing, measurement data of the year 2020 is analyzed and compared to data from the year 2019. To be able to quickly process the data, various stream processing algorithms as well as geospatial algorithms are used.

Die während der COVID-19-Pandemie eingeführten Lockdowns hatten indirekt messbare Auswirkungen auf die Luftqualität. In dieser Masterarbeit wird nach den Orten in Deutschland gesucht, in denen sich die Luftqualität am stärksten verbessert hat. Dazu werden mithilfe von Stream Processing Sensordaten aus dem Jahr 2020 ausgewertet und mit den Daten von 2019 verglichen. Um die große Menge an Sensordaten schnell zu verarbeiten, kommen hierbei Stream-Processing-Algorithmen sowie Algorithmen aus der Geoinformatik zum Einsatz.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Die DEBS Grand Challenge

Die DEBS Grand Challenge ist ein Wettbewerb, welcher im Rahmen der Distributed and Event-based Systems (DEBS) Conference stattfindet. Hierbei werden eine oder mehrere Aufgaben gestellt, die möglichst effizient gelöst werden sollen. Autoren von Lösungen können eine wissenschaftliche Arbeit einreichen, welche ihre Lösung erklärt, und diese auf der Konferenz präsentieren.

Diese Master-Thesis basiert auf der Query 1 der DEBS Grand Challenge 2021. Bei dieser ging es darum, Luftqualitätsdaten zu analysieren, um die Verbesserung der Luftqualität zu ermitteln.

1.1.1 Aspekte der Grand Challenge

Verarbeitung von Geodaten

Um die Luftqualität zu berechnen, werden Messungen bereitgestellt. Sie enthalten Werte für Partikelkonzentrationen, aus denen die Luftqualität berechnet werden kann.

Der Ort, an dem die Messung durchgeführt wurde, ist jedoch nicht in der Messung angegeben. Die bereitgestellten Messdaten beinhalten stattdessen den Längen- und Breitengrad, an dem die Messung stattgefunden hat. Um die Luftqualität für die verschiedenen Orte zu berechnen, muss daher für jede Messung zunächst analysiert werden, zu welchem Ort die Messung gehört (s. Kapitel 3.3.2).

Da diese Zuordnung für jede Messung durchgeführt werden muss, ist es erforderlich, diese Lokalisierung stark zu optimieren. Hierfür werden verschiedene Algorithmen aus der Geoinformatik eingesetzt (s. Kapitel 4.1).

Berechnung der Luftqualität

Um die Luftqualität an den verschiedenen Orten zu ermitteln, wird zunächst eine Möglichkeit benötigt, die Luftqualität aus den Messdaten zu errechnen. Hierfür verwenden die Veranstalter der Challenge den sogenannten Air Quality Index (AQI).

Der AQI wurde durch die amerikanische Environmental Protection Agency (EPA) definiert [1]. Zur Berechnung der Luftqualität wird die Konzentration von verschiedenen Partikeln in der Luft genutzt. Aus diesen wird ein Wert berechnet, der AQI. Dieser ermöglicht Aussagen über die Luftqualität zu treffen.

Jede Messung im Datensatz der DEBS Grand Challenge enthält zwei verschiedene Werte, p1 und p2. Diese entsprechen in der Terminologie der EPA jeweils den Werten PM_{10} und $PM_{2.5}$. Die Zahlen der EPA beziehen sich auf die Partikelgröße in µm. Somit beschreibt PM_{10} (in der Challenge p1 genannt) die Anzahl der Partikel mit einer Größe < 10µm. Entsprechend beschreibt p2 die Anzahl Partikel < 2.5µm.

Aus diesen Werten wird mithilfe einer Formel jeweils die Luftqualität berechnet und die schlechtere Luftqualität (der höhere AQI-Wert) als finaler Wert übernommen. Um zu verhindern, dass einzelne Messungen den AQI stark beeinflussen, wird ein Mittelwert über einen festen Zeitraum gebildet. Die EPA schreibt hierfür ein Fenster von 24 Stunden vor [1].

Auch in der Challenge wird der Mittelwert der p1-/p2-Werte über einen längeren Zeitraum gebildet: Zum Vergleich der Luftqualität werden zwei Fünf-Tages-Fenster genutzt. Für die 50 Orte, für die beim Vergleich zwischen diesen Fenstern die stärkste Verbesserung der Luftqualität ermittelt wird, wird zusätzlich der AQI innerhalb der letzten 24 Stunden berechnet. Warum die Organisatoren der Challenge ausgerechnet ein Fünf-Tage-Fenster gewählt haben, ist aus den veröffentlichten Informationen nicht ersichtlich.

Scalability

Bei der DEBS Grand Challenge steht neben der Korrektheit der Lösung auch deren Skalierbarkeit im Fokus [2]. Damit ist gemeint, wie sich die Performance verbessern lässt, wenn dem Computer weitere Ressourcen hinzugefügt werden. Dies lässt sich auf zwei Arten erreichen: Einerseits kann die Hardwareausstattung (Speicher, CPU) des Computers verbessert werden. Somit kann das Programm mehr auf dem selben Computer erreichen (vertical scalability). Alternativ können weitere Computer hinzugefügt werden, welche am selben Problem arbeiten. In diesem Fall wird von "horizontal scalability" gesprochen. Die Veranstalter der DEBS Grand Challenge geben an, bei Lösungen horizontal scalability zu bevorzugen [3].

Das Problem hierbei ist, dass Lösungen, welche gut horizontal skalieren, oft einen großen Overhead besitzen. In ihrer Arbeit "Scalability! But at what COST?" zeigen Frank McSherry et al. auf, dass der Overhead von Algorithmen für verteilte Systeme teilweise so groß ist, dass die verteilten Algorithmen erst ab mehreren hundert Cores die Performance eines einzelnen Cores mit einem einfacheren Algorithmus übertreffen [4].

In dieser Arbeit wurde daher wenig Wert auf horizontal scalability gelegt, sondern vielmehr versucht möglichst effizient alle Ressourcen eines Computers auszuschöpfen.

1.1.2 Verwandte Arbeiten

Die DEBS Conference fand von 28. Juni bis zum 2. Juli 2021 statt. Dabei wurden vier Arbeiten veröffentlicht, welche sich mit der DEBS Grand Challenge 2021 beschäftigen. Eine der Arbeiten wurde von den Veranstaltern geschrieben und beschreibt die Organisation des Events [2]. Die anderen drei Arbeiten beschreiben die implementierten Lösungen [5, 6, 7]. Im Kapitel 6.1 wird die in dieser Arbeit präsentierte Lösung mit zwei bei der Konferenz eingereichten Arbeiten verglichen.

1.2 Grundlagen

1.2.1 Rust

Rust ist eine noch junge Programmiersprache. Version 1.0 wurde erst 2015 veröffentlicht. Rust ist, wie C und C++, eine kompilierte Programmiersprache ohne Runtime, sodass Code ohne Overhead direkt auf der CPU ausgeführt werden kann. Anders als C und C++ legt Rust einen starken Fokus auf Sicherheit. Hierbei hat Rust aus den Fehlern von C/C++ gelernt und verhindert bereits zur Kompilierzeit ungültige Speicherzugriffe. Nur wenn Programmierer Rusts Kontrollen mit `unsafe` bewusst umgehen, kann es zu solchen Problemen kommen [8].

Ein Seiteneffekt von Rusts Programmiermodell ist, dass genau bekannt ist, auf welche Daten ohne Risiko von Multithreading-Problemen zugegriffen werden kann. Somit ist es einfach, Programme zu parallelisieren, ohne dass dabei unerwartete Fehler auftreten können. Dies wird in Rust oft als "fearless concurrency" bezeichnet [9, 10].

Ownership und Borrowing

Da Rust keine Runtime besitzt, muss die Speicherverwaltung durch den Programmierer durchgeführt werden. Um dabei Rusts Sicherheitsgarantien aufrecht erhalten zu können, existieren einige Regeln, welche vom Compiler durchgesetzt werden. Rust führt deshalb die Konzepte von Ownership und Borrowing ein.

Ownership bedeutet, dass immer nur ein "Besitzer" (owner) für jeden Wert existiert. Genauer gesagt:

- Jeder Wert in Rust gehört zu einer Variable, welche als "owner" bezeichnet wird.
- Es gibt immer nur einen owner.
- Wenn der owner den Gültigkeitsbereich (Scope) verlässt, wird der Speicher freigegeben ("dropped") [9].

Dies ermöglicht es genau nachzuvollziehen, wann der Speicher für nicht länger benötigte Variablen freigegeben werden kann. Somit wird anders als bei z.B. Java keine

Runtime benötigt die prüft, wann nicht mehr benötigter Speicher freigegeben werden kann. Auf der anderen Seite kann der Compiler anders als in C/C++ genau beurteilen, wann Speicher freigegeben werden kann, sodass Speicherlecks praktisch nicht möglich sind [1].

Es ist jedoch nicht immer möglich oder sinnvoll, ownership an eine andere Funktion zu übertragen. Borrowing ("ausleihen") dient dazu, Zugriffe auf Werte zu ermöglichen, welche man gerade nicht besitzt. Hierbei wird zwischen lesendem und schreibendem Zugriff unterschieden. Dadurch wird sichergestellt, dass entweder ein einziger schreibender Zugriff oder beliebig viele lesende Zugriffe auf denselben Wert erfolgen können.

Ein Teil des Compilers, der sogenannte "borrow checker", überprüft, ob Referenzen länger verwendet werden als erlaubt. Dies passiert z.B. dann, wenn mehrere schreibende (`&mut`) Referenzen für denselben Wert verwendet werden sollen oder der owner eines Wertes nicht so lange gültig ist wie die Referenz. In solchen Fällen löst der borrow checker einen Kompilierfehler aus und erklärt, wieso die Referenzen nicht wie gewünscht verwendet werden können. Somit werden durch den borrow checker ungültige Zugriffe verhindert. Dies führt dazu, dass Probleme wie Iteratoren-Invalidierung oder Zugriffe auf freigegebenen Speicher in Rust unmöglich gemacht werden, da sie durch Rusts Regeln ausgeschlossen werden.

Multithreading

Ownership und borrowing ermöglichen eine sichere Speicherverwaltung ohne die Nutzung einer Runtime. Diese Konzepte lassen sich aber auch einsetzen, um automatisch fehlerhaften Multithreading-Code zu erkennen.

Rust definiert zwei Traits[2], Send und Sync. Mit diesen kann effizient kontrolliert werden, ob gleichzeitig Zugriffe von mehreren Threads vorkommen können. Send und Sync werden jeweils vom Compiler automatisch implementiert, sofern alle in einer Datenstruktur enthaltenen Variablen jeweils Send und Sync sind. Nicht threadsichere structs deaktivieren die Implementierung dieser Traits. Dies führt dazu, dass zur Compilezeit festgestellt werden kann, welche Datenstrukturen sicher mit anderen Threads geteilt werden und welche nicht.

Mithilfe von ownership und borrowing sowie den Send- und Sync-Traits können sogenannte "data races", also lesende und schreibende Zugriffe zur selben Zeit auf die gleiche Variable, fast vollständig ausgeschlossen werden [3]. Data races sind problematisch, da bei diesen Zugriffen auf eine Variable nicht klar definiert ist, was das

[1] Es gibt zwar Möglichkeiten, Speicherlecks zu verursachen, diese haben dann aber keine Auswirkungen auf Rusts Sicherheitsgarantien. Eine längere Erklärung hierzu würde jedoch den Rahmen dieser Einführung sprengen.

[2] Rusts Version von dem, was in anderen Programmiersprachen als Interface bezeichnet wird.

[3] Durch inkorrekt verwendete Atomare Variablen oder fehlerhaften `unsafe`-Code können data races auftreten.

Ergebnis dieses lesenden Zugriffs ist. Da der Rust-Compiler annimmt, dass nur eine schreibende Zugriffsmöglichkeit existiert und basierend auf dieser Annahme Optimierungen durchführt (s. der folgende Abschnitt "Ermöglichte Optimierungen"), würden data races zu undefiniertem Verhalten führen.

Neben data races existieren noch weitere Arten von Multithreading-Problemen (z.B. Deadlocks), welche Rust aber nicht erkennen kann. Dennoch können zumindest data races in der Praxis zuverlässig erkannt und verhindert werden [10].

Wenn data races in Rust automatisch erkannt werden, existieren verschiedene Möglichkeiten, den Fehler zu beheben. In den meisten Fällen kann eine Sperre (engl. Lock) eingesetzt werden, um sicher zu gehen, dass nur ein Thread auf einmal die Daten modifizieren kann.

Ermöglichte Optimierungen

Ein weiterer Vorteil von Rusts Konzepten besteht darin, dass der Compiler erkennen kann, an welchen Stellen im Programm Variablen modifizert werden können. Anders als in C oder C++ kann der Compiler hier weitere Optimierungen vornehmen, da bekannt ist, dass keine lesenden und schreibenden Referenzen auf dieselbe Variable zur gleichen Zeit existieren dürfen.

Hierzu ein Beispiel aus Rusts offizieller Dokumentation [4]: Wie in Listing 1 zu erkennen, nimmt die Funktion `compute` zwei Parameter: Den Eingabe-Parameter `input` und den Ausgabe-Parameter `output`. Falls `input` größer als 10 ist, wird `output` auf 1 gesetzt. Anschließend wird geprüft, ob `input` größer als 5 ist. Falls ja, wird `output` verdoppelt. Falls keine der Bedingungen zutrifft, bleibt `output` unverändert.

In Rust kann der Compiler die Funktion zu `compute2` umschreiben, da sichergestellt ist, dass `input` und `output` unterschiedliche Werte referenzieren. Der Compiler kann ausschließen, dass `input` und `output` auf den selben Wert verweisen, da immer gilt: Es kann immer nur eine schreibende Referenz (`&mut`-Referenz) oder beliebig viele lesende Referenzen (`&`-Referenzen) auf denselben Wert geben. Diese Regeln werden vom Borrow Checker geprüft (s. voriger Abschnitt). Da die Funktion `demo` diese Regeln bricht, kommt es hier zu einem Compile-Fehler.

Wäre nicht gesichert, dass die einzelnen Parameter unterschiedliche Variabeln referenzieren, so wäre die Transformation von `compute` zu `compute2` fehlerhaft: Wenn `compute2` mit Parametern aufgerufen werden würde, welche die gleiche Variable referenzieren, so würde bei einer Änderung von `output` gleichzeitig `input` geändert. Im Gegensatz zu `compute` prüft `compute2` aber nur einmal, welchen Wert `input` besitzt. Die Funktionen würden daher ein unterschiedliches Verhalten aufweisen.

[4] `https://doc.rust-lang.org/nomicon/aliasing.html`

Durch diese Prüfung des Borrow Checkers kann der Rust-Compiler also sicher effizienteren Code erzeugen. Da andere Programmiersprachen wie z.B. C oder C++ diese Prüfungen nicht vornehmen können, bleiben ihnen diese Optimierungen verwehrt.

```
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
}

fn compute2(input: &u32, output: &mut u32) {
    let cached_input = *input; // keep *input in a register
    if cached_input > 10 {
        // x > 10 implies x > 5, so double and exit immediately
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}

fn demo() {
    let mut input = 20;
    // Solch eine Nutzung (aliasing) wird
    // von Rust-Compiler verhindert.
    // Falls diese Prüfung (wie in anderen Programmiersprachen)
    // nicht existieren würde, so würde
    compute(&input, &mut input);
    // 1 in input schreiben und

    input = 20;
    compute2(&input, &mut input);
    // würde 2 in input schreiben
}
```

Listing 1: Demonstration einer möglichen Optimierung durch den Rust-Compiler

Zero-Cost abstractions

Ein weiterer Punkt, wieso Rust äußerst schnell ist, sind dessen "zero-cost abstractions". Damit ist gemeint, dass im Quellcode beliebige Abstraktionen genutzt werden können, der Compiler aber trotzdem Code generiert, wie wenn man selbst Low-Level Code geschrieben hätte [9].

Ein Beispiel hierfür sind die Iteratoren, die im Code verwendet werden. Der Compiler kann ein Iterieren mithilfe eines Iterators in eine einfache Schleife umwandeln. Dies gilt auch dann, wenn durch Aufruf verschiedener Methoden auf den Iteratoren neue Iteratoren mit unterschiedlichen Datentypen erzeugt werden.

Zero-cost abstractions erlauben es somit, Abstraktionen zu erzeugen, die den Code les- und wartbarer machen, während die Performance des Codes zur Laufzeit nicht negativ beeinflusst wird.

Dies ist damit vergleichbar, wie Bjarne Stroustrup, Entwickler der Programmiersprache C++, das "zero-overhead principle" von C++ definiert:

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better [9].

In dieser Arbeit wurde stark von zero-cost abstractions profitiert. Insbesondere Iteratoren wurden ausgiebig genutzt. Das Arbeiten mit Iteratoren machte es möglich, mit wenig Aufwand den Code parallelisieren zu können.

Verwaltung von Abhängigkeiten

Neben der Sprache selbst ist es in Rust einfach, andere Bibliotheken zu verwenden. Dies liegt vor allem daran, dass das Buildsystem, welches den Quellcode verarbeitet (cargo), auch gleichzeitig für die Verwaltung von Abhängigkeiten zuständig ist. Somit ist es einfach, einem Projekt Software-Bibliotheken hinzuzufügen. Solche Bibliotheken werden in Rust als "crates" bezeichnet.

1.2.2 Stream Processing

Ein Streaming System verarbeitet Daten, die in einem möglicherweise endlosen Strom auftreten. Die Verarbeitung der Daten durch dieses System wird als Stream Processing bezeichnet. Im Rahmen der DEBS Grand Challenge werden verschiedene Techniken aus dem Bereich des Stream Processings genutzt, um die zu analysierenden Messdaten zu verarbeiten.

Um Daten eines potentiell endlosen Stroms überhaupt verarbeiten zu können, muss zunächst geklärt werden, welcher Teil der Daten verarbeitet werden soll. Dies wird

durch Windowing umgesetzt. Windowing bedeutet, die Daten einer Datenquelle (endlos oder begrenzt) an verschiedenen Zeitpunkten zur Verarbeitung einzuteilen [11].

Im Rahmen der Challenge werden drei verschiedene sogenannte Sliding Windows verwendet. Ein Sliding Window wird durch eine fixe Länge und eine fixe Periode definiert [11].

In der Challenge soll alle fünf Minuten (Periode) ein Vergleich der Luftqualität zwischen zwei Jahren stattfinden. Hierzu wird die Luftqualität des aktuellen sowie des vorherigen Jahres über einen Fünf-Tages-Zeitraum (Länge) ermittelt.

Verschiedene Frameworks existieren, um Streaming Systems zu realisieren. Die Organisatoren riefen in der Beschreibung die Teilnehmer der Challenge dazu auf, in ihren Lösungen Open-Source-basierte Software zu nutzen. Daraufhin wurde von allen Teilnehmern, die eine wissenschaftliche Arbeit eingereicht haben, ein Framework der Apache Foundation eingesetzt.

Die Apache Foundation unterstützt die Entwicklung verschiedener Open-Source Softwareprojekte. Darunter befinden sich mehrere Frameworks, welche die Entwicklung eines Streaming System vereinfachen. Zwei Gruppen nutzten Apache Flink[5] und eine Teilnehmerin setzte Apache Spark[6] ein [5, 6, 7]. Beide Frameworks sind in Java geschrieben und vereinfachen die Entwicklung von Programmen, die Streams auf mehreren Computern gleichzeitig analysieren können. Hierzu stellen die Frameworks Funktionalität zur Verfügung, um die Verarbeitung der Daten in einzelnene Verarbeitungsschritte einzuteilen. Diese Schritte werden dann auf mehreren Computern gleichzeitig ausgeführt.

Neben dieser Funktionalität ermöglichen die Frameworks es auch, die Daten zu persistieren, die Auswertung zu unterbrechen und später wieder aufzunehmen. Hierbei kann garantiert werden, dass jeder Datenpunkt nur einmal ausgewertet wird (exactly-once processing). Solche erweiterte Funktionalität kam jedoch bei keiner der Lösungen zum Einsatz.

Ein Ziel der Arbeit war zu prüfen, ob sich die Programmiersprache Rust zur Entwicklung eines Streaming System eignet, das die Daten der Grand Challenge schnell verarbeiten kann.

Während andere Teilnehmer der Challenge bei der Implementierung ihrer Lösung auf ein bestehendes Framework wie Apache Flink oder Apache Spark zurückgreifen konnten, ist dies in Rust zurzeit nicht möglich. Es existieren zwar einzelne Projekte wie z.B. amadeus [7], welche sich mit Stream Processing in Rust beschäftigen. Diese Projekte sind jedoch noch nicht so ausgereift wie die von der Apache Foundation unterstützte Software. Der Mangel an ausgereiften Lösungen ist mutmaßlich auf das

[5] https://flink.apache.org/
[6] https://spark.apache.org/
[7] https://constellation.rs/amadeus

noch junge Rust-Ökosystem zurückzuführen.

Auch wenn in Rust kein auf Stream Processing ausgerichtetes Framework verfügbar ist, bedeutet dies nicht, dass Rust zur Umsetzung eines solchen Streaming System ungeeignet ist. Tatsächlich stellt Rust in der Standardbibliothek viele Datenstrukturen und Funktionen zur Verfügung, die eine funktionale Verarbeitung der auszuwertenden Daten ermöglichen. Diese sind zwar nicht dafür vorgesehen, endlose Streams zu verarbeiten, lassen sich aber dennoch auf diese Art nutzen. Somit lässt sich ein minimales Streaming System entwickeln, das die Daten der Challenge auswerten kann.

1.2.3 gRPC und Protobuf

gRPC ist ein Framework zum Ausführen von Remote Procedure Calls (RPC). Remote Procedure Calls dienen dazu, von einem Computer eine Funktion auf einem anderen Computer aufzurufen. Im Rahmen der Challenge wird gRPC dazu genutzt, die Batches von Messungen abzurufen und Ergebnisse zu übermitteln.

Damit Remote Procedure Calls ausgeführt werden können, muss definiert werden, wie kommuniziert wird und wie einzelne Datenstrukturen übertragen werden können. gRPC nutzt HTTP/2 als Protokoll, um zwischen den einzelnen Computern zu kommunizieren.

Um die Daten zu übertragen, welche von den Funktionen genutzt werden (Parameter und/oder Rückgabewerte der Funktion), muss das Format festgelegt werden, in welchem die Daten serialisiert werden. gRPC unterstützt verschiedene Formate zum Datenaustausch. Da es erweiterbar ist, können beliebige Formate hinzugefügt werden. In der ersten Version von gRPC war aber nur Protobuf als Austauschformat implementiert [12]. Aus diesem Grund wird gRPC oft in Kombination mit Protobuf verwendet. So auch im Rahmen der DEBS Grand Challenge 2021.

Protobuf erlaubt es Datenstrukturen zu definieren. Mithilfe eines Compilers können aus den Definitionen in einer Protobuf-Datei die entsprechenden Datenstrukturen in verschiedenen Programmiersprachen generiert werden.

Neben den Datenstrukturen können in Protobuf auch Funktionen definiert werden, welche vom gRPC-Server angeboten werden. Diese können mittels gRPC aufgerufen werden, wobei die Datenstrukturen mithilfe von Protobuf serialisiert werden.

Die Organisatoren der DEBS Grand Challenge stellen eine Protobuf-Datei bereit, die alle für die Challenge relevanten Datenstrukturen und RPC-Funktionen definiert [3]. Dadurch wird es ermöglicht, ohne viel Entwicklungsarbeit auf die Daten zugreifen zu können. Dies macht es einfacher, die Lösung in einer beliebigen Programmiersprache entwickeln zu können.

2 Problemstellung

Die Problemstellung basiert auf der Query 1 der DEBS Grand Challenge 2021, setzt aber die Prioritäten für die Lösung etwas anders.

Die Organisatoren der DEBS Grand Challenge stellen fest, dass die Maßnahmen gegen die Coronaviruspandemie 2020 an vielen Orten auch Auswirkungen auf die Luftqualität hatten [2]. Die Maßnahmen haben weltweit an vielen Orten dazu geführt, dass die Luftqualität gegenüber dem Vorjahr gestiegen ist [13, 14]. In der Grand Challenge 2021 soll daher die Luftqualität an verschiedenen Orten innerhalb Deutschlands im Jahr 2020 mit der Luftqualität des Vorjahres verglichen werden. Um die Luftqualität berechnen und vergleichen zu können, werden Messdaten des weltweiten Sensorennetzwerks Luftdaten (luftdaten.info) bereitgestellt.

Über eine gRPC- und Protobuf-basierte API können die Daten abgerufen und die berechneten Ergebnisse übermittelt werden. Die Organisatoren der DEBS Grand Challenge stellen den Teilnehmern eine Protobuf-Datei zur Verfügung, welche die Datenstrukturen und API-Funktionen definiert. Der Protobuf-Code wird sowohl auf der Webseite mit der Problemstellung als auch in der Arbeit der Organisatoren erläutert [3, 2].

Aus dem zur Verfügung gestellten Protobuf-Code kann mithilfe des Protobuf-Compilers der Code für die Datenstrukturen in verschiedenen Programmiersprachen generiert werden (s. voriges Kapitel). Im Folgenden wird anhand des Protobufs-Codes die genaue Problemstellung erläutert.

2.1 Beschreibung des Datensatzes

Zur Analyse der Luftqualität werden viele Messungen ausgewertet. Eine Messung enthält die in Listing 2 dargestellten Daten. Dabei beschreibt p1 die Anzahl Partikel mit einer Größe $< 10\mu m$ und p2 die Anzahl Partikel mit einer Größe $< 2.5\mu m$, welche in der Luft enthalten sind. Die Koordinaten dienen dazu, eine Messung einem Ort zuordnen zu können. Für jeden Ort muss der Durchschnitt der p1- und p2-Werte über einen längeren Zeitraum bestimmt werden, um die Luftqualität zu berechnen (s. Kapitel 1.1.1, 3.3.5).

Eine bestimmte Anzahl an Messungen wird in einem sogenannten Batch zusammengefasst (s. Listing 3). Ein Batch enthält zwei Listen (`current` und `lastyear`), in denen die Messungen gespeichert sind. Dabei enthält `current` Messungen aus dem

```
message Measurement {
        Timestamp timestamp = 1;
        float latitude = 2;
        float longitude = 3;
        float p1 = 4; //Particles < 10µm (particulate matter)
        float p2 = 5; //Particles < 2.5µm (ultrafine particles)
}
```

Listing 2: Der Inhalt einer Messung

```
message Batch {
        int64 seq_id = 1;
        bool last = 2; //Set to true when it is the last batch
        repeated Measurement current = 3;
        repeated Measurement lastyear = 4;
}
```

Listing 3: Ein Batch. Dieses enthält viele Messungen in den Feldern current und lastyear.

Jahr 2020 und `lastyear` Messungen aus dem Jahr 2019. Die Anzahl an Messungen, die in einem Batch enthalten sein sollen, kann zu Beginn eines Benchmarks konfiguriert werden. Bei der finalen Evaluierung der Performance wurde von den Organisatoren der DEBS Grand Challenge eine Batch-Größe von 10.000 Messungen vorgeschrieben. Dabei bezieht sich die Batch-Größe auf die Gesamtanzahl aller Messungen in einem Batch, sowohl denen in `lastyear` als auch denen in `current`.

In den unterschiedlichen Jahren finden nicht unbedingt die gleiche Anzahl an Messungen statt. Dies führt dazu, dass für die gleiche Zeitdauer in einem Jahr viel mehr Messungen als in dem anderen Jahr verarbeitet werden müssen. Die Messungen sind daher nicht gleichmäßig auf beide Listen verteilt. So existieren z.B. Batches, welche 10.000 Messungen in `lastyear` und keine in `current` enthalten.

Zur Auswertung wurden zu Beginn dieser Arbeit 100.000 Batches mit jeweils 10.000 Messungen gespeichert. Insgesamt sind also genau eine Milliarde Messungen zu analysieren.

2.2 Problemstellung der DEBS Grand Challenge 2021

Das Ziel der Query 1 besteht darin, die Luftqualität an verschiedenen Orten zu vergleichen. Dazu soll alle fünf Minuten[1] eine Liste der 50 Orte erstellt werden, in denen sich die Luftqualität am meisten verbessert (ResultQ1 in Listing 4). Dabei sollen nur Orte

[1] Dieses Zeitintervall wurde von den Organisatoren gewählt, prinzipiell könnte die Auswertung auch häufiger oder seltener stattfinden.

beachtet werden, an denen in den letzten zehn Minuten mindestens einmal gemessen wurde.

```
message ResultQ1 {
 //both are used to correlate the latency
 int64 benchmark_id = 1;
 int64 batch_seq_id = 2;
 //top 50 improved cities
 repeated TopKCities topkimproved = 3;
}
message TopKCities {
 //begin with 1
 int32 position = 1;
 //full name of the city according the locations
 string city = 2;
 //5D average improvement compared to previous year
 int32 averageAQIImprovement = 3;
 //(all AQI values are first rounded to 3 digits,
 // then multiplied by 1000)
 //current AQI for P1 values (based on 24h average)
 int32 currentAQIP1 = 5;
 //current AQI for P2 values (based on 24h average)
 int32 currentAQIP2 = 6;
}
```

Listing 4: Ein Ergebnis (ResultQ1) enthält einen Eintrag für jeden der 50 Orte, in denen sich die Luftqualität am meisten verbessert hat.

Die Luftqualität wird mithilfe der p1- und p2-Werte der einzelnen Messungen berechnet. Hierzu wird der Mittelwert der p1- und p2-Werte über fünf Tage berechnet. Anschließend wird mithilfe der Mittelwerte und einer Formel der Air Quality Index (AQI) errechnet, welcher die Luftqualität repräsentiert (s. Kapitel 1.1.1, 3.3.5). Die Berechnung des Mittelwerts über einen längeren Zeitraum bewirkt, dass einzelne stark abweichende Messungen keinen großen Einfluss auf die Berechnung der Luftqualität haben. Dass hierfür ein Fünf-Tages-Zeitraum genutzt wird, wurde von den Organisatoren der DEBS Grand Challenge festgelegt. Die amerikanische Umweltbehöde EPA dagegen rechnet üblicherweise nur mit einem 24-Stunden-Zeitraum [1]. Eine weitere Abweichung der Challenge gegenüber dem EPA-Standard besteht darin, dass die Organisatoren eine höhere Präzision verlangen: Der AQI muss bis auf drei Nachkommastellen genau berechnet werden.

Zusätzlich zur Verbesserung gegenüber dem letzten Jahr wird auch die Luftqualität der letzten 24 Stunden, getrennt nach Partikelart (p1/p2), angegeben (s. TopKCities→currentAQIP1 und TopKCities→currentAQIP2 in Listing 4).

Nach jedem Batch wird ein Ergebnis wie in Listing 4 erwartet. Falls durch das Batch

ein Fünf-Minuten-Fenster vervollständigt wird, so soll das Ergebnis die Liste mit den 50 Orten mit der besten Luftqualität enthalten. Andernfalls soll die Liste leer sein. Dies führt zu einem Problem: Falls ein Batch mehr als fünf Minuten an Daten enthält, muss eines der beiden Ergebnisse ignoriert werden. Da die Batch-Größe jedoch nur 10.000 Messungen beträgt, tritt dieses Problem in der Praxis nicht auf. Dennoch wäre es sinnvoller gewesen, die Antwort mit einem Zeitstempel zu vergleichen: Dies würde es ermöglichen, die Ergebnisse unabhängig von den Batches zu berechnen.

```
message Point { double longitude = 1; double latitude = 2;}
message Polygon { repeated Point points = 1; }
message Location {
        string zipcode = 1;
        string city = 2;
        double qkm = 3;
        int32 population = 4;
        repeated Polygon polygons = 5;
}
```

Listing 5: Jeder Ort besitzt einen Namen und mindestens ein Polygon.

Bevor jedoch die Luftqualität für die Orte berechnet werden kann, muss für jede Messung ermittelt werden, in welchem Ort die Messung erfolgt ist. Auch hierfür stellen die Organisatoren einen Datensatz bereit. Für jeden Ort gibt es einen Namen, eine Liste von Polygonen, deren Inhalt die Fläche des Ortes markiert (s. Listing 5) sowie weitere Eigenschaften, die in der Challenge nicht gebraucht werden (zipcode, qkm, population). Wenn also die Koordinaten einer Messung sich innerhalb eines Polygons befinden, dann gehört die Messung zu dem Ort, zu dem das Polygon gehört.

Das Problem bei dieser Query ist die große Menge an Daten, welche ausgewertet werden muss: Eine Milliarde Messungen. Der verwendete Datensatz erstreckt sich in den zwei Vergleichsjahren jeweils über einen Zeitraum von drei Monaten und knapp fünf Tagen. Um die Daten in Echtzeit verarbeiten zu können, müssten daher rund 120 Messungen pro Sekunde analysiert werden. Selbst wenn jede Messung innerhalb von 0.1ms verarbeitet werden kann, würde eine vollständige Auswertung des Datensatzes mehr als einen Tag dauern. Um die Daten in einem möglichst kurzem Zeitfenster auswerten zu können, muss daher eine effiziente und performante Software entwickelt werden, welche viele Zehntausend Messungen pro Sekunde analysieren kann.

2.3 Abweichungen von der DEBS Grand Challenge 2021

Auch wenn die Query 1 der DEBS Grand Challenge 2021 wie beschrieben bearbeitet wurde, so wurden einige Aspekte angepasst oder ihre Wichtigkeit anders eingestuft.

Eine signifikante Abweichung von der Vorgabe besteht darin, dass der Datensatz nicht über das Netzwerk empfangen oder gesendet wurde. Dies hat vermutlich einige Auswirkungen auf die Performance der Lösungen. Die gRPC-API würde eine HTTP2-Verbindung nutzen, um Batches von Messungen zu empfangen und die Ergebnisse zu übermitteln. Da jedoch nur lokal gearbeitet wird, fällt der Overhead dieser Verbindung nun weg. Andererseits kommt jedoch der Overhead dazu, die Batches von einem Speichermedium (SSD oder Festplatte) einzulesen. Wie groß die Auswirkung auf die Performance ist, ist schwer abzuschätzen. Es ist aber davon auszugehen, dass die Performance weniger schwankt, da die Auswertung unabhängig von der Netzwerkqualität durchgeführt werden kann.

Außerdem stellen die Veranstalter der DEBS Grand Challenge noch eine weitere Aufgabe ("Query 2"). Aufgrund des Umfangs der beiden Aufgaben wurde jedoch zu Beginn dieser Arbeit beschlossen nur Query 1 umzusetzen. Die Lösungen der anderen Teilnehmer haben jedoch beide Queries bearbeitet, sodass dies berücksichtigt werden muss, wenn die Performance der entwickelten Lösungen verglichen wird (s. Kapitel 6.1.1).

Während bei der Challenge auch die Latenz der Auswertung berechnet wurde, wurde in dieser Arbeit allein auf die Verarbeitungsgeschwindigkeit der Lösungen geachtet, da diese leichter zu ermitteln ist. Da die in dieser Arbeit entwickelte Lösung nur einen Computer nutzt, besitzt sie keinen Overhead, welcher aus der Koordinierung mit anderen Computern resultiert. Dementsprechend ist ohnehin davon auszugehen, dass die Latenz der in dieser Arbeit entwickelten Lösung niedriger oder vergleichbar mit der der Lösungen der anderen Teilnehmer der Grand Challenge ist.

Zuletzt stellt sich bei der Umsetzung der Challenge die Frage, welche Aspekte bei der Lösung besonders beachtet werden sollen. Die Organisatoren der DEBS Grand Challenge legen hierbei besonders Wert auf Skalierbarkeit und empfehlen, ein Open Source-basiertes Streaming System zu nutzen [2]. Solche Systeme vereinfachen die Entwicklung einer Lösung, da mit geringem Aufwand komplexe Pipelines erstellt werden können, welche Datenströme automatisch auf mehreren Computern verteilen, zwischenspeichern und effizient verarbeiten können. Verteilte, besonders skalierbare Systeme besitzen jedoch verschiedene Probleme. Unter anderem ist es möglich, dass ein verteilter Algorithmus so ineffizient ist, dass dessen Performance erst ab einigen hundert Cores die Performance einer einzelnen Maschine übertrifft [4].

In dieser Arbeit wird daher Wert auf eine möglichst perfomante Lösung gelegt, wobei die horizontale Skalierbarkeit der Lösung zweitrangig ist.

Um eine möglichst perfomante Lösung zu entwickeln scheint die Programmiersprache Rust auf den ersten Blick eine gute Wahl zu sein: Anders als z.B. Java besitzt Rust keine Runtime, die ein Programm bremsen könnte, während sie trotzdem (im Gegensatz zu C/C++) sicher und leicht zu parallelisieren ist. Der Nachteil dabei liegt

jedoch darin, dass in Rust noch kein besonderes Framework existiert, das sich auf Stream Processing fokussiert hat. Rust enthält jedoch bereits in der Standardbibliothek viele Funktionen und Datenstrukturen, die einen funktionalen Programmierstil ermöglichen. Es stellt sich daher die Frage, ob diese sich für die Entwicklung eines Streaming System eignen.

3 Implementierung

Um die Luftdaten auswerten zu können, wurde der von den Organisatoren bereitgestellte Protobuf-Code genutzt, um daraus die Datenstrukturen in Rust zu generieren. Hierzu wurde die Rust-crate prost[1] eingesetzt. Somit war es möglich, die gRPC-API zu nutzen.

3.1 Herunterladen der Daten

Um die Daten herunterladen zu können, musste ein Benchmark erstellt werden. Ein Benchmark beschreibt im Rahmen der Challenge eine Ausführung des Programms, welches die Daten auswertet. Der Benchmark kann bei der Erstellung konfiguriert werden. Dabei kann unter anderem angegeben werden, welche der beiden Queries bearbeitet werden sollen. Um die Daten herunterzuladen wurde angegeben, dass sowohl Query 1 als auch Query 2 bearbeitet werden sollten. Dies stellt sicher, dass alle Daten erhalten werden, welche für beide Queries notwendig sind.

Neben den beiden Queries musste auch angegeben werden, wie viele Messungen in einem Batch enthalten sein sollen. Zur Auswertung der Daten wurde ein Wert von 10.000 Messungen per Batch gewählt, da dies der Anzahl Messungen entspricht, mit denen die Performance der Lösungen der Grand Challenge analysiert werden soll [3].

Insgesamt wurden 100.000 Batches mit jeweils 10.000 Messungen heruntergeladen, also insgesamt eine Milliarde Messungen. Jedes Batch wurde nach dem Herunterladen mithilfe der Protobuf-API serialisiert und in einzelne Dateien gespeichert. Gespeichert benötigt jedes Batch 293KiB Speicherplatz. Insgesamt ergeben sich daraus knapp 30GiB an Daten, die ausgewertet werden müssen.

Durch die lokale Speicherung wurde es möglich, mit einem Datensatz zu arbeiten, welcher sich bei verschiedenen Ausführungen des Programms garantiert nicht ändert. Der Datensatz war daher auch weiter verfügbar, als die Challenge offiziell beendet wurde und die Challenge-Plattform abgeschaltet wurde.

3.2 Analyse der Daten

Die heruntergeladenen Daten wurden analysiert, um die Eigenschaften der Daten herauszusuchen und mögliche Optimierungsansätze zu identifizieren.

[1] https://github.com/tokio-rs/prost

Bei der Analyse der Daten fielen einige Eigenschaften auf, die nicht explizit genannt wurden. Beispielsweise enthält die generierte Datenstruktur für Messungen den Zeitstempel in Form einer Option, was bedeutet, dass Messungen ohne Zeitstempel existieren könnten. Im verwendeten Datensatz tauchte solch eine Messung jedoch nicht auf. Tatsächlich fiel bei der Analyse der Zeitstempel eine andere hilfreiche Eigenschaft auf: Alle Messungen sind (über alle Batches hinweg) chronologisch sortiert. Dies vereinfacht die Verarbeitung der Messungen erheblich, da klar ist, dass alte Messungen niemals erneut ausgewertet werden müssen, wenn neue Messungen hinzukommen.

3.3 Implementierung

Die Auswertung der Daten wurde in verschiedene Schritte unterteilt. Die einzelnen Verarbeitungsschritte wurden modular implementiert:

- Einlesen der Messdaten
- Lokalisierung der Messungen
- Unterteilung einzelner Messungen in Fünf-Minuten-Abschnitte
- Aggregierung der Werte in Fünf-Minuten-Abschnitte
- Berechnung der Luftqualität über fünf Tage
- Erstellung der Rangliste
- Berechnung der Luftqualität der letzten 24 Stunden
- Ausgabe der Ergebnisse

Im folgenden wird jeder der einzelnen Schritte erläutert.

3.3.1 Einlesen der Messdaten

Das Einlesen der Messdaten erfolgt in einem eigenen Thread. Dieser lädt die Batches der Reihe nach aus den entsprechenden Dateien. Dazu wird der Inhalt der Datei in ein byte-Array geladen und anschließend mithilfe der Protobuf-API deserialisiert. Anschließend wird das geladene Batch an die Threads weitergeleitet, welche die Daten verarbeiten.

Somit können Daten gleichzeitig sowohl geladen als auch verarbeitet werden.

3.3.2 Lokalisierung der Messungen

Die Lokalisierung der einzelnen Messungen ist einer der ersten Schritte, die durchgeführt werden müssen. Erst wenn bekannt ist, zu welchem Ort eine Messung gehört, kann eine Aggregierung der Werte für diesen Ort durchgeführt werden.

Um den Ort zu identifizieren, zu dem eine Messung gehört, muss überprüft werden, welches Polygon eines Ortes die Koordinaten der Messung enthält. Hierfür wird die geo-crate verwendet [2]. Diese löst das Point-in-Polygon Problem (s. Kapitel 4.1.1), indem die Anzahl an Schnitten mit dem Polygon gezählt werden. Die Anzahl der Schnittprüfungen steigt jedoch linear mit der Zahl der Punkte, die das Polygon bilden. Da es über 8000 Orte mit Polygonen gibt, die im Schnitt jeweils ca. 370 Punkte enthalten, ist dieser naive Ansatz zu langsam.

Um die Performance zu verbessern, werden hierzu Optimierungen entworfen, die zum Ziel haben, möglichst selten diese langsame Funktion aufzurufen. Dies wird dadurch erreicht, dass zum einen die Anzahl zu prüfender Orte durch einen R*-Tree reduziert wird (s. Kapitel 4.1.2). Zum anderen findet ein Caching der Ergebnisse statt. Die Details dieser Optimierungen sind im Kapitel 4.1 zu finden.

3.3.3 Unterteilung in Fünf-Minuten-Abschnitte

Nachdem die Messungen lokalisiert sind, werden diese in Fünf-Minuten-Intervalle unterteilt. Dies muss einmal für alle Messungen des aktuellen Jahres und einmal für alle Messungen des Vorjahres getan werden.

Dies führt jedoch zu einem Problem: Rust ist darauf ausgelegt, dass die Lebensdauer eines Wertes fest bestimmt ist, sodass der Wert nach Ende des Gebrauchs freigegeben werden kann. Dies ist hier aber nicht möglich: Aus einem Iterator von Batches sollen zwei verschiedene Streams von lokalisierten Messungen generiert werden. Die daraus generierten Streams müssen jedoch unabhängig voneinander verwendet werden können, um die Messungen in Fünf-Minuten-Schritten zu sammeln. Somit wird erst zur Laufzeit klar, wie lange einzelne Batches im Speicher verbleiben müssen. Dies macht es schwierig, einen Stream in zwei Streams aufzuteilen.

Die Rust-Standardbibliothek stellt zwar eine Funktion namens `unzip` bereit, welche aus einem Iterator von Tupeln zwei verschiedene Listen generieren kann. Die Liste ist dabei materialisiert, d.h. alle Werte werden berechnet und in die Liste eingefügt. In diesem Fall würde das bedeuten, dass der gesamte Datensatz in den Speicher geladen werden muss, bevor die Liste erstellt werden kann. Bei Streams, welche potentiell unendlich lang sein können, ist diese Funktion daher nicht geeignet.

Zum Aufteilen in zwei Streams musste daher eine eigene Datenstruktur entwickelt werden. Diese speichert zeitweise für jedes Batch die zwei Listen von Messungen in

[2] https://github.com/georust/geo

jeweils einen Ringbuffer. Wenn ein Stream (z.B. der Stream für die Daten des aktuellen Jahres) die Daten eines weiteren Batches benötigt, so wird die gewünschte Liste (z.B. `current`) aus dem nächsten Batch geladen und die andere Liste (z.B. `lastyear`) zwischengespeichert, bis sie von dem anderen Stream genutzt wird. Da die Streams etwa gleich schnell abgearbeitet werden, müssen nur wenige Daten zwischengespeichert werden.

Dass hier ein eigener Mechanismus implementiert werden musste, zeigt einen Nachteil der Umsetzung in Rust auf: Es ist problematisch mit Werten zu arbeiten, die eine unterschiedliche Lebensdauer besitzen. Einfaches Iterieren und Aggregieren ist kein Problem, doch ein Aufteilen in mehrere Streams ist nicht trivial. Hierbei bereitet die möglicherweise unterschiedliche Lebensdauer der erzeugten Ausgaben Probleme. Festzustellen, wann der Speicher für ausgewertete Daten freigegeben werden kann, ist ein fundamentales Problem, das auch andere Streaming-Systeme meistern müssen. In anderen Programmiersprachen wie z.B. Java kann die Runtime aber automatisch erkennen, wann Daten nicht länger genutzt werden, sodass der Speicher freigegeben und erneut verwendet werden kann. In Rust hingegen musste diese Funktionalität von Hand implementiert werden.

Dass diese Funktionalität von Hand implementiert werden musste, zeigt einen Nachteil auf, den die Umsetzung in Rust ohne ein Streaming-Framework mit sich bringt. Allerdings muss darauf hingewiesen werden, dass hier nur die Standardbibliothek von Rust verwendet wurde. Diese stellt zwar Iterator-Funktionalität zur Verfügung, ist aber nicht speziell auf Streaming-Systeme zugeschnitten. Ein Streaming-Framework könnte solchen Problemen entgegenwirken, indem geeignete Datenstrukturen und Algorithmen bereitgestellt werden.

3.3.4 Aggregierung der Fünf-Minuten-Abschnitte

Nachdem die lokalisierten Messungen in Fünf-Minuten-Abschnitte unterteilt wurden, werden die darin enthaltenen Daten aggregiert. Später wird zur Berechnung der Luftqualität für jeden Ort der durchschnittliche p1- und p2-Wert benötigt. Ein Fünf-Minuten-Aggregat wird daher generiert, indem für jeden Ort die p1- und p2-Werte aufsummiert (`sum_p1` und `sum_p2`) und die Anzahl der Messungen pro Ort gezählt werden (`samples`). Solche Fünf-Minuten-Aggregate sind in Abbildung 1 dargestellt: Jedes Fünf-Minuten-Aggregat entspricht einem Kasten. Die Fünf-Minuten-Aggregate basieren auf der vorherigen Unterteilung der Messungen in Fünf-Minuten-Abschnitte. Die Aggregate sind daher disjunkt: Der Zeitraum der Messungen, welche in einem Aggregat enthalten sind, überschneidet sich nicht mit dem Zeitraum eines anderen Aggregates. Jedes Fünf-Minuten-Aggregat beginnt dort, wo das vorige endet.

Abbildung 1: Aggregierung von mehreren Fünf-Minuten-Aggregaten über fünf Tage. Ein Kasten entspricht den summierten Messwerten für einen Fünf-Minuten-Zeitraum am selben Ort.

3.3.5 Berechnung der Luftqualität

Um die Luftqualität zu berechnen, werden der durchschnittliche p1- und p2-Wert für diesen Ort benötigt. Hierfür werden mehrere Fünf-Minuten-Aggregate weiter aggregiert: Durch Teilen der aufsummierten p1- und p2-Werte durch den aufsummierten Teiler (`samples`) ergibt sich der durchschnittliche p1- und p2-Wert über fünf Tage. In Abbildung 1 ist visualisiert, wie dies für ein Fünf-Tages-Fenster funktioniert. Da 5 Tage in $1440 \cdot 5$ Minuten unterteilt werden können, müssen die Werte der letzten 1440 Fünf-Minuten-Fenster aufsummiert werden. Da die Rangliste alle fünf Minuten erstellt werden muss, verschiebt sich das Fenster dieser Fünf-Tages-Aggregierung bei jeder Berechnung um fünf Minuten, also um ein Fünf-Minuten-Aggregat. Es findet also eine Auswertung in einem Sliding Window (s. Kapitel 1.2.2) mit einer Länge von 1440 Werten und einer Periode von einem Wert statt.

Mithilfe einer Optimierung (näher beschrieben in Kapitel 4.2.2) kann die Aggregierung um ein Vielfaches schneller berechnet werden. Dies ist notwendig, da die Berechnung der durchschnittlichen Partikelkonzentration alle fünf Minuten mehrfach berechnet werden muss. Pro Ort müssen drei solcher Aggregierungen durchgeführt werden: Zwei über fünf Tage, um die Daten der beiden Jahre zu vergleichen, und einmal um die Luftqualität der letzten 24 Stunden zu bestimmen.

Nachdem die durchschnittliche Konzentration an Partikeln in der Luft berechnet wurde, kann nun die eigentliche Berechnung der Luftqualität erfolgen. Die Berechnung der Luftqualität basiert auf einer von der EPA zur Verfügung gestellten Tabelle und einer mathematischen Funktion, welche eine lineare Interpolation durchführt [1, 2]. Die Formel selbst ist für alle Stoffe die gleiche, allerdings variieren die Werte, zwischen denen die Interpolation stattfindet. In Abbildung 2 ist die Funktion zur Umrechnung der Werte visualisiert. Die vertikalen Linien markieren dabei die unterschiedlichen Tabellenzeilen, die unterschiedlichen Farben stellen die Grenzwerte der Luftqualität dar. Generell gilt: Je höher der AQI, desto schlechter die Luftqualität.

Nachdem sowohl für p1 als auch p2 ein AQI-Wert berechnet wurde, wird der größere AQI-Wert als finaler AQI-Wert genutzt. Die offizielle Berechnungsmethode rundet den AQI-Wert [1]. In der Challenge wird der AQI bis auf drei Stellen nach dem Komma gefragt.

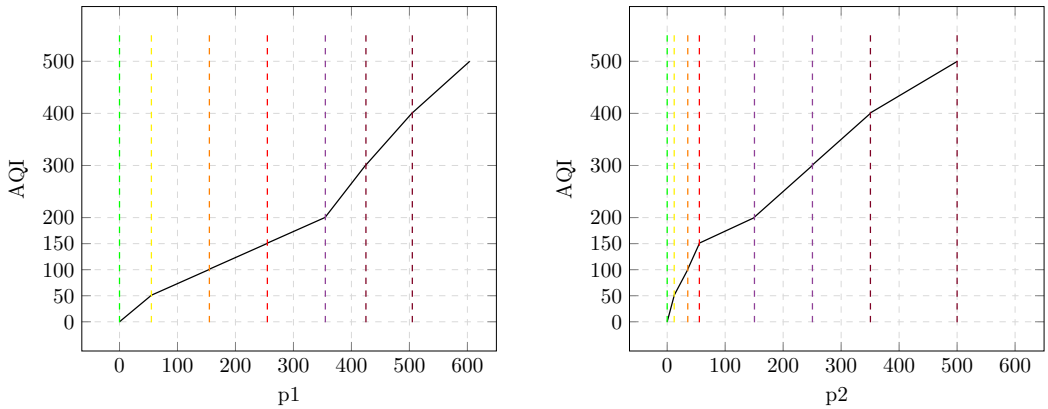

Abbildung 2: Verhältnis der Partikelkonzentration p1 und p2 zu AQI

3.3.6 Erstellung der Rangliste

Nachdem die Luftqualität berechnet wurde, kann die Verbesserung der Luftqualität im letzten Jahr berechnet werden, indem für jeden Ort die Luftqualitätswerte der beiden Jahre verglichen werden.

Zur Berechnung der Verbesserung an einem Ort wird die Differenz des AQI-Wertes des aktuellen Jahres mit dem AQI-Wert des letzten Jahres gebildet. Da für jeden Ort unabhängig voneinander die Differenz zwischen den AQI-Werten der beiden Jahre gebildet werden kann, kann diese Berechnung parallelisert werden. Somit eignet sich dieser Schritt hervorragend zur parallelen Verarbeitung der Daten (s. Kapitel 4.3.1).

Nachdem die Berechnung der Verbesserung der Luftqualität abgeschlossen ist, wird die Rangliste der 50 Orte mit der besten Luftqualität erstellt. Da es nur ca. 8000 Orte gibt, für die die Rangliste generiert werden muss, reicht zur Erstellung der Rangliste ein einfaches Sortieren der Ergebnisse. Hierbei genügt der Sortieralgorithmus der Rust-Standardbibliothek.

3.3.7 Luftqualität der letzten 24 Stunden

Zuletzt muss noch für die 50 Orte mit der besten Luftqualität die Luftqualität der letzten 24 Stunden basierend auf den p1- und p2- Werten in diesem Zeitraum berechnet werden. Es ist am einfachsten, den 24-Stunden-Durchschnitt der p1- und p2-Werte gleichzeitig mit dem Fünf-Tage-Durchschnitt zu bilden, da somit alle Aggregate gleichzeitig ausgewertet werden können. Der 24-Stunden-Durchschnitt wird daher zusammen mit dem Fünf-Tage-Durchschnitt gespeichert. Somit muss am Ende der Auswertung nur noch für die 50 Orte mit der besten Luftqualität mithilfe des p1- und p2-Durchschnitts die Luftqualität der letzten 24 Stunden berechnet werden (vgl. Abbildung 2).

Anschließend kann das Ergebnis genutzt werden. In der Originalversion der Challenge würde das Ergebnis mithilfe der gRPC zu den Organisatoren geschickt werden. Da der in dieser Arbeit entwickelte Code offline arbeitet, wird stattdessen das ermittelte Ergebnis einfach ausgegeben.

4 Optimierungen

Auch wenn Rust schnell ist, ersetzt Rust allein keine Optimierungen auf algorithmischer Ebene. An mehreren Stellen der Verarbeitung ist es möglich, 1000x weniger Daten verarbeiten zu müssen, indem geeignete Algorithmen eingesetzt werden.

In diesem Kapitel werden die einzelnen Optimierungen im Detail erläutert, welche im Rahmen dieser Arbeit eingesetzt wurden.

4.1 Schnellere Lokalisierung

Die erste Operation, die für jede Messung durchgeführt werden muss, ist die Zuordnung der Messung zu einem bestimmten Ort. Da für mehrere Millionen Messungen pro Sekunde geprüft werden muss, ob sie einem der über 8000 Orte angehören, müssen entsprechend performante Algorithmen entwickelt und eingesetzt werden.

4.1.1 Das Point-in-Polygon Problem

Zur Zuordnung der Messungen zu einem Ort werden von den Veranstaltern der DEBS Grand Challenge Daten bereitgestellt (vgl. Listing 5 in Kapitel 2.2) [2]. Für jeden Ort wird eine Liste der Polygonen angegeben, die den Ort begrenzen. Eine Messung, deren Koordinaten sich innerhalb eines solchen Polygons befinden, gehört somit zum entsprechenden Ort.

Ein Computer kann jedoch nicht wie ein Mensch einfach sehen, dass sich ein Punkt in einem Polygon befindet. Hierfür muss ein Algorithmus eingesetzt werden. Ein einfacher Algorithmus besteht darin, die Anzahl an Schnitten des Polygons mit einer Halbgeraden, welche vom zu prüfenden Punkt ausgeht, zu zählen. Ist die Anzahl der Schnitte ungerade, so liegt der Punkt im Polygon. Ist die Anzahl der Schnitte gerade, so liegt der Punkt außerhalb des Polygons [15]. Typischerweise wird eine Halbgerade entlang der x- oder y-Achse verwendet, da somit eine Prüfung, auf welcher Seite der Halbgeraden sich die Punkte des Polygons befinden, weniger Berechnungen erfordert.

Ein Beispiel für solche eine Prüfung ist in Abbildung 3 dargestellt. Hier wird überprüft, ob der blau markierte Punkt sich innerhalb des schwarz schattierten Polygons befindet. Hierzu wird eine Halbgerade entlang der x-Achse erstellt. Da die Halbgerade drei Kanten des Polygons schneidet (in der Abbildung rot markiert), ist klar, dass der Punkt im Polygon liegt.

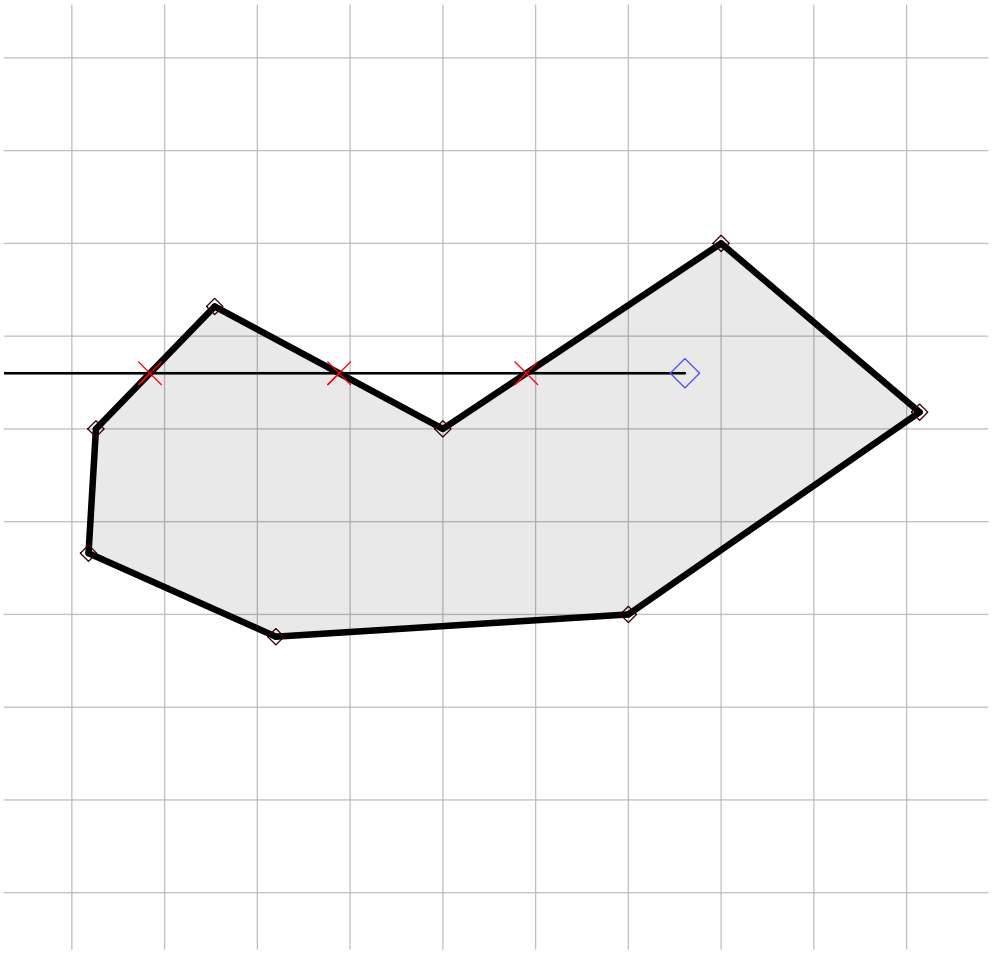

Abbildung 3: Visualisierung eines Point-in-Polygon Tests. Es wird geprüft, ob der blaue Punkt im Polygon enthalten ist.

Dieser Algorithmus ist zwar einfach, aber langsam, da er jede Kante des Polygons beachten muss. Andere Algorithmen, wie beispielsweise die "Dual perspective method" von Ali et al., können mehr als zehnmal so schnell sein [16].

Dennoch wurde der beschriebene Algorithmus eingesetzt, da dieser bereits in der geo-crate fertig implementiert war und somit direkt verwendet werden konnte. Einen anderen Algorithmus zu verwenden, wäre mit nicht unerheblichem Aufwand verbunden, da dieser wesentlich komplexer wäre (vgl. [16]).

Es existieren ohnehin verschiedene Möglichkeiten, den Code weiter zu optimieren, ohne einen anderen Algorithmus zu implementieren, welcher das Point-in-Polygon Problem löst: Indem zwei Verarbeitungsstufen hinzugefügt werden, wird ermöglicht, dass das Point-in-Polygon Problem seltener gelöst werden muss. Da das Problem somit seltener gelöst werden muss, wird die Performance verbessert.

4.1.2 R*-Tree

Ein erster, naiver Ansatz bei der Lokalisierung der Messungen besteht darin, bei einer Lokalisierung bei jedem der 8000 Orte für jedes der Polygone zu untersuchen, ob die Messung im jeweiligen Polygon enthalten ist. Dieser Ansatz ist jedoch ineffizient, da es nicht nötig ist, überhaupt alle Orte zu durchsuchen: Manche Orte können schnell ausgeschlossen werden, da man mithilfe eines Rechtecks, welche alle Polygone für diesen Ort enthält (Bounding Box), feststellen kann, dass die Polygone für diesen Ort einen Punkt gar nicht enthalten können, weil das Rechteck diesen nicht enthält. R-Trees ermöglichen es, viele Orte auf einmal auszuschließen. Ein R-Tree enthält viele dieser Rechtecke, sodass schnell gefunden werden kann, welcher der 8000 Orte überhaupt in Frage kommt.

R-Trees wurden von Antonin Guttman erfunden, um Operationen mit Geodaten zu beschleunigen [17]. Dabei ist das Ziel, möglichst schnell Einträge zu lokalisieren - genau das, was auch in dieser Challenge gefragt ist. R-Trees bestehen aus Rechtecken, welche entweder weitere Rechtecke oder eine Referenz zu einem Objekt enthalten. Ein solcher R-Tree ist in Abbildung 4 dargestellt. Mithilfe eines R-Trees kann eine Hierarchie von Rechtecken aufgebaut werden. Diese Hierarchie erlaubt es, schnell die Rechtecke zu finden, die einen beliebigen Punkt enthalten.

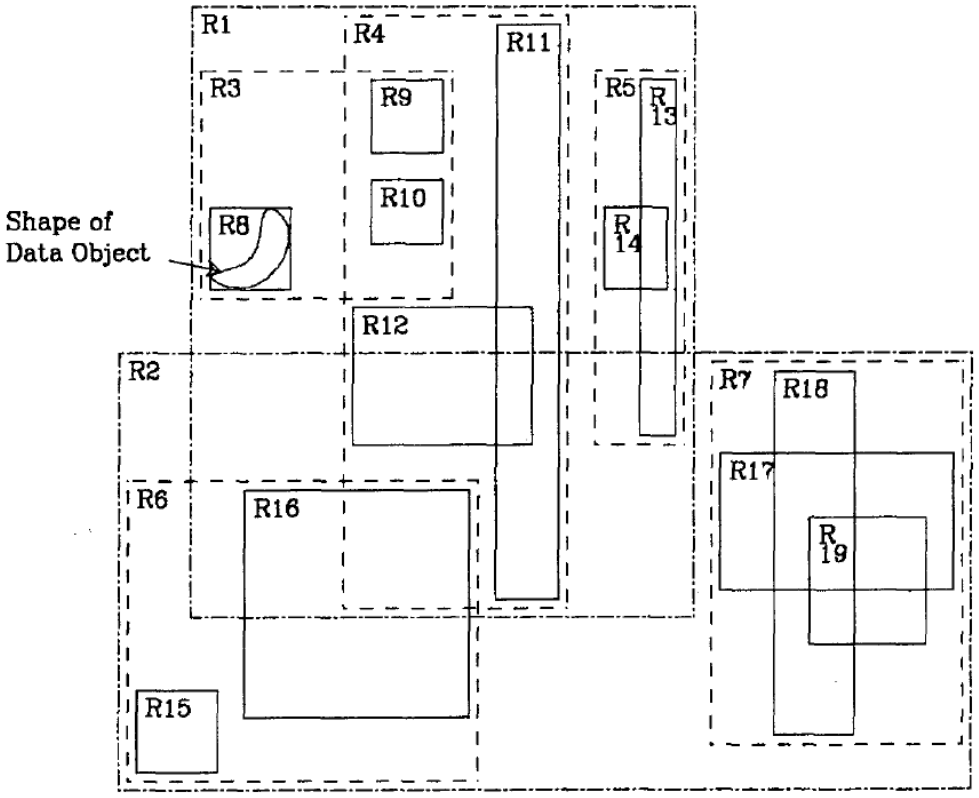

Abbildung 4: Visualisierung eines R-Trees. Grafik der Arbeit von Guttmann entnommen [17]

Später wurde der R-Tree von Beckmann et al. weiterentwickelt [18]. Dabei wurde ein Algorithmus erfunden, der es ermöglicht, die Performance beim Durchsuchen des R-Trees weiter zu verbessern, wenn die Erstellung des R-Trees auf eine bestimmte Weise erfolgt. Der verbesserte R-Tree wird als R*-Tree bezeichnet.

In dieser Arbeit wurde ein R*-Tree erzeugt, um schnell Orte (bzw. das dazugehörige Polygon) zu finden, in denen die Messung überhaupt enthalten sein könnte. Diese Optimierung ermöglicht es, den Point-in-Polygon Test nur für wenige Orte anstatt für jeden der über 8000 Orte durchführen zu müssen.

Hierzu wurde die Rust-crate rstar [1] genutzt. Diese ermöglicht das Erstellen eines R*-Trees aus einer Liste von Rechtecken. Die Daten zur Lokalisierung müssen daher nur in die Datenstrukturen der rstar-crate umgewandelt werden, indem die bounding box für jedes Polygon gebildet wird (vgl. "R8" in Abbildung 4).

Anschließend kann mit einem Methodenaufruf ein Iterator erzeugt werden, der für einen Punkt über die Orte iteriert, welche den Punkt enthalten könnten. Da jedoch durch die Erzeugung der bounding box die Fläche für einen Ort vergrößert wird, muss noch mit einem Point-in-Polygon Test überprüft werden, ob der Punkt sich tatsächlich in einem Polygon des zu prüfenden Ortes befindet.

4.1.3 Geocaching

Neben den anderen Optimierungen kann mithilfe eines Caches die Performance weiter erhöht werden. Durch Caching wird eine schnelle Möglichkeit geschaffen, zu überprüfen ob bekannt ist, ob ein Punkt sicher in einem Polygon enthalten ist. Dadurch kann es vermieden werden, einen neuen Point-in-Polygon Test durchzuführen.

Hierzu bedarf es jedoch eines speziellen Caches. Normalerweise werden in einem Cache die Eingabewerte einer Funktion als Schlüssel genutzt, um den Rückgabewert der Funktion für weitere Aufrufe der Funktion zwischenzuspeichern. Dies ist hier jedoch nicht ohne weiteres möglich. Die Funktion, die überprüft, ob ein Punkt sich in dem zu überprüfenden Polygon befindet, nimmt als Eingabewerte den Längen- und Breitengrad des Punktes entgegen. Da diese Eingabewerte jedoch aus Fließkommazahlen bestehen, kann hieraus kein sinnvoller Schlüssel für den Cache erzeugt werden. Bereits minimale Abweichungen im Nachkommabereich würden den Schlüssel ändern. Ein Runden der Koordinaten könnte zwar einen Schlüssel generieren. Dies könnte aber zu Fehlern bei Messungen nahe der Grenze eines Polygons führen. Einen solchen hashbasierten Ansatz haben Marić et al. umgesetzt: Sie rundeten in ihrem Programm die Koordinaten auf die erste Nachkommastelle [5]. Da dies jedoch zu Fehlern führen kann, wurde dieser hashbasierte Ansatz in dieser Arbeit nicht weiter verfolgt.

Dennoch ist es möglich, eine Art Cache zu generieren. Dies funktioniert wie folgt:

[1] https://github.com/Stoeoef/rstar

Wenn ein Cache-Eintrag für einen Punkt erzeugt wird, werden neben dem Ergebnis (Ist der Punkt innerhalb/außerhalb des Polygons?) die Koordinaten des Punktes sowie zusätzlich die Distanz des Punktes zur nächsten Kante des Polygons in dem Cache-Eintrag gespeichert.

Beim Abfragen des Caches wird für jeden Cache-Eintrag überprüft, ob die Distanz zwischen dem Eingabepunkt und dem Punkt des Cache-Eintrags kleiner ist als die im Cache-Eintrag gespeicherte Distanz. Falls ja, befinden sich beide auf der gleichen Seite des Polygons. Falls nein, lässt sich aus diesem Eintrag nichts schließen und es muss der nächste Eintrag überprüft oder ein Point-in-Polygon Test durchgeführt werden.

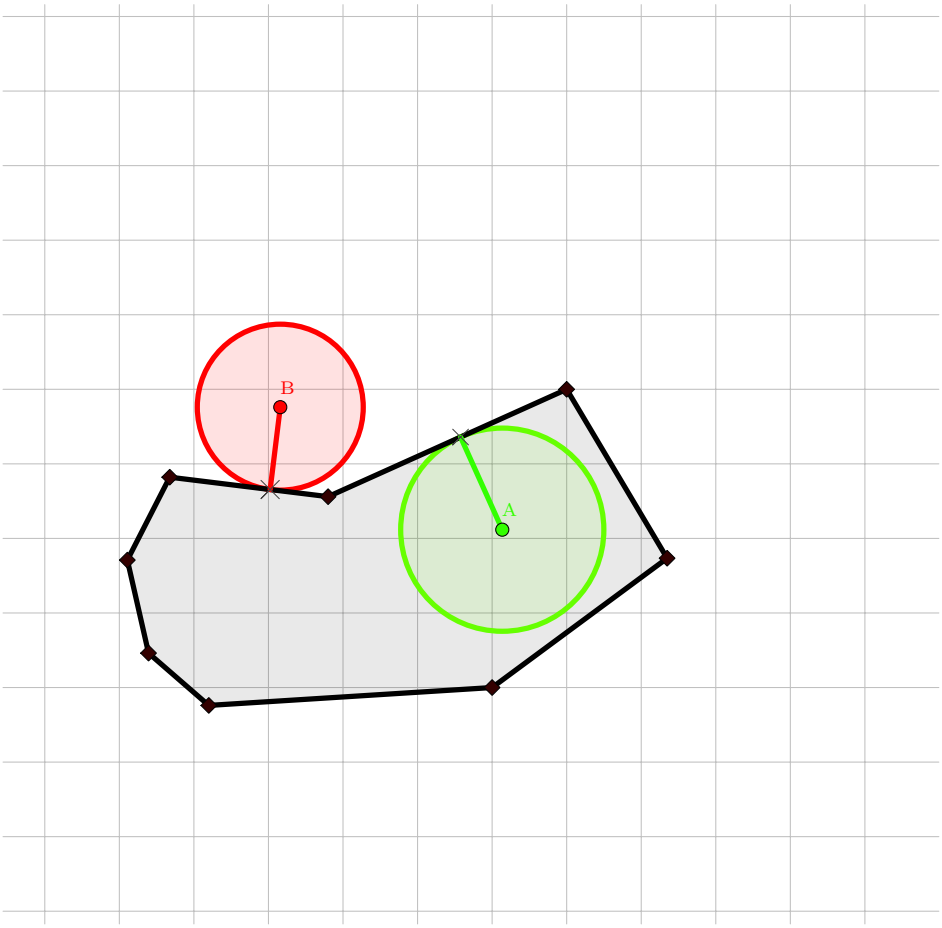

Abbildung 5: Visualisierung des entwickelten Caching-Algorithmus

Ein Beispiel für solch einen Cache ist in Abbildung 5 sichtbar. Angenommen, es soll geprüft werden, ob sich ein neuer Punkt im schwarzen Polygon befindet. Bei vorherigen Lokalisierungen wurden bereits Cache-Einträge an den Punkten A und B erstellt. Befindet sich der zu lokalisierende Punkt innerhalb des grünen Kreises um A, so ist sofort klar, dass der Punkt innerhalb des Polygons liegt. Falls sich der Punkt dagegen im roten Kreis um B befindet, so kann ausgeschlossen werden, dass

das Polygon den Punkt enthält. Wenn sich der Punkt außerhalb der beiden Kreise befindet, so reicht der Cache nicht aus, um zu entscheiden, ob das Polygon den Punkt enthält. In diesem Fall müsste das Point-in-Polygon Problem gelöst werden (vgl. 4.1.1).

Es kann einfach bewiesen werden, dass der entwickelte Caching-Algorithmus nie falsche Ergebnisse zurückliefert:

1. Um die Seite des Polygons zu wechseln, muss eine Kante überschritten werden
2. Die Distanz des Cache-Eintrags besteht aus der Distanz zur nächsten Kante
3. Da sich per Definition keine Kante innerhalb dieser Distanz befindet, müssen alle Punkte, die sich innerhalb dieser Distanz befinden, auf der gleichen Seite des Polygons liegen wie der Cache-Eintrag

Nachdem nun beschrieben wurde, wie der Cache funktioniert, existieren verschiedene Möglichkeiten, den Cache selbst noch zu optimieren. So werden z.B. nur Punkte mit einem gewissen Mindestradius im Cache gespeichert. Außerdem wird eine maximale Anzahl an Einträgen im Cache festgelegt. Beides dient dazu, die Anzahl der zu durchsuchenden Einträge im Cache möglichst gering zu halten.

Bei der Implementierung dieses Caches kam es zu einem Kompilierfehler. Dies lag daran, dass ohne entsprechende Vorsichtsmaßnahmen mehrere Threads gleichzeitig schreibend auf den Cache zugreifen könnten. Der Rust-Compiler konnte dies erkennen und gab eine entsprechende Fehlermeldung aus. Ohne eine solche Erkennung wären zur Laufzeit möglicherweise die Messungen den falschen Orten zugeordnet worden. Dies zeigt, dass durch Rusts Regeln wirksam Fehler in Programmen aufgedeckt und verhindert werden können.

Um Multithreading-Fehler zu vermeiden wurde der Cache umgeschrieben, sodass ein Einfügen auch mit mehreren Threads korrekt funktioniert. Somit können die Lokalisierungen mithilfe des Caches parallel stattfinden. Ein Nachteil hierbei liegt darin, dass die Einträge im Cache nicht deterministisch festgelegt sind, da Threads, welche Messungen lokalisieren, unterschiedlich schnell arbeiten können. Da dies die Korrektheit der Ergebnisse nicht beeinflusst und bis auf minimale Performance-Schwankungen keinen Unterschied machen sollte, ist dies akzeptabel.

Der entwickelte Cache erlaubt es, rund 80% der verbleibenden Point-in-Polygon Tests zu eliminieren, welche nach der Filterung durch den R*-Tree noch durchgeführt werden müssten. Somit kann auf mehrere hundert Millionen Point-in-Polygon Tests verzichtet werden, wodurch die Performance der Lösung verbessert wird.

Auch wenn der Algorithmus extra für diese Arbeit entwickelt wurde, ist dieser so trivial, dass nicht auszuschließen ist, dass ein vergleichbarer Algorithmus bereits früher gefunden wurde. Marić et al. haben ebenfalls einen Cache entwickelt, welcher die

Ergebnisse zwischenspeichert. Der Algorithmus wird von den Autoren als Hashing der Koordinaten beschrieben [5]. Wie genau dieses Hashing funktioniert, wird in der Arbeit aber nicht im Detail erläutert. Aus dem Quellcode der Lösung ist aber ersichtlich, dass die Koordinaten der Messungen auf eine Nachkommastelle gerundet wurden und aus den gerundeten Werten ein Hash gebildet wird. Durch das Runden könnte es aber dazu kommen, dass eine Messung einem falschen Ort zugewiesen wird. Die Lösung von Marić et al. ist dementsprechend ungenau, während die in dieser Arbeit entwickelte Lösung immmer ein korrektes Ergebnis liefert.

4.2 Stream processing

Um Daten möglichst redundanzarm auszuwerten, müssen möglichst große Zeiträume auf einmal ausgewertet werden. Dies kann erreicht werden, indem Daten auf eine bestimmte Art aggregiert werden. Dieser Abschnitt erläutert zwei verschiedene Optimierungen, welche die Auswertung der Daten beschleunigen.

4.2.1 Fünf-Minuten-Aggregierung

Die Challenge sieht verschiedene Zeiträume zur Auswertung vor:

- Alle 5 Minuten soll eine Auswertung stattfinden
- Orte müssen in den letzten 10 Minuten vorhanden sein, um bei der Auswertung berücksichtigt zu werden
- Die Luftqualität zum Vergleich mit dem letzten Jahr wird über 5 Tage berechnet
- Für die 50 Orte mit der stärksten Verbesserung wird separat die Luftqualität der letzten 24 Stunden basierend auf p1 und p2 berechnet

Der kleinste gemeinsame Nenner dieser Zeiträume beträgt fünf Minuten. Es bietet sich daher an, möglichst viele Messungen in Fünf-Minuten-Schritten zu aggregieren. Dies ermöglicht es, anstatt jeder Messung nur jedes Fünf-Minuten-Fenster mehrfach auswerten zu müssen. Typischerweise enthält ein Fünf-Minuten-Fenster für jeden Ort drei oder vier Messungen, sodass davon ausgegangen werden kann, dass eine Auswertung basierend auf Fünf-Minuten-Fenstern etwa um diesen Faktor schneller ist, als jede Messung wiederholt auszuwerten.

Wie genau die Daten eines Streams aggregiert werden sollen, hängt allgemein davon ab, welche Daten bei der weiteren Verarbeitung benötigt werden. Hier wird zur Berechnung der Luftqualität die durchschnittliche Partikelkonzentration in der Luft (p1- und p2-Werte) benötigt. Es ist daher sinnvoll diese in Fünf-Minuten-Abschnitten

aufzusummieren. Mehrere dieser Fünf-Minuten-Aggregate können dann zusammengefasst werden, um sowohl die Luftqualität über 24 Stunden als auch über 5 Tage zu berechnen, ohne dass die einzelnen Messungen mehrfach aufsummiert werden müssen.

4.2.2 Rolling sum

Ein naiver Weg, die Partikelkonzentrationen zu aggregieren, besteht darin, für jedes Fünf-Minuten-Fenster alle Daten der letzten fünf Tage zu aggregieren. Dies ist jedoch äußerst ineffizient, da hierfür 1440 Fenster ausgewertet werden müssten. Es lässt sich hier jedoch ausnutzen, dass die Addition der einzelnen Fünf-Minuten-Fenster kommutativ ist.

Somit kann die Auswertung beschleunigt werden, indem den Summen ein neues Fenster hinzuaddiert und das älteste Fenster subtrahiert wird. Dies ermöglicht es, nur zwei Additionen für das Fünf-Tage-Fenster anstatt der 1440 Additionen durchführen zu müssen, welche aus dem naiven Ansatz resultieren würden.

Zum besseren Verständnis ist solch eine Operation in 6 dargestellt. Angenommen, die Summe für das 5-Tage Fenster zum Zeitpunkt `n` ist bekannt. Dann kann der Wert für `n+1` berechnet werden, indem von der Summe der älteste Wert des alten Fensters, in der Abbildung rot markiert, abgezogen und stattdessen der neue Wert (grün markiert) dazuaddiert wird. Das Ergebnis ist nun dasselbe, wie wenn die Fünf-Minuten-Aggregate von `n-1438` bis `n+1` aufsummiert werden.

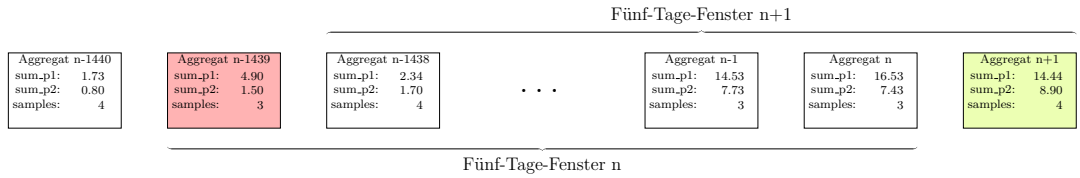

Abbildung 6: Optimierte Aggregierung von mehreren Fünf-Minuten-Aggregaten über fünf Tage. Bei der Auswertung des zweiten Fensters wird das rot markierte Aggregat abgezogen und das grün markierte dazuaddiert.

Problem Fließkommazahlen-Präzision

In der Mathematik ist Addition und Subtraktion kommutativ. Aufgrund der internen Repräsentation von Kommazahlen gilt bei den Computern keine Kommutativität. Es kann daher abhängig von der Reihenfolge der Addition und Subtraktion zu Rundungsfehlern und somit zu unterschiedlichen Ergebnissen kommen [19].

Rundungsfehler werden besonders dann problematisch, wenn viele Zahlen unterschiedlicher Größenordnungen verrechnet werden. Dies wird bei der optimierten Berechnung des Mittelwerts über fünf Tage sichtbar, da die kleinen p1- und p2-Werte von 1440 Fünf-Minuten-Aggregaten aufsummiert werden, woraus eine große Summe

entsteht. Da bei der Berechnung des Mittelwerts der nächsten Fenster von dieser Summe nur einzelne Fünf-Minuten-Aggregate und somit kleine Beträge abgezogen und dazuaddiert werden, kann es hier zu Rundungsfehlern kommen.

Den Rundungsfehlern und dessen Auswirkungen auf den Mittelwert kann ein Stück weit entgegengewirkt werden, indem Floating-Point-Register mit mehr Bits verwendet werden. Durch die Verfügbarkeit einer höheren Anzahl Bits kann eine größere Präzision erreicht werden.

4.3 Allgemeine Optimierungen

Die in den vorigen Abschnitten erläuterten Optimierungen sind recht spezifisch und für die Entwicklung anderer Software nur eingeschränkt hilfreich. In diesem Abschnitt sind zwei einfache Optimierungen beschrieben, die auch in vielen anderen Programmen eingesetzt werden könnten.

4.3.1 Parallelisierung

Der funktionale Programmierstil, in dem die Lösung implementiert wurde, macht es einfach, die Verarbeitung der Daten zu parallelisieren, soweit es möglich ist. Hierzu wird die Rust-crate rayon [2] genutzt. Diese implementiert die meisten funktionalen Methoden und Strukturen der Rust-Standardbibliothek neu und behält, wo möglich, die Schnittstellen bei. Dabei parallelisiert rayon den Code so stark wie möglich. Somit kann mithilfe minimaler Änderungen viel Code parallelisiert werden.

Rayon erkennt automatisch die Anzahl der CPUs, die genutzt werden können und startet die entsprechende Anzahl Threads. Sobald Daten parallel ausgewertet werden sollen, verteilt rayon automatisch die Daten auf die Threads, welche die Daten dann parallel verarbeiten.

Diese Vorgehensweise wird vor allem genutzt, um die Messungen zu lokalisieren. Außerdem wird so die Aggregierung der Werte für jeden Ort parallel durchgeführt.

4.3.2 Optimierung durch den Compiler

Wie viele C und C++-Compiler stellt auch der Rust-Compiler Optionen zur Verfügung, die festlegen, wie stark der generierte Code optimiert wird. Damit der Code überhaupt optimiert wird, sollten Programme mit `cargo --release` kompiliert werden.

Neben dem Aufruf des Compilers gibt es noch weitere Möglichkeiten, um schnelleren Code zu generieren. In der Konfigurationsdatei des Rust-Projektes "`Cargo.toml`" können weitere Optionen hinzugefügt werden. Eine wichtige Option stellt dabei die

[2] https://github.com/rayon-rs/rayon

Link Time Optimization (LTO) dar. Bei aktivierter LTO werden noch bei der Erstellung des finalen Programms aus den verschiedenen Software-Bibliotheken verschiedene Optimierungen durchgeführt. Dabei werden z.B. nicht benötigte Funktionen entfernt und wenig genutzte Funktionen zusammengefasst (inlining).

All diese Optimierungen durch den Compiler haben jedoch einen Preis: Der Compiler benötigt bei jedem Kompiliervorgang Zeit, um die gewählten Optimierungen durchzuführen. Dies führt somit zu einer längeren Kompilierzeit.

4.4 Mögliche, aber nicht realisierte Optimierungen

Nicht alle erkannten Möglichkeiten zur Optimierung des Programms wurden auch genutzt. Im Folgenden wird dargelegt, welche weiteren Optimierungen theoretisch möglich wären, aber nicht umgesetzt wurden. Diese Optimierungen wurden zwar in Betracht gezogen, aber aus unterschiedlichen Gründen verworfen. Die Performance der entwickelten Software könnte durch die Umsetzung der hier beschriebenen Optimierungen noch weiter verbessert werden.

4.4.1 Verwendung eines anderen Point-in-Polygon-Algorithmus

Wie in Abschnitt 4.1.1 beschrieben ist der Algorithmus, welcher das Point-in-Polygon Problem löst, zwar einfach, aber äußerst langsam. Indem die Anzahl der Nutzungen dieses Algorithmus reduziert wurden, konnte die Performance des Programms erheblich verbessert werden. Dennoch benötigt die Funktion, welche die Position des Punkts relativ zu einem Polygon testet, trotz aller Optimierungen unverhältnismäßig viel Laufzeit. In der finalen Version entfiel laut `perf` (einem Profiling-Tool) rund 18% der Laufzeit des Programms auf diese Funktion.

Es würde sich deshalb anbieten, den Algorithmus auszutauschen und durch einen effizienteren wie den von Ali et al. zu ersetzen. Die Autoren messen in ihren Benchmarks eine zehnmal höhere Verarbeitungsgeschwindigkeit gegenüber dem einfachen Algorithmus [16]. Falls diese Verarbeitungsgeschwindigkeit auch hier sichtbar wird, so könnte das Programm die Messungen um rund 16% schneller verarbeiten.

Aufgrund des Zeit- und Implementierungsaufwands sowie einer hohen Komplexität wurde jedoch darauf verzichtet, den Algorithmus zu implementieren.

4.4.2 Verarbeitung auf mehreren Computern

Einige Operationen, wie die Lokalisierung der Messungen und die Aggregierung der Fünf-Minuten-Aggregate für jeden Ort, werden parallel auf mehreren Prozessoren durchgeführt. Hier stellt sich die Frage, ob die Performance des Programms verbessert werden kann, indem die Verarbeitung auf mehrere Computer verteilt wird. Dabei muss

jedoch bedacht werden, dass die Verteilung auf mehrere Computer einen Overhead mit sich bringt, da Daten zwischen den Computern verschickt werden müssen. Dieser entsteht sowohl aus der Netzwerk-Kommunikation als auch aus dem Aufwand, die benötigten Daten zu serialisieren und zu deserialisieren.

Dass dieser Overhead problematisch werden kann, zeigt sich auch daran, dass Marić et al. in ihrer Arbeit als eine ihrer Optimierungen die Reduktion der Anzahl der Serialisierungen und Deserialiserungen nennen [5].

Um diesen Overhead zu vermeiden, bietet es sich an, die Daten möglichst früh zu verteilen - noch bevor die Messungen lokalisiert werden.

Eine Möglichkeit hierfür könnte darin bestehen, die zu analysierende Fläche (Deutschland) entlang der x- und/oder y-Achse aufzuteilen und jedem Computer einen Teil zuzuweisen. Um eine fehlerhafte Berechnung zu vermeiden, müsste hierbei jedoch besonders auf Orte geachtet werden, welche sich in mehreren zu analysierenden Flächen befinden.

Durch den benötigten Overhead zur Verteilung ist unklar, ob die Performance des Programms durch eine Verteilung auf mehrere Computer tatsächlich verbessert werden kann. Zudem wäre eine Umsetzung dieser Idee mit einem nicht unerheblichen Implementierungsaufwand verbunden. Neben dem Implementierungsaufwand würde zum Testen der Lösung ein Netzwerk von Computern mit einer Kommunikationsgeschwindigkeit im Gigabit-Bereich benötigt werden. Aus diesen Gründen wurde auf die Umsetzung dieses Konzepts verzichtet.

5 Plausibilitätsprüfung

Um sicher zu gehen, dass die Ergebnisse des entwickelten Programms korrekt sind, wurde eine zweite Lösung implementiert, um die Ergebnisse zu vergleichen. Unter der Annahme, dass die Fehler unabhängig sind, sollte die Wahrscheinlichkeit, dass beide Lösungen Fehler enthalten und dennoch das gleiche Ergebnis zurückliefern, äußerst niedrig sein. Zur Umsetzung wurde SQL in Form von PostgreSQL in Kombination mit PostGIS verwendet.

5.1 Umsetzung der SQL-basierten Lösung

Um mit dem gleichen Datensatz arbeiten zu können, muss dieser zunächst in PostgreSQL importiert werden. Da dieser nur im Protobuf-Format vorliegt, mussten hierzu Programme entwickelt werden, welche die Daten in ein Format konvertieren, das PostgreSQL einlesen kann. Hierzu musste zunächst ein Schema entwickelt werden. Das Schema für die Messungen besteht aus zwei Tabellen: Eine für die Daten des aktuellen Jahres und eine für die Daten des letzten Jahres. [1] Die Konvertierung muss sowohl für die Batches der Messungen als auch für den Datensatz, welcher die Orte mit Polygonen beschreibt, durchgeführt werden. Der Einfachheit halber wurden hierzu SQL-Befehle in eine Datei geschrieben und später mittels Include-Befehlen von der PostgreSQL-shell (psql) importiert. Um die Auswertung zu beschleunigen, wurde nur ein Teil der Daten importiert: Die ersten 10.000 Batches, also 10% des gesamten Datensatzes.

Nachdem der zu analysierende Datensatz importiert wurde, wurden Indices auf den einzelnen Tabellen angelegt, um die nachfolgenden Queries zu beschleunigen.

Zum Vergleichen der Ergebnisse wurde eine SQL-Query formuliert, welche für einen bestimmten Zeitpunkt die Rangliste erstellt. Das bedeutet, dass die Daten eines Fünf-Tages-Fenster aggregiert und mit dem des Vorjahres verglichen werden. Außerdem muss die durchschnittliche Luftqualität über die letzten 24 Stunden berechnet werden. Die Lokalisierung der Messungen ist dank PostGIS einfach: Mithilfe eines Joins auf die Tabelle, welche die Geometrie der Orte enthält, lässt sich schnell der Ortsname feststellen. Schwieriger ist es, die Luftqualität zu berechnen. Hierfür wurde eine

[1] Es hätte auch ein Flag genutzt werden können, um zwischen Messungen des aktuellen und des letzten Jahres unterscheiden zu können. Theoretisch hätte sogar allein der Zeitstempel zur Unterscheidung ausgereicht. Das Speichern in verschiedenen Tabellen erhöht aber die Lesbarkeit der Queries.

Stored Procedure entwickelt, welche aus den durchschnittlichen p1- und p2-Werten der Messungen die Luftqualität berechnet.

Die Query benötigt zur Erstellung einer einzigen Rangliste etwa 10 Minuten - länger als das in dieser Arbeit entwickelte Programm zum Analysieren des gesamten Datensatzes braucht. Da das Ziel darin bestand, möglichst einfach die Ergebnisse auf Plausibilität zu testen, ist solch eine Geschwindigkeit akzeptabel. Mehr als Stichproben können mit dieser Performance jedoch nicht realisiert werden.

Diese Erfahrung ähnelt der von Jacobs Adams in seiner Arbeit "The Pathologies of Big Data" [20]. Dort beschreibt er, wie eine Auswertung eines großen Datensatzes in PostgreSQL verglichen mit seinem C-Programm äußerst langsam ist, und diskutiert verschiedene Gründe dafür. Als Grund nennt Adams unter anderem die Performance-Einbußen, welche bei einem nicht-sequentiellen Zugriff auf Arbeitsspeicher und Festplatte/SSD entstehen. PostgreSQL muss viele Daten von der Festplatte einlesen, durchsuchen und Joins erzeugen, um ein Ergebnis zu generieren. Das von ihm entwickelte C-Programm dagegen konnte bei der Implementierung die Daten so strukturieren, dass der Datensatz so schnell wie möglich eingelesen und verarbeitet wird.

Wie bei Adams ist auch in dieser Arbeit die Performance von PostgreSQL der des selbst entwickelten Programms weit unterlegen. Dies liegt vermutlich daran, dass die entwickelte Lösung stark auf die Challenge zugeschnitten ist. PostgreSQL dagegen stellt viele verschiedene Features bereit, was es jedoch schwerer macht, geeignete Optimierungen zu finden und umzusetzen.

Insgesamt sind die Ergebnisse der beiden Lösungen plausibel - der Unterschied zwischen den zwei Implementationen beträgt bei den Stichproben höchstens 1/10000, was vermutlich durch Rundungsfehler zustande kommt.

5.2 Vergleich mit anderen Arbeiten

Die Plattform der Veranstalter, welche die Daten zur Verfügung stellt und Benchmarks durchführt, überprüft nach Angaben der Organisatoren die empfangenen Ergebnisse. Nur wenn sie korrekt sind, soll auch eine Bewertung der Ergebnisse stattfinden [2]. Demnach ist anzunehmen, dass die Teilnehmer der Grand Challenge bei der Auswertung der Messungen darauf geachtet haben, dass die Ergebnisse überwiegend korrekt sind.

In der Praxis scheint bei der Überprüfung der Ergebnisse jedoch ein gewisser Spielraum zu existieren. Dies wird unter anderem daraus ersichtlich, dass Marić et al. in ihrer Arbeit eine Optimierung erwähnen, welche ein schnellere Verarbeitung erlaubt aber zu kleinen Abweichungen führt, sodass die Lösung nur noch zu 96% genau ist [5]. Auch Morcos et al. merken in ihrer Arbeit an, dass es bei ihrer Lösung zu Abwei-

chungen von ca. 19% bei der Berechnung der Luftqualität kommen kann [6].

Diese Varianzen in der Berechnung machen es unmöglich, die Ergebnisse der verschiedenen Lösungen zu vergleichen.

5.3 Vergleich mit Werten der Organisatoren

Eine Alternative zum Vergleich mit anderen Arbeiten hätte eventuell darin bestehen können, die Plattform der Organisatoren zu nutzen, um die Ergebnisse zu verifizieren.

Da die Grand Challenge zum Zeitpunkt der Fertigstellung des Programms bereits beendet war, existierte aber keine Möglichkeit, die API zu nutzen um die entwickelte Lösung zu testen.

Die Organisatoren haben den Quellcode der Plattform veröffentlicht [2], daher sollte es mithilfe des Codes des gRPC-Servers möglich sein nachzuvollziehen, wie die Organisatoren die Luftqualität berechnet haben. Dieser Ansatz ließ sich jedoch ebenfalls nicht umsetzen: Der Server-Code beinhaltet zwar die Funktionalität, um die Latenz und Verarbeitungsdauer zwischen Abruf des Batches und Übersendung der Ergebnisse zu berechnen. Eine Validierung der Ergebnisse ist im Server-Code aber nicht zu finden.

Im git-Repository liegt aber auch eine zip-Datei mit Ergebnissen, die zum Vergleich zur Verfügung gestellt wurden. Im Vergleich mit den Ergebnissen der entwickelten Lösungen wurde festgestellt, dass die Ergebnisse ähnlich sind, da die Liste der besten Orte größtenteils übereinstimmt. Leider lässt sich ohne den Code, mit welchem die Ergebnisse berechnet wurden, nicht feststellen, wie die abweichenden Ergebnisse zustande kommen.

[2] unter https://github.com/jawadtahir/bandency

6 Ergebnisse

In diesem Kapitel wird die in dieser Arbeit entwickelte Lösung vorgestellt. Um die Performance der entwickelten Lösung besser einschätzen zu können, wurden neben der eigenen Lösung auch die Performance der anderen Lösungen gemessen. Somit kann ein Vergleich mit den Lösungen anderer Teilnehmer durchgeführt werden. Abschließend werden die Vor- und Nachteile der entwickelten Lösung dargelegt.

6.1 Vergleich mit anderen Lösungen

Von den drei veröffentlichten wissenschaftlichen Arbeiten haben zwei ihren Code veröffentlicht [5, 6]. Die Arbeit von Rim Moussa beansprucht eine äußerst geringe Verarbeitungsgeschwindigkeit und bleibt daher im folgenden unbeachtet [7].

Der Code beider Arbeiten ähnelt sich in vielen technischen Aspekten: Beide Arbeiten sind in Java geschrieben. Um die Messdaten zu verteilen und zu verarbeiten wurde Apache Flink genutzt. Zum Abrufen der Daten werden Googles gRPC- und Protobuf-Bibliotheken eingesetzt. Das Kompilieren und die Verwaltung der Bibliotheken funktionert mithilfe von Maven.

Um einen Vergleich der Performance zwischen den Lösungen der anderen Teilnehmer und der in dieser Arbeit entwickelten Lösung zu ermöglichen, wurden die Messungen mithilfe der beiden Lösungen ausgewertet. Dabei wurde gemessen, wie lange das Verarbeiten der Daten dauert.

Da die Lösungen darauf ausgelegt sind, die Messungen vom Server der DEBS Grand Challenge zu empfangen, mussten die Lösungen leicht abgewandelt werden, sodass sie den lokal gespeicherten Datensatz verarbeiten. Hierzu wurde Java-Code entwickelt, welcher die Batches aus den Dateien lädt und mittels der verwendeten Protobuf-API deseralisiert. Da der Code beider Arbeiten auf denselben Software-Frameworks aufbaut, konnte derselbe Code für beide Lösungen verwendet werden.

Abgesehen vom Hinzufügen des Codes zum Laden der Batches wurden die Lösungen nicht weiter modifiziert. Der Code der beiden Lösungen konnte nun mit Maven kompiliert und anschließend mit Apache Flinks Kommandozeileninterfaces ausgeführt werden.

6.1.1 Vergleich der Verarbeitungsgeschwindigkeit

Bei der Messung der Verarbeitungsgeschwindigkeit zeigte sich, dass die beiden Lösungen sich trotz ähnlicher Technologien um mehrere Größenordnungen in ihrer Performance unterschieden. Die Performance der unterschiedlichen Lösungen ist in Tabelle 1 dargestellt.

Beim Vergleich der Performance der Lösungen muss allerdings darauf hingewiesen werden, dass im Rahmen dieser Arbeit nur die Query 1 der Challenge bearbeitet wurde. Die Lösungen der Teilnehmer der DEBS Grand Challenge bearbeiten gleichzeitig noch die Query 2.

Lösung	Diese Arbeit	Marić et al. [5]	Morcos et al. [6]	Moussa [7]
	Nach eigenen Angaben in der veröffentlichten wissenschaftlichen Arbeit			
Anzahl ausgewerteter Batches	-	20.000	330.000	22
Laufzeit	-	ca. 1m50s	ca. 17m	ca. 15m30s
Messungen/s	-	1.800.000	3.200.000[a]	ca. 236[b]
CPU	-	2xIntel(R) Core(TM) i7-9700K @ 3.60GHz	1xRyzen 9 5900x[c]	???
	Messung auf Laptop mit 16GiB RAM und AMD Ryzen 7 5700U			
Anzahl ausgewerteter Batches	100.000	30.000	124	-
Laufzeit	6m7s	60m46s	1h10m10s	-
Messungen/s (ca.)	2.720.000	823.000	70	-
	Messung auf Server mit 128GiB RAM und 2x Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz			
Anzahl ausgewerteter Batches	100.000	30.000	10.000	-
Laufzeit	6m13s	29m29s	ca. 10h [d]	-
Messungen/s (ca.)	2.680.000	170.000[e]	ca. 278	-

[a] Dies entspricht etwa dem zweifachen der maximalen Verarbeitungsgeschwindigkeit, welche nach Angaben der Organisatoren theoretisch erreichbar ist.
[b] Ergibt sich aus 22 Batches in 15.5 Minuten, allerdings reichen 22 Batches nicht aus, um einen Fünf-Tages-Vergleich zu ermöglichen.
[c] Nicht in Arbeit angegeben, Information nach E-Mail-Kommunikation mit einem der Autoren erhalten.
[d] Nachdem nur 10000 Batches in etwa 10 Stunden verarbeitet wurden, wurde die Messung abgebrochen.
[e] Bei einer Messung mit 100.000 Batches wird nur 1/6 der Verarbeitungsgeschwindigkeit erreicht.

Tabelle 1: Performance-Vergleich zwischen den einzelnen Arbeiten

Bei der Auswertung fiel der langsame Code von Morcos et al. auf [6]. Um die Ursache der Performance-Abweichungen zu finden, wurden die Autoren der Arbeit angeschrieben. Da die Arbeit von Marić et al. ebenfalls auf Apache Flink aufbaut aber keinerlei Performance-Probleme zeigt, erscheint es unwahrscheinlich, dass eine fehlerhafte Apache Flink-Konfiguration die Ursache des Performance-Problems ist.

Trotz E-Mail-Kommunikation mit einem der Autoren der Arbeit von Morcos et al. konnte nicht abschließend geklärt werden, wieso ihr Code mehr als 10.000 Mal langsamer ist als in ihrer Arbeit angegeben. Aus einem Video[1], in dem Morcos et al. ihre Arbeit präsentieren ist aber anhand der Statistiken von Apache Flink ersichtlich, dass die Software auf den Computern der Autoren eine ähnliche Verarbeitungsgeschwindigkeit aufwies.

Neben der Gesamt-Performance der Lösungen ist ebenfalls interessant, wie diese sich bei der Verarbeitung entwickelt. Die Batches werden der Reihe nach verarbeitet. Indem gemessen wird, wann das nächste Batch geladen wird, lässt sich somit einfach visualisieren, wie schnell die Batches verarbeitet werden können. Dies ist in Abbildung 7 dargestellt.

[1] https://youtu.be/HNQPQ0e5Yww?t=9m28s

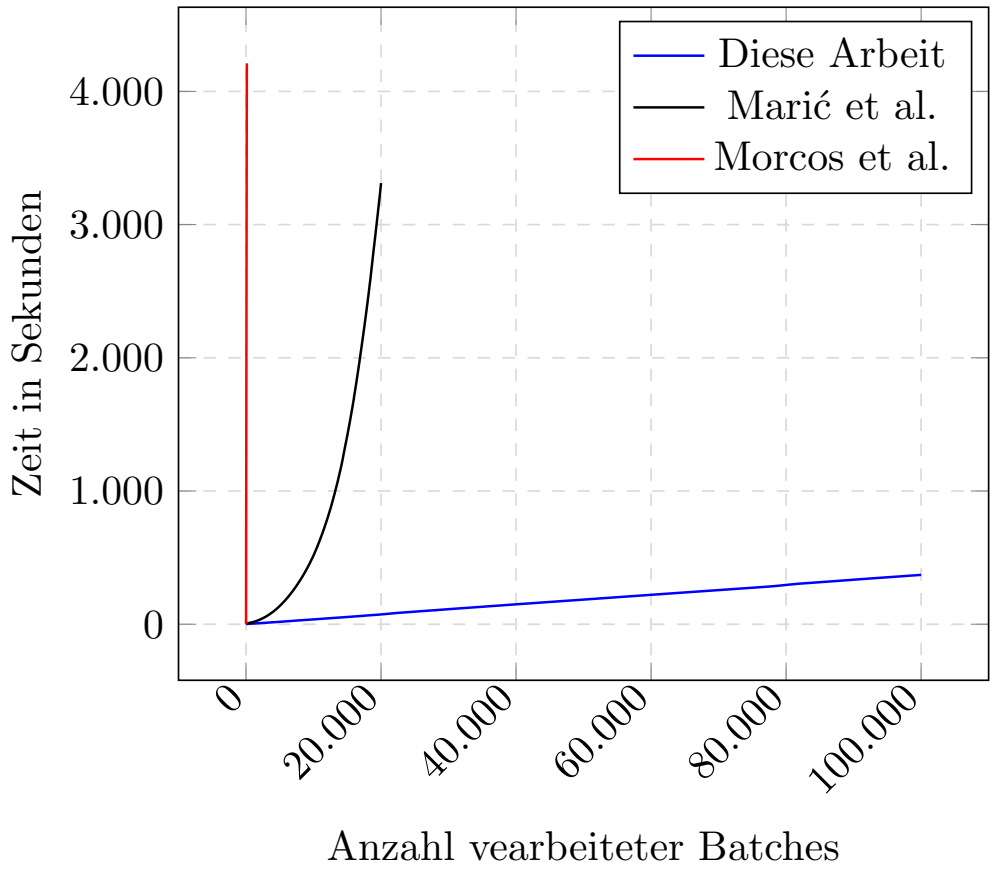

Abbildung 7: Visualisierung der Performance der Lösungen. In blau die in dieser Arbeit entwickelte Lösung. Schwarz: Marić et al. [5]. Rot: Morcos et al. [6].

Die Abbildung zeigt, wie viel Zeit benötigt wird um eine bestimmte Anzahl an Batches zu verarbeiten. Je flacher die Linie, desto schneller kann die Lösung die Daten verarbeiten. Wie in der Abbildung zu sehen, verarbeitet die in dieser Arbeit entwickelte Lösung alle 100.000 Batches innerhalb weniger Minuten, während die anderen Lösungen wesentlich langsamer sind. Die Laufzeit der Lösung von Morcos et al. (in der Abbildung rot) steigt so stark mit der Anzahl Batches, dass kaum erkennbar ist, dass überhaupt Fortschritt erzielt wird.

Ein besonders interessanter Verlauf ist bei der Darstellung der Lösung von Marić et al. zu erkennen: Hier werden die Daten zunächst schnell geladen, doch die Verarbeitungsgeschwindigkeit nimmt immer weiter ab.

6.1.2 Vergleich des Speicherverbrauchs

Neben der Verarbeitungszeit der Lösungen erscheint auch ein weiterer Punkt interessant: Der Speicherverbrauch der Lösungen. Ein hoher Speicherverbrauch kann ein Hinweis darauf sein, dass mehr Ressourcen als nötig im Speicher behalten werden,

sodass die Performance der Software möglicherweise negativ beeinflusst wird.

Das in dieser Arbeit entwickelte Rust-Programm benötigt während der Auswertung aller 100.000 Batches nur maximal 927MiB an RAM [2]. Der Speicherverbrauch steigt langsam, da immer mehr Cache-Einträge hinzukommen. Letztendlich wird der maximale RAM-Verbrauch aber durch die feste Maximalgröße des Caches begrenzt.

Den Speicherverbrauch von Java-Programmen zu messen ist nicht einfach. Beim Start von Apache Flink muss in einer Konfigurationsdatei spezifiziert werden, wie viel Speicher von der Java Virtual Machine (JVM) genutzt werden darf. Da nicht bekannt ist, wie viel Speicher eine Lösung benötigt, muss daher großzügig provisioniert werden, sodass dem Programm nicht der Speicher ausgeht. Erschwerend kommt hinzu, dass die JVM selbst Speicher benötigt um zu arbeiten, sodass zwischen dem Speicherverbrauch der JVM selbst und dem Speicherverbrauch der Anwendung unterschieden werden muss. Mittels der Nutzung der REST-API [3] von Apache Flink kann ein Wert von der JVM berechnet werden, welcher den tatsächlichen Speicherverbrauch angibt.

Mithilfe dieser API zeigt sich, dass die Lösung von Marić et al. bei der Analyse von 30.000 Batches ca 2.4 GiB an RAM benötigt. Hierbei ist der Overhead der JVM nicht eingerechnet.

Allerdings steigt der RAM-Bedarf des Programms immer weiter an, während die Performance der Software immer weiter abnimmt. Wenn der gesamte Datensatz mit 100.000 Batches ausgewertet wird, so werden ca 9.2GiB an RAM benötigt. Die Auswertungszeit für die etwa 3,3-fache Menge an Daten beträgt mehr als das 20-fache (über 10 Stunden). Mutmaßlich liegt dies daran, dass der verwendete Cache zur Lokalisierung der Messungen unbegrenzt wächst, womit sowohl ein höherer Speicherverbrauch als auch eine niedrigere Verarbeitungsgeschwindigkeit einhergeht. In ihrer Arbeit geben Marić et al. an, dass mit zwei Computern eine Verarbeitungsgeschwindigkeit von 1.8 Millionen Messungen pro Sekunde erreicht wird und geben für einen Computer etwa die doppelte Verarbeitungszeit an [5]. Eine entsprechende Performance auf einem Computer konnte nicht reproduziert werden. Die Messungen von Marić et al. basieren allerdings auf einer Auswertung von 20.000 Batches. Da die Verarbeitungsgeschwindigkeit mit steigender Batchanzahl abnimmt, könnte dies die Diskrepanz erklären.

Auf jeden Fall ist die in dieser Arbeit entwickelte Lösung der Lösung von Marić et al. sowohl im Speicherverbrauch als auch in der Verarbeitungsgeschwindigkeit überlegen. Dabei ist die Lösung von Marić et al. mit Abstand die schnellste der Teilnehmer der Grand Challenge, womit die in dieser Arbeit entwickelte Lösung hinsichtlich der Verarbeitungsgeschwindigkeit alle Lösungen der Teilnehmer der DEBS Grand Challenge übertrifft.

[2] Gemessen mit `time -v`

[3] http://localhost:8081/jobmanager/metrics?get=Status.JVM.Memory.Heap.Used

6.2 Vor- und Nachteile der entwickelten Lösung

Die entwickelte Lösung kann wie gefordert die Query 1 der DEBS Grand Challenge auswerten. Die entwickelte Lösung unterscheidet sich dabei jedoch stark von den Lösungen der anderen Arbeiten.

Einer der Nachteile der entwickelten Lösung besteht darin, dass die Query 2 der DEBS Grand Challenge nicht umgesetzt wurde. Autoren anderer Lösungen der DEBS Grand Challenge haben jedoch in ihren Lösungen beide Queries bearbeitet. Es ist daher nicht einfach abzuschätzen, um wie viel schneller die entwickelte Lösung ist.

Dennoch ist klar, dass die entwickelte Lösung um ein vielfaches schneller als die Lösungen der anderen Teilnehmer ist. Während Maric et al. ihren eigenen Angaben zufolge mit 2 Computern mit jeweils 8 physischen Cores eine Geschwindigkeit von ca. 1,8 Millionen Messungen pro Sekunde erreichen, erreicht der in dieser Arbeit entwickelte Code etwa eine Millionen zusätzliche Messungen pro Sekunde auf einem vergleichbaren System. Zudem nimmt die Performance der Lösung von Maric et al. immer weiter ab, während die Verarbeitungsgeschwindigkeit der Rust-basierten Lösung konstant bleibt.

Die bessere Performance wurde durch eine Vielzahl an Optimierungen erreicht, die hauptsächlich die Lokalisierung der Messungen beschleunigen. Hieraus ergibt sich ein Vorteil dieser Lösung, der mit moderatem Aufwand auf Lösungen der Teilnehmer der Grand Challenge übertragen werden kann.

In Hinblick auf die Optimierungen ist auch zu beachten, dass Marić et al. einige Optimierungen auf Kosten der Genauigkeit vorgenommen haben (vgl. Ende von Kapitel 4.1.3). Die in dieser Arbeit entwickelte Lösung dagegen achtet auf Präzision und versucht möglichst genaue Werte zu ermitteln.

Ein möglicher Nachteil der entwickelten Lösung ist es, dass diese in dieser Form nicht horizontal skalierbar ist. Es wäre theoretisch möglich, die Auswertung auf mehrere Computer zu verteilen (vgl. Kapitel 4.4.2). Es ist aber fraglich, ob dies die Verarbeitungsgeschwindigkeit weiter erhöhen würde. Da jedoch unklar ist, ob die Lösung durch eine Verteilung tatsächlich verbessert wird, kann dieser Nachteil auch als eine Abwägung zwischen Effizienz und Skalierbarkeit der Lösung angesehen werden.

Die Lösung in dieser Arbeit wurde ohne ein Framework umgesetzt, welches auf Stream processing ausgerichtet ist. Dies hat sowohl Vor- als auch Nachteile. Einerseits ermöglicht dies eine minimale und simple Lösung, welche wenig Overhead besitzt und eine hohe Verarbeitungsgeschwindigkeit erreicht. Andererseits besitzen Streaming Frameworks wie Apache Flink zusätzliche Fähigkeiten. Diese ermöglichen es unter anderem, die Auswertung der Daten auf mehrere Computer zu verteilen, die Auswertung nach Belieben zu unterbrechen, später fortzusetzen und trotzdem jedes Batch nur einmal auszuwerten (s. Kapitel 1.2.2).

Diese zusätzlichen Fähigkeiten wurden jedoch in der Challenge nicht benötigt. Daher genügt die entwickelte Software zur Lösung der Query 1 der DEBS Grand Challenge 2021. Eine Ergänzung der Software um die genannten zusätzlichen Fähigkeiten wäre jedoch mit nicht unerheblichem Entwicklungsaufwand verbunden.

7 Fazit

Im Rahmen der Arbeit wurde eine Software entwickelt, welches die Query 1 der DEBS Grand Challenge 2021 löst. Im Gegensatz zu den Lösungen der Teilnehmer der Challenge ist die entwickelte Lösung darauf ausgerichtet, auf einem einzigen Computer effizient und performant zu arbeiten.

Die Leistungsfähigkeit der entwickelten Lösung wurde durch verschiedene Optimierungen erreicht, wobei eine davon eigens für diese Challenge entwickelt wurde.

Bei der Entwicklung der Lösung hat sich gezeigt, dass Rust eine geeignete Wahl ist, um ein System zu entwickeln, welches die Daten der Challenge verarbeiten kann. Durch Rusts Programmiermodell wurden verschiedene Bugs erkannt und verhindert (vgl. Kapitel 4.1.3). Dabei ermöglichte Rust durch dessen Zero-Cost abstractions eine hohe Performance und eine überwiegend einfache Entwicklung. Das Fehlen eines Stream Processing Frameworks führte dazu, dass grundlegende Funktionalität selbst umgesetzt werden musste (s. Kapitel 3.3.3).

Durch die genutzten Optimierungen und der Schnelligkeit von Rust wird im Vergleich mit anderen Lösungen (s. Kapitel 6.1.1) auf derselben Hardware eine etwa zehnmal höhere Anzahl an Messungen pro Sekunde erreicht. Ein Nachteil der entwickelten Lösung ist, dass sie nicht ohne weiteres auf mehrere Computer skaliert werden kann (vgl. Kapitel 4.4.2). Aufgrund der höheren Performance ist dies aber auch nicht nötig: Die Messungen können mit der entwickelten Lösung etwa 20.000 mal schneller als in Echtzeit verarbeitet werden.

Die Organisatoren der DEBS Grand Challenge fordern bei der Analyse der Messungen nur eine Auswertung für alle Orte innerhalb Deutschlands. Mit der Performance der entwickelten Lösung sollte es möglich sein, auch größere Mengen an Daten zu verarbeiten. So könnte z.B. die Auswertung auf Orte weltweit ausgedehnt werden. Falls die Perforance der Lösung noch weiter verbessert werden soll, so kann hierbei auf eine der Optimierungen aus Kapitel 4.4 zurückgegriffen werden. Bei der Umsetzung wäre ein in Rust geschriebenes Framework wie z.B. amadeus [1] zwar hilfreich, sobald solch ein Framework einsatzbereit ist. Bis es so weit ist, hat diese Arbeit aber gezeigt, dass es auch ohne ein Framework praktikabel ist, Daten effizient zu verarbeiten, wenn geeignete Algorithmen eingesetzt werden um die verfügbare Hardware möglichst stark auszunutzen.

[1] https://constellation.rs/amadeus

Literaturverzeichnis

[1] U.S. Environmental Protection Agency. (2018) Technical assistance document for the reporting of daily air quality – the air quality index (aqi). [Online]. Available: https://www.airnow.gov/sites/default/files/2020-05/aqi-technical-assistance-document-sept2018.pdf

[2] J. Tahir, C. Doblander, R. Mayer, S. Frischbier, and H.-A. Jacobsen, "The debs 2021 grand challenge: Analyzing environmental impact of worldwide lockdowns," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 136–141. [Online]. Available: https://doi.org/10.1145/3465480.3467836

[3] DEBS Grand Challenge Organizers. (2021) Call for grand challenge solutions. [Online]. Available: https://2021.debs.org/call-for-grand-challenge-solutions/

[4] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, ser. HOTOS'15. USA: USENIX Association, 2015, p. 14.

[5] J. Marić, K. Pripužić, and M. Antonić, "Debs grand challenge: Real-time detection of air quality improvement with apache flink," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 148–153. [Online]. Available: https://doi.org/10.1145/3465480.3466930

[6] M. Morcos, B. Lyu, and S. Kalathur, "Solving the 2021 debs grand challenge using apache flink," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 142–147. [Online]. Available: https://doi.org/10.1145/3465480.3466929

[7] R. Moussa, "Scalable analytics of air quality batches with apache spark and apache sedona," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 154–159. [Online]. Available: https://doi.org/10.1145/3465480.3466931

[8] H. Xu, Z. Chen, M. Sun, and Y. Zhou, "Memory-safety challenge considered solved? an empirical study with all rust cves," *CoRR*, vol. abs/2003.03296, 2020. [Online]. Available: https://arxiv.org/abs/2003.03296

[9] C. N. Steve Klabnik, *The Rust Programming Language 2018*. No Starch, 2019.

[10] Z. Yu, L. Song, and Y. Zhang, "Fearless concurrency? understanding concurrent programming safety in real-world rust software," *CoRR*, vol. abs/1902.01906, 2019. [Online]. Available: http://arxiv.org/abs/1902.01906

[11] T. Akidau, S. Chernyak, and R. Lax, *Streaming Systems*. O'Reilly, 2018.

[12] gRPC maintainers. (2021) Faq | grpc. [Online]. Available: https://grpc.io/docs/what-is-grpc/faq/

[13] R. Nigam, K. Pandya, A. J. Luis, R. Sengupta, and M. Kotha, "Positive effects of COVID-19 lockdown on air quality of industrial cities (ankleshwar and vapi) of western india," *Scientific Reports*, vol. 11, no. 1, Feb. 2021. [Online]. Available: https://doi.org/10.1038/s41598-021-83393-9

[14] A. Shareef and D. R. Hashmi, "Impacts of COVID-19 pandemic on air quality index (AQI) during partial lockdown in karachi pakistan," *Journal of Health and Environmental Research*, vol. 6, no. 3, p. 93, 2020. [Online]. Available: https://doi.org/10.11648/j.jher.20200603.17

[15] M. Shimrat, "Algorithm 112: Position of point relative to polygon," *Commun. ACM*, vol. 5, no. 8, p. 434, Aug. 1962. [Online]. Available: https://doi.org/10.1145/368637.368653

[16] K. M. Ali and A. Guaily, "Dual perspective method for solving the point in a polygon problem," 2020.

[17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, p. 47–57, Jun. 1984. [Online]. Available: https://doi.org/10.1145/971697.602266

[18] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 322–331. [Online]. Available: https://doi.org/10.1145/93597.98741

[19] F. Ris, E. Barkmeyer, C. Schaffert, and P. Farkas, "When floating-point addition isn't commutative," *SIGNUM Newsl.*, vol. 28, no. 1, p. 8–13, Jan. 1993. [Online]. Available: https://doi.org/10.1145/156301.156303

[20] A. Jacobs, "The pathologies of big data," *Commun. ACM*, vol. 52, no. 8, p. 36–44, Aug. 2009. [Online]. Available: https://doi.org/10.1145/1536616.1536632

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Ort, den 19.08.2021 Maxim Balsacq