HAW HAMBURG

BACHELORARBEIT

Vergleich von relationalen und Zeitreihendatenbanken. Eine Untersuchung der Anfragekomplexität und -effizienz

Saskia Behn
[redacted]

Erstprüferin: Prof. Dr.-Ing. Lars Hamann
Zweitprüfer: Prof. Dr. Stefan Sarstedt

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Department Informatik
Campus Berliner Tor
Berliner Tor 7
20099 Hamburg

Inhaltsverzeichnis

Zusammenfassung

Diese Bachelorarbeit untersucht die Unterschiede in der Anfragekomplexität und -effizienz zwischen relationalen Datenbanken und Zeitreihendatenbanken. Relationale Datenbanken sind weit verbreitet und werden häufig für allgemeine Anwendungen genutzt, während Zeitreihendatenbanken speziell für das Speichern und Abfragen von Zeitreihendaten entwickelt wurden. Die Untersuchung umfasst eine Analyse der Abfragesprachen mithilfe der Halstead Metriken, sowie Tests zur Performancebewertung. Hierzu wurden große Datenmengen aus Wetterdaten verwendet und es wurden typische Abfragen aus realen Anwendungsfällen formuliert und verglichen. Die Ergebnisse zeigen, dass Zeitreihendatenbanken bei zeitbezogenen Abfragen eine signifikant höhere Effizienz aufweisen, insbesondere bei großen Datenmengen und komplexen Aggregationen. Relationale Datenbanken hingegen sind besser geeignet für einfache Abfragen.

This bachelor's thesis investigates the differences in query complexity and efficiency between relational databases and time series databases. Relational databases are widely used and commonly employed for general applications, whereas time series databases are specifically designed for storing and querying time series data. The investigation includes an analysis of the query languages using Halstead metrics, as well as performance evaluation tests. Large datasets of weather data were used, and typical queries from real-world use cases were formulated and compared. The results show that time series databases exhibit significantly higher efficiency for time-related queries, especially with large datasets and complex aggregations. In contrast, relational databases are better suited for simple queries.

1 Einleitung

Im Alltag begegnen uns Daten immer wieder. Sei es auf dem Bankkonto, in der Wetterapp oder bei der Nutzung von Sozialen Medien. Daten sind allgegenwärtig und umfassen eine Vielzahl an Informationen [1]. Daten helfen uns, fundierte Entscheidungen zu treffen. Dies gilt insbesondere für Unternehmen, die Daten gezielt für Geschäftsentscheidungen nutzen [2].

Die Menge der generierten Daten ist enorm, und es ist unerlässlich, diese Daten effizient zu speichern, zu verwalten und zu analysieren. Datenbanken spielen hierbei eine zentrale Rolle und sind ein wesentlicher Bestandteil der modernen Informationsverarbeitung [3].

Es gibt verschiedene Arten von Datenbanken, die sich in der Art und Weise unterscheiden, wie sie Daten speichern und verwalten. Zwei bekannte Typen sind relationale Datenbanken und Zeitreihendatenbanken. Diese beiden Datenbanktypen unterscheiden sich hauptsächlich in der Art der Datenspeicherung und der verwendeten Abfragesprachen [4].

In dieser Bachelorarbeit werden relationale Datenbanken und Zeitreihendatenbanken detailliert untersucht und miteinander verglichen.

Einige der zentralen Fragen, die in dieser Arbeit behandelt werden, sind: Wie unterscheiden sich Zeitreihendatenbanken von anderen Datenbanktypen? Welche spezifischen Funktionen bieten Zeitreihendatenbanken im Vergleich zu relationalen Datenbanken? In welchen spezifischen Anwendungsfällen übertreffen Zeitreihendatenbanken herkömmliche relationale Datenbanken? Dabei wird insbesondere auf die Unterschiede in der Abfragesprache eingegangen und wie diese die Nutzung und Effizienz der Datenbanken beeinflussen.

Durch die Beantwortung dieser Fragen soll geklärt werden, welche Unterschiede zwischen den beiden Arten von Datenbanken bestehen und welche sich für bestimmte Anwendungsfälle als vorteilhafter erweisen. Um dies zu erreichen, wird zunächst erklärt, was Zeitreihendatenbanken sind und welche besonderen Eigenschaften sie auszeichnen. Anschließend wird erläutert, welche Daten als Beispiel verwendet werden, woher diese stammen und wie sie strukturiert sind.

Nachdem die theoretischen Grundlagen geklärt sind, werden zwei Beispiel-Datenbanken vorgestellt: eine relationale Datenbank und eine Zeitreihendatenbank. Diese werden anschließend konkret miteinander verglichen, um die eingangs gestellten Fragen zu beantworten. Es wird aufgezeigt, in welchen Szenarien und für welche Anwendungsfälle die jeweiligen Datenbanktypen besonders geeignet sind und welche Vor- und Nachteile sie mit sich bringen.

Ziel dieser Arbeit ist es, ein tiefgehendes Verständnis für die Unterschiede und Gemeinsamkeiten von relationalen und Zeitreihendatenbanken zu vermitteln und deren spezifische Einsatzmöglichkeiten und Vorteile zu beleuchten. Dadurch soll ein fundiertes Wissen darüber erlangt werden, welcher Datenbanktyp in welchen Kontexten effizienter und effektiver eingesetzt werden kann.

2 Grundlagen

In diesem Kapitel werden die Grundlagen, auf den diese Arbeit basiert, beschrieben. Es werden die beiden Datenbanktypen – Zeitreihendatenbanken und relationale Datenbanken – beschrieben. Zusätzlich werden die Grundlagen der Halstead-Metriken vorgestellt.

2.1 Zeitreihendatenbanken

Zeitreihendatenbanken sind darauf spezialisiert, Daten in zeitlicher Reihenfolge zu speichern und zu verwalten. Sie ermöglichen die effiziente Verwaltung und Analyse großer Datenmengen und bieten die Möglichkeit, Daten in Echtzeit zu analysieren, Ereignisse vorherzusagen oder präventive Maßnahmen zu ergreifen. Im Gegensatz zu herkömmlichen Datenbanken legen Zeitreihendatenbanken den Schwerpunkt auf die zeitliche Beziehung zwischen den Daten, was eine tiefergehende Analyse von Zeitreihendaten ermöglicht.

Daten wie Wetterdaten, Finanzkurse oder Logdaten von IT-Systemen werden häufig in Zeitreihendatenbanken gespeichert [4].

Die Hauptkomponenten einer Zeitreihendatenbank sind die Zeitreihendaten, ein Zeitstempel, Metadaten, wie zum Beispiel ein Standort und die Abfragesprache, mit der die Daten abgerufen und analysiert werden können [5].

Zeitreihendaten sind eine spezielle Art von Daten, die über die Zeit hinweg gesammelt werden, typischerweise in regelmäßigen Intervallen oder Ereignissen. Diese Datenpunkte sind mit einem Zeitstempel versehen, der den genauen Zeitpunkt angibt, zu dem die Messung oder Erfassung erfolgt ist. Dieser Zeitstempel spielt eine zentrale Rolle bei der Analyse und Verarbeitung von Zeitreihendaten, da er es ermöglicht, die zeitliche Entwicklung der Daten zu verfolgen und Muster oder Trends zu identifizieren [4].

Zusätzlich zu den eigentlichen Messwerten enthalten Zeitreihendaten oft Metadaten. Diese Metadaten bieten zusätzliche Informationen über die Datenpunkte, wie zum Beispiel den Standort, die Quelle der Daten, die Messgeräteigenschaften oder andere beschreibende Attribute. Metadaten helfen dabei, die Daten besser einordnen zu können, ihre Herkunft zu verstehen und die Qualität der Daten zu bewerten [6].

Für die effiziente Abfrage und Analyse von Zeitreihendaten werden spezielle Abfragesprachen verwendet. Diese Sprachen sind speziell dafür entwickelt, die besonderen Anforderungen und Eigenschaften von Zeitreihendaten zu berücksichtigen. Sie bieten Funktionen und Werkzeuge, die das Filtern, Zusammenfassen und Bearbeiten von Datenströmen über die Zeit erleichtern. Mit diesen Abfragesprachen können Nutzer komplexe Analysen durchführen und Einblicke in das Verhalten der Daten im Zeitverlauf gewinnen [7].

Insgesamt ermöglichen Zeitreihendaten in Kombination mit Zeitstempeln und Metadaten eine detaillierte Analyse von zeitabhängigen Phänomenen und Prozessen. Die spezialisierten Abfragesprachen bieten dabei die nötige Flexibilität und Leistungsfähigkeit, um große Datenmengen effizient zu verarbeiten und umfangreiche zeitliche Analysen durchzuführen [7].

Bekannte Beispiele von Zeitreihendatenbanken sind unter anderem InfluxDB, Prometheus, TimescaleDB und OpenTSDB. Diese Zeitreihendatenbanken bieten eine Reihe von Funktionen, die speziell auf die Bedürfnisse der Verwaltung und Analyse von Zeitreihendaten zugeschnitten sind. Dazu gehört vor allem die effiziente Speicherung und Verarbeitung großer Datenmengen. Diese Systeme sind darauf ausgelegt, große Datenströme schnell und effizient zu speichern, ohne dabei übermäßig viel Speicherplatz zu verbrauchen [4].

Ein weiterer wesentlicher Vorteil ist die Fähigkeit zur Echtzeitanalyse. Zeitreihendatenbanken ermöglichen es, Daten kontinuierlich und in Echtzeit zu verarbeiten. Dadurch können Organisationen sofort auf Veränderungen reagieren, ohne auf verzögerte Analysen angewiesen zu sein [8].

Viele Zeitreihendatenbanken bieten eingebaute Analysefunktionen an. Dazu zählen häufig genutzte Funktionen wie Summationen oder Durchschnittsberechnungen über spezifische Zeitintervalle, gleitende Durchschnittswerte zur Datenverfeinerung sowie fortgeschrittene Analysen wie Vorhersagemodelle. Diese integrierten Funktionen ermöglichen es Nutzern, komplexe Analysen durchzuführen und Erkenntnisse aus den Daten zu gewinnen [5].

Insgesamt sind Zeitreihendatenbanken somit nicht nur auf die effiziente Verwaltung und Speicherung von Daten spezialisiert, sondern bieten auch die erforderlichen Werkzeuge zur Durchführung umfassender Analysen in Echtzeit. Sie spielen eine entscheidende Rolle in verschiedenen Bereichen wie Finanzwesen, Telekommunikation und vielen anderen, wo schnelle und präzise Datenanalyse von entscheidender Bedeutung ist [5].

2.2 Relationale Datenbanken

Relationale Datenbanken speichern Daten in einer strukturierten Form, nämlich in Tabellen. Jede Tabelle besteht aus Zeilen (auch Datensätze oder Tupel genannt) und Spalten (Attribute), wobei jede Zeile eine eindeutige ID erhält, die oft als Primärschlüssel bezeichnet wird. Die Spalten definieren die Art der gespeicherten Daten und stehen in einer definierten Beziehung zueinander, was eine hohe Datenintegrität und -konsistenz gewährleistet [4].

Die Abfragesprache, die für relationale Datenbanken verwendet wird, ist SQL (Structured Query Language). SQL ermöglicht es, Daten zu erstellen, abzurufen, zu aktualisieren und zu löschen [9].

Relationale Datenbanken zeichnen sich durch eine Reihe von Hauptmerkmalen aus, die ihre Beliebtheit und ihren Einsatz in verschiedenen Branchen erklären. Ein zentrales Merkmal relationaler Datenbanken ist die Gewährleistung von Datenintegrität und Konsistenz. Durch den Einsatz von Primärschlüsseln, Fremdschlüsseln und Constraints wird sichergestellt, dass die gespeicherten Daten konsistent und valide sind. Primärschlüssel identifizieren eindeutig jede Zeile in einer Tabelle, während Fremdschlüssel Beziehungen zwischen Tabellen definieren und die Referenzielle Integrität sicherstellen. Constraints helfen dabei, Regeln und Bedingungen festzulegen, die die Datenqualität bewahren [4].

Ein weiteres bedeutendes Konzept ist die Normalisierung, ein Prozess, der darauf abzielt, Redundanzen in der Datenbank zu minimieren und die Struktur so zu gestalten, dass die Datenintegrität und Effizienz verbessert werden. Durch die Normalisierung wird gewährleistet, dass jede Information nur einmal gespeichert wird, was Redundanzen vermeidet und die Konsistenz der Datenbank erhöht [4].

Relationale Datenbanken unterstützen auch Transaktionen gemäß den ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability). Diese Eigenschaften stellen sicher, dass Transaktionen atomar, konsistent, isoliert und dauerhaft ausgeführt werden [4].

Zu den beliebtesten RDBMS gehören unter anderem MySQL, PostgreSQL und Oracle Database. Diese Systeme bieten zwar zahlreiche Vorteile, sind jedoch aufgrund bestimmter Einschränkungen möglicherweise nicht für alle Anwendungsfälle geeignet. Insbesondere haben sie Schwierigkeiten, horizontal über verteilte Systeme hinweg zu skalieren, was zu Einschränkungen ihrer Leistung und Kapazität in Big-Data-Anwendungen führen kann. Zudem können die strengen Anforderungen an das Schema die Flexibilität beeinträchtigen, wodurch die Weiterentwicklung von Datenmodellen oder die Berücksichtigung komplexer Datentypen und Beziehungen erschwert wird [4].

Relationale Datenbanken bieten eine Reihe bedeutender Vorteile, die sie zu einer bevorzugten Wahl für viele Anwendungen und Systeme machen:

Strukturierte Datenverwaltung: Die tabellarische Struktur relationaler Datenbanken ermöglicht eine klare und geordnete Verwaltung von Daten. Jede Tabelle enthält Spalten (Attribute) und Zeilen (Datensätze), was eine einfache Organisation und Verwaltung der Informationen ermöglicht. Diese Struktur erleichtert auch die Verwaltung von Beziehungen zwischen verschiedenen Datensätzen über Fremdschlüssel [10].

Flexibilität bei Abfragen: Relationale Datenbanken verwenden die Structured Query Language (SQL), eine mächtige Abfragesprache, die es Benutzern ermöglicht, komplexe Abfragen zu formulieren. Mit SQL können Anwender Daten selektieren, filtern, gruppieren, sortieren und aggregieren. Dies erleichtert präzise Datenanalysen und Berichterstattung sowie die Extraktion spezifischer Informationen aus großen Datenmengen [10].

Robuste Transaktionsverwaltung: Relationale Datenbanken unterstützen ACID-Transaktionen die eine zuverlässige und fehlertolerante Verarbeitung von Daten gewährleisten. Atomicity stellt sicher, dass Transaktionen entweder vollständig ausgeführt oder gar nicht ausgeführt werden. Consistency sorgt dafür, dass die Datenbank nach jeder Transaktion in einem konsistenten Zustand bleibt. Isolation stellt sicher, dass Transaktionen unabhängig voneinander ausgeführt werden, ohne dass sich ihre Ergebnisse gegenseitig beeinflussen. Durability gewährleistet, dass einmal geschriebene Daten auch bei Systemausfällen oder Neustarts erhalten bleiben [10].

Diese Eigenschaften machen relationale Datenbanken zu einer stabilen und zuverlässigen Wahl für geschäftskritische Anwendungen, bei denen Datenintegrität, Flexibilität bei Abfragen und eine robuste Transaktionsverwaltung entscheidend sind. Sie sind besonders gut geeignet für Anwendungen, die komplexe Datenstrukturen und umfangreiche Datenmengen verwalten müssen, sowie für Systeme, die eine hohe Zuverlässigkeit und Leistung erfordern [10].

2.3 Halstead - Metriken

Die Halstead-Metriken sind eine Gruppe von Metriken, die entwickelt wurden, um die Komplexität eines Programms oder Systems zu bestimmen. Diese können sowohl auf Pseudocode als auch auf den tatsächlichen Code angewendet werden [11].

Die Grundlage dieser Metriken besteht darin, den Code in bestimmte Token zu zerlegen, die als Operatoren und Operanden bezeichnet werden [12].

Die Operatoren sind die Aktionen oder Befehle im Code wie zum Beispiel `SELECT` oder `FROM` in SQL. Die Operanden sind die Daten oder Werte, die in den Operationen verwendet werden. Ein Beispiel für Operanden ist Zeitstempel oder Wert [13].

Die Halstead-Metriken basieren auf vier grundlegenden Parametern, die aus der Analyse des Codes abgeleitet werden n1, n2, N1 und N2 [13].

n1 sind die Anzahl eindeutiger Operatoren und n2 die Anzahl eindeutiger Operanden. Hier werden die Token einmalig gezählt, aber nicht doppelt [13].

N1 ist die Gesamtanzahl der Operatoren und N2 ist die Gesamtanzahl der Operanden. Bei diesen Parametern, werden alle vorhandenen Token gezählt, also auch welche, die doppelt vorkommen [13].

Mit diesen Parametern werden nun die folgenden Metriken berechnet [11]:

Programmlänge (N):

$$N = N1 + N2$$

Größe des Vokabulars (n):

$$n = n1 + n2$$

Schwierigkeitsgrad (D):

$$D = (n1/2) * (N1/n2)$$

Volumen (V):

$$V = N * \log(n)$$

Aufwand (E):

$$E = D * V$$

Implementierungszeit (T):

$$T = E / 18$$

Diese Metriken bieten verschiedene Einblicke in die Komplexität und den Aufwand, der für das Verständnis und die Implementierung des Codes erforderlich ist. Zum Beispiel misst das Volumen (V) den Umfang des Programms in Bezug auf die Anzahl der Informationen, die verarbeitet werden müssen, während der Aufwand (E) die Gesamtkosten in Bezug auf die mentale Anstrengung darstellt, die zur Implementierung des Programms erforderlich ist [13].

2.3.1 Anwendung

Die Halstead-Metriken können verwendet werden, um die Komplexität und Effizienz verschiedener Abfragesprachen zu vergleichen [12]. In dieser Arbeit wird ein detaillierter Vergleich zwischen den Abfragesprachen von MariaDB (SQL) und InfluxDB (Flux) durchgeführt. Dies ermöglicht eine messbare Bewertung der Unterschiede in der Komplexität und des Aufwands, der für die Implementierung und Wartung von Abfragen in diesen Datenbanken erforderlich ist.

Im Kapitel 4 dieser Arbeit werden konkrete Beispiele und detaillierte Vergleiche zwischen SQL und Flux präsentiert. Dabei werden die Halstead-Metriken auf spezifische Abfragen angewendet, um deren Komplexität zu analysieren und die Vor- und Nachteile der jeweiligen Abfragesprachen aufzuzeigen.

Durch die Anwendung der Halstead-Metriken kann ein tieferes Verständnis für die Effizienz und Benutzerfreundlichkeit der beiden Datenbanktypen gewonnen und fundierte Empfehlungen für den Einsatz in verschiedenen Anwendungsszenarien gegeben werden.

3 Datenbanken

In diesem Kapitel wird zuerst darauf eingegangen, welche Daten als Beispiel verwendet wurden und wie sie erhalten wurden. Anschließend werden die beiden Datenbanken vorgestellt, die miteinander verglichen werden sollen. Dabei wird zunächst eine allgemeine Definition der Datenbanken gegeben, gefolgt von einer Beschreibung der Installation und Benutzung. Abschließend wird auf die jeweilige Abfragesprache eingegangen.

3.1 Daten

In diesem Abschnitt wird dargelegt, welche spezifischen Datensätze als Beispiele für die Analyse und den Vergleich verwendet wurden. Es wird beschrieben, wie diese Daten gesammelt oder generiert wurden und welche Eigenschaften sie besitzen. Diese Informationen sind entscheidend, um den Kontext und die Anwendbarkeit der Analysen zu verstehen.

3.1.1 Vorüberlegung

Um die beiden Datenbanken vergleichen zu können, wurden große Mengen an Daten benötigt. Verschiedene Anwendungsfälle wie z.B. Aktienkurse, Wetterdaten, Windgeschwindigkeiten und Geburtenraten sind denkbar. Für diese Bachelorarbeit wurde entschieden, die Analysen mithilfe von Wetterdaten durchzuführen, da diese konsistent über lange Zeiträume hinweg gesammelt werden und vielfältige Möglichkeiten zur Analyse bieten.

3.1.2 Datenbeschaffung

Die Daten wurden von der Webseite des Deutschen Wetterdienstes entnommen [14]. Über die Webseite gelangt man zu dem *CDC – Climate Data Center*, wo historische Wetterdaten frei zugänglich sind (siehe Abb. 1).

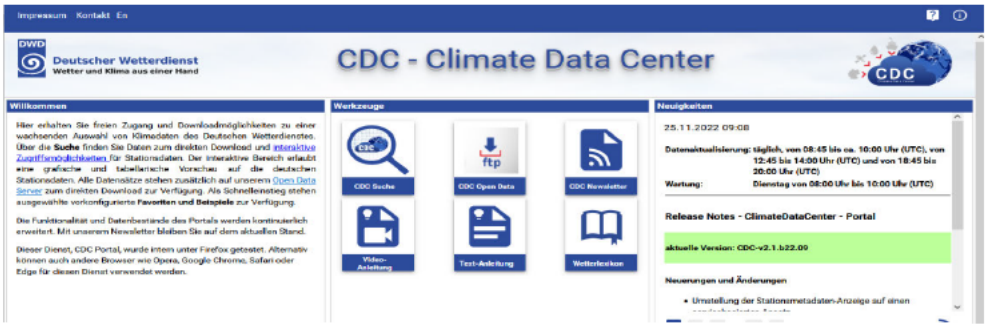


Abbildung 1 Webseite Deutscher Wetterdienst

Um die Daten von der Webseite zu erhalten, wählt man den gewünschten Zeitraum aus.

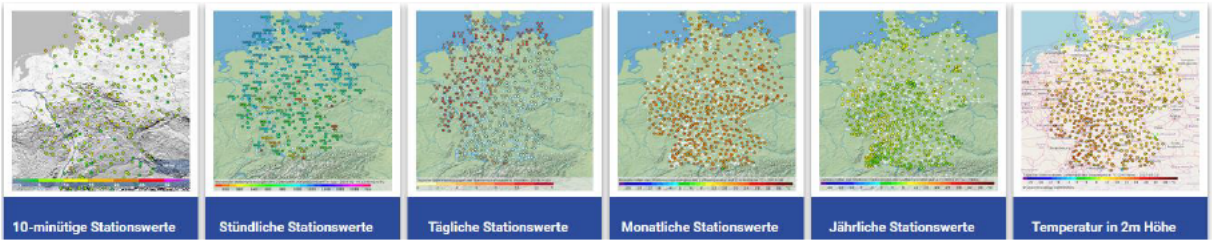


Abbildung 2 Stationswerte Zeitraum

Es wurde sich für die Stündlichen Stationswerte entschieden, da sie eine hohe Granularität bieten und so eine große Datenmenge für die Analyse bereitstellen.

Für diese Bachelorarbeit wurden die Daten einer Messstation in `Hamburg-Fuhlsbüttel (ID 1975)` ausgewählt. Diese Station liefert Wetterdaten seit dem 1. Januar 1949 bis heute, mit Messungen, die jede Stunde aufgezeichnet wurden.

3.1.3 Datenumfang und -format

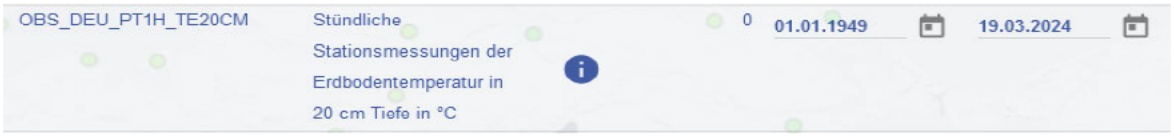


Abbildung 3 Zeitraum

Auf diesem Bild (Abb. 3) ist zu sehen, in welchem Zeitraum die Daten sind. Es kann ein bestimmter Zeitraum gewählt werden, je nachdem wie viele Daten oder aus welchen Jahren die Daten benötigt werden.

Der gewählte Zeitraum vom 01. Januar 1949 bis 19. März 2024 umfasst 659.141 Datenpunkte, da stündlich eine Temperaturmessung erfolgt. Je mehr Daten vorhanden sind, desto präziser und aussagekräftiger ist die spätere Analyse.

Abbildung 4 Download der Daten

Für den Download großer Datenmengen ist die Angabe einer E-Mail-Adresse erforderlich (siehe Abb. 4). Nachdem die Adresse eingegeben und der Download initiiert wurde, kam die E-Mail nach ca. 2 Minuten mit einem Download-Link. Innerhalb von 24 Stunden konnten die Daten heruntergeladen werden.

Die heruntergeladenen Daten befanden sich in einem ZIP-File und bestanden aus 659.141 Datenzeilen. Jede Zeile enthält folgende Werte im Format:

> *Code , ID , Zeitstempel , Wert , QualitaetByte , QualitaetNiveau*
> Messstation, ID der Messstation, Datum und Uhrzeit, Temperatur, Qualität des Wertes, Qualitätsniveau

Ein Beispiel dazu ist:

> *OBS_DEU_PT10M_T2M,"1975","2008-04-30T09:20:00","15.6","11","2"*

Für den Vergleich der Abfragesprachen und Datenbanken wurden nur der Zeitstempel und der Wert (Temperatur) benötigt. Deshalb musste die Datei angepasst werden. Aufgrund der großen Datenmenge wäre eine manuelle Bearbeitung zu zeitaufwendig. Daher wurde ein Python-Programm geschrieben, um die Datei zu bearbeiten.

3.1.4 Datenaufbereitung mit Python

Das Python-Programm liest die Datei ein, teilt die Zeilen in ihre Felder auf und schreibt eine neue Datei, die von den Datenbanken eingelesen werden kann. Die umgewandelten Daten haben folgendes Format:

> *"2024-03-19T23:00:00","9.7"*

Das Programm liest jede Zeile der Originaldatei, extrahiert den Zeitstempel und den Temperaturwert, und speichert sie in einem Datumsformat, welches von der Datenbank gelesen werden kann, in der neuen Datei. So konnten alle 659.141 Datenzeilen effizient umgewandelt und für den Datenbankvergleich vorbereitet werden.

3.2 InfluxDB

3.2.1 Definition

InfluxDB ist eine Open Source Datenbank, die von der Firma Influx Data Inc. entwickelt wurde. Sie ist mit der Programmiersprache Go geschrieben und die neueste Version ist 2.0 [15].

InfluxDB zeichnet sich durch eine schnelle Datenspeicherung und -verarbeitung aus, insbesondere im Vergleich zu herkömmlichen Datenbanken. Abfragen werden mit der Flux-Sprache oder Influx Query Language erstellt, auf die später noch genauer eingegangen wird [16].

3.2.2 Installation

Die neueste Version von InfluxDB kann über die offizielle InfluxData Webseite heruntergeladen werden. Eine ZIP-Datei mit InfluxDB wurde heruntergeladen und entpackt. Im entpackten Ordner befindet sich die Datei `influxd.exe`, der Influx Daemon, der den Influx-Server startet.

Um InfluxDB zu nutzen, öffnet man ein Command-Fenster und navigiert zum Pfad, in dem sich der InfluxDB Daemon befindet. Durch Ausführen von `influxd.exe` wird der Influx-Server gestartet. Anschließend öffnet man im Browser die Seite `localhost:8086`, um die Benutzeroberfläche zu erreichen [17].

Nach dem Start des Servers wird ein Benutzerkonto erstellt (siehe Abb. 5).

Es werden ein Benutzername, ein Passwort und eine Organisation

(in diesem Fall die HAW Hamburg) angegeben.

Abbildung 5 Benutzerkonto

3.2.3 Daten laden

InfluxDB zeichnet sich durch verschiedene Funktionen aus, die es zu einer leistungsfähigen Lösung für die Speicherung und Verwaltung großer Mengen an Zeitreihendaten machen. Es bietet Kapazität und Skalierbarkeit, um große Datenmengen effizient zu speichern und zu verarbeiten.

Um nun mit Daten arbeiten zu können, muss nun ein sogenanntes `Bucket` erstellt werden (siehe Abb. 6). Dieser wird benannt, um die Datenbank zu definieren [18].

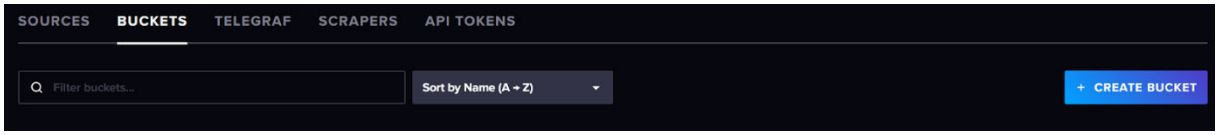


Abbildung 6 Bucket

Eine weitere wichtige Funktion von InfluxDB sind Retention Policies (siehe Abb. 8). Diese ermöglichen es, festzulegen, wie lange Daten aufbewahrt werden sollen, bevor sie automatisch gelöscht werden. Dies ist besonders hilfreich für die langfristige Datenarchivierung und -verwaltung, da es die manuelle Auswahl von Daten zum Löschen überflüssig macht [19].

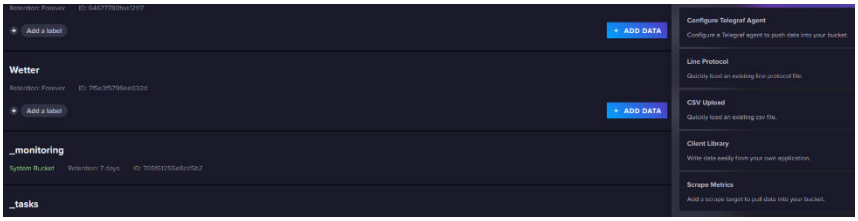


Abbildung 7 Daten laden

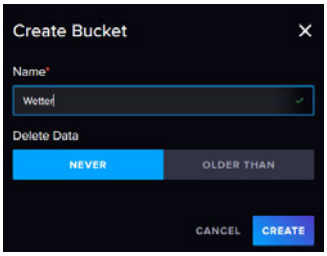


Abbildung 8 Bucket erstellen

Es gibt verschiedene Möglichkeiten, Daten in InfluxDB zu laden (siehe Abb. 7). Eine Möglichkeit ist das Line Protocol, mit dem Daten manuell eingefügt werden können. Die Syntax dafür ist wie folgt [20]:

// Syntax

<measurement>[,<tag_key>=<tag_value>[,<tag_key>=<tag_value>]]
<field_key>=<field_value>[,<field_key>=<field_value>] [<timestamp>]

In der Datenbankstruktur für Zeitreihendaten sind mehrere Schlüsselkonzepte entscheidend für die Organisation und das Verständnis der Daten. Hier sind die wesentlichen Elemente:

`Measurements` sind die Kategorien oder Typen von Daten, die gespeichert werden. Diese werden typischerweise als Strings bezeichnet und repräsentieren den Namen der Datensätze in der Datenbank. Zum Beispiel könnten Daten unter dem Measurement `Wetterdaten` gespeichert sein [21].

`Fields` sind Schlüssel-Wert-Paare, die innerhalb jedes Datensatzes enthalten sind. Die `Field-Values` sind die eigentlichen Messwerte oder Datenpunkte, die im Laufe der Zeit gesammelt werden, wie z.B. Temperaturwerte. Die `Field-Keys` sind Metadaten, die beschreiben, welche Art von Daten in den Field-Values enthalten sind. Beispielsweise könnte `Temperatur` ein Field-Key sein, während der gemessene Temperaturwert der Field-Value ist [21].

`Tags` sind optionale Schlüssel-Wert-Paare, die zusätzliche Metadaten zu jedem Datensatz hinzufügen. Tags sind nützlich, um Datenpunkte zu kategorisieren oder zu filtern. Zum Beispiel könnten Tags Informationen wie den Standort der Messung enthalten [21].

Der `Timestamp` ist ein wichtiger Bestandteil jedes Datenpunkts und gibt den genauen Zeitpunkt der Datenerfassung an. Er wird typischerweise als Unix-Timestamp in Nanosekunden angegeben, was eine präzise zeitliche Einordnung der Daten ermöglicht [21].

Der Unix-Timestamp ist eine Zahl, die vom 01. Januar 1970 sekündlich hochgezählt wird. Der Timestamp bei Influx rechnet den Unix-Timestamp nochmal in Nanosekunden um. Vom 20.03.2024 um 23:00 Uhr ist der Timestamp also 1.710.972.000.000.000.000 [22].

Diese Strukturelemente helfen, Zeitreihendaten effizient zu organisieren, abzurufen und zu analysieren, indem sie sowohl die eigentlichen Messwerte als auch die zugehörigen Metadaten klar und strukturiert darstellen.

Zwischen den Tags und den Fields wird jeweils ein Leerzeichen gesetzt, ebenso zwischen den Fields und dem Timestamp. Der Rest wird mit einem Komma getrennt [15].

// Beispiel

Wetterdaten,Ort=Hamburg Wert=2.1 1710972000000000000

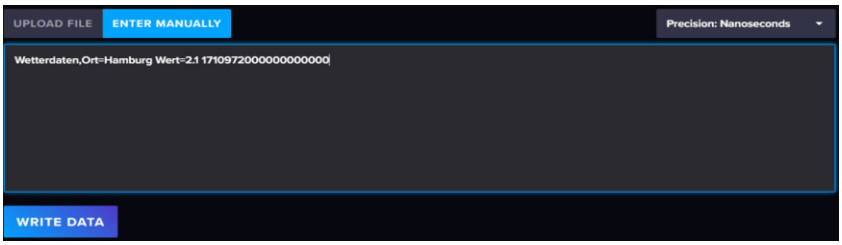


Abbildung 9 Daten manuell einfügen

Wenn die Daten erfolgreich in die Datenbank importiert worden sind erhält man folgendes Fenster (Abb. 10).

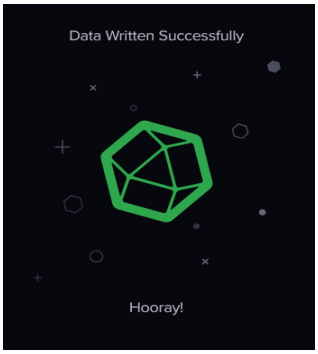


Abbildung 10 Erfolgreiche Eingabe

Um große Datenmengen effizient zu laden, wurde die CSV-Upload-Funktion verwendet. Zuerst wurden 5 Beispiel-Datensätze manuell angepasst und getestet, wie sie korrekt formatiert werden müssen. Das Format für den CSV-Upload sieht wie folgt aus:

, , table, _time, _value, _field, _measurement

, , 0, 2023-01-01T00:52:00Z, 15.43, mem, m

Es wird also die Zeit, Value, Field und Measurement gebraucht. Am Anfang wird noch eine Zahl als Aufzählung angegeben sowie zwei leere Plätze [23] [24].

Ein Beispiel für die Wetterdaten:

, , 1, 1949-01-01T00:00:00Z, 2.1, temp, wetterdaten

Damit nicht alle 659.141 Daten manuell geändert werden müssen, wurde ein Python Programm geschrieben, um die Datei zu ändern.

Die Datei wurde geöffnet und eine neue Datei erstellt. Mit einer Schleife wurde durch jede Zeile iteriert, die Zeilen wurden in ihre einzelnen Attribute aufgeteilt und diese Attribute wurden dann in die neue Datei geschrieben. Dabei wurden die einzelnen Elemente hinzugefügt.

Das ganze sieht dann so aus:

neu.write(',,' + str(i) + ',' + teile[0]+ 'Z,' + zeile_ohne_umbruch + ',temp,wetterdaten \n')

Wobei `teile[0]` die Zeit ist und `zeile_ohne_umbruch` der Wert.

Mit dieser Datei wurde nun versucht, die Daten in Influx zu laden. Da das immer noch nicht funktioniert hat, obwohl die Annotation korrekt ist, wurde sich dazu entschieden, die Daten nicht per CSV Import zu importieren, sondern das einspielen der Daten mit der in InfluxDB mitgelieferten Python Schnittstelle zu importieren [25].

Um Python zu benutzen, damit die Daten in Influx geladen werden können, muss zuerst Python mit Influx verbunden werden.

Dafür wird ein bestimmter Token benötigt. Der wird auf der Benutzeroberflächen Seite zur Verfügung gestellt. Unter Sources kann Python ausgewählt werden, dort kann nun der Token kopiert und in die Python Datei eingefügt werden. Außerdem wird die URL `http://localhost:8086` und die Organisation `HAW Hamburg` benötigt.

Damit kann nun ein Client erstellt werden, indem die drei Parameter (url, Organisation und Token) angegeben werden.

write_client = influxdb_client.InfluxDBClient(url=url, token=token, org=myorg)

Mit diesem Client kann nun eine API zu Influx erstellen werden.

write_api = write_client.write_api(write_options=SYNCHRONOUS)

Nachdem die Verbindung hergestellt wurde, wird die Datei mit den Daten geöffnet und mit einer Schleife durch jede Zeile gegangen. Dabei wird nun die Zeit und der Wert aus den Daten extrahiert [25].

Da der Wert ein String ist, muss dieser noch in einen Float umgewandelt werden.

```
# Umwandlung in einen Float-Wert [26]
  try:
      float_wert = float(myvalue)
      print("Float-Wert:", float_wert)
  except ValueError:
      print("Konnte den String nicht in einen Float-Wert umwandeln.")
```

Dann wird ein Point Objekt erstellt [25].

```
point = Point("wetterdaten").field("temp", float_wert).time(mytime)
```

Mit diesem Objekt werden die Daten in den Influx Bucket geladen.

```
write_api.write(bucket=bucket, org=myorg, record=point)
```

Die ganzen Daten mit Hilfe von Python in die Datenbank zu laden hat insgesamt 2 Stunden und 30 Minuten gedauert.

3.2.4 Abfragesprache

Nachdem die Daten nun in der Datenbank gespeichert sind, können Abfragen erstellen werden. InfluxDB unterstützt zwei Abfragesprachen: Flux und Influx Query Language (InfluxQL).

InfluxQL ähnelt SQL und bietet hohe Leistung bei einfachen Abfragen. Allerdings ist es weniger flexibel und hat eingeschränkte Funktionalitäten bei der Bearbeitung von Zeitreihen. Flux hingegen ist eine Skript-Abfragesprache, die sehr flexibel ist und auch bei komplexen Abfragen hohe Leistung bietet. Da Flux zukünftig in InfluxDB verstärkt eingesetzt werden soll, wird es kontinuierlich weiterentwickelt [16].

Für diese Bachelorarbeit wurde Flux verwendet. Abfragen werden im Data Explorer der Benutzeroberfläche erstellt und im Skript-Editor eingegeben. Außerdem kann angegeben werden, wie die Daten zusehen sein sollen (siehe Abb. 11).

Hier wird hauptsächlich die `Table` Ansicht verwendet, da der Überblick über die Daten dort besser ist.

Abbildung 11 Ansicht der Ergebnisse

Eine einfache Abfrage in Flux sieht wie folgt aus [27]:

from(bucket: "example-bucket")

|> range(start: -1h)

Es startet immer mit `from` und dort wird der Bucket, der verwendet wird, angegeben. In diesem Fall liegen die Daten in dem Bucket `Wetter`.

`Range` gibt die Zeitspanne an, in der die Daten durchsucht werden sollen [27]. Ein einfaches Flux – Beispiel:

from(bucket: "Wetter")

|> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:00:00Z)

_start	_stop	_time	_value	_field	_measurement
1949-01-01 01:00:00 GMT+1	2024-03-20 00:00:00 GMT+1	1949-01-01 01:00:00 GMT+1	2,10	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:00:00 GMT+1	1949-01-01 02:00:00 GMT+1	3	temp	wetterdaten

Abbildung 12 Ergebnisse der einfachen Flux Abfrage

Zurückgegeben wird dann der Start, Stop, der Zeitpunkt, der Wert, Field und Measurement.

Wobei am wichtigsten der Zeitpunkt und der Wert ist.

Um spezifischere Abfragen zu erstellen, kann `filter()` verwendet werden, ähnlich wie `WHERE` in SQL [28]:

|> filter(fn: (r) => r._value > 0 and r._value < 10)

`fn` steht für `function` und ist ein Parameter. Diese Funktion bestimmt welche Datenpunkte durch den Filter durchlaufen. `r` steht für `record` und stellt einen einzelnen Datenpunkt dar [28]. In diesem Beispiel werden die Werte (_value) gefiltert, sodass nur diejenigen erhalten bleiben, die größer als 0 und kleiner als 10 sind.

Dann gibt es noch das `aggregateWindow`, das verwendet wird, um bestimmte Werte über einen bestimmten Zeitraum zusammenzufassen und diese aggregierten Ergebnisse zurückzugeben [29].

|> aggregateWindow(every: 1y, fn: max, createEmpty: false)

Dieses Beispiel gibt als Ergebnis den Maximal Wert jedes Jahres zurück, was mit `fn: max` bestimmt wird. `Every: 1y` bestimmt das Zeitfenster. In diesem Falls ist das Zeitfenster 1 Jahr, das bedeutet, dass die Daten in jährliche Zeitfenster aufgeteilt werden.

Dabei gibt verschiedene Möglichkeiten wie zum Beispiel jedes Jahr,

jeden Monat oder jeden Tag.

Units	Meaning
y	year (12 months)
mo	month
w	week (7 days)
d	day
h	hour (60 minutes)
m	minute (60 seconds)
s	second

Abbildung 13 Abkürzungen der Zeitangaben

Das Bild zeigt die jeweiligen Möglichkeiten mit den Abkürzungen, die bei `every` verwendet werden können (siehe Abb. 13).

Diese beiden Parameter sind erforderlich [29].

Weitere Parameter, die verwendet werden können, aber nicht zwingend erforderlich sind, umfassen unter anderem `createEmpty`. Dieser Parameter legt fest, ob leere Tabellenzeilen angezeigt werden sollen oder nicht. Wenn er nicht angegeben wird, ist der Standardwert `true`, was bedeutet, dass leere Zeilen angezeigt werden [29].

Ein weiterer Parameter ist `period`, der die Dauer eines Zeitfensters angibt. Mit `offset` kann das Zeitfenster entweder in die Zukunft verschoben werden (bei einer positiven Zahl) oder in die Vergangenheit (bei einer negativen Zahl). Schließlich gibt es `location`, das den geografischen Ort angibt [29].

Diese zusätzlichen Parameter bieten erweiterte Kontrollmöglichkeiten bei der Aggregation und Analyse von Zeitreihendaten.

3.3 MariaDB

3.3.1 Definition

MariaDB ist eine relationale Datenbank und eine Abspaltung von MySQL. Entwickelt von der MariaDB Foundation, ist sie Open-Source und bietet eine Vielzahl von Funktionen, die MySQL erweitern. MariaDB unterstützt verschiedene Programmiersprachen wie Python, Java, C und C# sowie alle gängigen Betriebssysteme wie Linux und Windows. Sie kombiniert traditionelle SQL-Funktionen mit zusätzlichen NoSQL-ähnlichen Features und wurde entwickelt, um schnell, zuverlässig und benutzerfreundlich zu sein. Dies macht sie sowohl für kleine Projekte als auch für umfangreiche Anwendungen geeignet [30].

MariaDB bietet eine Vielzahl von Funktionen, die es zu einer zuverlässigen und leistungsfähigen Datenbanklösung macht. Eine der wichtigsten Eigenschaften ist die Unterstützung von Transaktionen und die ACID-Konformität (Atomicity, Consistency, Isolation, Durability). Diese Prinzipien gewährleisten, dass Transaktionen zuverlässig und konsistent ausgeführt werden, was besonders wichtig für Anwendungen ist, die hohe Anforderungen an Datenintegrität und Zuverlässigkeit stellen [31].

Darüber hinaus legt MariaDB großen Wert auf Sicherheit und Benutzerverwaltung. Es gibt umfassende Mechanismen zur Implementierung von Sicherheitsmaßnahmen, um unbefugten Zugriff zu verhindern. Dazu gehören die Benutzerverwaltung, rollenbasierte Zugriffskontrolle und Verschlüsselung, die dazu beitragen, die Daten vor unbefugtem Zugriff zu schützen [32].

Ein weiteres herausragendes Merkmal von MariaDB ist die Skalierbarkeit und Hochverfügbarkeit. Die Datenbank kann sowohl horizontal als auch vertikal skaliert werden, um den Anforderungen wachsender Datenmengen und Anfragen gerecht zu werden. Dies schließt die Unterstützung für Master-Slave-Replikationen und Cluster-Setups ein, die für Hochverfügbarkeitsszenarien besonders wichtig sind. Diese Eigenschaften machen MariaDB zu einer robusten und vielseitigen Lösung für verschiedene Datenbankanforderungen [33].

3.3.2 Installation

Die Installation von MariaDB ist einfach und die Installationsdateien können über die offizielle MariaDB.org heruntergeladen werden [34]. Die Installation umfasst oft Werkzeuge wie HeidiSQL, die verwendet werden können, um auf MariaDB zuzugreifen und sie zu verwalten. Nach der Installation kann eine Verbindung zu MariaDB über die entsprechende Client-Software hergestellt werden.

HeidiSQL und MariaDB können mit folgenden Einstellungen verbunden werden.

In data/my.ini kann der Port gewählt werden, über den MariaDB erreicht werden kann. Der MariaDB Server läuft hier im localhost (127.0.0.1) (siehe Abb. 14).

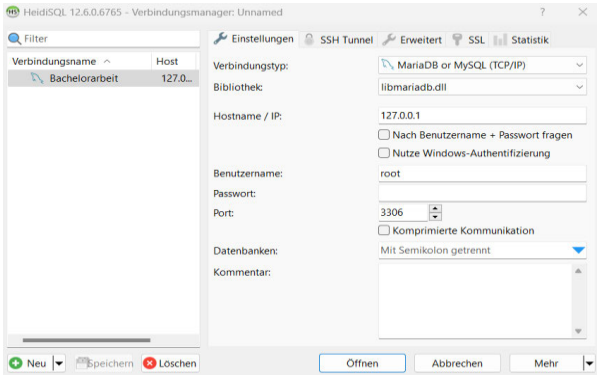


Abbildung 14 MariaDB verbinden

3.3.3 Daten laden

Nach der Installation und Konfiguration von MariaDB wird eine neue Datenbank erstellt, z.B. `Wetterdaten`, und innerhalb dieser Datenbank eine Tabelle, z.B. `Wetter` (siehe Abb. 15). Die Tabelle wird so konfiguriert, dass sie die benötigten Attribute wie Zeitstempel (DATETIME) und Wert (FLOAT) enthält.

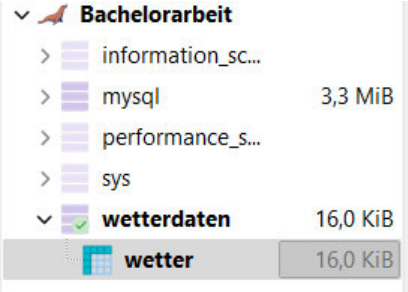


Abbildung 15 Datenbank und Tabelle erstellen

Das Laden von Daten in MariaDB erfolgt häufig über eine CSV-Datei. Hierbei wird in HeidiSQL die Option zum Importieren einer CSV-Datei genutzt.

Im oberen Bereich des Fensters befindet sich ein Reiter mit einem grünen Symbol, auf dem "CSV" steht (siehe Abb. 16).

Abbildung 16 Reiter von HeidiSQL

Diese wird ausgewählt und es öffnet sich ein Fenster. Unter Quell-Datei wird die Datei ausgewählt, in dem die Daten liegen.

Unter Berücksichtigung der Optionen wie Zeilenüberspringung und Feldtrennzeichen werden die Daten aus der CSV-Datei direkt in die erstellte Tabelle importiert. Dieser Vorgang ist effizient und ermöglicht es, große Datenmengen schnell in die Datenbank zu laden.

In diesem Fall soll keine Zeile ignoriert werden, da die erste Zeile der Datei direkt mit den Daten beginnt. Bei Kontrollzeichen wird das Feld `Felder getrennt mit` von *;* zu , geändert, da die Daten mit einem Komma getrennt sind und nicht mit einem Semikolon (siehe Abb 17).

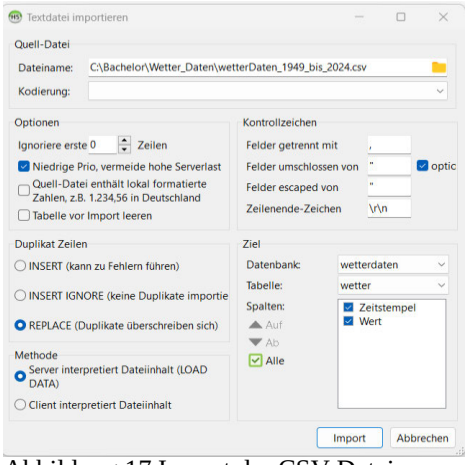


Abbildung 17 Import der CSV Datei

Ein Überprüfen des Imports erfolgt durch eine SELECT-Abfrage, um sicherzustellen, dass alle Daten korrekt geladen wurden. Das Laden der Daten hat nur wenige Sekunden gedauert.

3.3.4 Abfragesprache

MariaDB verwendet SQL (Structured Query Language) als primäre Abfragesprache. SQL ist eine Standard-Sprache für relationale Datenbanken und bietet eine leistungsfähige Möglichkeit, Daten zu verwalten. Mit SQL können Daten abgefragt, eingefügt, aktualisiert und gelöscht werden. Die Syntax von SQL ermöglicht komplexe Abfragen, Aggregationen und Verknüpfungen zwischen Tabellen, was sie zu einem vielseitigen Werkzeug für Datenbankadministratoren und Entwickler macht [35].

MariaDB unterstützt außerdem die Verwendung von Indizes, um Abfragen zu beschleunigen und die Datenbankleistung zu optimieren. Durch das richtige Setzen von Indizes auf häufig abgefragten Spalten können die Antwortzeiten erheblich verbessert werden. Indizes sind ein wesentliches Werkzeug zur Performance-Optimierung in MariaDB und tragen dazu bei, dass Abfragen effizienter ausgeführt werden können [36].

4 Vergleich

In diesem Kapitel werden die beiden Datenbanken InfluxDB und MariaDB miteinander verglichen, wobei der Schwerpunkt auf den Abfragesprachen Flux und SQL liegt. Neben den Abfragesprachen werden auch die Datenbanken selbst hinsichtlich ihrer Installation und Benutzerfreundlichkeit untersucht. Durch diesen Vergleich soll aufgezeigt werden, welche Abfragesprache komplexer ist und welche möglicherweise einfacher anzuwenden ist. Darüber hinaus wird ermittelt, welche Datenbank für welche Anforderungen am besten geeignet ist und in welchen Szenarien sie optimal eingesetzt werden kann.

Zur Bewertung der Abfragesprachen wird die Halstead-Metrik verwendet, die in Kapitel 2.3 detailliert erklärt wurde. In diesem Kapitel wird die Metrik anhand von Beispielen angewendet, um eine fundierte Analyse zu ermöglichen.

Die Operatoren und Operanden werden gemäß den Kriterien von Testwell CMT++ und Testwell CMTJava kategorisiert und definiert. Dadurch können die Halstead-Metriken präzise angewendet werden [37].

4.1 Überprüfung der Ergebnisse

Um sicherzustellen, dass die Abfragen die korrekten Ergebnisse liefern, wurden die Berechnungen manuell durchgeführt. Zum Beispiel wurde für die Durchschnittstemperatur jedes Monats eines jeden Jahres eine Stichprobe von einigen Monaten aus den Daten entnommen und deren Durchschnitt berechnet. Wenn die Ergebnisse der Stichproben mit den Ergebnissen der Abfragen übereinstimmen, zeigt dies, dass die Abfragen korrekt sind. Dieser Prozess wurde für alle Abfragen durchgeführt, um ihre Genauigkeit zu überprüfen. Für einige Abfragen wurden alle ausgegebenen Ergebnisse überprüft, während bei Abfragen mit extrem vielen Ergebnissen nur Stichproben überprüft wurden.

4.2 Einfache Abfrage

In dieser Abfrage soll von jedem Jahr der Maximalwert ermittelt werden.

4.2.1 Einfache Abfrage in SQL

So sieht die beschriebene Abfrage in SQL aus:

SELECT YEAR(zeitstempel), MAX(wert)

FROM wetterdaten

GROUP BY YEAR(zeitstempel)

ORDER BY YEAR(zeitstempel);

Mit dem SQL-Befehl `SELECT YEAR(zeitstempel), MAX(wert)` wird das Jahr aus dem Zeitstempel extrahiert und der höchste Wert unter allen Werten ermittelt. Die Anweisung `FROM wetterdaten` gibt an, welche Datenbanktabelle durchsucht werden soll, in diesem Fall die Tabelle `wetterdaten`. Mit `GROUP BY YEAR(zeitstempel)` werden die Daten nach Jahren gruppiert. Innerhalb dieser Gruppierungen wird der höchste Wert mit `MAX(wert)` ermittelt, sodass die höchste Temperatur jedes Jahres gefunden wird. Abschließend sorgt `ORDER BY YEAR(zeitstempel)` dafür, dass die Ergebnisse chronologisch nach Jahren sortiert ausgegeben werden.

Um diese Abfrage mit der Healstead Metrik bewerten zu können, wird aus dieser Abfrage die Operatoren und Operanden bestimmt [37].

Die Operatoren sind: *SELECT, YEAR, MAX, FROM, GROUP BY, ORDER BY, (,), ,*

Die Operanden sind: *Zeitstempel, Wert*.

Somit kann auch n1, n2, N1 und N2 bestimmt werden.

n1 = 9, n2 = 2, N1 = 17 und N2 = 4.

Mit diesen Parametern können nun die Formeln, die im Kapitel 2.3 definiert wurden berechnet werden.

Die Ergebnisse der einzelnen Formeln sehen so aus:

N = 21

n = 11

D = 38,25

V = 21,869

E = 836,489

T = 46,472

4.2.2 Einfache Abfrage in Flux

Die beschriebene Abfrage in Flux sieht wie folgt aus:

```
from(bucket: "Wetter")
    |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:00:00Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> aggregateWindow(every: 1y, fn: max, createEmpty: false)
    |> yield(name: "max_temperature")
```

Der Befehl `FROM` wählt den Bucket aus, in dem die gespeicherten Daten durchsucht werden sollen, um sie abzurufen. `Range` definiert mit `start` und `stop` den Zeitraum, der durchsucht werden soll. In diesem Fall erstreckt sich der Zeitraum von 01.01.1949 00:00 Uhr bis 19.03.2024 23:00 Uhr, was alle Daten in der Datenbank umfasst. Alternativ kann ein kleinerer Zeitraum gewählt werden, um nur bestimmte Daten zu durchsuchen.

Die `filter`-Funktion ermöglicht zusätzliche Datenfilterung. Hier wird nach dem Measurement `Wetterdaten` und dem Field `temp` gefiltert, die bei der Datenspeicherung definiert wurden.

Die Funktion `filter(fn: (r) => r["_measurement"] == "wetterdaten")` sorgt dafür, dass nur Datensätze berücksichtigt werden, deren Messung (_measurement) `wetterdaten` ist. Dies entspricht in SQL der Auswahl einer bestimmten Tabelle.

Der Filter `filter(fn: (r) => r["_field"] == "temp")` stellt sicher, dass nur Datensätze berücksichtigt werden, bei denen das Feld (_field) `temp` ist. Dies entspricht in SQL der Auswahl einer bestimmten Spalte.

Diese Filterfunktionen sind wichtig, um gezielt auf bestimmte Kategorien von Daten zuzugreifen oder bestimmte Kriterien in den Datensätzen zu definieren und sie sind ein zentraler Bestandteil der Abfragemöglichkeiten in InfluxDB.

`aggregateWindow(every: 1y, fn: max, createEmpty: false)` spezialisiert die Abfrage weiter. `fn` gibt an, dass der Maximalwert ausgegeben werden soll. Es wird definiert, dass leere Zeilen nicht angezeigt werden sollen (`createEmpty: false`). Mit `every: 1y` wird festgelegt, dass für jedes Jahr der Maximalwert ermittelt werden soll.

Die letzte Zeile `yield(name:"max_temperature")` gibt das Ergebnis unter dem Namen `max_temperature` aus.

Die Operatoren sind: *(,),* , , |>, *[,],* ==, =>, *:*

Die Operanden sind: **Identifier(Schlüsselwörter):** *from, range, filter, aggregateWindow, yield*

Sonstige Identifier: *bucket, start, stop, fn, r, _measurement, wetterdaten, _field, temp, every, 1y, max, createEmpty, name, max_temperature, Wetter, false*

Zeitangaben: *1949-01-01T00:00:00Z, 2024-03-19T23:00:00Z*

Somit ist n1 = 9, n2 = 24, N1 = 41, N2 = 30.

Die Ergebnisse der Formeln sehen wie folgt aus:

N = 71

n = 33

D = 7,688

V = 107,814

E = 828,874

T = 46,049

4.3 Komplexe Abfrage

In dieser Abfrage wird der Tagesdurchschnitt der Temperatur für den Tag berechnet, an dem die höchste Tagestemperatur gemessen wurde.

4.3.1 Komplexe Abfrage in SQL

In SQL sieht die beschriebene Abfrage wie folgt aus:

```
SELECT DATE(zeitstempel), AVG(wert)
FROM wetter
WHERE DATE(zeitstempel) = (
        SELECT DATE(zeitstempel)
        FROM wetter
        WHERE wert = (SELECT MAX(wert) FROM wetter)
        LIMIT 1)
GROUP BY DATE(zeitstempel);
```

Die Hauptabfrage `SELECT DATE(zeitstempel), AVG(wert) FROM wetter` wählt alle Datensätze aus der Tabelle `wetter` aus, wobei nur das Datum (`DATE(zeitstempel)`) und der Durchschnittswert (`AVG(wert)`) der Temperatur (`wert`) betrachtet werden.

Die Bedingung `WHERE DATE(zeitstempel) = (...)` stellt sicher, dass nur die Daten des Tages mit der höchsten Temperatur verwendet werden.

Die Subabfrage `SELECT DATE(zeitstempel) FROM wetter WHERE wert = (SELECT MAX(wert) FROM wetter) LIMIT 1` sucht in der Tabelle `wetter` nach dem Datum, an dem die höchste Temperatur gemessen wurde. Sie beginnt damit, den maximalen Temperaturwert in der Tabelle zu ermitteln (`SELECT MAX(wert) FROM wetter`). Anschließend filtert sie die Datensätze, indem sie nur diejenigen auswählt, deren Temperaturwert diesem Maximum entspricht (`WHERE wert = (SELECT MAX(wert) FROM wetter)`). Von diesen Datensätzen wird schließlich das Datum des ersten Vorkommens zurückgegeben (`SELECT DATE(zeitstempel)`), um den Tag mit der höchsten Temperatur zu identifizieren. Die Verwendung von `LIMIT 1` stellt sicher, dass nur ein Datum zurückgegeben wird, selbst wenn mehrere Tage die gleiche maximale Temperatur aufweisen.

Nach der Bestimmung des Datums mit der höchsten Temperatur gruppiert die Hauptabfrage die Daten nach Datum (`GROUP BY DATE(zeitstempel)`) und berechnet den Durchschnitt der Temperaturen (`AVG(wert)`) für diesen Tag.

Daraus werden nun die Operatoren und Operanden bestimmt [37].

Die Operatoren sind: *SELECT, DATE, AVG, FROM, WHERE, GROUP BY, MAX, LIMIT, = , (,), ,*

Die Operanden sind: *Wert, Zeitstempel,1*

n1 = 12, n2 = 3, N1 = 35, N2 = 9

N = 25

n= 11

D= 40,5

V = 26,035

E = 1054,418

T = 58,579

4.3.2 Komplexe Abfrage in Flux

In Flux sieht die beschriebene Abfrage wie folgt aus:

```
import "date"
// Schritt 1: Finden Sie den Zeitstempel des wärmsten Tages in UTC
maxTemp =
  from(bucket: "wetter")
      |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:59:59Z)
      |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
      |> filter(fn: (r) => r["_field"] == "temp")
      |> max()
      |> findRecord(fn: (key) => true, idx: 0)

// Extrahiere das Datum aus dem Ergebnis von maxTemp und setze die Uhrzeit auf 00:00:00 UTC
startRange = date.truncate(t: maxTemp._time, unit: 1d)

// Definiere den Endzeitpunkt als den Beginn des nächsten Tages (UTC)
endRange = date.add(d: 1d, to: startRange)

// Daten aus einem Bucket laden basierend auf dem gefundenen Datum
data = from(bucket: "wetter")
      |> range(start: startRange, stop: endRange)
      |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
      |> filter(fn: (r) => r["_field"] == "temp")
      |> mean()
Data
```

Zunächst wird das Modul `date` importiert, um Datum- und Zeitoperationen durchführen zu können. Dann wird im ersten Schritt der Zeitstempel des wärmsten Tages ermittelt. Dazu wird eine Abfrage auf dem Bucket `wetter` durchgeführt, die den Zeitraum vom 1. Januar 1949 bis zum 19. März 2024 abdeckt. Innerhalb dieses Zeitraums werden die Daten nach dem Messwert `wetterdaten` und dem Feld `temp` gefiltert, um nur Temperaturwerte zu berücksichtigen. Mit der Funktion `max()` wird der höchste Temperaturwert gefunden. Der dazugehörige Datensatz wird mit `findRecord(fn: (key) => true, idx: 0)` extrahiert.

`findRecord(fn: (key) => true, idx: 0)` dient dazu, den ersten Datensatz in der Ergebnismenge zu finden und zurückzugeben. `fn: (key) => true` ist die Bedingung (eine anonyme Funktion), die angibt, dass jeder Datensatz ausgewählt wird. Da die Bedingung immer `true` zurückgibt, wird der erste Datensatz der Ergebnismenge ausgewählt. `idx: 0` gibt an, dass der erste Datensatz (Index 0) zurückgegeben werden soll [38].

Im nächsten Schritt wird das Datum aus dem Ergebnis von `maxTemp` extrahiert und die Uhrzeit auf 00:00:00 UTC gesetzt. Dies erfolgt mittels `date.truncate(t: maxTemp._time, unit: 1d)`, wobei der Zeitstempel auf den Beginn des Tages gekürzt wird [39].

Anschließend wird das Ende des Zeitraums als der Beginn des nächsten Tages definiert, indem ein Tag zum Startdatum addiert wird (`endRange = date.add(d: 1d, to: startRange)`).

Danach werden erneut Daten aus dem Bucket `wetter` geladen, basierend auf dem gefundenen Datum. Der Zeitraum wird durch `startRange` und `endRange` definiert. Wiederum werden die Daten nach dem Messwert `wetterdaten` und dem Feld `temp` gefiltert, um nur die relevanten Temperaturwerte zu erhalten. Schließlich wird mit der Funktion `mean()` der Durchschnitt der Temperaturen innerhalb dieses Tages berechnet.

Die berechneten Daten werden durch das Schlüsselwort `data` ausgegeben, wodurch die Durchschnittstemperatur des wärmsten Tages in der Datenbank ermittelt und angezeigt wird.

Die Operatoren und Operanden wurden anhand [37] ermittelt und definiert.

Die Operatoren sind: **Reserved (Reservierte Wörter):** *import*

=, (,), |>, . , [,], ==, , , =>

Die Operanden sind: **Identifiers***: maxTemp, from, bucket, wetter, range, start, stop, filter, fn, r,_measurement, wetterdaten, _field, temp. Max, findRecord, key, idx, _time, date, truncate, t, unit, startRange, add, d, to, endRange, data, mean,*

Constants*: 1d, 0, true, 1949-01-01T00:00:00Z, 2024-03-19T23:59:59Z*

n1 = 11, n2 = 35, N1 = 90, N2 = 66

N = 156

n = 46

D = 14,143

V = 259,39

E = 3668,553

T = 203,809

4.4 Bewertung

Um die Ergebnisse der verschiedenen Abfragen miteinander zu vergleichen und zu bewerten, können wir die Operatoren und Operanden für jede Abfrage in einer Tabelle darstellen.

Die beiden Abfragen umfassen ein einfaches und ein komplexes Beispiel. Außerdem wurden noch zwei weitere Abfragen zur besseren Vergleichbarkeit hinzugefügt.

Einmal die Abfrage um `den Durchschnittswert von jedem Jahr zu berechnen` und die zweite Abfrage, die hinzugefügt wurde ist `Alle Tage / Stunden die wärmer als 35 grad sind`.

Abfrage SQL	Max Wert von jedem Jahr	Tagesdurchschnitts-temp vom Tag mit der höchstem Tagestemp.	Durchschnitt Wert von jedem Jahr	Alle Tage/ Stunden die wärmer als 35 grad sind
N1	17	35	17	4
N2	4	9	4	3
n1	9	12	9	4
n2	2	3	2	3
N	21	44	21	7
n	11	15	11	7
D	38,25	70	38,25	2,667
V	21,869	51,748	21,869	5,916
E	836,489	3622,36	836,489	15,778
T	46,472	201,242	46,472	0,877

Abfrage Flux	Max Wert von jedem Jahr	Tagesdurchschnitts-temp vom Tag mit der höchstem Tagestemp.	Durchschnitt Wert von jedem Jahr	Alle Tage/ Stunden die wärmer als 35 grad sind
N1	41	90	41	39
N2	30	66	30	26
n1	9	11	9	9
n2	24	35	25	17
N	71	156	71	65
n	33	46	34	26
D	7,688	14,143	7,38	10,324
V	107,814	259,39	108,735	91,973
E	828,874	3668,553	802,464	949,529
T	46,049	203,809	44,581	52,752

Die Analyse der beiden Abfragesprachen SQL und Flux zeigt deutliche Unterschiede in Bezug auf die Komplexität und Handhabung von Datenabfragen. Die Untersuchung umfasst vier spezifische Abfragen: den Maximalwert eines jeden Jahres, den Tagesdurchschnitt der höchsten Tagestemperatur, den Durchschnittswert eines jeden Jahres und die Identifikation aller Tage oder Stunden, die wärmer als 35 Grad sind.

Für die Abfrage des Maximalwerts eines jeden Jahres zeigt sich, dass die SQL-Abfrage 17 Operatoren und 4 Operanden enthält, während die Flux-Abfrage mit 41 Operatoren und 30 Operanden deutlich komplexer ist. Dieser Unterschied in der Komplexität deutet darauf hin, dass Flux zwar mehr Flexibilität bietet, dies jedoch auf Kosten einer höheren Anzahl von Operationen und einer größeren Komplexität geschieht.

In der Abfrage, die den Tagesdurchschnitt der höchsten Tagestemperatur berechnet, sind die Unterschiede noch ausgeprägter. Die SQL-Abfrage umfasst 35 Operatoren und 9 Operanden, während die Flux-Abfrage mit 90 Operatoren und 66 Operanden arbeitet. Dies ist auf die zusätzlichen Schritte in der Flux-Abfrage zurückzuführen, die erforderlich sind, um den Höchstwert zu berechnen und die entsprechenden Datumstransformationen durchzuführen.

Bei der Abfrage des Durchschnittswerts eines jeden Jahres zeigt sich ein ähnliches Bild wie bei der ersten Abfrage. Die SQL-Abfrage ist mit 17 Operatoren und 4 Operanden wieder deutlich einfacher gehalten, während die Flux-Abfrage 41 Operatoren und 30 Operanden erfordert. Dies unterstreicht erneut die höhere Komplexität der Flux-Abfragen.

Die Abfrage zur Identifikation aller Tage oder Stunden, die wärmer als 35 Grad sind, zeigt ebenfalls einen deutlichen Unterschied in der Komplexität. Die SQL-Abfrage benötigt nur 4 Operatoren und 3 Operanden, während die Flux-Abfrage 39 Operatoren und 26 Operanden umfasst. Die zusätzliche Filterbedingung und die spezifische Handhabung der Temperaturen tragen zur erhöhten Komplexität der Flux-Abfrage bei.

Zusammengefasst lässt sich feststellen, dass Flux-Abfragen durchgehend komplexer sind als die entsprechenden SQL-Abfragen. Dies zeigt sich in der höheren Anzahl von Operatoren und Operanden. Die höhere Komplexität von Flux-Abfragen deutet auf eine größere Flexibilität hin, bringt jedoch auch potenziell höheren Wartungsaufwand mit sich. Im Vergleich sind die SQL-Abfragen einfacher und effizienter zu verstehen und zu warten. Einfachere SQL-Abfragen könnten somit effizienter und leichter zu debuggen sein, während Flux-Abfragen aufgrund ihrer Komplexität mehr Rechenressourcen beanspruchen könnten.

Diese vergleichende Analyse verdeutlicht die Unterschiede in der Komplexität zwischen SQL- und Flux-Abfragen und hilft bei der Entscheidungsfindung, welche Abfragesprache für bestimmte Aufgaben besser geeignet ist. Während SQL durch Einfachheit und Effizienz überzeugt, bietet Flux eine größere Flexibilität, die jedoch mit einer höheren Komplexität verbunden ist.

4.5 Performance

Ein weiterer wichtiger Aspekt für den Vergleich der beiden Sprachen und Datenbanken ist die Schnelligkeit. Verschiedene Abfragen wurden mit beiden Datenbanken durchgeführt, um zu ermitteln, welche die Ergebnisse schneller liefert. Die Datenbanken selbst haben die Ausführungszeiten angegeben.

Die getesteten Abfragen waren:

1. *Alle Tage und Stunden, die wärmer als 35 Grad sind*
2. *Wärmster Tag und Stunde der Daten*
3. *Durchschnittstemperaturen jedes Monats jedes Jahres*
4. *Max Wert jedes Jahres*
5. *Tägliche Durchschnittstemperatur für jeden Monat*
6. *Tagesdurchschnittstemperatur von dem Tag mit der höchsten Tagestemperatur berechnen*

Abfrage:	SQL	Flux
1	~ 0,3 Sekunden	~ 0,3 Sekunden
2	~ 0,6 Sekunden	~ 0,2 – 0,3 Sekunden
3	~ 0,5 Sekunden	~ 0,7 Sekunden
4	~ 0,5 Sekunden	~ 0,2 Sekunden
5	~ 0,6 Sekunden	~ 0,5 Sekunden
6	~ 1 Sekunde	~ 0,2 Sekunden

Die Analyse der Ergebnisse zeigt deutlich, dass Flux in mehreren Fällen schneller ist als SQL. Ein Vergleich der Abfragen verdeutlicht diese Unterschiede:

Bei der Abfrage nach allen Tagen und Stunden, die wärmer als 35 Grad sind, liefern sowohl SQL als auch Flux das Ergebnis nahezu gleichzeitig, wobei beide jeweils etwa 0,3 Sekunden benötigen. Bei der Ermittlung des wärmsten Tages und der wärmsten Stunde der Daten zeigt sich Flux als deutlich schneller, mit einer Ausführungszeit von etwa 0,2 bis 0,3 Sekunden im Vergleich zu den etwa 0,6 Sekunden, die SQL benötigt.

Interessanterweise liefert SQL die Durchschnittstemperaturen jedes Monats jedes Jahres schneller (etwa 0,5 Sekunden) als Flux (etwa 0,7 Sekunden). Hingegen zeigt sich Flux bei der Abfrage nach dem Maximalwert jedes Jahres als wesentlich schneller, mit einer Ausführungszeit von nur etwa 0,2 Sekunden im Vergleich zu den etwa 0,5 Sekunden, die SQL benötigt.

Bei der Berechnung der täglichen Durchschnittstemperatur für jeden Monat ist Flux etwas schneller, mit einer Ausführungszeit von etwa 0,5 Sekunden, während SQL etwa 0,6 Sekunden benötigt. Schließlich zeigt sich Flux auch bei der Berechnung der Tagesdurchschnittstemperatur des Tages mit der höchsten Tagestemperatur als deutlich schneller, mit einer Ausführungszeit von etwa 0,2 Sekunden im Vergleich zu den etwa 1 Sekunde, die SQL benötigt.

Die Ergebnisse der Schnelligkeitsanalyse zeigen, dass Flux in mehreren Fällen schneller als SQL ist, insbesondere bei komplexen Abfragen wie der Ermittlung des Tagesdurchschnitts der höchsten Tagestemperatur. Obwohl SQL bei der Berechnung der monatlichen Durchschnittstemperaturen schneller ist, zeigt sich insgesamt, dass Flux bei vielen Abfragen effizienter arbeitet.

5 Fazit

Die vergleichende Analyse der beiden Datenbanken InfluxDB und MariaDB, unter Berücksichtigung der Abfragesprachen Flux und SQL, zeigt wesentliche Unterschiede in Bezug auf Komplexität, Flexibilität und Performance.

5.1 Komplexität und Flexibilität

Die Halstead-Metrik wurde verwendet, um die Komplexität der Abfragesprachen SQL und Flux zu bewerten. Die Ergebnisse zeigen, dass Flux-Abfragen durchgehend komplexer sind als die entsprechenden SQL-Abfragen, was sich in einer höheren Anzahl von Operatoren und Operanden manifestiert. Zum Beispiel enthält die SQL-Abfrage zur Ermittlung des Maximalwerts eines jeden Jahres 17 Operatoren und 4 Operanden, während die Flux-Abfrage mit 41 Operatoren und 30 Operanden wesentlich komplexer ist. Ähnliche Unterschiede zeigen sich bei der Berechnung des Tagesdurchschnitts der höchsten Tagestemperatur, wo die SQL-Abfrage 35 Operatoren und 9 Operanden benötigt, im Vergleich zu 90 Operatoren und 66 Operanden bei der Flux-Abfrage.

Diese höhere Komplexität von Flux-Abfragen weist auf eine größere Flexibilität hin, die es ermöglicht, komplexe Analysen und spezialisierte Anforderungen effizient zu bewältigen. Allerdings geht diese Flexibilität mit einem höheren Wartungsaufwand einher. Im Gegensatz dazu sind SQL-Abfragen aufgrund ihrer Einfachheit und Effizienz oft einfacher zu verstehen, zu warten und zu debuggen, insbesondere bei weniger komplexen Anfragen und routinemäßigen Datenbankoperationen.

Die Kombination von SQL und Flux ermöglicht es, je nach den spezifischen Anforderungen und der Komplexität der Datenabfragen die geeignete Lösung auszuwählen. Während SQL für klassische Datenbankanforderungen und einfache Abfragen bevorzugt werden kann, bietet Flux eine leistungsstarke Option für die Analyse von Zeitreihendaten und komplexen Abfrageoperationen.

5.2 Perfomance

Die Performance-Analyse verdeutlicht, dass Flux in vielen Szenarien schneller als SQL ist. Bei weniger komplexen Abfragen zeigt SQL eine robuste Performance, während Flux besonders bei der Verarbeitung großer Datenmengen und der detaillierten Analyse von Zeitreihen glänzt. In solchen Szenarien bietet Flux oft eine bessere Leistung. Bei einfachen Abfragen, wie der Ermittlung von Tagen und Stunden mit Temperaturen über 35 Grad, zeigen beide Sprachen vergleichbare Leistungen.

Jedoch übertrifft Flux SQL bei komplexeren Abfragen, wie der Bestimmung des wärmsten Tages, der höchsten Tagestemperatur und der Berechnung des Tagesdurchschnitts dieser Spitzenwerte, deutlich. In bestimmten Fällen zeigt Flux eine Leistungssteigerung um das Dreifache bis Fünffache im Vergleich zu SQL. Dennoch hat SQL seine Stärken, insbesondere bei Aufgaben wie der Berechnung der monatlichen Durchschnittstemperaturen über mehrere Jahre hinweg, wo es schneller als Flux arbeitet.

Die Geschwindigkeit der Datenabfrage ist entscheidend bei der Wahl zwischen SQL und Flux. Während SQL durch seine Einfachheit und Effizienz überzeugt, bietet Flux neben seiner größeren Flexibilität auch eine bemerkenswerte Geschwindigkeit in der Datenverarbeitung. Diese Eigenschaften machen Flux besonders geeignet für komplexe Abfragen und die Verarbeitung großer Datenmengen. Letztlich sollte die Entscheidung zwischen SQL und Flux auf den spezifischen Anforderungen und Prioritäten des jeweiligen Projekts basieren.

5.3 Entwicklungsaufwand und Verständnis

Der Entwicklungsaufwand und das Verständnis für SQL und Flux zeigen deutliche Unterschiede je nach Komplexität der Abfragen und den spezifischen Anforderungen des Projekts. SQL bietet einen klaren Vorteil bei einfachen und mittelkomplexen Abfragen aufgrund seiner Vertrautheit und der weit verbreiteten Nutzung in relationalen Datenbankumgebungen. Aufgrund der optimierten Strukturen für relationale Datenbanken ist SQL in der Regel leichter zu erlernen und zu verwenden. Dies erleichtert nicht nur die Entwicklung, sondern auch das Debugging und die Wartung von Abfragen.

Im Gegensatz dazu erfordert Flux bei komplexen Abfragen oft einen höheren Entwicklungsaufwand. Dies liegt an seiner spezialisierten Natur, die auf die Verarbeitung und Analyse von Zeitreihendaten optimiert ist. Flux bietet zwar eine erweiterte Flexibilität und leistungsstarke Funktionen für spezifische Analyseanforderungen, jedoch gehen diese Vorteile mit einer steileren Lernkurve einher. Entwickler und Datenanalysten müssen sich intensiver mit den speziellen Funktionalitäten von Flux vertraut machen, um diese effektiv nutzen zu können. Dieser initiale Aufwand bei der Einarbeitung kann sich langfristig jedoch auszahlen, insbesondere in Umgebungen, die komplexe Zeitreihendaten verarbeiten und anspruchsvolle Analyseoperationen durchführen müssen.

Die Entscheidung für SQL oder Flux hängt daher stark von den spezifischen Anforderungen des Projekts ab. Für Projekte mit überwiegend einfachen Abfragen und klassischen relationalen Datenbankanforderungen bietet sich SQL aufgrund seiner Einfachheit und Effizienz an. Es ermöglicht eine schnelle Entwicklung und ist robust für routinemäßige Datenbankoperationen.

Für Projekte, die komplexe Datenanalysen und zeitbasierte Abfragen erfordern, kann Flux die bessere Wahl sein. Trotz des höheren initialen Entwicklungsaufwands bietet Flux eine leistungsstarke Plattform für die Verarbeitung großer Datenmengen und die Durchführung detaillierter Zeitreihenanalysen.

Letztlich sollte die Entscheidung zwischen SQL und Flux darauf basieren, welche Abfragesprache die spezifischen Anforderungen des Projekts am besten erfüllt, sowohl in Bezug auf die Performance als auch auf den Entwicklungsaufwand und das Verständnis für die Datenbankoperationen.

5.4 Zusammenfassung

SQL und Flux bieten jeweils einzigartige Vorteile, die je nach den Anforderungen und der Komplexität der Datenabfragen entscheidend sind. SQL zeichnet sich durch seine Einfachheit und Effizienz aus, was es besonders geeignet macht für weniger komplexe Abfragen und routinemäßige Datenbankoperationen. Durch seine weit verbreitete Nutzung und optimierten Strukturen für relationale Datenbanken ermöglicht SQL eine schnelle Entwicklung, einfaches Debugging und eine robuste Performance.

Im Gegensatz dazu bietet Flux eine größere Flexibilität und leistungsstarke Funktionen für die Analyse von Zeitreihendaten und komplexe Abfrageoperationen. Trotz der höheren Komplexität der Flux-Abfragen zeigt sich Flux bei anspruchsvollen Datenanalysen oft schneller und effizienter. Insbesondere bei zeitbasierten Abfragen und der Verarbeitung großer Datenmengen profitiert Flux von seiner spezialisierten Natur und schnellen Ausführungszeiten.

Für Projekte mit überwiegend einfachen Anforderungen und klassischen relationalen Datenbankoperationen ist SQL die bevorzugte Wahl. Es bietet eine stabile Performance und eine einfache Handhabung, die sowohl für Entwickler als auch für Datenanalysten zugänglich ist.

Für komplexe Datenanalysen, spezialisierte Anforderungen und zeitintensive Abfragen ist Flux jedoch besser geeignet. Trotz des höheren Entwicklungsaufwands belohnt Flux mit seiner Flexibilität und Leistungsfähigkeit in komplexen Szenarien. Diese Eigenschaften machen es ideal für Anwendungen, die kontinuierliche Datenverarbeitung und detaillierte Zeitreihenanalysen erfordern.

Letztlich sollte die Entscheidung zwischen SQL und Flux auf den spezifischen Anforderungen und der Komplexität der Datenabfragen basieren. Beide Abfragesprachen bieten einzigartige Stärken, die je nach den Herausforderungen und Zielen des Projekts zum Tragen kommen können.

Literaturverzeichnis

[1] J. Wiegand, „Was sind Daten?,“ [23.10.2023]. [Online]. Available: https://praxistipps.chip.de/definition-was-sind-daten-einfach-erklaert_168074. [Zugriff 2024].

[2] L. Lins, „Datengestützte Entscheidungsfindung,“ [Online]. Available: https://datenbasiert.de/datengestuetzte-entscheidungsfindung/. [Zugriff am 04.2024].

[3] D. M. Siepermann, „Was ist "Datenbank"?,“ [Online]. Available: https://wirtschaftslexikon.gabler.de/definition/datenbank-30025. [Zugriff am 04.2024].

[4] L. Laurent, „Datenbanken,“ [30.08.2023]. [Online]. Available: https://appmaster.io/de/blog/arten-von-datenbankverwaltungssystemen. [Zugriff am 04.2024].

[5] WeEncrypt, „Zeitreihendatenbanken: Eine umfassende Einführung und Analyse,“ [15.12.2023]. [Online]. Available: https://weencrypt.pro/glossar/zeitreihendatenbanken-eine-umfassende-einfuehrung-und-analyse/. [Zugriff am 04.2024].

[6] D.-I. S. Luber, „Was sind Metadaten?,“ [11.01.2023]. [Online]. Available: https://www.security-insider.de/was-sind-metadaten-a-358dc75ad6978ece7b58d89b045f5ec6/. [Zugriff am 06.2024].

[7] Redaktion ComputerWeekly.de, „Zeitreihendatenbank (Time Series Database, TSDB),“ [01.2021]. [Online]. Available: https://www.computerweekly.com/de/definition/Zeitreihendatenbank-Time-Series-Database-TSDB. [Zugriff am 06.2024].

[8] R. Stolz, „Real Time Databases vs Time Series Databases vs Real-Time Analytics,“ [01.07.2021]. [Online]. Available: https://preset.io/blog/2021-7-1-time-series%20databases/. [Zugriff am 06.2024].

[9] DataScientest, „Relationale Datenbanken,“ [29.04.2023]. [Online]. Available: https://datascientest.com/de/relationale-datenbanken. [Zugriff am 06.2024].

[10] Oracle, „What is a relational database?,“ [Online]. Available: https://www.oracle.com/de/database/what-is-a-relational-database/. [Zugriff am 06.2024].

[11] J. Hudak, W. Nichols, J. McHale, M.-Y. Nam ,J. Delange, „Evaluating and Mitigating the Impact of,“ [12.2015]. [Online]. Available: https://insights.sei.cmu.edu/documents/1254/2015_005_001_448093.pdf. [Zugriff am 05.2024].

[12] Schneider Electric, „Metrik: Halstead-Komplexität,“ [2022]. [Online]. Available: https://product-help.schneider-electric.com/Machine%20Expert/V2.0/de/CodeAnly/CodeAnly/D-SE-0095969.html. [Zugriff am 05.2024].

[13] A. Zeller, „Software-Metriken,“ [Online]. Available: https://www.st.cs.uni-saarland.de/edu/se2/metriken.pdf. [Zugriff am 05.2024].

[14] Deutscher Wetterdienst, „Wetterdaten,“ [Online]. Available: https://cdc.dwd.de/portal/ . [Zugriff am 04.2024].

[15] U. Berger, „InfluxDB - eine Einführung,“ [2020]. [Online]. Available: https://programm.froscon.org/2020/system/event_attachments/attachments/000/000/614/original/influxdb.pdf. [Zugriff am 04.2024].

[16] Bitmotec, „InfluxDB – Einführung in die Open-Source-Zeitreihendatenbank - Abfragesprachen,“ [05.06.2023]. [Online]. Available: https://www.bitmotec.com/blog/influxdb-einfuehrung-in-die-open-source-zeitreihendatenbank/. [Zugriff am 04.2024].

[17] Influx Data, Inc, „Influx Dokumentation - Install,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/v2/install/. [Zugriff am 04.2024].

[18] InfluxData, Inc, „Influx Dokumentation - Create Bucket,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/v2/admin/buckets/create-bucket/. [Zugriff am 04.2024].

[19] InfluxData, Inc, „Data retention in InfluxDB,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/v2/reference/internals/data-retention/. [Zugriff am 06.2024].

[20] InfluxData, Inc, „Influx Dokumentation - Line Protocol,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/cloud/reference/syntax/line-protocol/. [Zugriff am 04.2024].

[21] InfluxData, Inc, „Influx Glossar,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/v1/concepts/glossary/#field-key. [Zugriff am 06.2024].

[22] F. Schürmeyer, „Unix-Timestamp,“ [03.04.2022]. [Online]. Available: https://hellocoding.de/blog/tools/generatoren/unix-timestamp. [Zugriff am 04.2024].

[23] InfluxData, Inc, „Influx Dokumentation - Annotation CSV,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/cloud/reference/syntax/annotated-csv/#annotated-csv-in-flux. [Zugriff am 04.2024].

[24] InfluxData, Inc, „Influx Dokumentation - Write CSV Data to InfluxDB,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/cloud/write-data/developer-tools/csv/. [Zugriff am 04.2024].

[25] InfluxData, Inc, „Influx Dokumentation - Python,“ [2024]. [Online]. Available: https://docs.influxdata.com/influxdb/cloud/api-guide/client-libraries/python/. [Zugriff am 04.2024].

[26] Ionos, „Python - String zu Float konvertieren,“ [07.02.2024]. [Online]. Available: https://www.ionos.de/digitalguide/websites/web-entwicklung/python-string-float/. [Zugriff am 04.2024].

[27] InfluxData, Inc, „InfluxDB - Query Data,“ [2024]. [Online]. Available: https://docs.influxdata.com/flux/v0/query-data/influxdb/. [Zugriff am 06.2024].

[28] InfluxData, Inc, „Influx Dokumentation - fliter(),“ [2024]. [Online]. Available: https://docs.influxdata.com/flux/v0/stdlib/universe/filter/. [Zugriff am 04.2024].

[29] InfluxData, Inc, „Influx Dokumentation - aggregateWindow(),“ [2024]. [Online]. Available: https://docs.influxdata.com/flux/v0/stdlib/universe/aggregatewindow/. [Zugriff am 04.2024].

[30] PureStorage, „Was ist MariaDB?,“ [Online]. Available: https://www.purestorage.com/de/knowledge/what-is-mariadb.html. [Zugriff am 04.2024].

[31] MariaDB, „ACID: Concurrency Control with Transactions,“ [2024]. [Online]. Available: https://mariadb.com/kb/en/acid-concurrency-control-with-transactions/. [Zugriff am 06.2024].

[32] MariaDB, „MariaDB Datensicherheit,“ [2024]. [Online]. Available: https://mariadb.com/database-topics/security/. [Zugriff am 06.2024].

[33] RemoteScout24, „Alles über MariaDB,“ [25.04.2023]. [Online]. Available: https://remotescout24.com/de/blog/602-alles-ueber-mariadb. [Zugriff am 06.2024].

[34] MariaDB, „MariaDB Download,“ [2024]. [Online]. Available: https://mariadb.org/download/?t=mariadb&p=mariadb&r=11.4.2&os=windows&cpu=x86_64&pkg=msi&mirror=wilhelm. [Zugriff am 06.2024].

[35] Business Systemhaus AG, „Was ist SQL? Eigenschaften & Befehle,“ [Online]. Available: https://bsh-ag.de/it-wissensdatenbank/sql/. [Zugriff am 05.2024].

[36] DataScientest, „SQL Index,“ [16.09.2023]. [Online]. Available: https://datascientest.com/de/sql-index-was-ist-das-wozu-dient-es. [Zugriff am 06.2024].

[37] Verifysoft Technology GmbH, „Messung von Halstead-Metriken mit Testwell CMT++ und Testwell CMTJava,“ [08.12.2023]. [Online]. Available: https://verifysoft.com/de_halstead_metrics.html. [Zugriff am 06.2024].

[38] InfluxData, Inc, „InfluxDB FindRecord(),“ [2024]. [Online]. Available: https://docs.influxdata.com/flux/v0/stdlib/universe/findrecord/. [Zugriff am 06.2024].

[39] InfluxData, Inc, „InfluxDB Truncate,“ [2024]. [Online]. Available: https://docs.influxdata.com/flux/v0/stdlib/date/truncate/. [Zugriff am 06.2024].

Anhang

6 Abfragen

6.1 SQL

Alle Tage/ Stunden, die wärmer als 35 Grad sind:

```
SELECT *

FROM wetter

WHERE wert > 35
```

#	Zeitstempel	Wert
1	1992-08-09 12:00:00	35,9
2	1992-08-09 13:00:00	36,8
3	1992-08-09 14:00:00	37,2
4	1992-08-09 15:00:00	36,7
5	1992-08-09 16:00:00	35,2
6	2006-07-20 12:00:00	36,3
7	2006-07-20 13:00:00	36
8	2006-07-20 14:00:00	35,9
9	2006-07-20 15:00:00	35,6
10	2015-07-04 13:00:00	35,3
11	2015-07-04 14:00:00	35,5
12	2015-07-04 15:00:00	36
13	2015-07-04 16:00:00	35,9
14	2022-07-20 11:00:00	35,5
15	2022-07-20 12:00:00	37,7
16	2022-07-20 13:00:00	38,2
17	2022-07-20 14:00:00	38,4
18	2022-08-04 13:00:00	35,8
19	2022-08-04 14:00:00	35,7

Wärmster Tag und Stunde der Daten:

```
 SELECT MAX(wert) AS höchste_temperatur, zeitstempel, HOUR(zeitstempel) AS
                                stunde
FROM wetter

WHERE wert = (SELECT MAX(wert) FROM wetter)

GROUP BY zeitstempel;
```

#	höchste_temperatur	zeitstempel	stunde
1	38,4	2022-07-20 14:00:00	14

Durchschnittstemperaturen jedes Monats jedes Jahres:

```
SELECT YEAR(zeitstempel) AS jahr, MONTH(zeitstempel) AS monat, AVG(wert) AS
    durchschnittstemperatur
FROM wetter
GROUP BY jahr, monat
ORDER BY jahr, monat;
```

#	jahr	monat	durchschnittstemperatur
1	1.949	1	2,520026889038823
2	1.949	2	3,0802083401940763
3	1.949	3	2,1935483933957194
4	1.949	4	9,48791666334081
5	1.949	5	11,617204308790225
6	1.949	6	13,49388890001509
7	1.949	7	16,105510767429106
8	1.949	8	15,984677421149387
9	1.949	9	16,25027776029375
10	1.949	10	11,096370970339624
11	1.949	11	4,825833339720137
12	1.949	12	3,995295698974802
13	1.950	1	-0,013709674569307476
14	1.950	2	3,106845242281755
15	1.950	3	5,270295693569126
16	1.950	4	6,778333324980404
17	1.950	5	12,604973120394574
18	1.950	6	16,577222222420904
19	1.950	7	16,7186828044153
20	1.950	8	17,49247310623046
21	1.950	9	13,041944440868166
22	1.950	10	8,245967742996992
23	1.950	11	4,441388890881919
24	1.950	12	-0,7637096778840147
25	1.951	1	1,4196236508207456
26	1.951	2	1,5092261927091473
27	1.951	3	2,100000003392818
28	1.951	4	7,079722225345257
29	1.951	5	10,85188171672084
30	1.951	6	15,185416650772094

#	jahr	monat	durchschnittstemperatur
646	2.002	10	8,034408598958004
647	2.002	11	4,737916672084895
648	2.002	12	-0,6350806487864384
649	2.003	1	0,6696236560281407
650	2.003	2	-0,76949404838628
651	2.003	3	5,048118274058065
652	2.003	4	8,77263889786684
653	2.003	5	13,310618270148513
654	2.003	6	17,695416694879533
655	2.003	7	19,37311829033718
656	2.003	8	19,3989247275937
657	2.003	9	14,365833355320824
658	2.003	10	5,999865596342872
659	2.003	11	7,191111107677635
660	2.003	12	3,285618284066278
661	2.004	1	0,5700268757839998
662	2.004	2	3,4014367782885486
663	2.004	3	4,826344082392351
664	2.004	4	9,732638902879424
665	2.004	5	11,991532280880918
666	2.004	6	14,834305570522945
667	2.004	7	16,19220430812528
668	2.004	8	18,833602164381293
669	2.004	9	14,398749993575944
670	2.004	10	10,382123668908433
671	2.004	11	5,19972222648147
672	2.004	12	3,40739247302014
673	2.005	1	3,910887098580759
674	2.005	2	0,6026785704418129
675	2.005	3	3,707123664598311

Max Wert jedes Jahres:

```
SELECT YEAR(zeitstempel), MAX(wert)
FROM wetter
GROUP BY YEAR(zeitstempel)
ORDER BY YEAR(zeitstempel);
```

#	YEAR(zeitstempel)	MAX(wert)
1	1.949	29,5
2	1.950	29,7
3	1.951	30,8
4	1.952	33,1
5	1.953	30,4
6	1.954	30,8
7	1.955	28,1
8	1.956	28,5
9	1.957	34,5
10	1.958	27,1
11	1.959	34,4
12	1.960	29,6
13	1.961	31,8
14	1.962	27,2
15	1.963	34,5
16	1.964	32,4
17	1.965	25,7
18	1.966	31,7
19	1.967	30,4
20	1.968	32,7
21	1.969	32,1
22	1.970	29,2
23	1.971	31,2
24	1.972	30,1
25	1.973	32,3
26	1.974	31,5
27	1.975	33,8
28	1.976	32,4
29	1.977	28,5
30	1.978	30,2
31	1.979	30,7

#	YEAR(zeitstempel)	MAX(wert)
46	1.994	34,6
47	1.995	31,8
48	1.996	32,9
49	1.997	32,4
50	1.998	29,7
51	1.999	33,1
52	2.000	34
53	2.001	32,4
54	2.002	33,5
55	2.003	33,9
56	2.004	29,9
57	2.005	32,6
58	2.006	36,3
59	2.007	32,8
60	2.008	31,9
61	2.009	33,7
62	2.010	34,1
63	2.011	29,2
64	2.012	34,3
65	2.013	34
66	2.014	31,1
67	2.015	36
68	2.016	31,5
69	2.017	28,4
70	2.018	35
71	2.019	34,8
72	2.020	32,6
73	2.021	34,1
74	2.022	38,4
75	2.023	31,5
76	2.024	16,5

Tägliche Durchschnittstemperatur für jeden Monat:

```
SELECT YEAR(zeitstempel) AS jahr, MONTH(zeitstempel) AS monat,
        DAY(zeitstempel) AS tag, AVG(wert) AS durchschnittstemperatur
FROM wetter
GROUP BY jahr, monat, tag
ORDER BY jahr, monat, tag;
```

#	jahr	monat	tag	durchschnittstemperatur
1	1.949	1	1	4,445833295583725
2	1.949	1	2	5,125000029802322
3	1.949	1	3	2,8166666900118194
4	1.949	1	4	1,1708333306014538
5	1.949	1	5	2,425000016887983
6	1.949	1	6	5,12500003973643
7	1.949	1	7	5,529166668653488
8	1.949	1	8	3,1666666666666665
9	1.949	1	9	0,4708333273107807
10	1.949	1	10	-2,1499999777103462
11	1.949	1	11	1,0916666683430474
12	1.949	1	12	2,075000005463759
13	1.949	1	13	1,3458333294838667
14	1.949	1	14	4,550000001986821
15	1.949	1	15	2,112499994536241
16	1.949	1	16	2,716666671757897
17	1.949	1	17	5,129166682561238
18	1.949	1	18	5,6875
19	1.949	1	19	7,8916667103767395
20	1.949	1	20	5,358333319425583
21	1.949	1	21	2,700000005463759
22	1.949	1	22	-0,22916666294137636

#	jahr	monat	tag	durchschnittstemperatur
20609	2.005	6	10	12,208333253860474
20610	2.005	6	11	9,93333331743876
20611	2.005	6	12	9,604166666666666
20612	2.005	6	13	11,34166669845581
20613	2.005	6	14	15,945833325386047
20614	2.005	6	15	18,0958331823349
20615	2.005	6	16	18,43749996026357
20616	2.005	6	17	17,604166587193806
20617	2.005	6	18	15,895833333333334
20618	2.005	6	19	19,224999944369
20619	2.005	6	20	22,94166664282481
20620	2.005	6	21	21,40000009536743
20621	2.005	6	22	17,075000047683716
20622	2.005	6	23	20,987499952316284
20623	2.005	6	24	23,987499952316284
20624	2.005	6	25	18,44166672229767
20625	2.005	6	26	16,037500023841858
20626	2.005	6	27	14,12499996026357
20627	2.005	6	28	15,458333293596903
20628	2.005	6	29	17,174999992052715
20629	2.005	6	30	17,987499992052715
20630	2.005	7	1	16,03749990463257
20631	2.005	7	2	18,216666499773662
20632	2.005	7	3	21,0625

Tagesdurchschnittstemperatur von dem Tag mit der höchsten Tagestemperatur berechnen

```
SELECT DATE(zeitstempel), AVG(wert)
FROM wetter
WHERE DATE(zeitstempel) = (
        SELECT DATE(zeitstempel)
        FROM wetter
       WHERE wert = (SELECT MAX(wert) FROM wetter)
       LIMIT 1)
GROUP BY DATE(zeitstempel);
```

#	DATE(zeitstempel)	AVG(wert)
1	2022-07-20	27,145833492279053

6.2 Flux

Alle Tage/ Stunden, die wärmer als 35 Grad sind:

```
from(bucket: "Wetter")
    |> range(start: 1949-01-01T01:00:00Z, stop: 2024-03-19T23:59:59Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> filter(fn: (r) => r["_value"] > 35)
```

_start	_stop	_time	_value	_field	_measurement
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	1992-08-09 14:00:00 GMT+2	35,90	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	1992-08-09 15:00:00 GMT+2	36,80	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	1992-08-09 16:00:00 GMT+2	37,20	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	1992-08-09 17:00:00 GMT+2	36,70	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	1992-08-09 18:00:00 GMT+2	35,20	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2006-07-20 14:00:00 GMT+2	36,30	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2006-07-20 15:00:00 GMT+2	36	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2006-07-20 16:00:00 GMT+2	35,90	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2006-07-20 17:00:00 GMT+2	35,60	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2015-07-04 15:00:00 GMT+2	35,30	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2015-07-04 16:00:00 GMT+2	35,50	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2015-07-04 17:00:00 GMT+2	36	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2015-07-04 18:00:00 GMT+2	35,90	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2022-07-20 13:00:00 GMT+2	35,50	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2022-07-20 14:00:00 GMT+2	37,70	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2022-07-20 15:00:00 GMT+2	38,20	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2022-07-20 16:00:00 GMT+2	38,40	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2022-08-04 15:00:00 GMT+2	35,80	temp	wetterdaten
1949-01-01 02:00:00 GMT+1	2023-11-03 00:00:00 GMT+1	2022-08-04 16:00:00 GMT+2	35,70	temp	wetterdaten

Wärmster Tag und Stunde der Daten:

```
from(bucket: "Wetter")
    |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:00:00Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> max()
    |> yield(name: "max_value_per_month")
```

_start	_stop	_time	_value	_field	_measurement
1970-01-01 01:00:00 GMT+1	2024-03-20 00:00:00 GMT+1	2022-07-20 16:00:00 GMT+2	38,40	temp	wetterdaten

Durchschnittstemperaturen jedes Monats jedes Jahres:

```
from(bucket: "Wetter")
    |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:59:59Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> aggregateWindow(every: 1mo, fn: mean, createEmpty: false)
    |> yield(name: "average_temp")
```

_start	_stop	_time	_value	_field	_measurement
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-02-01 01:00:00 GMT+1	2,52	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-03-01 01:00:00 GMT+1	3,08	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-04-01 01:00:00 GMT+1	2,19	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-05-01 02:00:00 GMT+2	9,49	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-06-01 02:00:00 GMT+2	11,62	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-07-01 02:00:00 GMT+2	13,49	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-08-01 02:00:00 GMT+2	16,11	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-09-01 02:00:00 GMT+2	15,98	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-10-01 02:00:00 GMT+2	16,25	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-11-01 01:00:00 GMT+1	11,10	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-12-01 01:00:00 GMT+1	4,83	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-01-01 01:00:00 GMT+1	4,00	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-02-01 01:00:00 GMT+1	-0,01	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-03-01 01:00:00 GMT+1	3,11	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-04-01 01:00:00 GMT+1	5,27	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-05-01 01:00:00 GMT+1	6,78	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-06-01 01:00:00 GMT+1	12,60	temp	wetterdaten

Max Wert jedes Jahres:

```
from(bucket: "Wetter")
    |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:59:59Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> aggregateWindow(every: 1y, fn: max, createEmpty: false)
    |> yield(name: "max_temperature")
```

_start	_stop	_time	_value	_field	_measurement
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1950-01-01 01:00:00 GMT+1	29,50	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1951-01-01 01:00:00 GMT+1	29,70	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1952-01-01 01:00:00 GMT+1	30,80	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1953-01-01 01:00:00 GMT+1	33,10	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1954-01-01 01:00:00 GMT+1	30,40	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1955-01-01 01:00:00 GMT+1	30,80	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1956-01-01 01:00:00 GMT+1	28,10	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1957-01-01 01:00:00 GMT+1	28,50	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1958-01-01 01:00:00 GMT+1	34,50	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1959-01-01 01:00:00 GMT+1	27,10	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1960-01-01 01:00:00 GMT+1	34,40	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1961-01-01 01:00:00 GMT+1	29,60	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1962-01-01 01:00:00 GMT+1	31,80	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1963-01-01 01:00:00 GMT+1	27,20	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1964-01-01 01:00:00 GMT+1	34,50	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1965-01-01 01:00:00 GMT+1	32,40	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1966-01-01 01:00:00 GMT+1	25,70	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1967-01-01 01:00:00 GMT+1	31,70	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1968-01-01 01:00:00 GMT+1	30,40	temp	wetterdaten

Tägliche Durchschnittstemperatur für jeden Monat:

```
from(bucket: "Wetter")
    |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:59:59Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> aggregateWindow(every: 1d, fn: mean, createEmpty: false)
    |> yield(name: "daily_average_temp")
```

_start	_stop	_time	_value	_field	_measurement
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-02 01:00:00 GMT+1	4,45	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-03 01:00:00 GMT+1	5,13	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-04 01:00:00 GMT+1	2,82	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-05 01:00:00 GMT+1	1,17	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-06 01:00:00 GMT+1	2,42	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-07 01:00:00 GMT+1	5,13	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-08 01:00:00 GMT+1	5,53	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-09 01:00:00 GMT+1	3,17	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-10 01:00:00 GMT+1	0,47	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-11 01:00:00 GMT+1	-2,15	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-12 01:00:00 GMT+1	1,09	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-13 01:00:00 GMT+1	2,07	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-14 01:00:00 GMT+1	1,35	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-15 01:00:00 GMT+1	4,55	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-16 01:00:00 GMT+1	2,11	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-17 01:00:00 GMT+1	2,72	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-18 01:00:00 GMT+1	5,13	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-19 01:00:00 GMT+1	5,69	temp	wetterdaten
1949-01-01 01:00:00 GMT+1	2024-03-20 00:59:59 GMT+1	1949-01-20 01:00:00 GMT+1	7,89	temp	wetterdaten

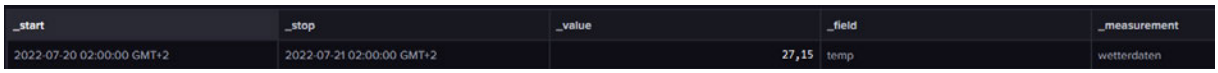
Tagesdurchschnittstemperatur von dem Tag mit der höchsten Tagestemperatur berechnen

```
import "date"
// Schritt 1: Finden Sie den Zeitstempel des wärmsten Tages in UTC
maxTemp =
  from(bucket: "Wetter")
    |> range(start: 1949-01-01T00:00:00Z, stop: 2024-03-19T23:59:59Z)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> max()
    |> findRecord(fn: (key) => true, idx: 0)
```

```
// Extrahiere das Datum aus dem Ergebnis von maxTemp und setze die Uhrzeit auf 00:00:00 UTC
startRange = date.truncate(t: maxTemp._time, unit: 1d)

// Definiere den Endzeitpunkt als den Beginn des nächsten Tages (UTC)
endRange = date.add(d: 1d, to: startRange)

// Daten aus einem Bucket laden basierend auf dem gefundenen Datum
data = from(bucket: "Wetter")
    |> range(start: startRange, stop: endRange)
    |> filter(fn: (r) => r["_measurement"] == "wetterdaten")
    |> filter(fn: (r) => r["_field"] == "temp")
    |> mean()
data
```



_start	_stop	_value	_field	_measurement
2022-07-20 02:00:00 GMT+2	2022-07-21 02:00:00 GMT+2	27,15	temp	wetterdaten

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel:

Vergleich von relationalen und Zeitreihendatenbanken.
Eine Untersuchung der Anfragekomplexität und -effizienz

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

29.07.2024	
Datum	Unterschrift