HAW HAMBURG

BACHELOR THESIS
Malte Behrmann

Szenarienbasierte Evaluation von Zeitreihen-Datenbanksystemen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Malte Behrmann

Szenarienbasierte Evaluation von Zeitreihen-Datenbanksystemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 21. März 2025

Malte Behrmann

Thema der Arbeit

Szenarienbasierte Evaluation von Zeitreihen-Datenbanksystemen

Stichworte

Zeitreihen-Datenbanksysteme, Vergleich, Evaluation, InfluxDB, TimescaleDB, MongoDB

Kurzzusammenfassung

Die effiziente Speicherung und Verarbeitung von Zeitreihendaten spielt eine wichtige Rolle in vielen Anwendungsbereichen. Aufgrund der besonderen Anforderungen von Zeitreihendaten, wurden in den letzten Jahren viele spezialisierte Zeitreihen-Datenbanksysteme entwickelt – darunter kommerzielle, aber auch auch freie open-source Varianten. Diese Arbeit vergleicht InfluxDB, TimescaleDB und MongoDB als Repräsentanten dreier Kategorien von open-source Zeitreihen-Datenbanksystemen – zunächst konzeptionell und anschließend experimentell anhand von drei realitätsnahen Szenarios. Für den experimentellen Vergleich wurde ein modulares und erweiterbares Testsystem entwickelt, das die Versuche automatisiert durchführt. Die Ergebnisse zeigen, dass spezialisierte Datenbanksysteme in vielen Fällen klare Vorteile bieten, jedoch kein einzelnes System universell empfohlen werden kann. Daher wurden auf Basis der Ergebnisse Nutzungsempfehlungen für verschiedene Anwendungsfälle abgeleitet.

Malte Behrmann

Title of Thesis

Scenario-based evaluation of Time Series Database Systems

Keywords

Time Series Database Systems, Comparison, Evaluation, InfluxDB, TimescaleDB, MongoDB

Abstract

The efficient storage and processing of time series data plays a crucial role in many application areas. Due to the specific requirements of time series data, numerous specialized time series database systems have been developed in recent years – both commercial and open-source variants. This thesis compares InfluxDB, TimescaleDB, and MongoDB as representatives of three categories of open-source time series database systems – first conceptually and then experimentally, using three realistic scenarios. For the experimental comparison, a modular and extendable testing system was developed to automate the evaluation of the databases. The results show that specialized database systems offer clear advantages in many cases; however, no single system can be universally recommended. Based on these findings, usage recommendations for various application scenarios were derived.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

DBMS Datenbankmanagementsystem

DBS Datenbanksystem

HTTP Hypertext Transfer Protocol

LSMT Log-Structured Merge-Tree

MA Metrikaufzeichnung

MQL MongoDB Query Language

RDBS relationale Datenbanksysteme

SME Systemmetrikerfassung

SQL Structured Query Language

TSC Time Series Collection

TSDBS Zeitreihen-Datenbanksystem (engl. Time Series Database System)

TSI Time Series Index

TSMT Time-Structured Merge-Tree

TSV Testsystem-Verwaltung

TU Testumgebung

VM virtuelle Maschine

WAL Write Ahead Log

1 Einleitung

Zeitreihen spielen eine zentrale Rolle in zahlreichen Anwendungsbereichen, da sie Datenpunkte in Abhängigkeit der Zeit erfassen können und diese analysierbar machen. Ob beispielsweise in der Finanzwelt zur Überwachung von Aktienkursen, in der Industrie für Sensor- und Maschinendaten oder in der Wissenschaft zur Klimaforschung [15, 45] – die effiziente Speicherung und Verarbeitung von Zeitreihendaten ist essenziell. Aufgrund der besonderen Anforderungen von Zeitreihendaten, wie hohen Schreibraten und der effizienten Durchführung von Zeitreihenanalysen, sind spezialisierte Zeitreihen-Datenbanksysteme erforderlich, die gezielt dafür optimiert sind.

Da es eine Vielzahl von Zeitreihen-Datenbanksystemen mit unterschiedlichen Eigenschaften und Konzepten gibt, wird in dieser Arbeit zunächst eine Kategorisierung vorgenommen, um die verschiedene Systeme zu analysieren. Auf Basis ihrer Architektur und ihres Funktionsumfangs werden Zeitreihen-Datenbanksysteme in drei Kategorien eingeteilt:

1. reine Zeitreihen-Datenbanksysteme
2. Erweiterungen von relationalen Datenbanksystemen
3. Erweiterungen von NoSQL-Datenbanksystemen

Für jede dieser Kategorien wird ein Repräsentant ausgewählt, der detailliert untersucht und mit den anderen Systemen verglichen wird. Die Repräsentanten InfluxDB, TimescaleDB und MongoDB dienen dabei als Beispiele für die Kategorien und werden hinsichtlich ihrer Leistungsfähigkeit in drei verschiedenen Szenarios gegenübergestellt.

Der Vergleich erfolgt sowohl konzeptionell als auch experimentell. Zunächst werden die theoretischen Grundlagen von Zeitreihen und Zeitreihenanalysen sowie die Eigenschaften relationaler und NoSQL-Datenbanksysteme erläutert. Anschließend folgt eine detaillierte

Erklärung der internen Funktionsweise der ausgewählten Repräsentanten. Im konzeptionellen Vergleich werden die Zeitreihen-Datenbanksysteme hinsichtlich ihrer Strukturanforderungen der Zeitreihen, Abfragesprachen und weiterer funktionaler Aspekte analysiert, um ihre Eignung für verschiedene Anwendungsfälle zu bewerten.

Im experimentellen Teil der Arbeit werden die Systeme anhand realitätsnaher Szenarios untersucht, darunter die Verarbeitung von Sensordaten in einem Smart-Home-Umfeld, die Analyse großer Zeitreihen aus dem Taxiverkehr sowie das Monitoring kurzlebiger Dienste. Durch eine systematische Evaluierung in einer dynamisch konfigurierbaren Testumgebung lassen sich Aussagen über Eignung der verschiedenen Systeme für spezifische Anwendungsbereiche treffen.

Für den experimentellen Vergleich wird ein modulares und erweiterbares Testsystem entwickelt, das die Datenbanksysteme unter realistischen Bedingungen automatisiert analysiert. Es führt die Szenarios aus und erfasst Anfragelatenzen, zentrale Systemmetriken und den Stromverbrauch, wodurch eine Bewertung der Systeme möglich wird.

Diese Untersuchung soll einen Überblick über die Leistungsfähigkeit und Eignung verschiedener Zeitreihen-Datenbanksysteme bieten. Die Ergebnisse dienen als Entscheidungshilfe für die gezielte Auswahl eines geeigneten Systems für den jeweiligen Anwendungsfall.

2 Grundlagen

2.1 Zeitreihen

Zeitreihen spielen in unserer zunehmend vernetzten und digitalisierten Welt eine zentrale Rolle und begegnen uns in vielen Bereichen des Alltags – z. B. bei Wetterberichten, Aktienkursen wie dem DAX (siehe Abb. 2.1) oder den monatlichen Arbeitslosenzahlen. Auch in der Industrie sind Zeitreihen unverzichtbar, etwa um Maschinen zu überwachen und frühzeitig Anzeichen für Verschleiß zu erkennen. Ebenso werden Zeitreihen in der Wissenschaft angewendet, um Analysen und Auswertungen durchzuführen. In Rechenzentren werden Zeitreihen häufig zur Überwachung von Servern eingesetzt und stellen einen weit verbreiteten Anwendungsfall dar [15, 45].

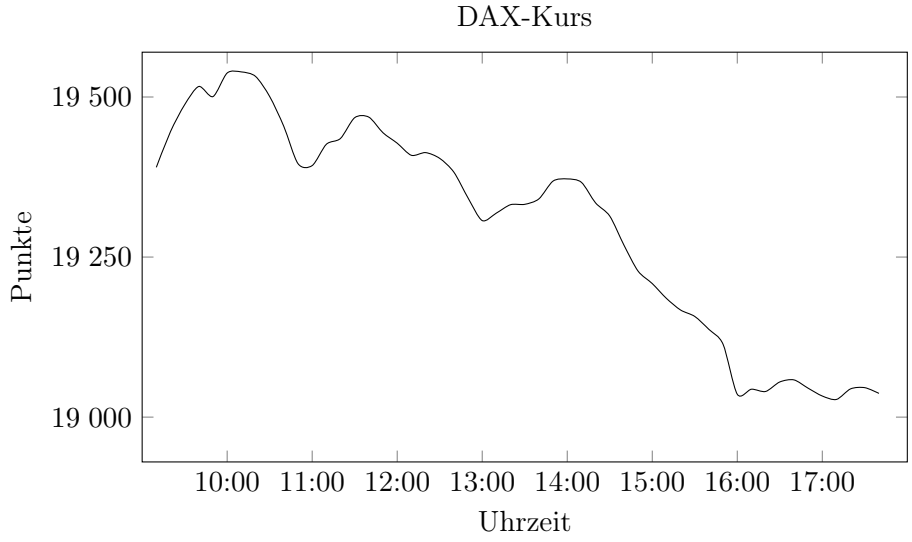


Abbildung 2.1: DAX-Kurs vom 06.11.2024 (Daten aus [14])

In der Statistik werden Zeitreihen wie folgt definiert: Eine Zeitreihe ist eine geordnete Folge $(x_t)_{t \in T}$, die aus Werten einer gewissen Größe besteht, wobei sich die Werte in der Menge der reellen Zahlen ($\mathbb{R}$) befinden. In der Regel sind dabei die zeitlichen Abstände identisch. Die Parametermenge $T = \mathbb{N}$ stellt die Zeitpunkte dar, die häufig aber auch mit negativen Zahlen (also $T = \mathbb{Z}$) genutzt wird, da so statistische Verfahren einfacher zu definieren sind, weil keine Sonderfälle für die Werte nahe dem Nullpunkt beachtet werden müssen [78, 45]. In dieser Arbeit wird die vereinfachte Definition von Chatfield [9] für Zeitreihen genutzt, da hier nicht im Detail auf die statistischen Verfahren zur Zeitreihenanalyse eingegangen wird.

Nach Chatfield [9] wird eine Zeitreihe als eine Folge von Beobachtungen eines bestimmten Merkmals definiert, die entweder kontinuierlich oder in regelmäßigen Abständen über die Zeit hinweg erfasst werden. Diese Definition wird ebenfalls von Brockwell und Davis [7] genutzt.

Zeitreihen können auf verschiedene Weisen klassifiziert werden. Zum einen unterscheidet man zwischen *kontinuierlichen (engl. continuous)* und *diskreten (engl. discrete)* Zeitreihen [9, 7]. Kontinuierliche Zeitreihen treten dort auf, wo Werte laufend entstehen. Ein Beispiel dafür ist ein Temperatursensor, dessen Ausgangsspannung sich durchgängig entsprechend der Temperatur ändert. Diskrete Zeitreihen bestehen hingegen aus Beobachtungen, die nur zu bestimmten, meist gleichmäßigen Zeitpunkten erhoben werden. Die Art der Werte spielt dabei keine Rolle: Kontinuierliche Zeitreihen können diskrete Werte speichern (z. B. Tür auf/zu) und diskrete Zeitreihen können kontinuierliche Werte beinhalten (z. B. stündliche Temperatur). In der Praxis entsteht laut Shumway und Robert [79] aus nahezu jeder eigentlich kontinuierlichen Zeitreihe eine diskrete Zeitreihe, wenn die Daten gespeichert werden müssen. Dies liegt inhärent an der Art und Weise, wie die Daten erfasst und von Rechnern verarbeitet werden können.

Eine weitere Möglichkeit Zeitreihen zu klassifizieren ist die Unterscheidung in stationäre und nicht-stationäre Zeitreihen. Stationär bedeutet dabei, dass sich stochastische Eigenschaften nicht über die Zeit hinweg ändern, wodurch Prognosen zukünftiger Werte möglich werden [45, 7]. Zudem gibt es viele weitere Eigenschaften, anhand derer Zeitreihen ebenfalls klassifiziert werden können – diese sind jedoch vor allem im Bereich der Statistik relevant.

2.2 Zeitreihenanalysen

Zeitreihen und deren Analysen sind keineswegs eine moderne Erfindung. Bereits im 19. Jahrhundert wurden Logbücher von Schiffen ausgewertet, um mithilfe von Daten wie Geschwindigkeit, Standort und Windverhältnissen die Routen der Schiffe zu optimieren [15]. Zeitreihen werden auch eingesetzt, um Trends in Daten zu identifizieren. Ein Beispiel dafür ist die Arbeit von Keeling, der 1958 auf Hawaii begann, die CO_2-Konzentration in der Atmosphäre zu messen und diese in der bislang längsten kontinuierlichen Zeitreihe dieser Art aufzuzeichnen. Seine Messungen konnten dazu beitragen, den menschlichen Einfluss auf den Klimawandel nachzuweisen. Aufgrund ihrer hohen Bedeutung wurde dieser Zeitreihe der Name „Keeling-Kurve“ gegeben [15, 63]. Auch in der Luftfahrt werden Zeitreihen genutzt, um das Verhalten von Flugzeugen zu analysieren [15]. Flugdatenschreiber, umgangssprachlich oft als „Black Box“ bezeichnet, zeichnen die wichtigsten Flugdaten auf, um im Fall eines Unglücks den Hergang rekonstruieren zu können.

Die Zeitreihenanalyse ist ein Teilbereich der Statistik, der sich nach Chatfield [9] in die Beschreibung, Erklärung, Vorhersage und Regelung von Zeitreihendaten unterteilen lässt:

- *Beschreibung*: In der beschreibenden Analyse werden die Struktur und grundlegende Eigenschaften einer Zeitreihe untersucht. Beispielsweise können Trends oder gewisse Muster erkannt werden. Hierzu ist es u. a. sinnvoll, die Daten zuerst in einem Diagramm visuell darzustellen. Zudem gibt es weitere fortgeschrittene Techniken für die Modellierung und Analyse, die z. B. Ausreißer erkennen können.

- *Erklärung*: Die erklärende Analyse soll die Ursachen und Zusammenhänge aufzeigen, die zu den zu analysierenden Daten geführt haben. Daraus können im Anschluss Kausalitäten in den Daten erkannt werden. Beispiele für Methoden, die dieser Kategorie zuzuordnen sind, sind die Korrelations- und Regressionsanalyse.

- *Vorhersage*: Die vorhersagende bzw. prognostizierende Zeitreihenanalyse befasst sich mit der Schätzung zukünftiger Werte auf Basis der vorliegenden Daten. Hierzu werden statistische Verfahren, wie z. B. ARMA (engl. für Autoregressive Moving Average), genutzt, die auf den Einsatz in Zeitreihen spezialisiert sind.

- *Regelung*: Die Regelung bezieht sich auf die Anwendung der Zeitreihenanalyse zur Steuerung von Entscheidungen, wie z. B. in der Finanzwirtschaft. Hier kann die

Analyse von Zeitreihendaten genutzt werden, um über den Kauf oder Verkauf von Aktien basierend auf prognostizierten Kursverläufen zu entscheiden.

Um die immer größer werdenden Mengen an Zeitreihen zu speichern, zu verwalten und zu analysieren, wurde über die Jahre eine Vielzahl an verschiedenen Zeitreihen-Datenbanksystemen auf den Markt gebracht. Dabei gibt es kommerzielle Angebote, aber auch viele open-source Datenbanksysteme, die miteinander konkurrieren. Die Notwendigkeit, spezialisierte Datenbanken für Zeitreihen zu entwickeln besteht u. a. darin, dass die Analyse von Zeitreihendaten besondere Anforderungen hat. In Kapitel 3 wird dies weiter diskutiert.

Diese Arbeit beschäftigt sich jedoch nicht mit den Details der Zeitreihenanalyse, sondern mit Datenbanksystemen, die besonders auf Zeitreihen und deren Speicherung und Verarbeitung spezialisiert sind. Grundlage dafür sind häufig sowohl relationale als auch NoSQL-Datenbanksysteme, die deswegen in den nächsten beiden Abschnitten genauer beleuchtet werden sollen.

2.3 Relationale Datenbanksysteme

Relationale Datenbanksysteme (RDBS) basieren auf dem Relationenmodell, das von Codd [11] im Jahr 1970 vorgestellt wurde, sowie der relationalen Algebra, die Codd [12] im Jahr 1972 entwickelte. Ein grundlegendes Prinzip von RDBS ist, dass die gespeicherten Daten in einer wohldefinierten Form, dem sogenannten Schema, stets korrekt und konsistent sind. Im Folgenden werden wichtige Aspekte relationaler Datenbanksysteme beschrieben.

2.3.1 Relationenmodell

Die grundlegende Idee des Relationenmodells besteht darin, Daten mithilfe von Relationen zu modellieren [47, 82]. Dadurch wird es unter anderem möglich, die Operationen auf den Daten durch mathematische Beweise auf Vollständigkeit und Korrektheit zu überprüfen. Eine weit verbreitete Darstellung der Relationen sind Tabellen mit Spalten und Zeilen, wie sie auch von SQL (engl. für Structured Query Language) genutzt wird. Bevor auf die Struktur der Relationen eingegangen werden kann, werden einige grundlegende Begriffe anhand der Beispieltabelle 2.1 erklärt.

Tabelle 2.1: Beispiel einer Tabelle im Relationenmodell (angelehnt an [82])

Autos

AutoNr	*Marke*	*Farbe*
1	Opel	silber
2	VW	rot
3	Opel	schwarz
4	Audi	rot

- *Attribute bzw. Merkmale* sind die Spalten einer Tabelle (im Beispiel „AutoNr“, „Marke“ und „Farbe“).
- *Tupel* sind die Zeilen der Tabelle, welche auch Datensatz genannt werden. Ein wichtiger Aspekt ist, dass die Tupel innerhalb einer Tabelle einmalig sind.
- *Identifikationsschlüssel* bestehen aus einem oder mehreren Attributen, die zusätzlich zwei Eigenschaften erfüllen müssen: Sie müssen jedes Tupel eindeutig identifizieren, und es darf keinen anderen Schlüssel mit weniger Attributen geben. Im Beispiel ist das Attribut „AutoNr“ ein Identifikationsschlüssel.
- *Datenwerte* sind die Zellen der Tabelle bzw. die einzelnen Werte der Tupel.

Im Relationenmodell wird eine Tabelle durch eine Relation von Tupeln dargestellt, die aus einer Teilmenge des kartesischen Produkts über den Wertebereichen der Attribute – den Domänen – bestehen:

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n \tag{2.1}$$

wobei D_i die Domäne des Attributs i ist.

2.3.2 Relationale Algebra

Die relationale Algebra definiert verschiedene Grundoperationen auf den Relationen des Relationenmodells (siehe Abschnitt 2.3.1) [82, 77], die im Folgenden kurz erläutert werden:

- *Projektion*: Die Projektion $\pi_{A_1,\ldots,A_n}$ wählt die $A_1, \ldots, A_n$ Attribute (Spalten) aus der Eingaberelation aus, die in der Ergebnisrelation enthalten sein sollen. Attribute, die nicht in $A_1, \ldots, A_n$ enthalten sind, werden nicht übernommen.

- *Selektion*: Die Selektion σ_Θ übernimmt nur die Tupel (Zeilen) aus der Eingaberelation in die Ergebnisrelation, die das Prädikat Θ erfüllen.
- *Umbenennung*: Die Umbenennung $\rho_{A_1,\ldots,A_n}$ gibt den Attributen (Spalten) $A_1, \ldots, A_n$ einen neuen Namen.

Zu den Grundoperationen zählen ebenfalls das kartesische Produkt, die Mengenvereinigung und -differenz, die wie in der klassischen Mengenlehre angewendet werden.

Außerdem gibt es weitere Operationen, die aus Kombinationen der oben genannten Grundoperationen entstehen. Diese sollen in dieser Arbeit jedoch nicht näher beleuchtet werden und lassen sich in [82] nachlesen.

2.3.3 Transaktionen und ACID

Die meisten relationalen Datenbanken bieten die Funktionalität von Transaktionen [82, 47, 77] an, die dafür sorgen, dass mehrere Operationen auf dem Datenbankmanagementsystem (DBMS) als eine Einheit gruppiert werden können. Anwendungen, die das DBMS nutzen, können Transaktionen starten (`BEGIN`) und diese entweder durch Übernahme der Änderungen (`COMMIT`) abschließen oder durch Rückgängigmachen der Änderungen (`ROLLBACK`) wieder beenden. Transaktionen erfüllen dabei die ACID-Eigenschaften, die von Haerder und Reuter [22] erdacht wurden. ACID steht dabei für die folgenden Begriffe:

- ***A**tomicity (Atomarität):* Die Eigenschaft der Atomarität gibt an, dass alle Transaktionen entweder vollständig mit allen Änderungen übernommen werden oder dass im Falle eines Fehlers bzw. eines Zurückrollens (`ROLLBACK`) keine der Änderungen gespeichert werden.
- ***C**onsistency (Konsistenz):* Die Konsistenz beschreibt, dass die Datenbank zu jedem Zeitpunkt einen gültigen (konsistenten) Zustand hat. Das heißt, dass eine Anwendung, die das DBMS nutzt, nicht darauf ausgelegt sein muss, mit inkonsistenten Daten zu arbeiten.
- ***I**solation (Isolation):* Die Isolation von Transaktionen garantiert, dass Anwendungen, die das DBMS nutzen, ihre Anfragen ungestört von anderen, möglicherweise parallel laufenden Anwendungen durchführen können.

- ***D**urability (Dauerhaftigkeit):* Nachdem eine Transaktion abgeschlossen und gespeichert wurde, bleibt diese dauerhaft erhalten. Auch im Falle eines Systemfehlers gehen keine Daten verloren.

Durch die Garantie der Isolation müssen Transaktionen in einigen Fällen auf andere warten. Da dies zu einem Performanz-Problem führen kann und diese Eigenschaft nicht in allen Fällen benötigt wird, wurden verschiedene Isolationsstufen *(engl. isolation level)* eingeführt, die je nach Stufe nur gewisse Isolationseigenschaften erhalten. Die Stufen sind zwar in SQL angegeben, variieren jedoch leicht in den verschiedenen Implementationen der Datenbanksysteme (siehe z. B. PostgreSQL [87]).

2.4 NoSQL-Datenbanksysteme

NoSQL-Datenbanksysteme [76, 16] können als eine Antwort auf das starre Datenmodell relationaler Datenbanksysteme gesehen werden, die um das Jahr 2009 vermehrt aufkamen. Ihre Kernkompetenz liegt in der Flexibilität und Skalierbarkeit, die in der Regel durch nicht fest definierte Schemata und die Nutzung verteilter Systeme erreicht werden. NoSQL soll dabei nicht beschreiben, dass diese DBS kein SQL unterstützen, sondern vielmehr, dass sie nicht nur SQL (**N**ot **o**nly **SQL**) nutzen.

2.4.1 BASE / CAP-Theorem

Einige NoSQL-Datenbanksysteme nutzen BASE als Konsistenzmodell [76, 47, 17], welches im Gegensatz zu ACID (siehe Abschnitt 2.3.3) geringere Anforderungen an die Konsistenz des DBS stellt und stattdessen die Verfügbarkeit in den Fokus rückt. Die Abkürzung wurde dabei bewusst so gewählt, dass sie den Kontrast zu ACID zeigt. BASE steht für ***B**asicly **A**vailable*, ***S**oft State* und ***E**ventually Consistent* und sagt aus, dass die Knoten eines verteilten Systems fast immer verfügbar *(Basicly Available)* sind. Die Konsistenz der Daten wird jedoch nur dann garantiert, wenn auf dem System keine Änderungen mehr vorgenommen werden und die Knoten auf einen gemeinsamen Stand konvergiert sind. Man spricht hierbei von einem System, das „Eventually Consistent“ ist. Der Systemzustand in dem Zeitraum, bevor die Konsistenz eintritt, wird als „Soft State“ bezeichnet.

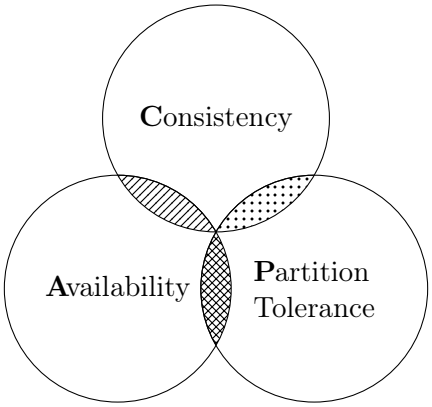


Abbildung 2.2: Visuelle Darstellung des CAP-Theorems, wobei die gemusterten Bereiche jeweils die möglichen Kombinationen der CAP-Eigenschaften zeigen

Eng mit BASE verwandt ist das CAP-Theorem, welches von Brewer [5] entwickelt und von Gilbert und Lynch [19] bewiesen wurde, und besagt, dass verteilte Rechnersysteme nur jeweils zwei der drei CAP-Eigenschaften Konsistenz *(**C**onsistency)*, Verfügbarkeit *(**A**vailability)* und Partitionierungstoleranz *(**P**artition Tolerance)* erfüllen können (siehe Abb. 2.2), wodurch folgende Kombinationen entstehen [47, 16, 17]:

- *CA*: Das System soll immer konsistent und verfügbar sein, somit darf es nicht zu einer Teilung bzw. Partitionierung des Systems kommen. Beispiele für CA-Systeme sind die DBS, die ACID (siehe Abschnitt 2.3.3) nutzen.
- *CP*: Systeme, die konsistent und partitionierungstolerant sind, reduzieren im Falle einer Teilung die Verfügbarkeit, um keine Inkonsistenzen zu erzeugen. Manche NoSQL DBS nutzen dieses Modell bzw. lassen sich entsprechend konfigurieren.
- *AP*: Im Gegensatz zu CP-Systemen können AP-Systeme während einer Partitionierung inkonsistent werden – sie bleiben jedoch jederzeit verfügbar. Dieses Modell wird ebenfalls von manchen NoSQL DBS genutzt.

Jedoch gibt es auch Kritik am CAP-Theorem, da sich die wenigsten Systeme hart in eine der Kategorien einteilen lassen. NoSQL-Systeme lassen sich oft feingranular bezüglich der Konsistenz oder Verfügbarkeit im Partitionierungsfall konfigurieren – teils ist dies sogar pro einzelner Anfrage möglich (siehe [62]). Zudem ist das CAP-Theorem lediglich im Falle einer Partitionierung relevant, welche in der Regel nur selten auftritt. Solange ein System nicht partitioniert ist, können Konsistenz und Verfügbarkeit gleichermaßen garantiert werden. Um dies zu adressieren, wurde von Abadi [1] PACELC (engl. für If **P**artition,

Availability and **C**onsistency, **E**lse, **L**atency and **C**onsistency) als Verbesserung von CAP vorgestellt, auf das hier jedoch nicht weiter eingegangen werden soll. Details hierzu lassen sich in [18] nachlesen.

2.4.2 Kategorien von NoSQL-Datenbanksystemen

NoSQL-Datenbanksysteme lassen sich in vier Kategorien gruppieren [76, 16]:

- *Key-Value-orientiert*: Diese Datenbanksysteme speichern ihre Daten in Form von Schlüssel-Wert-Paaren, wobei die Schlüssel den Index der Datenbank darstellen. Die Werte können aus einfachen Zeichenketten (engl. Strings) oder auch komplexeren Typen wie Listen bestehen. Ein Kernaspekt, warum Key-Value-DBS in vielen Fällen im Einsatz sind, ist die hohe Performanz, die sich durch das einfache Datenmodell dieser Systeme erzielen lässt. Das Datenbanksystem Redis ist ein häufig verwendeter Vertreter von Key-Value-DBS [74].

- *Dokumentenorientiert*: Datenbanksysteme, die dieses Modell nutzen, speichern Daten als Dokument. Mit Dokumenten werden dabei Datenstrukturen beschrieben, die entweder skalare Daten, Listen oder weitere Dokumente enthalten. Innerhalb eines Dokuments werden die einzelnen enthaltenen Daten als Felder bezeichnet. Meistens werden Dokumente im JSON-Format gespeichert, wobei XML aber auch teils möglich ist. Dokumentenorientierte Datenbanksysteme sind in der Regel schemafrei, das heißt, die Struktur der Dokumente muss nicht vorher angegeben werden. Ein gängiges Beispiel für diese Kategorie ist MongoDB [61], auf das in Kapitel 3.4 im Kontext von Zeitreihen-Datenbanksystemen genauer eingegangen wird.

- *Spaltenorientiert*: Spaltenorientierte Datenbanksysteme funktionieren ähnlich wie relationale DBS (siehe Abschnitt 2.3), jedoch werden die Daten hier nicht in Form von Tupeln (Zeilen) sondern spaltenweise gespeichert. Dies kann bei gewissen Operationen, wie z. B. „Summe über alle Datenwerte einer Spalte“, die Performanz erheblich verbessern, da die Daten weitestgehend sequenziell aus dem Speicher gelesen werden können. Ein weiterer Unterschied zu relationalen DBS besteht darin, dass das Schema der Datenbank deutlich flexibler ist: Es können jederzeit beliebig viele Spalten hinzugefügt oder entfernt werden. Ein verbreitetes Open-Source-System hierfür ist Apache Cassandra [83].

- *Graphorientiert*: Speichert und verarbeitet ein DBS seine Daten als Graph mit Knoten und Kanten, so spricht man von einem graphorientierten Datenbanksystem. Dieses Datenmodell ermöglicht es, direkt in dem DBMS Graphalgorithmen zu nutzen, wie z. B. die Berechnung des kürzesten Wegs von Knoten A nach Knoten B. Die höhere Komplexität dieses Modells im Vergleich zu anderen Modellen wirkt sich jedoch negativ auf die Performanz aus. Neo4J [65] ist diesbezüglich ein bekanntes Graphdatenbanksystem.

3 Zeitreihen-Datenbanksysteme

In diesem Kapitel werden Zeitreihen-Datenbanksysteme kategorisiert und für jede der Kategorien ein Repräsentant gewählt, auf dessen interne Funktionsweise zusätzlich eingegangen wird. Die Repräsentanten werden in den Kapiteln 5 und 6 zunächst konzeptionell und anschließend experimentell verglichen. In Kapitel 5 wird auf den genauen Funktionsumfang der jeweiligen DBS weiter eingegangen. Am Ende des Kapitels wird außerdem auf verwandte Arbeiten verwiesen.

3.1 Kategorisierung und Auswahl von Repräsentanten

Die Initiative DB-Engines [71] führt 61 verschiedene Datenbanksysteme mit Zeitreihenfunktionalität auf. Aufgrund der Vielzahl verschiedener Systeme wurde in dieser Arbeit entschieden, die Systeme zu kategorisieren und jeweils einen Repräsentanten pro Kategorie auszuwählen. Die Kategorien wurden ähnlich wie bei Bader [3] primär anhand des verwendeten Datenmodells gewählt. Somit wurden die Kategorien *reine Zeitreihen-Datenbanksysteme*, *Erweiterungen von relationalen DBS* und *Erweiterungen von NoSQL DBS* definiert. Bader [3] hat zudem eine Kategorie für proprietäre DBS eingeführt, die hier jedoch nicht genutzt wird, da in dieser Arbeit ausschließlich open-source DBS miteinander verglichen werden.

Die Repräsentanten wurden u. a. anhand ihrer Popularität, gemessen von DB-Engines [71], ausgewählt (siehe Abb. 3.1). Zudem wurde auf die Aktivität der Entwicklung sowie die freie Verfügbarkeit der Software und des Quellcodes (Open-Source) geachtet.

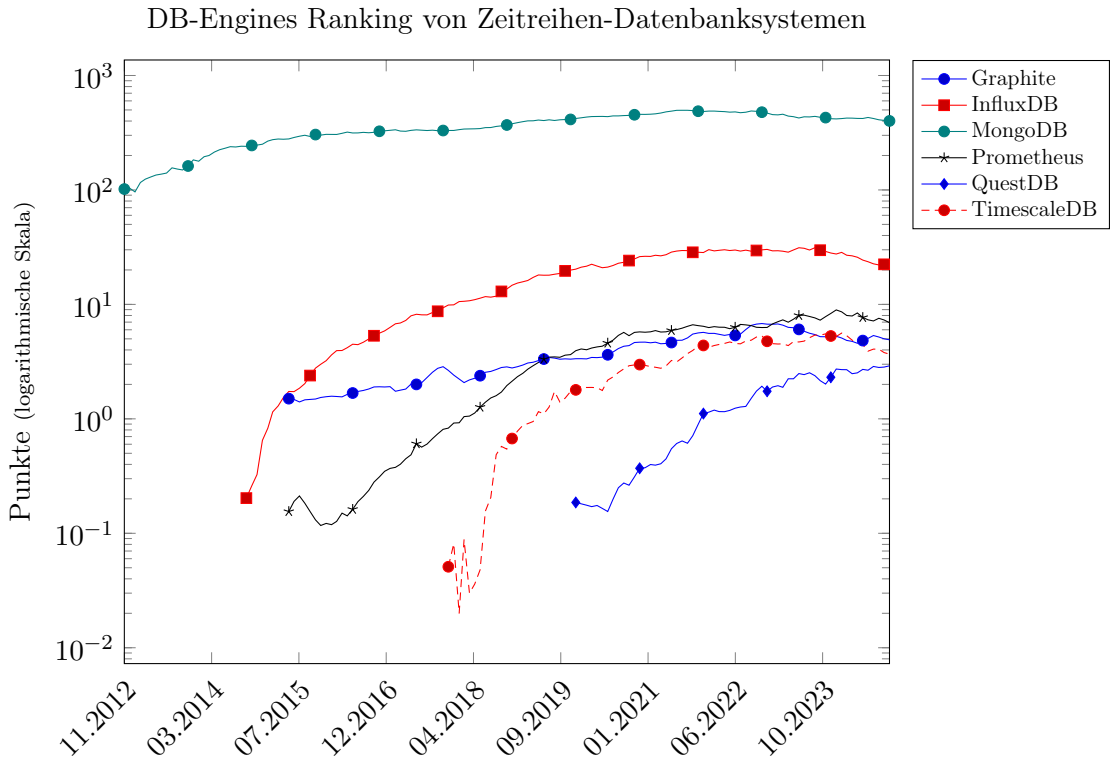


Abbildung 3.1: Ranking von ausgewählten Zeitreihen-Datenbanksystemen – gemessen anhand ihrer Popularität seit Beginn der Erfassung (Daten aus [71])

Die Kategorien sind wie folgt aufgebaut:

- *reine Zeitreihen-Datenbanksysteme*: Zu dieser Kategorie zählen Datenbanksysteme, die von Grund auf als Zeitreihendatenbank entwickelt wurden.

  Repräsentant dieser Kategorie: InfluxDB [42]
  Andere DBS, die auch dieser Kategorie angehören: Prometheus [70], Graphite [21] und OpenTSDB [84]

- *Erweiterung von relationalen DBS*: Hierbei handelt es sich um Datenbanksysteme, die entweder auf einem relationalen DBS aufbauen, bzw. ein solches um Zeitreihenfunktionalität erweitern.

  Repräsentant dieser Kategorie: TimescaleDB [95]
  Andere DBS, die auch dieser Kategorie angehören: Clickhouse [10], IBM Informix [25] und SingleStore [80]

- *Erweiterung von NoSQL DBS*: Systeme in dieser Kategorie sind jene, die als Basis ein NoSQL DBS nutzen.

  Repräsentant dieser Kategorie: MongoDB [61]
  Andere DBS, die auch dieser Kategorie angehören: Redis [74], Couchbase [13] und RavenDB[24]

Als Baseline wurde zudem ein klassisches relationales Datenbanksystem gewählt, um später vergleichen zu können, wie sich die spezialisierten Zeitreihen-Datenbanksysteme gegenüber zu regulären DBS unterscheiden. Repräsentant hierfür ist das DBS PostgreSQL [88], welches aus zwei Gründen ausgesucht wurde: Zum einen führt DB-Engines [71] PostgreSQL als zweitpopulärstes RDBS auf, und zum anderen ergibt sich durch diese Wahl die Möglichkeit, festzustellen, inwiefern sich das DBS TimescaleDB, das selbst eine Erweiterung von PostgreSQL ist, von seinem zugrundeliegenden DBS unterscheidet.

3.2 Reine Zeitreihen-Datenbanksysteme: InfluxDB

Für die Kategorie von reinen Zeitreihen-Datenbanksystemen wurde InfluxDB [42] als Repräsentant gewählt, da dieses System laut DB-Engines [71] die größte Popularität unter den Zeitreihen-Datenbanksystemen hat. InfluxDB wird seit 2013 kontinuierlich weiterentwickelt – derzeit von der Firma InfluxData, die speziell hierfür gegründet wurde. Das System wird in verschiedenen Varianten angeboten: Zum einen gibt es eine open-source Version (InfluxDB OSS), die frei verfügbar ist, aber einen eingeschränkten Funktionsumfang besitzt. Zum anderen gibt es verschiedene kommerzielle, proprietäre Produkte, die zusätzliche Funktionen wie beispielsweise die Verteilung der Daten auf mehrere Rechner unterstützen und entweder auf eigenen Rechnern oder über einen von InfluxData angebotenen Cloud-Dienst ausgeführt werden können. In dieser Arbeit wird die quelloffene Variante InfluxDB OSS betrachtet, die nachfolgend InfluxDB genannt wird.

InfluxDB nutzt die Anfragesprachen InfluxQL und Flux, um Daten aus der Datenbank zu lesen [39], wobei die Weiterentwicklung für Flux mit der derzeit in Entwicklung befindlichen Version 3 von InfluxDB eingestellt wurde [40]. InfluxQL ist eine SQL-ähnliche Sprache, die auf die Anfrage von Zeitreihendaten spezialisiert ist.

3.2.1 Datenmodell

Als von Grund auf entwickeltes Zeitreihendatenbanksystem nutzt InfluxDB ein eigenes Datenmodell, das für die Speicherung von Zeitreihendaten optimiert ist [41].

InfluxDB speichert die Daten einer Anwendung in Buckets, die mit Datenbanken in anderen DBS zu vergleichen sind. Jedem Bucket kann eine so genannte Retention Policy zugewiesen werden, die dafür sorgt, dass alte Daten nach der eingestellten Zeit automatisch entfernt werden. Dies ist insbesondere nützlich, um Speicherplatz zu sparen. Tabelle 3.1 zeigt fiktive Daten, wie sie in einem Bucket gespeichert werden. Anhand dieser Tabelle werden nachfolgend die restlichen Begriffe des Datenmodells erklärt.

Jedem Datenpunkt wird in InfluxDB ein Zeitstempel (`_time`) zugewiesen, der Uhrzeit und Datum in Nanosekundenauflösung speichern kann. Zudem lassen sich verschiedene Daten in so genannten Measurements (`_measurement`) gruppieren. In der Beispieltabelle 3.1 werden vier verschiedene Messwerte unter dem Measurement „weather“ zusammengefasst. Jeder Datenpunkt, der in der Datenbank gespeichert wird, kann dabei verschiedene Felder (`_field`) und dazugehörige Werte (`_value`) besitzen. Somit können mehrere Werte verschiedener Zeitreihen durch einen einzelnen Datenpunkt in die Datenbank eingefügt werden. Zusätzlich können Werten weitere Eigenschaften zugewiesen werden, die Tags genannt werden – diese sind zudem indexiert, wodurch Anfragen beschleunigt werden. Das Beispiel zeigt die Tags „city“ und „location“, die z. B. dafür genutzt werden können, um alle Messwerte von einem gewissen Standort abzurufen.

Eine Zeitreihe, wie sie in Kapitel 2.1 definiert wurde, ergibt sich somit aus den Werten, die einer eindeutigen Measurement-Field-Tag-Kombination zugeordnet sind. InfluxDB nennt diese eindeutige Kombination „Series Key“ (ab hier Zeitreihenschlüssel genannt). Im Beispiel gibt es zwei verschiedene Zeitreihenschlüssel und somit auch zwei Zeitreihen:

1. `[weather, city=HH, location=outside, temperature]`
2. `[weather, city=HB, location=inside, humidity]`

Tabelle 3.1: Beispieldaten wie sie in einem InfluxDB Bucket gespeichert werden

_time	_measurement	city	location	_field	_value
2024-01-01T00:00:00Z	weather	HH	outside	temperature	-1.0
2024-01-01T00:00:00Z	weather	HB	inside	humidity	0.7
2024-01-01T12:00:00Z	weather	HH	outside	temperature	5.2
2024-01-01T12:00:00Z	weather	HB	inside	humidity	0.6

3.2.2 Time-Structured Merge-Tree

Um Zeitreihendaten effizient zu speichern, nutzt InfluxDB einen so genannten „Time-Structured Merge-Tree“ (TSMT), der ähnlich wie Log-Structured Merge-Trees (LSMT) arbeitet [28]. LSMTs wurden von O'Neil et al. [68] entwickelt und werden in vielen DBS wie z. B. Apache Cassandra [83] genutzt. Durch eine Kombination von Datenstrukturen im Arbeitsspeicher und im persistentem Speicher werden Lese- und Schreiboperationen optimiert. LSMTs bestehen aus zwei sortierten Bäumen (siehe Abb. 3.2) – dem C_0-Baum, der im Arbeitsspeicher meistens als AVL- oder Rot-Schwarz-Baum aufgebaut ist, und dem C_1-Baum, der im persistenten Speicher die Form eines leicht angepassten B-Baums besitzt. Wenn Daten in einen LSMT geschrieben werden, werden sie zunächst in den C_0-Baum geschrieben. Dort verbleiben sie solange, bis der C_0-Baum, eine gewisse Größe überschritten hat oder eine gewisse Zeit vergangen ist. Wenn eine dieser Bedingungen eintritt, werden die Daten in den C_1-Baum mit dem so genannten „rolling merge“-Verfahren überführt. Der C_1-Baum wird in mehreren Dateien gespeichert, die anschließend nur lesbar sind. Das heißt, dass eine Datei nur einmal geschrieben wird und später nicht wieder geändert werden kann. Sollen Daten aus dem LSMT gelöscht oder geändert werden, müssen die entsprechenden Dateien vollständig neu geschrieben werden. Dieses Prinzip verbessert die Lese- und Schreibperformanz, da große Blöcke von Daten am Stück gelesen bzw. geschrieben werden können, wodurch insgesamt weniger Lese-Schreib-Operationen pro Sekunde (engl. Input/Output Operations per Second, bzw. IOPS) ausgeführt werden. Um im Falle eines Systemfehlers die Daten des C_0-Baums wiederherstellen zu können, wird zusätzlich ein Write Ahead Log (WAL) geführt, der jede noch nicht persistente Schreiboperation auf der Datenbank beinhaltet. Weitere Details zu LSMTs lassen sich in [68] nachlesen.

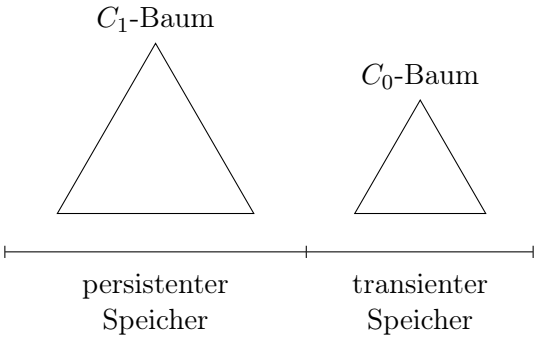


Abbildung 3.2: Visualisierung des Speicherorts und der typischen Größenverhaltnisse von C_0- und C_1-Baum im Log-Structured Merge Tree ([68] nachempfunden)

In den in InfluxDB verwendeten TSMTs gibt es ebenfalls einen WAL und einen C_0-Baum, der hier jedoch Cache genannt wird [28]. Aus dem Cache werden die Daten entweder nach einer gewissen Zeit oder einer bestimmten Datenmenge in den C_1-Baum übernommen. Dieser wird in mehreren verschiedenen TSM-Dateien gespeichert, die jeweils Blöcke von Zeitreihendaten enthalten, welche mit einem für den jeweiligen Datentyp optimierten Verfahren komprimiert werden. Die TSM-Dateien werden mit wachsendem Alter über die Zeit in vier Stufen, so genannten Levels, miteinander kombiniert, um die Anzahl von Dateien im Dateisystem möglichst klein zu halten. Somit werden jüngere Zeitreihendaten in kleinen TSM-Dateien und ältere Zeitreihendaten in größeren TSM-Dateien gespeichert. Dies hat insbesondere Relevanz, da wie in LSMTs die Dateien nur lesbar sind. Laut Hersteller sollte möglichst vermieden werden, ältere Daten zu ändern oder Daten in nicht-zeitlicher Reihenfolge in die Datenbank einzufügen, da sonst große Dateien neu geschrieben werden müssen.

3.2.3 Shards

Um alte Daten nach Ablauf der Retention Policy schneller löschen zu können, speichert InfluxDB die Zeitreihendaten in Shards [28], die entweder der Dauer der Retention Policy entsprechen oder – falls keine Retention Policy festgelegt ist – sieben Tage umfassen. Jeder Shard stellt dabei eine für sich abgeschlossene Einheit dar, die alle Komponenten, die im vorherigen Abschnitt 3.2.2 besprochen wurden, beinhaltet. Wenn alte Daten gelöscht werden sollen, können so komplette Shards entfernt werden, ohne die Datenstrukturen in den anderen Shards zu ändern.

3.2.4 Time Series Index

Für jeden Shard wird in InfluxDB ein Time Series Index (TSI) gepflegt [37, 36]. Dabei handelt es sich um eine Metadatenbank in Form eines Inverted Index, die Metadaten zu den im Shard befindlichen Zeitreihendaten speichert. Inverted Indexes sind Datenstrukturen, die speichern, in welchen Dokumenten ein gewisser Eintrag vorhanden ist [44]. Die Verwendung eines Inverted Index beschleunigt somit Anfragen, die beispielsweise die Daten für einen bestimmten Tag abfragen wollen. Der TSI wird in InfluxDB in einem Log-Structured Merge-Tree (siehe Abschnitt 3.2.2) gespeichert und wurde eingeführt, da es zuvor zu Performanzproblemen bei großen Mengen verschiedener Zeitreihen kam.

3.3 Erweiterung von relationalen DBS: TimescaleDB

TimescaleDB [95] wird seit 2018 als Erweiterung des relationalen open-source DBS PostgreSQL [88] vom gleichnamigen Unternehmen Timescale entwickelt. Laut DB-Engines [71] ist TimescaleDB das populärste Zeitreihen-Datenbanksystem, welches ein DBS mit relationalem Datenmodell erweitert. Durch die Entscheidung, TimescaleDB als PostgreSQL-Erweiterung aufzubauen, bleibt neben der Möglichkeit, Zeitreihendaten zu speichern, auch die Funktionalität von PostgreSQL erhalten. Ebenso wird es ermöglicht, Operationen (z. B. `JOIN`) auszuführen, die gleichzeitig sowohl Zeitreihendaten als auch reguläre relationale Daten nutzen. Als Anfragesprache nutzt TimescaleDB eine mit Zeitreihenfunktionalität erweiterte Form von SQL. Die in PostgreSQL vorhandene Transaktionsfunktionalität inkl. ACID (siehe Kapitel 2.3.3) wird durch die hinzugefügten Operationen nicht beeinträchtigt.

3.3.1 Datenmodell

Das Datenmodell von TimescaleDB ist in den Grundzügen identisch zu dem von PostgreSQL, jedoch wurden zusätzlich so genannte Hypertables eingeführt [96, 94], die zur Speicherung von Zeitreihendaten optimiert sind. Hypertables sind Tabellen, die ihre Daten in einstellbaren Zeiteinheiten und optional in einer weiteren Dimension in Untertabellen partitionieren (siehe Abb. 3.3) – diese Untertabellen werden Chunks genannt. Durch die Aufteilung der Daten entstehen viele kleinere Tabellen, wovon beim Einfügen neuer Werte jeweils nur der letzte Chunk bearbeitet werden muss. Ziel der Chunks ist es, ihre Größe so zu wählen, dass sie im Arbeitsspeicher in einem Cache zwischengespeichert werden können, wodurch der Zugriff auf Festplatten/SSDs vermieden wird. Da auf ältere Chunks seltener zugegriffen werden muss, können diese somit die meiste Zeit ausschließlich im persistenten Speicher liegen. Für Anwendungen, die Daten in Hypertables schreiben bzw. diese lesen, wird die Aufteilung in Chunks durch TimescaleDB transparent abstrahiert, sodass keine spezielle Anwendungslogik notwendig ist.

3.3.2 Wide-/ Narrow-Table-Modell

Im Gegensatz zu anderen Zeitreihen-Datenbanksystemen, wie z. B. InfluxDB (siehe Abschnitt 3.2) können Zeitreihen in TimescaleDB auch mehrere Werte pro Zeitpunkt besitzen [93] – dies wird „Wide-Table“-Modell genannt. Wenn hingegen nur ein Wert pro

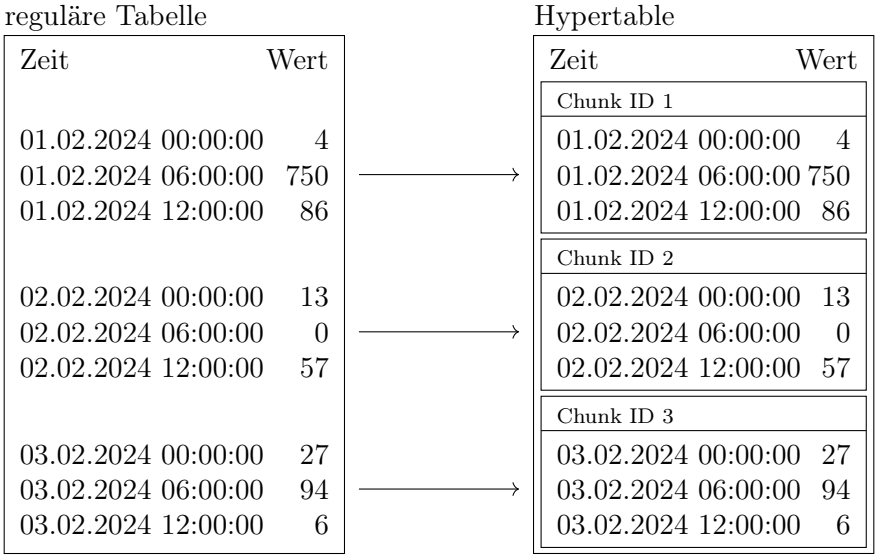


Abbildung 3.3: Aufteilung einer regulären Tabelle in einen Hypertable mit einer Chunk-Größe von einem Tag ([96] nachempfunden)

Zeitreihe pro Zeitpunkt gespeichert wird, spricht man von einem „Narrow-Table“-Modell. Durch die Speicherung als Wide-Table müssen für Anfragen über mehrere Zeitreihen keine `JOIN`-Operationen ausgeführt werden.

3.3.3 Kompression von Chunks

TimescaleDB bietet zudem an, dass die Daten innerhalb eines Chunks komprimiert werden können [89, 90]. Auf Wunsch kann dies entweder für alle Chunks, nur für manuell gewählte Chunks oder für Chunks, die ein gewisses Alter überschritten haben, geschehen. Bei der Kompression werden mehrere Zeilen des Chunks anhand des Attributes `segmentby` in einer Zeile kombiniert. Dies geschieht, indem die Werte der Zeilen spaltenweise in Listen umgewandelt und in dem entsprechenden Datenwert der neu entstehenden Zeile gespeichert werden. Durch diese Art der Datenspeicherung entfällt zum einen der Overhead, der bei der Speicherung jeder einzelnen Zeile entsteht, und zum anderen können auf den kontinuierlich im Speicher befindlichen Daten weitere Kompressionsalgorithmen angewendet werden. Die Sortierung der Daten innerhalb der Listen lässt sich durch das Attribut `orderby` beeinflussen, welches eine Spalte angibt, nach der die Daten aufsteigend sortiert werden. Um den Zugriff auf die Daten zu beschleunigen, speichert TimescaleDB in der kombinierten Zeile den minimalen und den maximalen Wert

des Sortierkriteriums. Die Standardeinstellung für das Attribut `orderby` ist die Spalte, die die Zeitpunkte der Zeitreihe speichert. Jedoch lassen sich je nach Anwendungsfall oft bessere Einstellungen für `segmentby` und `orderby` finden. Details hierzu lasen sich in der Dokumentation von TimescaleDB nachlesen. Zu beachten ist dabei, dass eine Kompression über Zeilen mehrerer Chunks nicht möglich ist. Zudem lassen sich bis auf die Spezialindizes der Attribute `segmentby` und `orderby` keine Indizes in der Tabelle erstellen.

3.4 Erweiterung von NoSQL DBS: MongoDB

Bei MongoDB handelt es sich um ein NoSQL DBS, welches ein dokumentenorientiertes Datenmodell (siehe Kapitel 2.4.2) nutzt. MongoDB [61] wird seit 2009 von der Frima MongoDB Inc. entwickelt und ist mittlerweile in Version 8.0 verfügbar. Im Gegensatz zu TimescaleDB (siehe Abschnitt 3.3) ist die Zeitreihenfunktionalität hier kein Zusatzmodul, das nachträglich installiert werden kann, sondern eine Erweiterung, die von der Herstellerfirma mit Version 5.0 direkt in das DBS integriert wurde. MongoDB wurde u. a. als Repräsentant dieser Kategorie gewählt, da DB-Engines [71] dieses als populärstes NoSQL DBS aufführt.

3.4.1 Datenmodell

In MongoDB werden Dokumente einer Datenbank in so genannten Collections gespeichert, die mit Tabellen in relationalen DBS verglichen werden können [58, 50]. Um Zeitreihendaten besser zu unterstützen, wurde eine neue Art von Collections hinzugefügt, die Time Series Collections (TSC) genannt werden. Diese TSCs sind auf auf Zeitreihen spezialisiert und optimiert bzgl. Speicherung und Abfrage der Daten.

In einer TSC lassen sich mehrere Zeitreihen speichern. Aufgrund dessen muss bei Erstellung einer TSC angegeben werden, in welchem Feld der Dokumente sich die Metadaten befinden, die die verschiedenen Zeitreihen identifizieren [57, 51]. Dieses Feld lässt sich durch das Attribut `metaField` bestimmen. Zusätzlich muss angegeben werden, in welchem Feld der Dokumente die Zeitpunkte der Zeitreihen gespeichert werden sollen. Dies geschieht über das Attribut `timeField`. Die Namen der beiden Felder sind dabei anfangs frei wählbar, lassen sich jedoch nach Erstellung der TSC nicht mehr ändern. Eine wichtige Eigenschaft des Metadatenfeldes ist dabei, dass in diesem auch geschachtelte

Dokumente oder Listen abgelegt werden können. Somit können Zeitreihen durch verschiedene Attribute identifiziert werden. Abbildung 3.4 zeigt ein beispielhaftes Dokument einer TSC.

```
{
    time: ISODate("2024-01-01T12:00:00Z"),
    sensorInfo: {
        city: "Hamburg",
        location: "outside",
    }
    temperature: 5,
    humidity: 0.75,
}
```

Abbildung 3.4: Beispieldokument mit Wetterdaten einer Time Series Collection in MongoDB (`timeField` = `time` und `metaField` = `sensorInfo`)

Ähnlich wie bei TimescaleDB lassen sich in MongoDB mehrere Werte pro Zeitreihe und Zeitpunkt speichern, wodurch man hier auch von einem Wide-Table-Modell sprechen kann (siehe Abschnitt 3.3.2). Im Beispieldokument lassen sich die beiden Werte `temperature` und `humidity` erkennen, die demselben Zeitpunkt zugeordnet sind.

3.4.2 Buckets

MongoDB speichert die Zeitreihen einer TSC in Blöcken von Dokumenten [50], die Buckets genannt werden und anhand der folgenden Kriterien gebildet werden:

- Einerseits werden die Metadatenfelder der Dokumente genutzt, um verschiedene Zeitreihen voneinander zu separieren.
- Zusätzlich werden die Dokumente einer Zeitreihe in gewisse Zeitintervalle gruppiert, auf deren Bestimmung im folgenden Abschnitt weiter eingegangen wird.

Durch die Aufteilung der Daten in Buckets können Abfragen einzelner Zeitreihen beschleunigt werden, da die Daten in kontinuierlichen Speicherbereichen liegen. Zudem gibt es in MongoDB einen In-Memory-Cache, der die Metadaten der in Benutzung befindlichen Buckets speichert. Dieser Cache soll u. a. dazu dienen, Latenzen zu verringern und parallele Schreiboperationen zu koordinieren.

Neue Buckets werden immer dann erstellt, wenn ein Dokument geschrieben wird, das ein bisher noch nicht bekanntes Metadatenfeld besitzt oder einen Zeitpunkt hat, der in keinem Intervall der bisherigen Buckets liegt. Zudem wird ein neuer Bucket angelegt, wenn die Maximalgröße des zu beschreibenden Buckets durch das neue Dokument überschritten wird.

3.4.3 Granularität

MongoDB berechnet das Zeitintervall, in dem ein Dokument gespeichert wird, anhand des Attributes `granularity`, das für jede TSC gesetzt werden muss [56, 57]. Diese Granularität kann dabei auf zwei verschiedene Arten angegeben werden. Zum einen kann sie auf `seconds`, `minutes` oder `hours` eingestellt werden, wodurch entsprechend Zeitintervalle von einer Stunde, einem Tag oder einem Monat entstehen. Wenn Dokumente in eine TSC eingefügt werden sollen, wird deren Zeitstempel jeweils anhand der eingestellten Granularität abgerundet und, basierend auf diesem Wert, das entsprechende Intervall gesucht. Die zweite Möglichkeit, die Granularität anzugeben, besteht darin, die Intervallgröße (`bucketMaxSpanSeconds`) und den Rundungsbereich (`bucketRoundingSeconds`) manuell anzugeben. MongoDB empfiehlt bei der Nutzung dieser Variante, die beiden Attribute auf den gleichen Wert zu setzen.

3.5 Zusammenfassung

Tabelle 3.2 dient als Übersicht der in diesem Kapitel vorgestellten Datenbanksysteme.

Tabelle 3.2: Übersicht der in Kapitel 3 gezeigten DBS

	InfluxDB	TimescaleDB	MongoDB
Kategorie	reines Zeitreihen-Datenbanksystem	Erweiterung von RDBS	Erweiterung von NoSQL DBS
Zugrundeliegende Datenstruktur	Shards, bestehend aus Time-Structured Merge Trees	Hypertables, bestehend aus Chunks	Time Series Collections, bestehend aus Buckets
Wide-Table-Modell	✗	✓	✓
automatisches Löschen alter Daten	✓	✓	✓
Kompression	✓	✓	✓
Einfügen von nicht zeitlich geordneten Daten	✓ / ✗ nur empfohlen, solange sich Datenbereich in C_0-Baum befindet	✓	✓ / ✗ nur empfohlen, solange Datenbereich im selben Granularitätsintervall liegt
Ändern alter Daten	nicht empfohlen	✓	nicht empfohlen
Einfügen von Daten in Blöcken	✓ sehr empfohlen	✓	✓
Verteilbar auf mehrere Knoten	✓ kommerzielle Version	✓ / ✗ nur Replikation	✓ kommerzielle Version
Lizenz	MIT / Apache 2.0	Apache 2.0 / TSL (Timescale License)	SSPL (ähnlich zu GPLv3)

3.6 Verwandte Arbeiten

In den letzten Jahren wurden verschiedene Untersuchungen und Benchmarks von Zeitreihen-Datenbanksystemen durchgeführt. Bader [3] verglich bereits 2016 zehn Systeme und stellte den Benchmark TSDBBench vor. Dabei kamen zwei Szenarios zum Einsatz, die jedoch keine realen Anwendungen repräsentierten. Manche der getesteten Systeme werden inzwischen nicht mehr weiterentwickelt.

Hao et al. [23] führten mit TS-Benchmark eine weitere Evaluierung durch, bei der Sensordaten von Windkraftanlagen genutzt wurden, die mithilfe neuronaler Netze synthetisch erzeugt wurden. Untersucht wurden – wie in dieser Arbeit – InfluxDB und TimescaleDB sowie zusätzlich Druid und OpenTSDB. Eine klassische relationale Datenbank als Baseline wurde nicht berücksichtigt.

Praschl et al. [69] analysierten sechs Zeitreihen-Datenbanksysteme, darunter auch die in dieser Arbeit betrachteten Repräsentanten. Ihre Methodik ähnelt dem experimentellen Vergleich dieser Arbeit, jedoch wurde lediglich ein einziges Szenario untersucht, wodurch nur ein spezifisches Anwendungsgebiet abgedeckt werden konnte. Zudem wurde kein konzeptioneller Vergleich des Funktionsumfangs der Systeme durchgeführt.

Ein Unterschied zu den bisherigen Untersuchungen besteht darin, dass keiner dieser Vergleiche den Energieverbrauch der Datenbanksysteme berücksichtigt hat. Zudem wurden in diesen Arbeiten mittlerweile veraltete Versionen der Datenbanksysteme getestet, die durch neuere Versionen abgelöst wurden und eine verbesserte Leistung versprechen.

4 Szenarios

In diesem Kapitel werden die Entscheidungsgrundlagen erläutert, die zur Auswahl der Szenarios geführt haben. Zudem werden die einzelnen Szenarios detailliert beschrieben. In den folgenden drei Abschnitten werden jeweils die für das entsprechende Szenario durchzuführenden Zeitreihenanalysen sowie die Rahmenbedingungen und der Kontext, in dem das Szenario ausgeführt wird, näher erläutert.

4.1 Auswahl der Szenarios

Die in diesem Kapitel vorgestellten Szenarios wurden so ausgewählt, dass sie realistische Einsatzgebiete von Zeitreihen-Datenbanksystemen darstellen. Dabei wurden jeweils folgende Punkte beachtet:

Szenario A wurde so gewählt, dass es einem typischen Anwendungsfall von Zeitreihen-Datenbanksystemen entspricht – nämlich der Überwachung von Sensoren. Der Fokus liegt auf einem Heimanwendungskontext, woraus sich folgende Anforderungen ergeben:

- *Ressourcennutzung*: In diesem Kontext müssen DBS häufig auf Rechnern mit geringer Leistung laufen. Hieraus entstehen weitere Punkte, die im nachfolgenden Abschnitt 4.2.2 genauer erläutert werden.
- *einfache Bedienbarkeit*: Für private Anwender ist es besonders wichtig, dass die Einrichtung, Nutzung und der Betrieb des DBS einfach ist, da diese meistens nicht die Erfahrung besitzen, die im industriellen Kontext vorhanden ist.

Szenario B soll eine Grundlage bieten, um zu testen, wie sich die Zeitreihen-Datenbanksysteme bei einer geringen Anzahl von Zeitreihen verhalten, die sich über einen langen Zeitraum erstrecken und viele Datenpunkte enthalten. Dies ist insbesondere interessant, da so analysiert werden kann, wie sich die in Kapitel 6.1 beschriebenen Metriken verhalten, wenn Operationen auf großen Zeitreihen durchgeführt werden.

Als Gegenstück zu Szenario B soll Szenario C testen, wie sich die DBS bei großen Anzahlen von kurzlebigen Zeitreihen verhalten. Insbesondere kann so der Overhead verdeutlicht werden, der für die Speicherung, aber auch Verarbeitung jeder Zeitreihe anfällt.

4.2 Szenario A: Smart Home

Dieses Szenario simuliert ein Wohngebäude (Smart Home) mit einer einstellbaren Anzahl an Sensoren bzw. Geräten, die Datenpunkte generieren. Dazu müssen die in diesem Szenario genutzten Testdaten nicht zwingend sinnvolle Zusammenhänge untereinander abbilden, jedoch ist es wichtig, dass ihre Struktur echten Daten ähnelt. Dies liegt daran, dass alle zu testenden Zeitreihen-Datenbanksysteme eine Form der Datenkompression unterstützen. Das Kompressionsverhältnis, also das Verhältnis zwischen der Größe der komprimierten und der unkomprimierten Daten, hängt dabei stark von deren Struktur ab. Lange Folgen von sich wiederholenden Werten lassen sich so besser komprimieren als zufällige Werte.

Um die Testdaten für dieses Szenario zu generieren, werden zwei Kategorien von Generatoren verwendet: Die erste Kategorie umfasst zeitintervallgesteuerte Verfahren, die Datenpunkte in regelmäßigen Abständen erzeugen, während die zweite Kategorie nur Werte bei Eintritt eines Ereignisses generiert. Da bei der Recherche keine Generatoren gefunden wurden, die den hier gestellten Anforderungen entsprechen, wurden die folgenden Generatoren entworfen:

- Zeitintervallgesteuert
  - *elektrische Verbraucher*: Für die Generierung der Verbrauchsdaten werden zufällige Ein- und Ausschaltzeiten bestimmt, die jeweils in einstellbaren Intervallen liegen. Für die Zeiträume, in denen ein Verbraucher eingeschaltet ist, werden Leistungswerte um einen Sollwert mit einem zufälligen Rauschen kombiniert, um leichte Schwankungen im Verbrauch des Geräts zu simulieren. Die restliche Zeit ist der Verbrauch null.
  - *Temperatur*: Zur Simulation von Temperaturen an verschiedenen Orten im Gebäude wird eine Sinus-Funktion, die durch eine hyperbolische Tangensfunktion an den Maxima gestaucht wird, mit einem zufälligen Rauschen kombiniert. Die

so resultierende Funktion soll Schwankungen zwischen Tag- und Nachttemperaturen sowie Messungenauigkeiten der Sensoren darstellen:

$$T(t) = \frac{v}{2} \cdot \left(\tanh \left(\sin \left(\frac{2\pi \cdot t}{p} + s \right) \cdot d \cdot \frac{p}{2\pi} \right) + 1 \right) + o + rand(-n, n) \tag{4.1}$$

wobei v den Temperaturunterschied zwischen Tag und Nacht, p die Periode und s die Verschiebung der Funktion darstellt. d gibt an, wie stark die Sinus-Funktion gestaucht wird und o ist die Nachttemperatur. Die Funktion $rand(-n, n)$ generiert zufällige Werte im Intervall $[-n, n)$

Abbildung 4.1 zeigt, wie $T(t)$ zusammengesetzt ist.

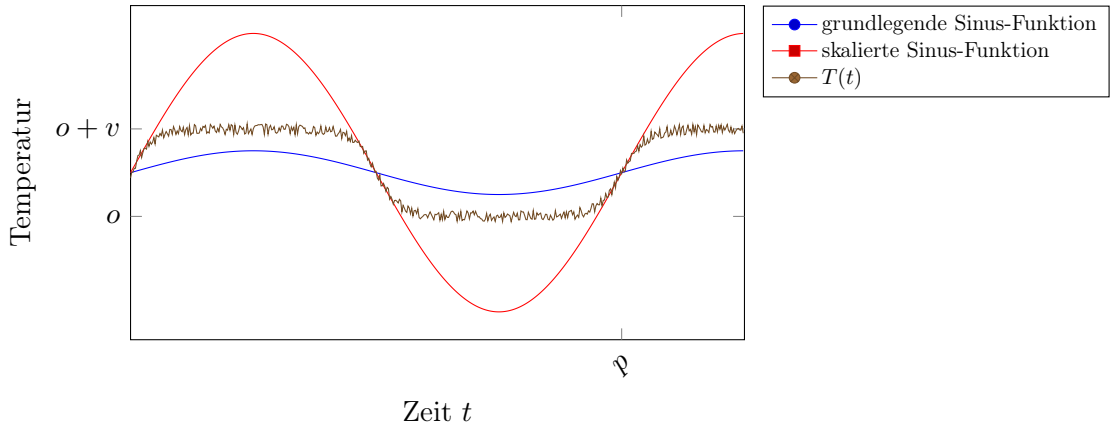


Abbildung 4.1: Visualisierung des Aufbaus und der Stauchung der Maxima der Temperaturfunktion $T(t)$

- Ereignisgesteuert
  - *Fenster und Türen*: Dieser Generator erzeugt zufällig, nach einer einstellbaren Wahrscheinlichkeit, Öffnungs- und Schließereignisse für jedes Fenster und jede Tür.
  - *Beleuchtung*: Im Gegensatz zur Generierung von Fenster- und Türereignissen, die nur zwei Werte – auf und zu – besitzen, werden hier prozentuale Helligkeitswerte simuliert. Die Art und Weise dieser Simulation ist dabei ähnlich zu der zuvor genannten. Hier wird jedoch für jedes Ereignis eine neue zufällige Helligkeit erzeugt.

Die Parameter der Generatoren lassen sich in Kapitel 6.3.3 nachlesen.

4.2.1 Analysen Szenario A

Folgende Analysen werden bei diesem Szenario durchgeführt:

- *Abfrage der Anzahl von Öffnungen von Fenstern und Türen*: Mit dieser Abfrage soll getestet werden, wie sich die DBS bei Zähloperationen auf Zeitreihen mit unregelmäßigen Abständen von Datenpunkten verhalten.
- *Abfrage des Gesamtstromverbrauchs*: Diese Anfrage prüft, welchen Einfluss Aggregationsoperationen auf die Zeitreihen-Datenbanksysteme haben.
- *Abfrage der minütlichen Durchschnittstemperatur*: Ähnlich zu der vorherigen Abfrage wird hier das Verhalten von Aggregationen geprüft. Jedoch werden die Daten hier zusätzlich in Zeitfenster gruppiert.

4.2.2 Rahmenbedingungen Szenario A

Da dieses Szenario im Heimanwendungskontext stattfindet, werden hier Systemkonfigurationen getestet, die geringe Ressourcenkapazitäten im Vergleich zu Servern in Rechenzentren besitzen. Somit werden Systeme getestet, die einen oder zwei Prozessorkerne besitzen und 512 MB oder 1 GB Arbeitsspeicher installiert haben. Dabei werden alle Kombinationen beider Parameter genutzt. Es entstehen somit vier verschiedene Konfigurationen.

In diesem Szenario soll dabei nur indirekt geprüft werden, wie groß die verarbeitbaren Daten werden können. Vielmehr ist das Ziel, herauszufinden, mit wie wenigen Systemressourcen ein DBS in einem realistischen Haushalt auskommen kann. Dazu wird im Experiment die Anzahl von Sensoren bzw. Datenquellen kontinuierlich vergrößert, bis eine Anzahl von insgesamt 400 erreicht wird. Anhand der gemessenen Latenzen kann so abgelesen werden, wie viele Generatoren das jeweilige DBS verarbeiten kann. Diese maximale Menge wurde so gewählt, dass bei einer Hinzufügerate von ca. einem Generator alle 3 Sekunden der gesamte Vorgang etwa 20 Minuten dauert. Das bedeutet auch, dass der Einfügeprozess im Gegensatz zu den Analysen in diesem Szenario die größere Rolle spielt, da Einfügeoperationen nahezu kontinuierlich und Abfrageoperationen nur sporadisch auftreten.

Zu beachten ist bei diesem Szenario, dass jeder Sensor über eine eigene Sitzung mit dem DBS kommuniziert, da in einem echten Gebäude die Sensoren verteilt angebracht

sind und jeweils z. B. über eine eigene WLAN-Verbindung mit dem DBS kommunizieren müssen.

Die in diesem Szenario auszuführenden Anfragen werden zudem 1000 mal pro Experiment wiederholt, da sie nur auf einem kleinen Datenbestand arbeiten können und eine genaue Messung sonst aufgrund der kurzen Anfragedauer schwierig möglich wäre.

4.3 Szenario B: Taxis in New York City

Im Gegensatz zu Szenario A (siehe Kapitel 4.2), in dem die Daten synthetisch generiert werden, wird hier ein Datensatz verwendet, der aus realen Daten von Taxifahrten in New York City (NYC) besteht. Der Datensatz wird von der NYC Taxi & Limousine Commission (TLC) zur Verfügung gestellt [66] und umfasst dabei fast alle Taxifahrten, die seit Anfang 2009 stattgefunden haben.

Die TLC teilt den Datensatz in vier Teile, die anhand des Taxityps gebildet werden. In diesem Szenario werden nur die Daten der gelben Taxen (engl. yellow cab) genutzt, da diese den größten Zeitraum, der sich von Januar 2009 bis Oktober 2024 erstreckt, abdecken. Dieser Teildatensatz umfasst ca. 30 GB an Rohdaten im Apache Parquet-Format – dies entspricht ungefähr 1,8 Milliarden Datenpunkten. Für dieses Szenario wurde der Datensatz weiter bereinigt, da er mehrfach vorkommende Zeitpunkte enthielt. In der hier verwendeten Version wurden diese Duplikate entfernt. Zudem lagen die Daten vor Januar 2011 in einem Format vor, das mit dem restlichen Datensatz nicht kompatibel war, weshalb nur Daten ab diesem Zeitpunkt berücksichtigt wurden. Darüber hinaus wurde die Anzahl der Spalten auf die für die Anfragen relevanten Attribute – Abfahrzeitpunkt, Strecke, Passagiere und Fahrpreis – reduziert.

4.3.1 Analysen Szenario B

Folgende Analysen werden bei diesem Szenario durchgeführt:

- *durchschnittlicher Fahrpreis pro Monat*: Diese Anfrage testet, wie sich die DBS bei der Aggregation von Daten verhalten, die nach bestimmten Zeitfenstern gruppiert sind.

- *Abfrage aller Fahrgastzahlen der ersten Jahreshälfte von 2023*: Hier soll prüft werden, wie schnell alle Daten einer Zeitreihe in einem gewissen Zeitraum abgefragt werden können.
- *Abfrage aller Fahrten, deren Strecke größer als x sind*: Diese Analyse soll testen, welchen Einfluss Filteroperationen auf die DBS haben.
- *Stunde mit den meisten Fahrten*: Bei dieser Operation wird untersucht, ob und wie gut sich komplexe Operationen, d. h. Kombinationen verschiedener Grundoperationen, durchführen lassen.

4.3.2 Rahmenbedingungen Szenario B

In diesem Szenario werden Rechner mit zwei, vier, acht oder 16 Prozessorkernen und 16 GB Arbeitsspeicher simuliert, die Server in Rechenzentren oder bei Cloud-Providern darstellen sollen. Die Größe des Arbeitsspeichers wird hier nicht variiert, da hier im Gegensatz zu Szenario A getestet werden soll, wie viel des Arbeitsspeichers genutzt wird und nicht, mit wie wenig Arbeitsspeicher das DBS lauffähig ist.

4.4 Szenario C: Monitoring kurzlebiger Dienste

Szenario C soll ein Rechenzentrum darstellen, welches Metriken von Diensten aufzeichnet und auswertet. Die Dienste werden hier mithilfe von mehreren Instanzen gehostet, die jeweils bis zu 10 Minuten lang laufen. Dies könnte in der Realität z. B. durch die Verwendung eines Autoscalers der Fall sein, der entsprechend der Last mehr oder weniger Ressourcen zur Verfügung stellt. Hierdurch entstehen somit viele kurze kurzlebige Zeitreihen, die im Zeitreihen-Datenbanksystem gespeichert und verarbeitet werden müssen.

Wie auch in Szenario A werden hier synthetische Testdaten verwendet, die realen Daten ähneln sollen. Dazu wird für eine einstellbare Anzahl von Instanzen Prozessor- und Arbeitsspeicherauslastung wie folgt erzeugt:

- *Prozessorauslastung*: Zur Generierung der Prozessorlast L wird eine Grundlast G, die zufällig um einen angestrebten Wert schwankt, mit Lastspitzen S kombiniert, die an zufällig verteilten Zeitpunkten erzeugt werden. So soll ein Dienst dargestellt werden, der größtenteils kleine und teilweise größere Anfragen bearbeitet.

Lastspitzen besitzen dabei zwei Parameter – Höhe h und Dauer d. Beide werden dabei anhand zweier normalverteilten Zufallsfunktionen generiert, die den Erwartungswert μ und die Standardabweichung σ besitzen. Durch die Normalverteilung werden so mit höherer Wahrscheinlichkeit Höhen und Dauern um μ und seltener weit entfernte Werte generiert.

Die Grundlast G für einen Zeitpunkt $t \in \mathbb{N}$ wird wie folgt definiert:

$$G(t) = \begin{cases} g & t = 0 \\ (L(t-1) + rand(-v, v)) \cdot (1 - x) + g \cdot x & \text{sonst} \end{cases} \tag{4.2}$$

wobei g die angestrebte Grundlast darstellt und $x \in [0,1]$ ein Faktor ist, der angibt, wie stark die Werte in Richtung der angestrebten Grundlast korrigiert werden. Während einer Lastspitze wird statt x der Faktor $x' > x$ genutzt, um schneller auf die Spitzenlast zu gelangen. Die Funktion $rand(-v, v)$ erzeugt gleichmäßig verteilte Werte $\in \mathbb{R}$ im Intervall $[-v, v)$.

Kombiniert ergibt sich so die Prozessorlast L:

$$L(t) = \min(0, \max(1, G(t) + S(t)) \tag{4.3}$$

wobei $S(t) = h$ wenn zum Zeitpunkt t eine Lastspitze der Höhe h auftritt und $S(t) = 0$ falls nicht.

- *Arbeitsspeicherauslastung*: Die Arbeitsspeicherauslastung A wird ähnlich zu Prozessorauslastung berechnet, jedoch schwankt diese weniger und besitzt keine Spitzen. Sie wird wie folgt definiert:

$$A(t) = \begin{cases} a & t = 0 \\ (A(t-1) + rand(-v, v)) \cdot (1 - x) + a \cdot x & \text{sonst} \end{cases} \tag{4.4}$$

  wobei a der angestrebte genutzte Speicher ist, der einmalig zufällig berechnet wird. Die restlichen Parameter sind analog zu denen bei der Prozessorgrundlast G.

Diese Verfahren wurden ebenfalls, wie in Szenario A, entwickelt, da keine bereits existierenden Generatoren verfügbar waren, die für diesen Anwendungsfall passend sind.

Die im experimentellen Vergleich genutzten Parameter der hier gezeigten Generatoren sind in Kapitel 6.3.3 zu finden. Abbildung 4.2 zeigt nach dem hier beschriebenen Verfahren generierten Zeitreihen.

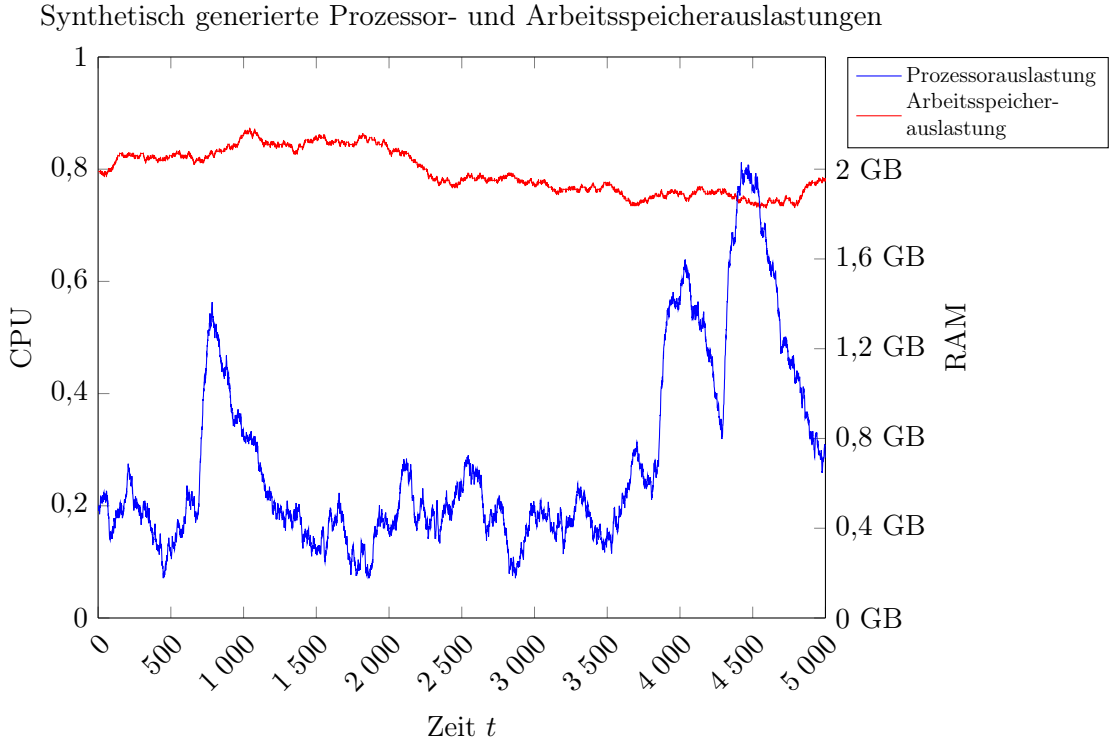


Abbildung 4.2: Nach dem in Kapitel 4.4 beschriebenen Verfahren generierte Prozessor- und Arbeitsspeicherauslastungen

4.4.1 Analysen Szenario C

Es wurde bei diesen Anfragen darauf geachtet, dass die Struktur ähnlich zu den Anfragen von Szenario B (siehe Abschnitt 4.3) ist, um einen Vergleich zwischen Szenario B und C zu ermöglichen. Für dieses Szenario werden somit die folgenden Analysen durchgeführt:

- *kombinierte Prozessorlast pro Minute der zu dem jeweiligen Zeitpunkt laufenden Instanzen*: Mit dieser Anfrage wird geprüft, wie sich die DBS bei Rechenoperationen auf großen Anzahlen von Zeitreihen verhalten.
- *Abfrage aller Instanzmetriken in einer bestimmten Stunde*: Hier wird geprüft, wie schnell sämtliche Daten aller Zeitreihen in einem gewissen Intervall abgefragt werden können.
- *Abfrage der Instanzen, bei denen Prozessorauslastung größer x und Arbeitsspeicherauslastung größer y sind*: Diese Anfrage soll feststellen, mit welcher Geschwindigkeit Filteroperationen auf den Zeitreihen durchgeführt werden können.

4.4.2 Rahmenbedingungen Szenario C

In diesem Szenario werden die gleichen Rahmenbedingungen wie in Szenario B (siehe Abschnitt 4.3.2) verwendet, um, wie bereits im vorherigen Abschnitt erwähnt, einen Vergleich zwischen den beiden Szenarios zu ermöglichen.

Die folgende Tabelle 4.1 zeigt eine Übersicht der Szenarios im Vergleich.

Tabelle 4.1: Übersicht der Szenarios

	Szenario A	Szenario B	Szenario C
Kontext	Heimanwender	Analyse von Taxifahrten	Rechen-zentrum
Anzahl der Zeitreihen	mittel	klein	groß
Menge der Datenpunkte	klein	groß	groß
Größe der Anfragen	klein	groß	groß

5 Konzeptioneller Vergleich

In diesem Kapitel werden die drei Zeitreihendatenbanksysteme konzeptionell miteinander verglichen. Dazu wird auf Gemeinsamkeiten bzw. Unterschiede der Systeme bezüglich des Funktionsumfangs eingegangen.

5.1 Strukturanforderungen der Zeitreihen

Die zu vergleichenden Zeitreihen-Datenbanksysteme sind unterschiedlich flexibel bezüglich der Datenstruktur der Zeitreihen. Datenbanksysteme, die fest strukturierte Daten halten, nennt man dabei schemabehaftet. DBS ohne feste Struktur nennt man hingegen schemafrei.

InfluxDB gehört zur Kategorie der schemafreien DBS, da hier – abgesehen von der Definition von Buckets (siehe Kapitel 3.2.1) – keine feste Erstellung eines Schemas möglich ist. Jedoch ist zu beachten, dass das generelle Datenmodell mit Measurements, Tags und Fields eingehalten werden muss.

Da TimescaleDB eine Erweiterung des relationalen DBS PostgreSQL ist, übernimmt es dessen Struktureigenschaften (siehe Kapitel 2.3). Somit kann TimescaleDB als schemabehaftet eingestuft werden. Die Struktur der Zeitreihen muss hier also bei bereits Erstellung der entsprechenden Tabelle angegeben werden.

MongoDB kann wiederum als schemafrei kategorisiert werden, da die Dokumente keine fest definierte Struktur haben müssen. Lediglich die in der Time Series Collection konfigurierten Felder `metaField` und `timeField` müssen in den Dokumenten vorhanden sein. Falls jedoch ein Schema gewünscht wird, das vom DBS geprüft wird, ist es bei MongoDB optional möglich, ein solches Schema zu definieren.

5.2 Anfragesprachen

Die DBS nutzen jeweils unterschiedliche Anfragesprachen mit verschiedenen Eigenschaften, die nachfolgend beschrieben werden.

InfluxDB bietet, wie bereits in Kapitel 3.2 erwähnt, zwei Anfragesprachen an:

- Zum einen wird die vom Hersteller entwickelte Sprache Flux angeboten, mit der Skripte zu Anfrage und Verarbeitung der gespeicherten Daten geschrieben werden können. Die Idee besteht darin, die Daten als Strom zu verarbeiten, auf dem Funktionen zur Filterung, Manipulation oder Aggregation ausgeführt werden. In Flux-Skripten ist es zudem u. a. möglich, `if-else`-Strukturen zu nutzen und eigene Funktionen zu definieren. Die genaue unterstützte Syntax lässt sich in der Flux Dokumentation [27] nachlesen.
- Die zweite von InfluxDB unterstützte Sprache ist InfluxQL [35]. Diese Sprache wurde mit einer ähnlichen Syntax zu SQL entworfen, die jedoch einen geringeren Funktionsumfang besitzt.

Beide Sprachen bieten zudem spezielle Funktionen, die für Zeitreihenanalysen genutzt werden können. Auf diese wird u. a. in Abschnitt 5.4 eingegangen.

TimescaleDB verwendet, wie auch PostgreSQL, SQL als Anfragesprache, die jedoch durch spezielle Funktionen für Zeitreihenoperationen erweitert wurde. Hierdurch ist es häufig möglich, dieselben Anfragen wie bei einem regulären relationalen DBS zu nutzen, wobei durch die Verwendung von spezialisierten Funktionen teilweise eine bessere Performanz erzielt werden kann. Wie auch bei SQL ist es möglich, mehrere Ausdrücke ineinander zu Schachteln, um komplexere Anfragen zu erstellen.

Die von MongoDB genutzte Anfragesprache MQL orientiert sich an der dokumentenorientierten Struktur ihrer Daten. Jede Anfrage in MQL ist aus geschachtelten JSON- bzw. BSON-Dokumenten aufgebaut, bei denen die Schlüssel die auszuführenden Operationen darstellen und die Werte die Parameter der Funktionen sind. Dabei beginnen Operationen immer mit einem Dollar-Symbol ($). Ebenso wie bei InfluxDB und TimescaleDB ist es möglich, mehrere Anfragen zu kombinieren. Dies wird bei MongoDB „Pipeline“ genannt. Das Konzept ist hier ähnlich zu dem von Flux, da hier die Dokumente jeweils Funktionen durchlaufen, die Dokumente filtern, verändern oder aggregieren. Syntaktisch wird dies als JSON-Array dargestellt.

Beispiele der hier gezeigten Sprachen werden in Kapitel 6.3.5 aufgeführt.

5.3 Verknüpfung von Zeitreihen (Joins)

Bei manchen Anfragen ist es notwendig, Informationen aus mehreren Zeitreihen bzw. Datenquellen zu verknüpfen. Dies wird in klassischen relationalen DBS „Join“ genannt. Die DBS in diesem Vergleich unterstützen alle verschiedene Varianten von Join-Operationen, die folgend beschrieben werden.

Von den beiden von InfluxDB unterstützten Sprachen kann nur Flux Joins durchführen – in InfluxQL ist dies nicht möglich.Des Weiteren muss für die Verwendung von Joins in Flux das `join`-Modul [32] geladen werden, welches neben den vier regulären Join-Operationen – `inner`, `left outer`, `right outer` und `full outer` – auch einen zeitbasierten Join unterstützt. Zu beachten ist jedoch, dass die Funktionen `union` und `pivot` laut Hersteller häufig schneller sind, wobei diese nicht in allen Fällen angewendet werden können. Die Funktion `union` [38] vereint dabei zwei Datenströme von Zeitreihen, sodass die Werte beider Datenströme im Ergebnisstrom ausgegeben werden. Bei der `pivot`-Funktion [34] werden die Daten hingegen anhand eines anzugebenden Feldes gruppiert.

TimescaleDB unterstützt die vollen Join-Operationen zwischen Hypertables, die von PostgreSQL zwischen regulären Tabellen unterstützt werden. Zudem ist es möglich, Joins zwischen Hypertables und Nicht-Hypertables durchzuführen, um so z. B. zusätzliche Informationen zu Zeitreihen hinzuzufügen.

Im Gegensatz zu den bereits gezeigten DBS unterstützt MongoDB nur eine Left-Outer-Join-Operation, die durch die `$lookup`-Funktion [53] bereitgestellt wird. Dabei wird den Dokumenten der Grundmenge eine Liste hinzugefügt, in der sich die dazugehörigen Dokumente der zu vereinigenden Dokumentmenge befinden.

5.4 Zeitreihenoperationen

Zeitreihen-Datenbanksysteme bieten teils Funktionen an, um Zeitreihenanalysen (siehe Kapitel 2.2) direkt im DBS durchzuführen. Hierbei gibt es jedoch wiederum Unterschiede zwischen den DBS, die in diesem Abschnitt beschrieben werden.

InfluxDB bietet über die Sprache Flux Funktionen an, die statistische Berechnungen, gleitende Durchschnitte mit verschiedenen Verfahren, Operationen auf Zeitfenstern und Vorhersagen, z. B. über die Holt-Winters-Methode [97], durchführen können. Die genaue

Liste der unterstützten Funktionen ist in der Dokumentation [26] näher beschrieben. Zudem ist es möglich, Flux-Skripte zu erstellen, die weitere nicht standardmäßig unterstützte Verfahren implementieren, die direkt auf dem DBS ausgeführt werden.

TimescaleDB unterstützt standardmäßig eine kleinere Anzahl von spezialisierten Operationen auf Zeitreihen als InfluxDB. Diese Zeitreihenoperationen werden in TimescaleDB „Hyperfunctions“ genannt und unterstützen u. a. die Verarbeitung von Zeitreihen in Fenstern, die Erstellung von Histogrammen und das Füllen von Lücken in den Daten durch Interpolation. Für weitere Zeitreihen- und Statistikoperationen kann das „Timescale Toolkit“ als zusätzliche Erweiterung in PostgreSQL installiert werden. Auf die dadurch hinzukommenden unterstützten Funktionen soll hier jedoch nicht eingegangen werden, da sich dieser Vergleich nur auf TimescaleDB beschränkt. Sie lassen sind jedoch in der Dokumentation [91] der Erweiterung ersichtlich.

Das DBS MongoDB unterstützt hingegen keine speziell für Zeitreihen gedachten Funktionen. Jedoch können teilweise die regulären MQL-Operationen genutzt werden, um Funktionen wie die Verarbeitung von Daten in Zeitfenstern nachzubilden.

5.5 Konsistenz und Transaktionen

Wie in Kapitel 2.3.3 beschrieben, legen manche Systeme besonderen Wert auf die Konsistenz der gespeicherten Daten, wohingegen andere mehr Wert auf Verfügbarkeit legen (siehe Kapitel 2.4.1).

InfluxDB gehört dabei zu der letzteren Kategorie und setzt auf „eventual consistency“ (siehe Kapitel 2.4.1). Daten, die in das DBS geschrieben werden, sind somit bereits vor dem Zeitpunkt, an dem sie persistent gespeichert wurden, zum Lesen verfügbar [29]. Zudem bietet InfluxDB keine Unterstützung von Transaktionen (siehe Kapitel 2.3.3) über mehrere Anfragen, wobei dafür gesorgt wird, dass während einer Anfrage keine Änderungen der zugrundeliegenden Daten durchgeführt werden.

TimescaleDB übernimmt als PostgreSQL-Erweiterung die ACID-Eigenschaften (siehe Kapitel 2.3.3) des grundlegenden relationalen DBS und ermöglicht ebenfalls die Nutzung von Transaktionen [92]. Wie bereits in Kapitel 2.3.3 erwähnt, lässt sich auch hier die Isolationsstufe pro Transaktion Einstellen um ggf. bessere Performanz zu erzielen.

Bei MongoDB lässt sich hingegen die Konsistenz pro Sitzung oder Anfrage in verschiedenen Stufen konfigurieren [59], die in der Dokumentation [60, 55] genau beschrieben sind.

Dazu muss das so genannte „read concern“ bzw. „write concern“ entsprechend für Lese- bzw. Schreib-Operationen auf die gewünschte Stufe eingestellt werden. Es ist zu beachten, dass das Schreiben und Ändern von Dokumenten immer atomar erfolgt. Das bedeutet, dass niemals teilweise geschriebene Dokumente gelesen werden. Transaktionen werden in MongoDB auf Time Series Collections nur für lesende Operationen unterstützt [58].

5.6 Programmierschnittstellen

Um mit den Datenbanksystemen mit Anwendungssoftware zu interagieren, bieten diese Programmierschnittstellen und dazugehörige Bibliotheken an. Dieser Abschnitt erläutert, wie sich diese im Detail unterscheiden.

InfluxDB nutzt als Schnittstelle eine RESTful HTTP API an, welche auf Port 8086 erreichbar ist [31]. Über die Endpunkte dieser Schnittstelle ist es möglich, Daten in das DBS einzufügen und diese wieder mit Flux oder InfluxQL abzufragen. Des Weiteren lässt sich das DBS hierüber konfigurieren. Aufgrund der Verwendung von HTTP als grundlegendes Protokoll kann jede Programmiersprache, die HTTP-Anfragen durchführen kann, mit dem DBS interagieren. Der Hersteller stellt für gängige Sprachen Bibliotheken bereit, die die Nutzung der API abstrahieren. Zudem ist die Schnittstelle zustandslos, was bedeutet, dass keine dauerhafte Sitzung mit dem DBS erforderlich ist.

Im Gegensatz zu InfluxDB nutzt TimescaleDB ein binäres Protokoll, das entweder auf Grundlage von TCP/IP (Port 5432) oder UNIX-Sockets verwendet werden kann [85]. Da es sich hierbei um dasselbe Protokoll wie bei PostgreSQL handelt, gibt es von diversen Anbietern Bibliotheken zur Verwendung der Schnittstelle. Ein Beispiel für eine solche Bibliothek ist sqlx [46], die auch im nachfolgenden experimentellen Vergleich verwendet wurde.

Wie auf TimescaleDB wird bei MongoDB ein binäres Protokoll – das so genannte „Wire Protocol“ – eingesetzt [54], das über TCP/IP (Port 27017) oder UNIX-Sockets genutzt werden kann. Dieses ist jedoch mit dem von TimescaleDB bzw. PostgreSQL inkompatibel. Der Hersteller bietet hier wie bei InfluxDB Treiber an, mit denen das DBS an eine Anwendung angebunden werden kann, wobei die MongoDB Bibliotheken für mehr Programmiersprachen verfügbar sind.

Tabelle 5.1 fasst die in diesem Kapitel vorgestellten Funktionen und Eigenschaften des DBS zusammen.

Tabelle 5.1: Funktionen und Eigenschaften der DBS

	InfluxDB	TimescaleDB	MongoDB
Strukturanforderung	schemafrei	schemabehaftet	schemafrei
Anfragesprache	InfluxQl / Flux (ab v3.0 veraltet)	erweitertes SQL	MQL
Joins	✓	✓	eingeschränkt möglich durch `$lookup`
Zeitreihenoperationen	Statistik, gleitende Durchschnitte, Fensteroperationen, Vorhersagen	Fensteroperationen, Histogramme, Füllen von Lücken (erweiterbar durch Timescale Toolkit)	keine spezialisierten Operationen
Konsistenz	eventual	stark	einstellbar
Transaktionen	✗	✓	✓ / ✗ (nur lesend)
Programmier-schnittstellen	RESTful HTTP API	binäres Protokoll, basierend auf TCP/IP oder UNIX-Sockets	binäres Protokoll, basierend auf TCP/IP oder UNIX-Sockets

6 Experimenteller Vergleich

In diesem Kapitel wird erläutert, wie der experimentelle Vergleich der Zeitreihen-Datenbanksysteme aufgebaut und durchgeführt wird. Die Experimente und das Testsystem, welche in diesem Kapitel beschrieben werden, wurden auf folgendem System ausgeführt:

- *Prozessor:* AMD Ryzen 9 3900x (12 Kerne à 3,8 GHz, mit Hyperthreading)
- *Arbeitsspeicher:* 48 GB DDR4 (4 Riegel à 3000 MHz)
- *Speicher:* 1 TiB Western Digital NVME SSD
- *Kernel:* Linux 6.13.3

Die Datenbanksysteme wurden dabei in folgenden Versionen getestet:

- *InfluxDB*: 2.7.11
- *TimescaleDB*: 2.18.2
- *MongoDB*: 8.0.4
- *PostgreSQL*: 17.4

6.1 Metriken

Der experimentelle Vergleich in diesem Kapitel wird anhand von Metriken durchgeführt, die vom Testsystem aufgezeichnet werden. Die Metriken lassen sich dabei in zwei Kategorien unterteilen: Zum einen die klassischen Systemmetriken, die innerhalb der Testumgebung des DBS (mehr dazu in Abschnitt 6.2.2) ermittelt werden. Zum anderen wird im Kontext von Green-IT der Energieverbrauch ermittelt. Beide Kategorien werden in den nächsten Abschnitten genauer beschrieben. Zudem wird für jede Anfrage die Latenz gemessen, wobei diese Messungen von den Szenarios selbst getätigt werden.

6.1.1 Klassische Systemmetriken

Angelehnt an [23], [69] und [43] werden die folgenden Metriken der Testumgebung für die Experimente erhoben:

- Prozessorlast (Durchschnitt in Prozent über alle Kerne)
- genutzter Arbeitsspeicher (in Byte)
- gesamter Arbeitsspeicher (in Byte)
- genutzte Festplattenkapazität (in Byte)
- gesamte Festplattenkapazität (in Byte)
- Festplatten Lese-/Schreibzugriffe (in Byte/s)
- Netzwerk Lese-/Schreibzugriffe (in Byte/s)

Diese Metriken wurden gewählt, da sie es ermöglichen, die Ressourcennutzung des Systems abzulesen. Dies ist besonders relevant für die Dimensionierung der Hardwarekomponenten der Rechner, auf denen die DBS in echten Anwendungsfällen eingesetzt werden sollen. Anhand der Testergebnisse des experimentellen Vergleichs können somit Schlüsse gezogen werden, welche DBS auf welchen Systemen für welche Anwendungsfälle besonders geeignet sind.

6.1.2 Green-IT und Strommessung

Der Begriff Green-IT beschreibt Bestrebungen, IT-Systeme, wie z. B. Rechenzentren, aber auch Rechner im privaten Umfeld, umweltfreundlicher zu gestalten [64]. Ziel ist es dabei, größtenteils den Stromverbrauch zu senken und damit den CO2-Ausstoß zu minimieren. Auch Aspekte wie die Senkung von Kosten sind jedoch relevant. Vermehrt aufgekommen ist dieser Begriff um das Jahr 2008, wobei es auch bereits vorher vereinzelte Projekte in diesem Bereich gegeben hat.

In den hier durchgeführten Experimenten soll Aufgrund der Relevanz von Green-IT zusätzlich zu den Metriken, die oben genannt wurden (siehe Abschnitt 6.1), der Energieverbrauch der Zeitreihen-Datenbanksysteme gemessen werden. Mehr zur Art und Weise der Messungen wird in Abschnitt 6.2.3 beschrieben.

6.2 Testsystem

Für die nachfolgenden Experimente wurde ein Testsystem entwickelt, das die Ausführung dieser weitestgehend automatisiert. Dazu wird u. a. die Testumgebung eingerichtet, das Zeitreihen-Datenbanksystem installiert und das entsprechende Szenario gestartet. Zudem werden die oben genannten Metriken (siehe Abschnitt 6.1) vollautomatisch vom Testsystem erfasst und dem jeweiligen Experiment zugeordnet.

6.2.1 Aufbau

Das Testsystem ist in zwei Teile aufgeteilt (siehe Abb. 6.1): Zum einen gibt es den Teil, der die Verwaltung der Experimente, die Ausführung der Szenarien und das Aufzeichnen der Metriken übernimmt. Dieser Teil wird Testsystem-Verwaltung (TSV) genannt. Den zweiten Teil stellt die Testumgebung (TU) dar, in der das Zeitreihen-Datenbanksystem (engl. Time Series Database System bzw. TSDBS) und die Erfassung der Metriken ausgeführt wird.

Die TSV und Metrikerfassung wurden in der Programmiersprache Rust [75] entwickelt. Diese Sprache wurde hier gewählt, da sie zum einen eine effiziente ressourcenschonende Programmierung ermöglicht. Dies ist insbesondere für die Metrikerfassung wichtig, da so das Testergebnis so wenig wie möglich verfälscht wird. Zum anderen bietet Rust ein statisches Typsystem und garantiert Speichersicherheit, wodurch Programmierfehler reduziert werden können.

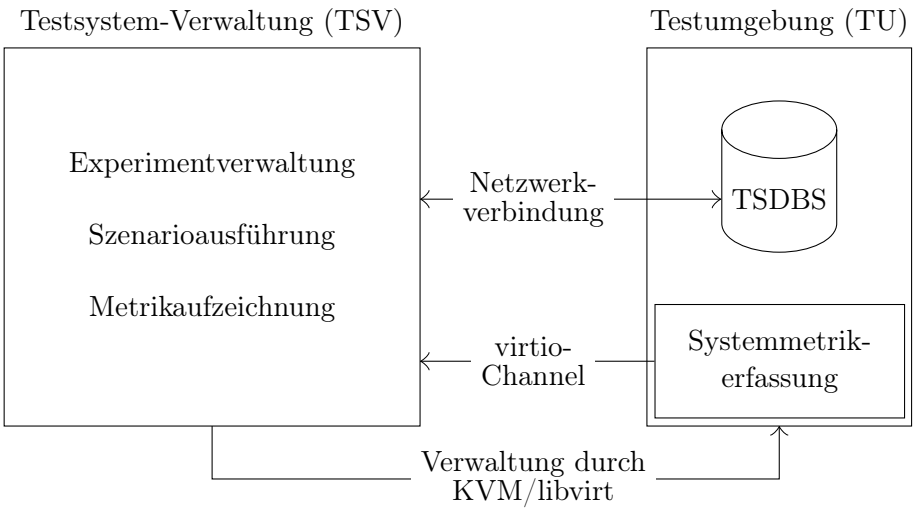


Abbildung 6.1: Schematische Übersicht des Aufbaus des Testsystems (für UML-Komponentendiagramm siehe Kapitel 6.2.4)

In den folgenden Abschnitten wird genauer auf den Aufbau der beiden Teile des Testsystems eingegangen. Details zur Einrichtung des Testsystems lassen sich im Anhang A.2 nachlesen.

6.2.2 Testumgebung (TU)

Es gibt verschiedene Möglichkeiten, die Testumgebung auszuführen, die jeweils eigene Vor- und Nachteile haben. Nachfolgend werden diese beschrieben und die endgültige Wahl begründet.

- *direkt auf dem Wirtsystem (engl. Host)*: Vorteilhaft ist hier, dass die Komplexität des Aufbaus niedrig ist und direkt über die Loopback-Netzwerkschnittstelle mit dem DBS kommuniziert werden kann. Es treten jedoch auch folgende Probleme auf: Zum einen stören die auf dem Wirtsystem laufenden fremden Prozesse und das Testsystem selbst die Messergebnisse. Zum anderen ist es schwierig, zwischen den verschiedenen DBS zu wechseln, da entweder alle DBS parallel installiert sein müssen oder jedes DBS vor dem jeweiligen Experiment dynamisch installiert und anschließend wieder deinstalliert werden muss.

- *in einem Container*: Der Container sorgt dafür, dass sich Prozesse, die darin ausgeführt werden, in einer getrennten Gruppe zu den restlichen Prozessen des Wirtes befinden. Weitere Details dazu lassen sich in der Linux-Kernel-Dokumentation [48] nachlesen. Hierdurch ergibt sich der Vorteil, dass für jedes Experiment ein eigener neuer Container erzeugt werden kann, in dem nur das jeweilige DBS installiert ist. Durch diese Trennung stören andere Prozesse des Wirtes hier zudem weniger als die zuvor genannte Variante. Jedoch besteht hierbei das Problem, dass bei Containern der Kernel des Wirtsystems mitgenutzt wird, wodurch so wiederum Störeinflüsse entstehen.

- *in einer virtuellen Maschine (VM)*: Im Gegensatz zu Containern wird bei virtuellen Maschinen ein gesamter virtueller Rechner (Gast) inklusive Prozessoren, Arbeitsspeicher, Festplatten, Netzwerkschnittstellen etc. erstellt, wodurch die Ressourcen des Wirtes ineffizienter genutzt werden. Hier läuft somit ein eigenes Betriebssystem, welches größtenteils getrennt vom Wirt ausgeführt wird. Solange der Wirt nicht überlastet wird, sind die Störeinflüsse auf den Gast jedoch minimal. Zudem lassen sich in einer VM Parameter wie die Anzahl der Prozessorkerne oder die Größe des Arbeitsspeichers frei konfigurieren.

- *auf einem separaten Rechner*: Bezüglich der Störungen durch andere Prozesse wäre es optimal, die Testumgebung auf einem eigenen Rechner mit separater Hardware zu betreiben. Allerdings ist der Aufwand hierfür deutlich größer als bei den anderen Varianten, da die Kommunikation zum restlichen Testsystem sichergestellt werden muss und auch hier die Installationsprobleme auftreten, die bereits in der ersten Variante beschrieben wurden. Darüber hinaus müssen zur Testung verschiedener Hardwarekonfigurationen manuelle Änderungen am System vorgenommen werden.

Aufgrund dieser Überlegungen wurde entschieden, die Testumgebung in einer virtuellen Maschine (VM) auszuführen, da die reduzierte Performanz im Vergleich zu Containern in diesem Anwendungsfall nicht relevant ist. Zudem wurde von einem separaten Rechner abgesehen, da einerseits die benötigte Hardware nicht zur Verfügung stand und andererseits der entstehende Aufwand den Nutzen nicht gerechtfertigt hätte. Das Testsystem wurde jedoch so entwickelt, dass mit nur geringen Änderungen die Experimente auf einem anderen Rechner ausgeführt werden können.

Zur Virtualisierung wurde hier KVM [72] in Kombination mit libvirt [73] genutzt. KVM ist die in den Linux-Kernel integrierte Virtualisierungssoftware, die es ermöglicht auf Linux-Systemen virtuelle Maschinen auszuführen. Zudem bietet KVM eine Reihe an paravirtualisierten Komponenten wie z. B. Festplatten, Netzwerkkarten etc. an, für die bereits Treiber im Linux-Kernel vorinstalliert sind, wodurch der Einrichtungsaufwand des Gastes verringert wird. libvirt bietet eine Programmierschnittelle an, mit der die Verwaltung von KVM automatisiert werden kann. Im Testsystem werden so automatisch virtuelle Maschinen konfiguriert, gestartet und am Ende des Experimentes wieder gestoppt. KVM und libvirt werden von der Firma RedHat entwickelt. Es wurde sich gegen andere Virtualisierungslösungen wie Oracle VirtualBox [67] oder VMware Workstation [6] entschieden, da die automatische Konfiguration bei diesen Produkten aufwändig ist. Zudem war VMware Workstation zur Zeit dieser Entscheidung kostenpflichtig. Nach der Übernahme von VMWare durch Broadcom wurde dies wurde jedoch mittlerweile geändert.

Für das in der VM ausgeführte Betriebssystem gibt es zwei Anforderungen:

1. Das System soll eine realistische Umgebung für das DBS darstellen und
2. es sollen hier wie oben erwähnt die Störeinflüsse minimal sein.

Anhand dieser Anforderungen wurde sich für eine minimale Installation der Linux-Distribution Debian [81] entschieden, welche bereits vorkonfiguriert als Festplattenabbild von der Website des Projektes heruntergeladen werden kann. Gegen die Distribution Alpine [2], die auch häufig als minimales System genutzt wird, wurde sich hier entschieden, da hier Inkompatibilitäten zu manchen DBS bestehen, weil Alpine nicht die Linux-Standard-C-Bibliothek glibc nutzt, sondern auf die eigene Implementation musl setzt. Microsoft Windows [49] wurde hier aufgrund der mangelnden Konfigurierbarkeit bzgl. installierten und ausgeführten Komponenten, der hohen Ressourcennutzung im Vergleich zu Debian und wegen nicht vorhandenen Lizenzen ebenfalls nicht genutzt.

Um das Betriebssystem nach Start der VM zu konfigurieren und das Zeitreihen-Datenbanksystem zu installieren, wird die Automatisierungssoftware cloud-init [8] genutzt, die von der Firma Canonical entwickelt wird. Die Software wurde für die Einrichtung von VMs bei Cloud-Providern konzeptioniert, kann aber auch im Nicht-Cloud-Kontext eingesetzt werden. Zudem ist cloud-init bereits in dem vom Debian-Projekt bereitgestellten Festplattenabbild vorinstalliert, wodurch keine manuelle Konfiguration nötig ist. Dies ist auch der Grund, warum diese Software genutzt wurde. Wenn die VM startet, fragt cloud-init über HTTP bei der Konfigurationsverwaltung des Testsystems an, wie das System eingerichtet werden muss. Die dabei entstehende Konfigurationsdatei wird vom Testsystem im yaml-Format automatisch generiert.

In der Testumgebung wird neben dem Zeitreihen-Datenbanksystem die Systemmetrikerfassung ausgeführt. Diese zeichnet die Systemmetriken der Testumgebung (siehe Abschnitt 6.1.1) mithilfe der Bibliothek sysinfo [20] auf und sendet sie über einen virtio-Channel an die TSV. Virtio-Channels sind bidirektionale Übertragungskanäle, die von KVM bereitgestellt werden und für die bereits Treiber im Linux-Kernel verfügbar sind. Sie stellen einen Weg dar, um zwischen Wirt und Gast zu kommunizieren. Ein solcher Kanal wurde genutzt, um sicherzustellen, dass die Messergebnisse nicht beeinträchtigt werden. Im Gegensatz zur Alternative, bei der die Daten über die Netzwerkschnittstelle übertragen werden – von der ebenfalls Messwerte erfasst werden – wird die Messung so nicht gestört.

6.2.3 Testsystem-Verwaltung (TSV)

Die Testsystem-Verwaltung besteht im Kern aus vier Komponenten, die jeweils folgende Aufgaben besitzen:

1. *Szenarioausführung*: Ausführung der Szenarios inkl. Durchführung der Anfragen
2. *Metrikaufzeichnung und Strommessung*: Aufzeichnung der Metriken, die von der Systemmetrikerfassung und der Strommessung gemessen wurden. Zusätzlich Speicherung der Latenzen der Anfragen
3. *VM-Verwaltung*: Einrichtung der VMs inkl. Konfiguration von Ressourcen und Kommunikation mit libvirt
4. *Experimentverwaltung*: Steuerung des Ablaufs der Experimente bzw. Koordination der anderen Komponenten

In den folgenden Abschnitten wird die Funktionsweise der Komponenten genauer beschrieben.

Szenarioausführung

Diese Komponente stellt die Ausführung der Szenarios im Testsystem dar. Szenarios sind dabei immer in zwei Schritte geteilt: Der erste Teil – die Initialisierung – erstellt das Schema des DBS, falls es nicht schemafrei ist, und befüllt das DBS je nach Szenario mit den entsprechenden Testdaten. Der zweite Teil des Szenarios befasst sich mit der Ausführung der Anfragen. Dabei wird jeweils die Latenz der Anfragen in dieser Komponente gemessen und an die Metrikerfassungskomponente übertragen.

Die derzeitige Implementation des Testsystems sieht vor, dass Szenarios in Rust geschrieben werden können, wobei es durchaus möglich ist, ein externes Programm wie Python als Szenario zu nutzen. In diesem Fall könnte die Mitteilung der Latenz-Werte über die Standard-Ausgabe (`stdout`) des Python Prozesses erfolgen.

Metrikaufzeichnung und Strommessung

Die Metrikaufzeichnung (MA) stellt das Gegenstück zu der Systemmetrikerfassung (SME) dar. Die Kommunikation mit der SME geschieht, wie bereits in Abschnitt 6.2.2 beschrieben, über einen virtio-Channel, der auf der Seite des Hosts durch einen UNIX-Socket repräsentiert wird.

Um den Stromverbrauch der DBS zu messen, wird die Bibliothek „scaphandre“ [4] genutzt, die die in Linux integrierte intel-rapl-Schnittstelle verwendet. Diese Schnittstelle ist dabei nicht nur mit Prozessoren der Firma Intel, sondern auch mit AMD-Prozessoren kompatibel. Diese Schnittstelle ermöglicht es, den Energieverbrauch der CPU in µJ (mikro Joule) abzufragen, dieser wird jedoch in µW (mikro Watt) umgerechnet, um einen zeitunabhängigen Wert zu erhalten. Zu beachten ist dabei, dass lediglich der Verbrauch des Prozessors gemessen wird. Andere Verbraucher wie Hauptplatine, Festplatten, etc. werden nicht gemessen, da es für diese keine bzw. keine einheitliche Schnittstelle gibt. Zudem müsste hier aus dem Gesamtverbrauch der Verbrauch der VM extrahiert werden. Dies wäre jedoch nur schwer möglich, da die hierfür benötigten Nutzungswerte nicht verfügbar sind.

Wenn Metriken von der MA empfangen werden, werden sie zunächst bis zum Abschluss des jeweiligen Experimentes in einer Datenstruktur im Arbeitsspeicher abgelegt, die die drei Metrikkategorien – Systemmetriken, Stromverbrauch und Latenzen – jeweils in einer eigenen Liste speichert. Anschließend werden die Daten in ihrer Rohform in eine CSV-Datei pro Metrikkategorie geschrieben. Zusätzlich werden statistische Kennzahlen wie das arithmetische Mittel, die Standardabweichung, der Median etc. von der Metrikerfassung berechnet und gespeichert.

VM-Verwaltung

Diese Komponente übernimmt die Konfiguration und Steuerung der VMs, für die libvirt und cloud-init genutzt werden, die bereits im Abschnitt 6.2.2 kurz erwähnt wurden.

Eine virtuelle Maschine basiert in libvirt auf einer XML-Datei, die eine Beschreibung ihrer Eigenschaften enthält, wie z. B. Name, Prozessoranzahl, Arbeitsspeicher und angeschlossene (virtuelle) Geräte. Zudem wird in diesem XML-Dokument die Adresse des cloud-init-Konfigurationsendpunktes angegeben, über den das Betriebssystem der VM eingerichtet wird. Wenn eine VM vom Testsystem erstellt werden soll, wird eine neue

XML-Datei anhand einer Vorlage mit den gewünschten Parametern gefüllt und an libvirt übergeben. Danach wird die so neu erstellte VM automatisch gestartet.

Damit neue VMs immer mit einem neu installierten Betriebssystem starten können, wird vor Erstellung der VMs ein so genannter „linked clone“ (verknüpfte Kopie) vom Grund-Festplattenabbild erstellt, der lediglich die vorgenommenen Änderungen speichert. Nachdem eine VM nicht mehr benötigt wird, kann diese Kopie gelöscht werden.

Experimentverwaltung

Die Experimentverwaltung ist der Teil des TSV, der alle anderen Komponenten steuert. Ihr Ziel ist es, für jedes Szenario jede Kombination von Systemkonfiguration und Anfrage durchzuführen. Zudem wird jede Anfrage zehnmal ausgeführt, damit Schwankungen zwischen den einzelnen Durchläufen berücksichtigt werden können.

Für die Ausführung der Experimente eines Szenarios auf einem DBS werden die folgenden Schritte durchgeführt:

1. Zunächst wird eine VM gestartet, in der das jeweilige DBS installiert und im Anschluss das Szenario initialisiert wird. Das durch diesen Vorgang entstandene Festplattenabbild mit installiertem DBS und eingefügten Daten wird bei der Durchführung der Experimente als Grundlage genutzt. Hierdurch wird vor allem Zeit gespart, da jedes Szenario pro DBS nur ein Mal initialisiert werden muss.
2. Im zweiten Schritt wird jede Anfrage für jede in den Rahmenbedingungen des Szenarios erwähnte Systemkonfiguration zehnmal in einer eigenen VM ausgeführt. So wird sichergestellt, dass sich die Anfragen untereinander nicht beeinflussen können.

6.2.4 Architektur

In diesem Abschnitt wird die Architektur der Komponenten des Testsystems beschrieben. Dabei werden die Bezeichnungen der Komponenten genutzt, die auch im Quelltext verwendet wurden. Auf die Klassen der Komponenten wurde hier verzichtet, da hier die Struktur der Komponenten betrachtet werden soll.

Abbbildung 6.2 zeigt eine Kombination der ersten Schicht des UML-Komponentendiagramms und der Verteilungssicht der Komponenten auf die beiden Teile des Systemsytems – der Testverwaltung (`databasesystem_tester`) und der Metrikerfassung

(`instrumentation_gatherer`). Die Bezeichnungen und Funktionen der Komponenten hier entsprechen denen aus Abschnitt 6.2.3 wie folgt:

- `testcase`: Experimentverwaltung
- `vm`: VM-Verwaltung
- `scenario`: Szenarioausführung
- `metrics`: Metrikaufzeichnung und Strommessung

Bis auf die Komponente `testcase` bestehen alle Komponenten wiederum aus weiteren Komponenten, die im Folgenden beschrieben werden.

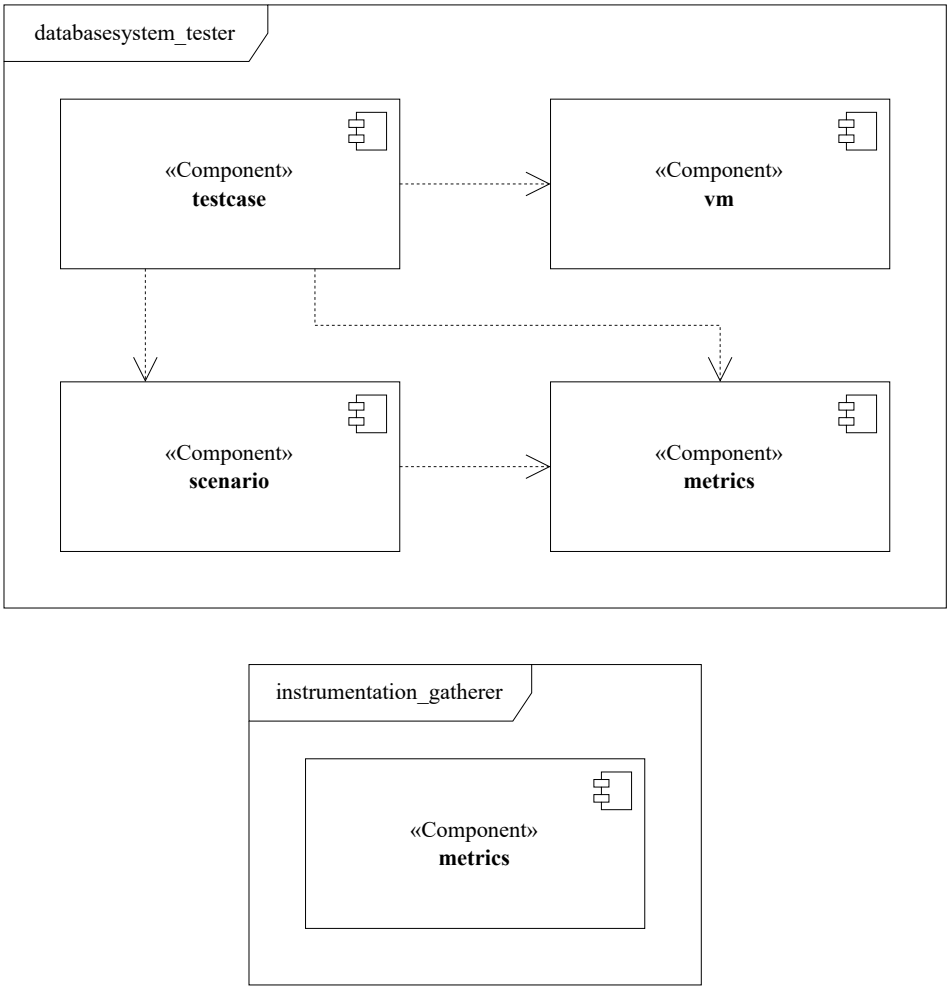


Abbildung 6.2: Schicht 1 des UML-Komponentendiagramms bzw. Verteilungssicht

Komponente „vm“

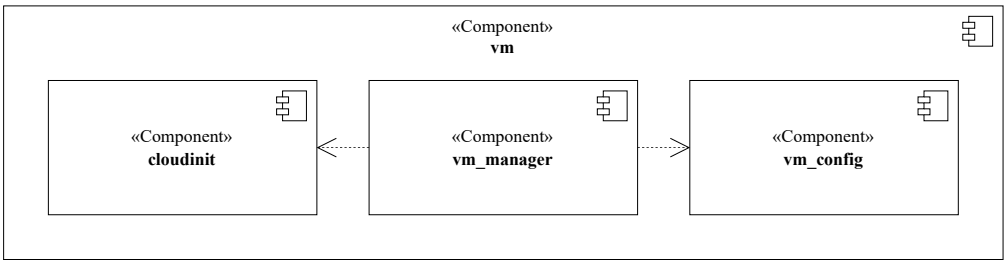


Abbildung 6.3: Schicht 2 des UML-Komponentendiagramms der Komponente `vm`

Die Komponente `vm` (Abb. 6.3) besteht aus drei Unterkomponenten: Der `vm_manager` kommuniziert zum libvirt und verwaltet den Lebenszyklus und die Konfiguration der VMs. Die Komponente `vm_config` liest eine Konfigurationsdatei ein, die unter anderem Skripte zur Installation der Datenbanksysteme sowie allgemeine Einstellungen für die Testumgebung, wie etwa die Netzwerkkonfiguration, enthält. Diese so eingelesene Konfiguration wir vom `vm_manager` an die Komponente `cloudinit` weitergereicht, die diese im entsprechenden yaml-Format für die VM anbietet (siehe Abschnitt 6.2.2).

Komponente „scenario“

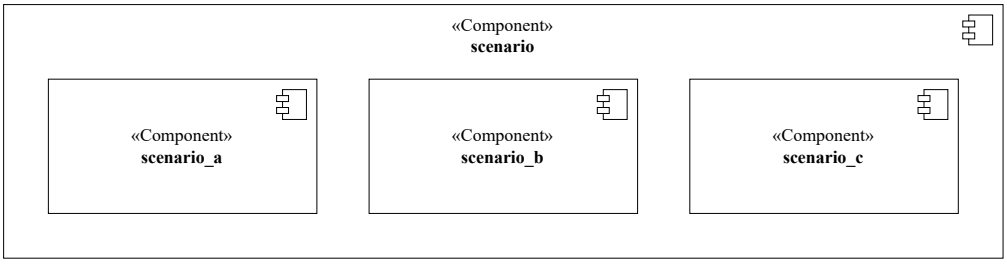


Abbildung 6.4: Schicht 2 des UML-Komponentendiagramms der Komponente `scenario`

In der Komponente `scenario` (Abb. 6.4) gibt es für jedes Szenario wiederum eine Unterkomponente. Der Aufbau dieser Unterkomponenten hängt dabei von den Szenarios ab. Szenario A und C besitzen jeweils eine Generatorkomponente, die die jeweiligen Testdaten erzeugt. Szenario B hat diese Komponente nicht, da hier lediglich aus den parquet-Dateien des Datensatzes gelesen werden muss. Des Weiteren haben alle Szenarios eine Komponente, die die Verbindungen zu den verschiedenen Datenbanksystemen abstrahiert. Weitere Details zur Implementierung der Szenarios befinden sich im Abschnitt 6.3

Komponente „metrics“

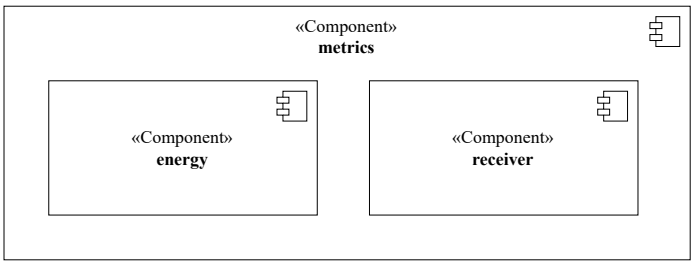


Abbildung 6.5: Schicht 2 des UML-Komponentendiagramms der Komponente `metics`

Die Komponente `metrics` (Abb. 6.5) besteht aus zwei Teilen. Zum einen der `energy`-Komponente, die den Stromverbrauch der VM misst und der `receiver`-Komponente, die die Gegenstelle zur Systemmetrikerfassung in der Testumgebung darstellt.

6.3 Implementierung der Szenarios

Die folgenden Unterabschnitte zeigen, wie die Szenarios im Testsystem aufgebaut und implementiert sind. Zudem wird auf die Anfragen an die Zeitreihen-Datenbanksysteme eingegangen.

6.3.1 Aufbau der Szenarios

Wie bereits in Abschnitt 6.2.3 erwähnt, bestehen die Szenarios aus zwei Stufen – Initialisierung und Ausführung der Anfragen. Diese Struktur wird im Quelltext durch eine Schnittstelle (siehe Abb. 6.6) abgebildet, die zwei entsprechende Methoden besitzt. Zusätzlich hat diese Schnittstelle Methoden, über die der Name, die möglichen Anfragen und die zu testenden Systemkonfigurationen des Szenarios abgefragt werden können. Jedes der Szenarios implementiert diese Schnittstelle, wodurch die Experimentverwaltung (bzw. die Komponente `testcase`) keine szenariospezifische Logik benötigt.

Den Methoden `run()` und `init()` wird über deren Parameter das zu verwendende DBS in Form eines Enums inkl. der Adresse und den Zugangsdaten übergeben. Anhand dieser Informationen stellen die Szenarios eine Verbindung zum DBS her, wobei in Szenario A mehrere Verbindungen aufgebaut werden. Bei der Methode `run()` wird zudem

die auszuführende Anfrage und ein Bezeichner des aktuellen Durchlaufs in Form einer Zeichenkette über zwei Parameter mitgeteilt.

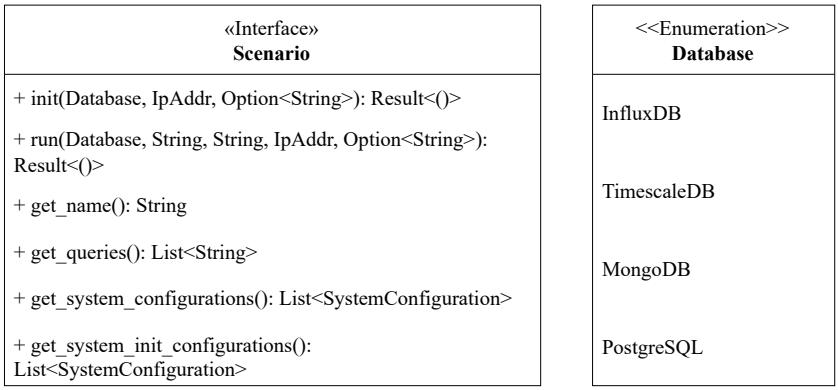
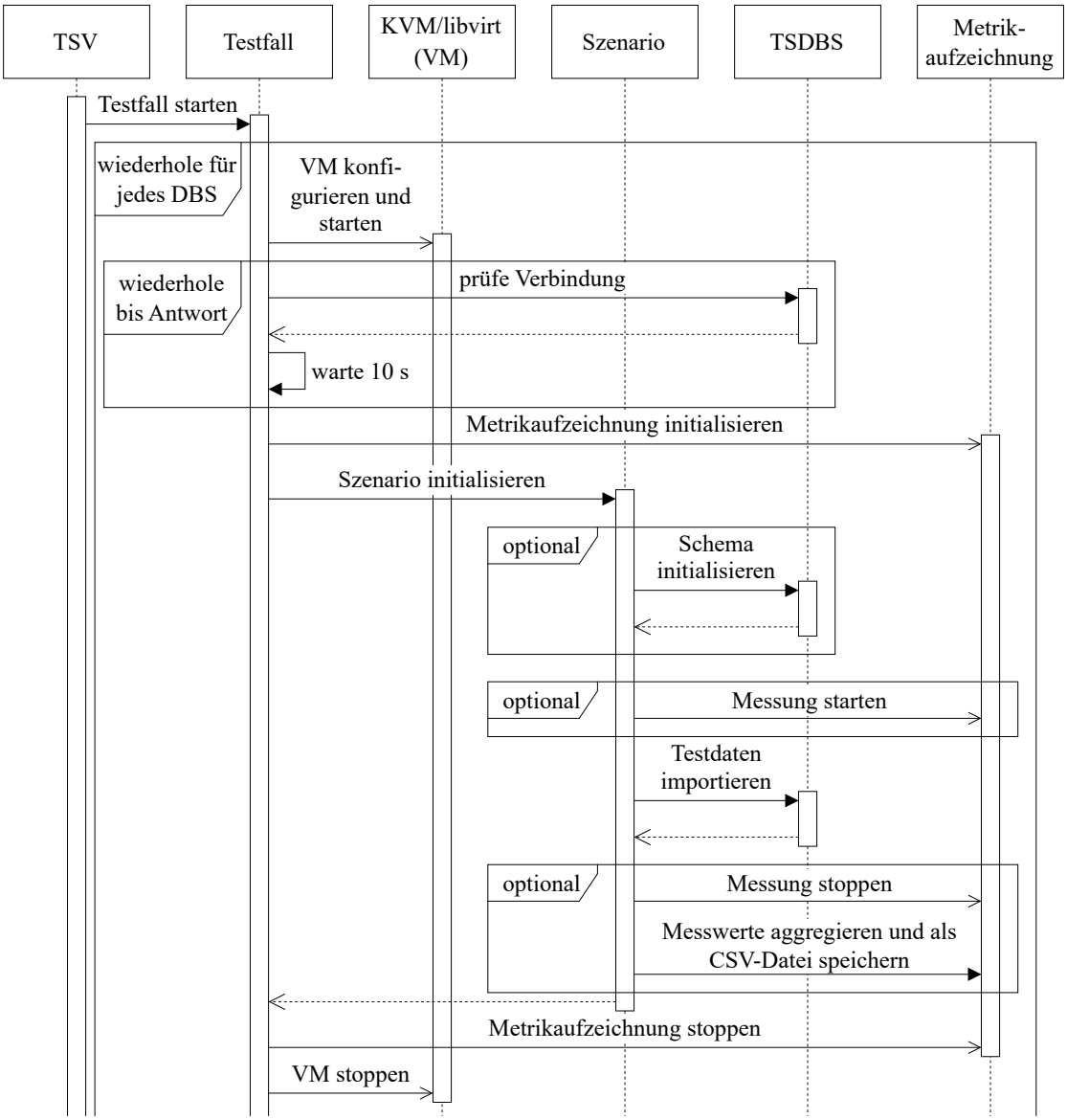


Abbildung 6.6: UML-Diagramm der Schnittstelle `Scenario` und des Enum `Database`

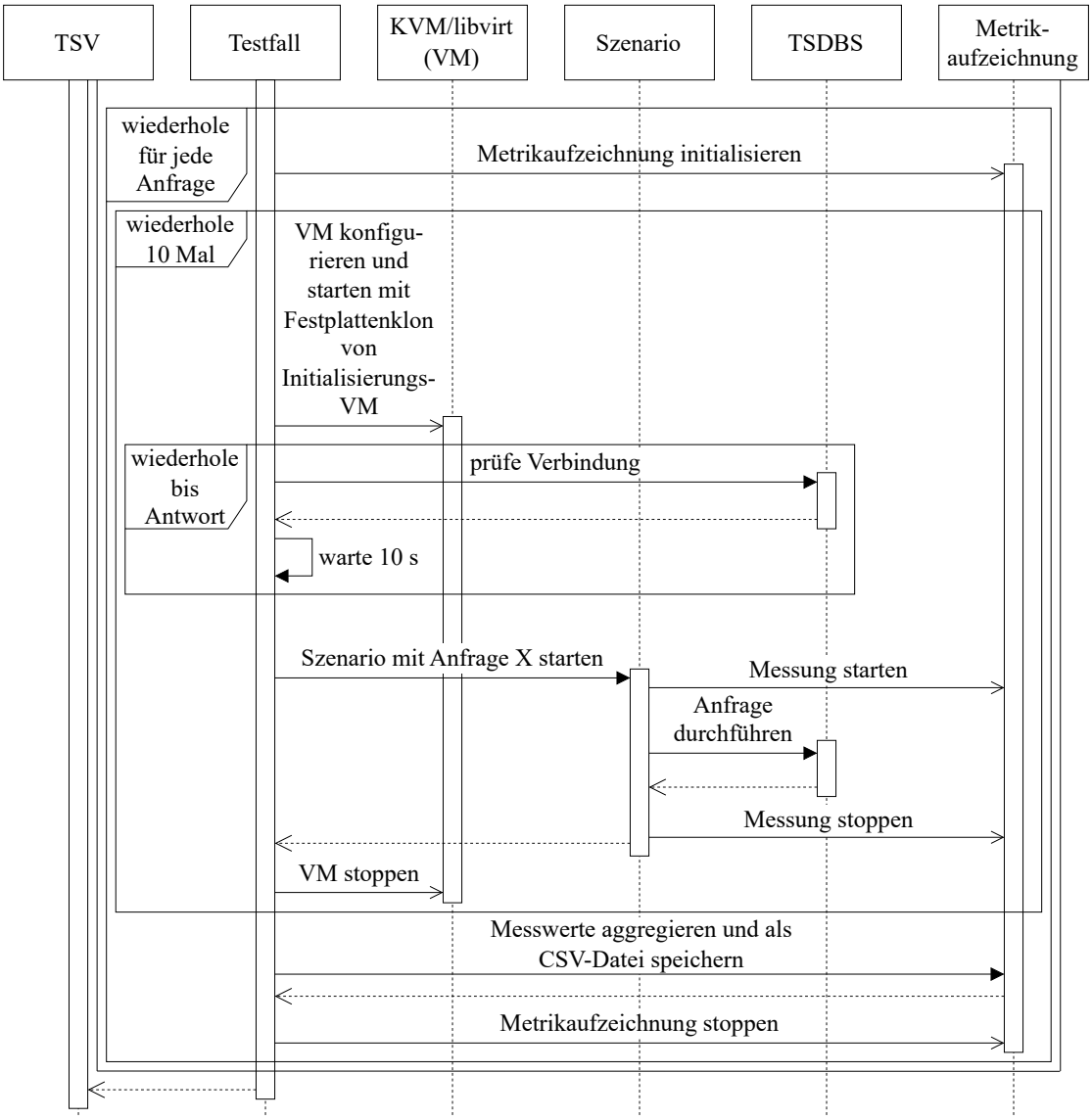
Um auch die Szenario-Implementierungen möglichst generisch bezüglich der Datenbanksysteme zu halten, hat jedes Szenario eine eigene Datenbank-Schicht, die pro DBS jeweils jede Anfrage implementiert. Hier wird ebenfalls eine Schnittstelle genutzt, in der für jede Anfrage eine entsprechende Methode definiert ist.

6.3.2 Szenarioablauf

Das UML-Sequenzdiagramm in Abbildung 6.7 stellt den vereinfachten Ablauf eines Szenarios dar.



(a) Teil 1/2 (Fortsetzung auf der folgenden Seite)



(b) Teil 2/2

Abbildung 6.7: Ablauf eines Szenarios als UML-Sequenzdiagramm

6.3.3 Implementierung der Generatoren

Die Generatoren aus Szenario A und C sind alle als Iterator implementiert. Iteratoren in der Programmiersprache Rust besitzen im Kern die `next()`-Methode. Sie liefert jeweils das nächste Element des Generators, wenn der Generator nicht bereits alle zu generierende Elemente ausgegeben hat. Für die Generatoren wurde in dieser Methode jeweils das in Kapitel 4 beschriebene Verfahren implementiert. Da Iteratoren ein grundlegendes Konzept in der Sprache sind, sind sie gut darin integriert und können so flexibel genutzt werden.

Damit die Experimente reproduzierbar sind, wurde für die Zufallszahlengeneratoren jeweils ein so genannter Seedwert gesetzt, der dafür sorgt, dass immer die gleiche zufälligen Werte erzeugt werden. Zu beachten ist dabei jedoch, dass in zukünftigen Versionen der Zufallszahlen-Bibliothek andere Zahlenfolgen generiert werden könnten.

Die Parameter der Generatoren für die Szenarios A und C wurden experimentell so bestimmt, dass sie möglichst realistisch anmutende Daten erzeugen. In folgenden beiden Unterabschnitten werden die ermittelten Werte der Generatoren genannt.

Parameter der Generatoren von Szenario A

- Generator-Erzeugungsrate: 3 s
- Maximale Generatoranzahl: 400
- Generierungsintervalle der Generatoren:
  - elektrische Verbraucher: 1 s
  - Temperatur: 2 s
  - Fenster und Türen: ≈ 50 s
  - Beleuchtung: ≈ 50 s
- Parameter der Temperaturgeneratoren:
  - Periode: $p = 86400$
  - Temperaturunterschied zwischen Tag und Nacht: $v = 4$

  - Verschiebung: $s = 0$
  - Nachttemperatur: $o = 17$
  - Stauchung: $d = 0{,}05$

Parameter der Generatoren von Szenario C

- Generator Prozessorauslastung:
  - angestrebte Grundlast: $g = 0{,}2$
  - Lastrauschen: $v = 0{,}01$
  - Korrekturfaktor: $x = 0{,}005$
  - Spitzenwahrscheinlichkeit: $p_s = 0{,}001$
  - Spitzenhöhe: $\mu_h = 0{,}8$ $\sigma_h = 0{,}15$
  - Spitzendauer: $\mu_d = 200$ $\sigma_d = 500$
  - Spitzenkorrekturfaktor: $x' = 0{,}02$
- Generator Arbeitsspeicherauslastung:
  - angestrebte Speichernutzung: $\mu_a = 2\,000\,000$ $\sigma_a = 500\,000$
  - Lastrauschen: $v = 2000$
  - Korrekturfaktor: $x = 0{,}00005$

6.3.4 Einfügen der Testdaten

Zu Beginn jedes Szenarios müssen zunächst die Testdaten für das jeweilige Szenario in das DBS einfügt werden. Bei Szenario A geschieht dies wie in Kapitel 4.2.2 beschrieben, indem pro simuliertem Gerät eine Verbindung zum DBS aufgebaut wird und die Daten im Anschluss in Echtzeit übertragen werden. Da bei Szenario B und C nur der Anfrageprozess betrachtet wird, können die Daten hier so schnell eingefügt werden, wie es das DBS ermöglicht. Dazu werden die einzelnen Datenpunkte in Blöcken zusammengefasst und jeweils am Stück übertragen. Bei InfluxDB wird das so genannte „Line protocol“ [33]

genutzt, bei dem jeder Datenpunkt als eine Zeile einer Zeichenkette an eine HTTP-Schnittstelle übertragen wird. In MongoDB wird die Funktion `insertMany()` [52] des Treibers genutzt, mit der mehrere Dokumente über einen Aufruf eingefügt werden können. Bei TimescaleDB und PostgreSQL wird die `COPY FROM STDIN` Funktionalität [86] von SQL genutzt, die es erlaubt, CSV-Daten direkt an die Datenbank zu senden.

6.3.5 Anfragen

Dieser Abschnitt zeigt die Syntax und Komplexität der Anfragen beispielhaft anhand der vierten Anfrage von Szenario B (siehe Abschnitt 4.3.1), die die Stunde mit den meisten Taxifahrten berechnen soll. Diese Anfrage wurde für dieses Beispiel gewählt, da sie die komplexeste Anfrage aller Szenarios darstellt.

InfluxDB

```
import "date"

from(bucket: "bucket")
  |> range(start: 0)
  |> filter(fn: (r) => r["_measurement"] == "trips")
  |> filter(fn: (r) => r["_field"] == "passenger_count")
  |> aggregateWindow(every: 1h, fn: sum, createEmpty: false)
  |> drop(columns: ["_start", "_stop"])
  |> map(fn: (r) => ({r with hour: date.hour(t: r._time)}))
  |> group(columns: ["hour"], mode: "by")
  |> sum(column: "_value")
  |> group()
  |> sort(columns: ["hour"], desc: true)
  |> limit(n: 1)
  |> yield(name: "count")
```

Abbildung 6.8: Vierte Anfrage von Szenario B in Flux für InfluxDB

In der in Abbildung 6.8 gezeigten Anfrage wird die Anfragesprache Flux verwendet. Diese wird zwar, wie in Abschnitt 3.2 erwähnt, mit der kommenden Version von InfluxDB nicht mehr weiterentwickelt, jedoch ist sie für InfluxDB v2 derzeit die empfohlene Variante.

In der Anfrage werden mit `range(start: 0)` zunächst alle Datenpunkte ab dem Startzeitpunkt angefragt – dies ist notwendig, da InfluxDB sonst einen Fehler ausgeben würde.

Anschließend werden mittels der zwei `filter()`-Funktionen nur die Datenpunkte der Zeitreihe mit den Fahrgastzahlen ausgewählt, welche dann in Ein-Stunden-Intervallen aufaddiert werden. Die folgenden Zeilen extrahieren die Uhrzeit, addieren alle Werte der jeweiligen Stunde und geben die Stunde mit den meisten Fahrgästen aus.

TimescaleDB

```
SELECT EXTRACT(HOUR FROM bucket_hour) as hour,
       SUM(bucket_count) as count
FROM (
    SELECT time_bucket('1 hour', tpep_pickup_datetime)
              as bucket_hour,
           SUM(*) as bucket_count
    FROM nyc_taxi_trips
    GROUP BY bucket_hour
)
GROUP BY hour
ORDER BY count DESC
LIMIT 1;
```

Abbildung 6.9: Vierte Anfrage von Szenario B in erweitertem SQL für TimescaleDB

Abbildung 6.9 zeigt die Anfrage in erweitertem SQL für TimescaleDB. Sie besteht dabei aus zwei ineinander geschachtelten SQL-Ausdrücken. Der innere Teil gruppiert die Daten in Ein-Stunden-Intervalle mithilfe der `time_bucket()`-Funktion und aggregiert sie, indem sie addiert werden. Die resultierenden Daten werden dann durch den äußeren Ausdruck nach ihrer zugehörigen Stunde gruppiert und summiert. Schlussendlich wird die Stunde mit den meisten Fahrgästen zurückgegeben.

MongoDB

Für die Anfrage bei MongoDB (siehe Abb. 6.10) wird hier eine MQL-Pipeline (siehe Kapitel 5.2), die aus vier Schritten besteht, genutzt. Der erste Schritt gruppiert und addiert die Daten anhand der jeweiligen Stunde. Danach werden durch die `$project`-Stufe die gewünschten Felder der Dokumente ausgewählt. In den letzten beiden Stufen werden die Ergebnisse absteigend sortiert und das größte Ergebnis ausgegeben.

```
[ { $group: {
    _id: {
      hour: {
        $hour: { date: "$tpep_pickup_datetime" }
      }
    },
    group_count: { $sum: "$passenger_count" }
  } },
  { $project: {
      _id: 0,
      hour: "$_id.hour",
      group_count: 1
  } },
  { $sort: { hour: -1 } },
  { $limit: 1 } ]
```

Abbildung 6.10: Vierte Anfrage von Szenario B als MQL-Pipeline für Mongodb

Postgresql

```
SELECT EXTRACT(HOUR FROM tpep_pickup_datetime) as hour,
       SUM(*) as count
FROM nyc_taxi_trips
GROUP BY hour
ORDER BY count DESC
LIMIT 1;
```

Abbildung 6.11: Vierte Anfrage von Szenario B in SQL für PostgreSQL

Die Anfrage des Baseline-DBS PostgreSQL (siehe Abb. 6.11) ist die kürzeste der vier Anfragen. Sie extrahiert für jede Zeile der Tabelle die jeweilige Stunde, gruppiert die Daten anschließend nach dieser, addiert sie und gibt das größte Ergebnis aus.

6.4 Datensätze

Die drei Szenarios nutzen jeweils unterschiedliche Datensätze. Szenario B verwendet dabei einen Datensatz, bestehend aus realen Daten von Taxis aus New York City (siehe Kapitel 4.3), wohingegen Szenario A und C mit synthetisch generierten Daten arbeiten. Diese werden mithilfe der Generatoren, die in den jeweiligen Kapiteln beschrieben sind,

erzeugt (siehe Kapitel 4.2 und 4.4). Die Parameter der Generatoren lassen sich dabei in Kapitel 6.3.3 ersichtlich.

n Szenario A werden Generatoren erzeugt, deren Anzahl bis zu 400 steigt und die gleichmäßig über die verschiedenen Kategorien verteilt sind. Hierbei wird sichergestellt, dass die Menge der generierten Daten immer gleich groß ist. Die Simulation der Geräte und Sensoren wird hier in Echtzeit ausgeführt.

In Szenario C wird bei jedem Zeitschritt (100 ms) mit einer Wahrscheinlichkeit von 0,05 eine neue Instanz bzw. ein Generator, der diese simuliert, erzeugt. Mit dieser Wahrscheinlichkeit kann dabei die Anzahl der Instanzen und damit auch die Anzahl der Zeitreihen bestimmt werden. Die gesamte simulierte Zeit beträgt dabei einen Tag, die jedoch so schnell wie möglich ausgeführt wird.

Die folgende Tabelle 6.1 zeigt u. a. die Anzahl der Datenpunkte und der Zeitreihen der Datensätze:

Tabelle 6.1: Übersicht der Datensätze

Szenario	Datenpunkte	Zeitreihen	Art der Daten
A	≈ 100 Tsd.	400	snythetisch
B	≈ 337 Mio.	3	real
C	≈ 170 Mio.	≈ 100 Tsd.	synthetisch

6.5 Ergebnisse

Die folgenden Abschnitte stellen die Ergebnisse der Experimente pro Szenario dar und beschreiben diese. Am Anfang jedes Szenarioabschnitts befindet sich eine Tabelle, die die genutzte Festplattenkapazität pro DBS zeigt. Zudem werden für jede Anfrage ein Boxplot der Latenzen und sechs Diagramme dargestellt, die die Systemmetriken (siehe Abschnitt 6.1.1) und die Energienutzung der Testumgebung beinhalten. Letztere Diagramme werden im Folgenden nur erwähnt und nicht genau beschrieben, da ihr Aufbau immer gleich ist: In einem Zwei-mal-Drei-Raster zeigen die Diagramme in der Reihenfolge von links nach rechts und von oben nach unten die durchschnittliche Prozessorauslastung, die durchschnittliche Arbeitsspeichernutzung, die Summe der von der Festplatte gelesenen Daten, die Summe auf die Festplatte geschriebenen Daten, die Summe der Datenmenge der Netzwerk Ein- und Ausgabe und die Summe des Energieverbrauchs der DBS. Aus Platzgründen wurden die Bezeichnungen der DBS wie folgt

abgekürzt: I $\widehat{=}$ InfluxDB, T $\widehat{=}$ TimescaleDB, M $\widehat{=}$ MongoDB und P $\widehat{=}$ PostgreSQL. Des Weiteren werden in den Beschreibungen der Latenzdiagramme die Mediane angegeben. Für Szenario A werden außerdem die Latenzen des Einfügeprozesses gezeigt. Alle Diagramme zeigen dabei die getesteten Systemkonfigurationen in Gruppen von vier für Szenario A und in Gruppen von drei für Szenario B und C. Zudem wird für Szenario B und C nur die Anzahl der Prozessorkerne dargestellt, da sich die Größe des Arbeitsspeichers hier nicht ändert.

Die Rohdaten aus denen die folgenden Diagramme erzeugt wurden, sind zum einen auf der beigefügten CD und unter der im Anhang A.2 genannten Internet-Adresse im CSV-Format verfügbar.

6.5.1 Szenario A

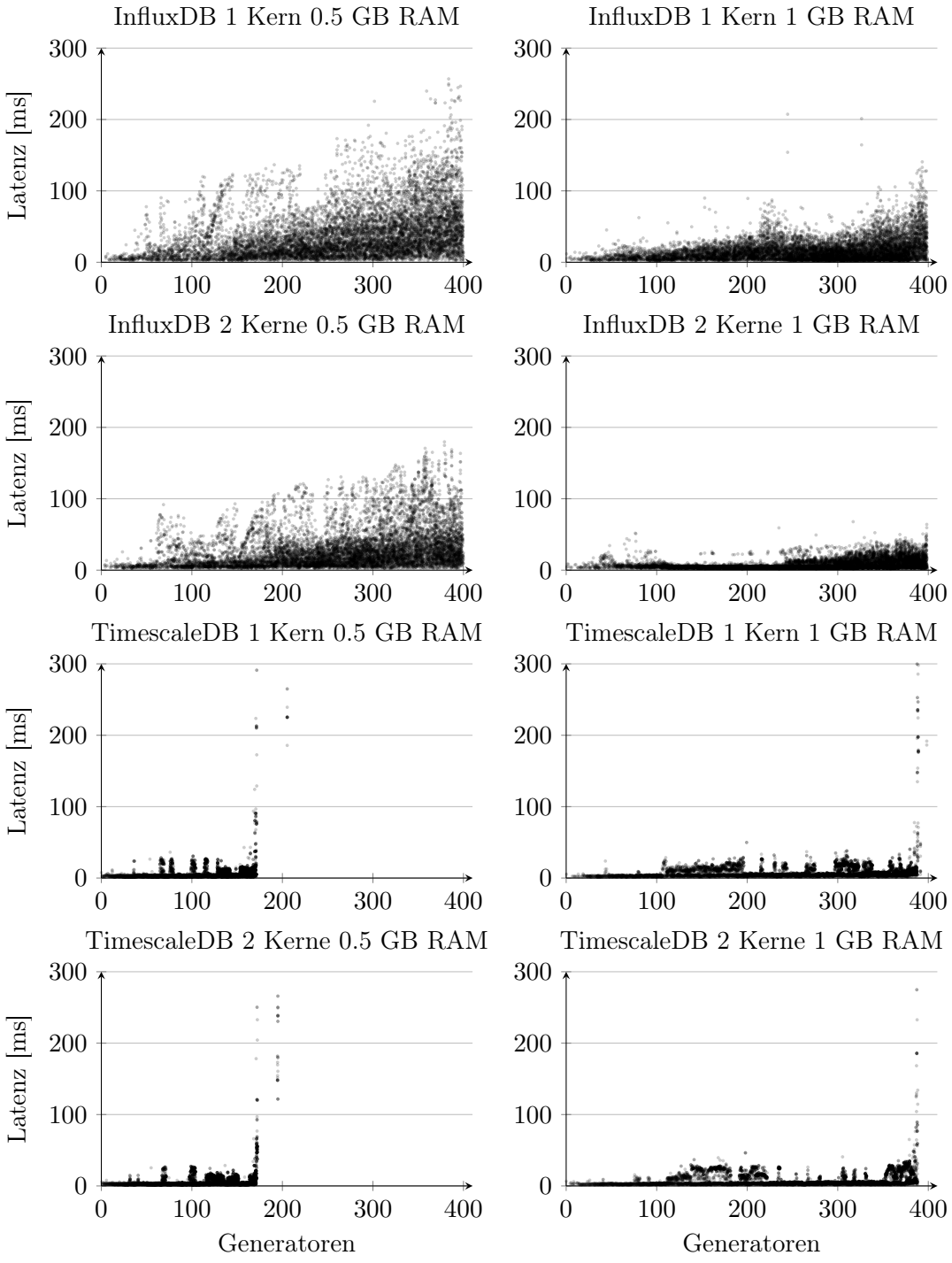
Tabelle 6.2 zeigt die genutzte Festplattenkapazität der DBS:

Tabelle 6.2: Genutzte Festplattenkapazitäten pro DBS

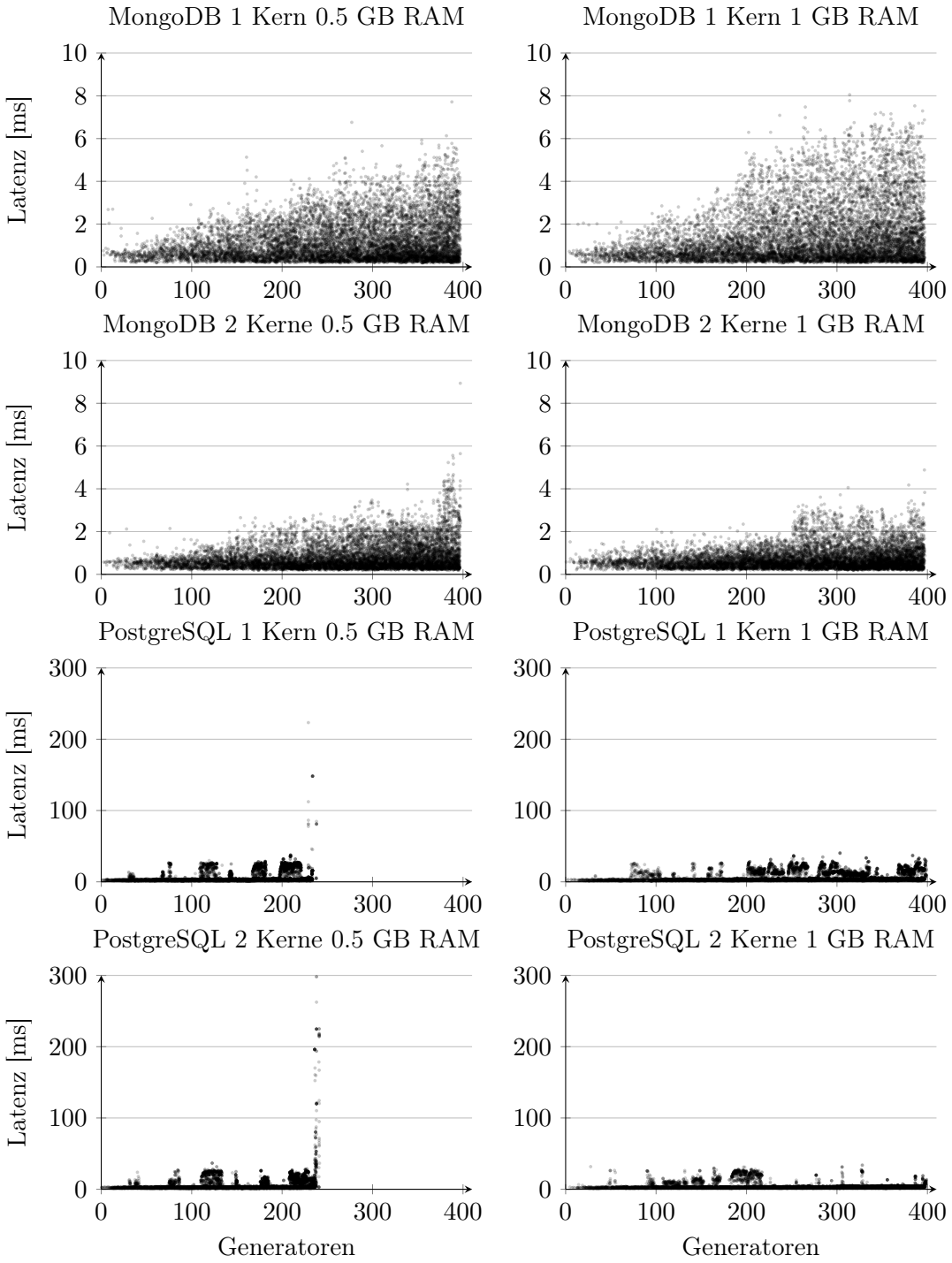
	InfluxDB	TimescaleDB	MongoDB	PostgreSQL
Festplattennutzung	5,09 GB	5,37 GB	5,79 GB	5,26 GB

Einfügen der Daten

Die Diagramme in Abbildung 6.12 zeigen die Einfügelatenzen der Zeitreihen-Datenbanksysteme mit den jeweiligen Systemkonfigurationen. Dabei ist zu beachten, dass die Diagramme teils keine Werte bis zu einer Anzahl von 400 Generatoren darstellen, da in den entsprechenden Fällen das DBS vor Erreichen dieser Generatoranzahl abgestürzt ist. Dies ist insbesondere bei allen TimescaleDB-Experimenten und bei den PostgreSQL-Experimenten der Fall, die 0,5 GB Arbeitsspeicher nutzen. Des Weiteren wurde für MongoDB eine Skala von null bis zehn Millisekunden verwendet, da die Ergebnisse so besser sichtbar werden. Alle anderen Diagramme nutzen im Gegensatz dazu Skalen, die von null bis 300 reichen.



(a) Teil 1/2 (Fortsetzung auf der folgenden Seite)



(b) Teil 2/2

Abbildung 6.12: Einfügelatenzen der DBS mit jeweiligen Systemkonfigurationen für Szenario A

Anfrage 1 – Abfrage der Anzahl von Öffnungen von Fenstern und Türen

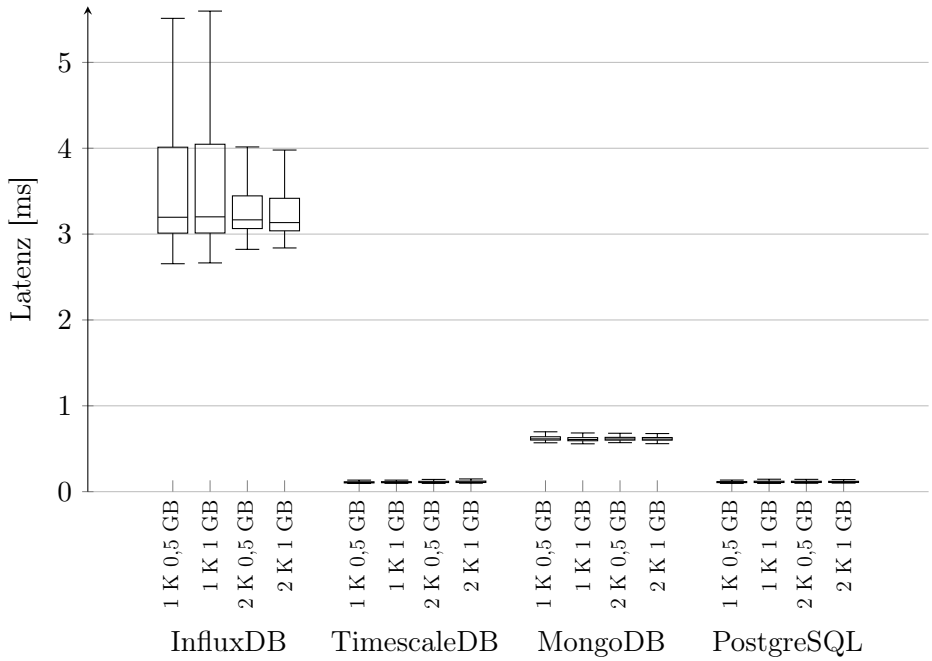


Abbildung 6.13: Latenzen der ersten Anfrage von Szenario A (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.13 zeigt die Latenzen der ersten Anfrage von Szenario A in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 0,11 Millisekunden
2. *PostgreSQL*: ca. 0,12 Millisekunden
3. *MongoDB*: ca. 0,61 Millisekunden
4. *InfluxDB*: ca. 3,13 bis 3,20 Millisekunden

Abbildung 6.14 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

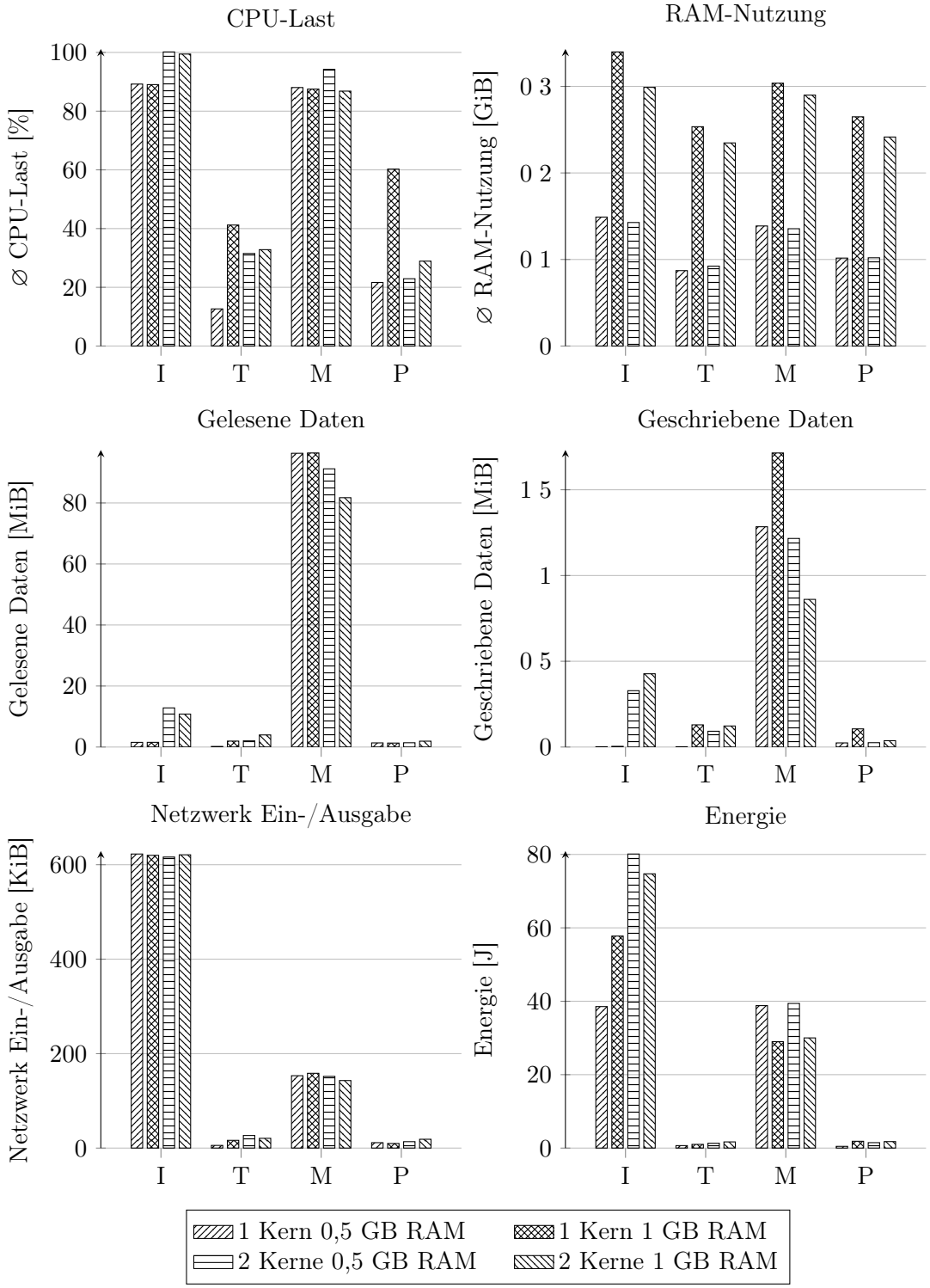


Abbildung 6.14: System- und Energiemetriken der ersten Anfrage von Szenario A (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 2 – Abfrage des Gesamtstromverbrauchs

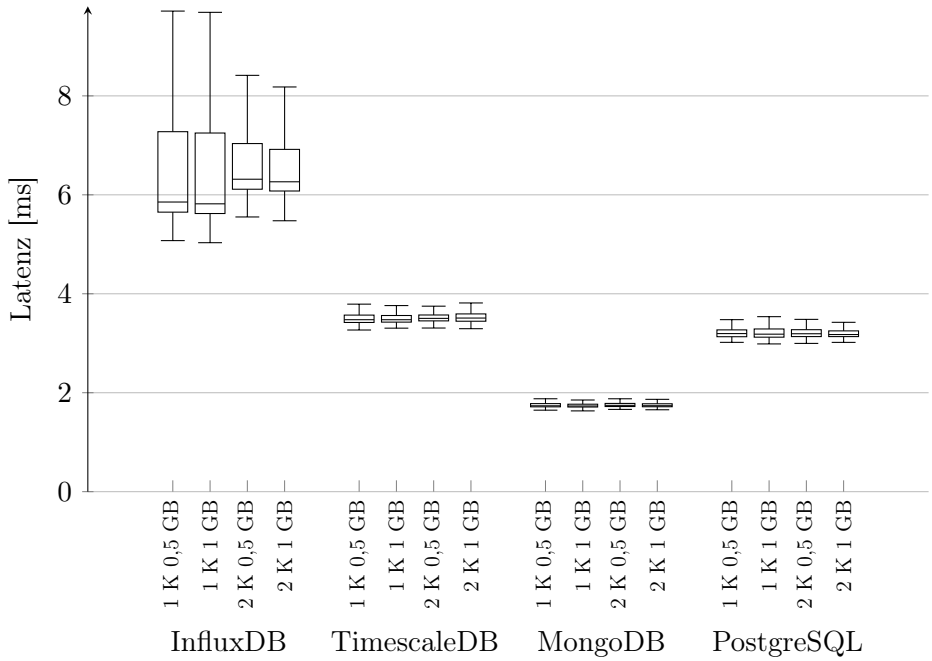


Abbildung 6.15: Latenzen der zweiten Anfrage von Szenario A (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.15 zeigt die Latenzen der zweiten Anfrage von Szenario A in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *MongoDB*: ca. 1,74 Millisekunden
2. *PostgreSQL*: ca. 3,18 bis 3,20 Millisekunden
3. *TimescaleDB*: ca. 3,48 bis 3,51 Millisekunden
4. *InfluxDB*: ca. 5,82 bis 6,32 Millisekunden

Abbildung 6.16 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

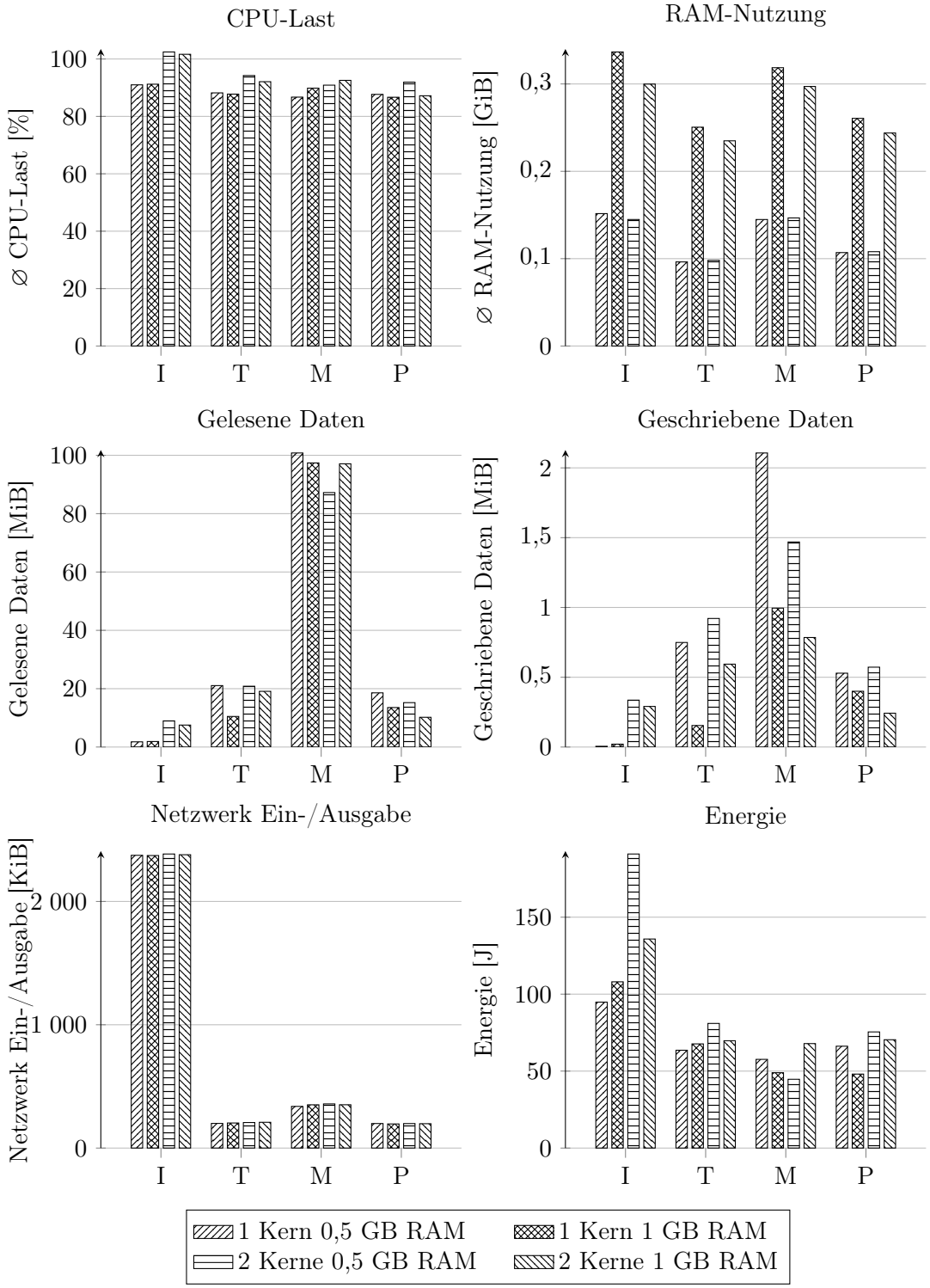


Abbildung 6.16: System- und Energiemetriken der zweiten Anfrage von Szenario A (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 3 – Abfrage der minütlichen Durchschnittstemperatur

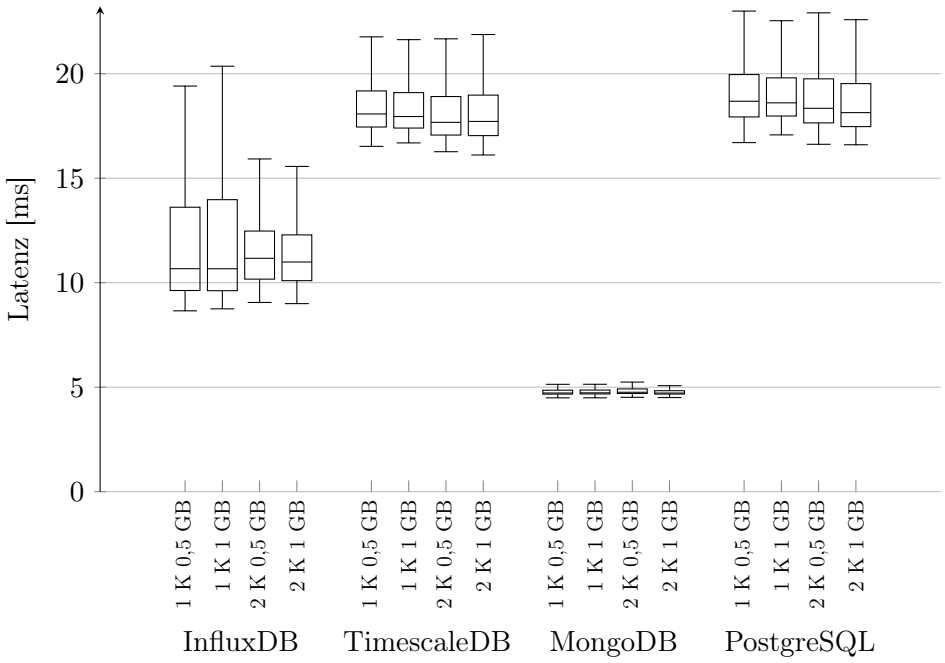


Abbildung 6.17: Latenzen der dritten Anfrage von Szenario A (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.17 zeigt die Latenzen der dritten Anfrage von Szenario A in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *MongoDB*: ca. 4,72 und 4,76 Millisekunden
2. *InfluxDB*: ca. 10,67 bis 11,17 Millisekunden
3. *TimescaleDB*: ca. 17,67 bis 18,07 Millisekunden
4. *PostgreSQL*: ca. 18,14 bis 18,68 Millisekunden

Abbildung 6.18 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

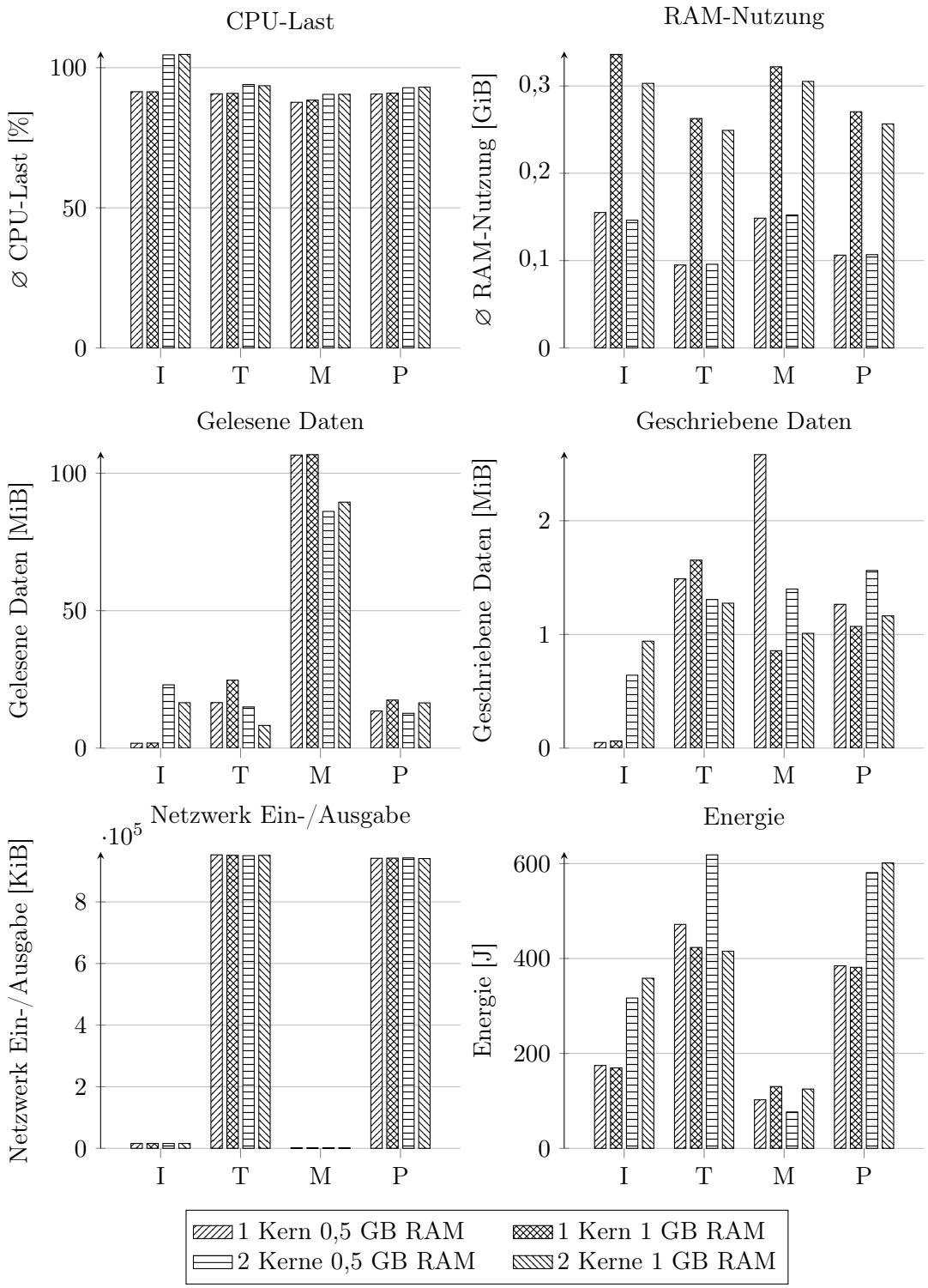


Abbildung 6.18: System- und Energiemetriken der dritten Anfrage von Szenario A (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

6.5.2 Szenario B

Tabelle 6.3 zeigt die genutzte Festplattenkapazität der DBS:

Tabelle 6.3: Genutzte Festplattenkapazitäten pro DBS

	InfluxDB	TimescaleDB	MongoDB	PostgreSQL
Festplattennutzung	8,50 GB	34,93 GB	8,27 GB	25,11 GB

Anfrage 1 – durchschnittlicher Fahrpreis pro Monat

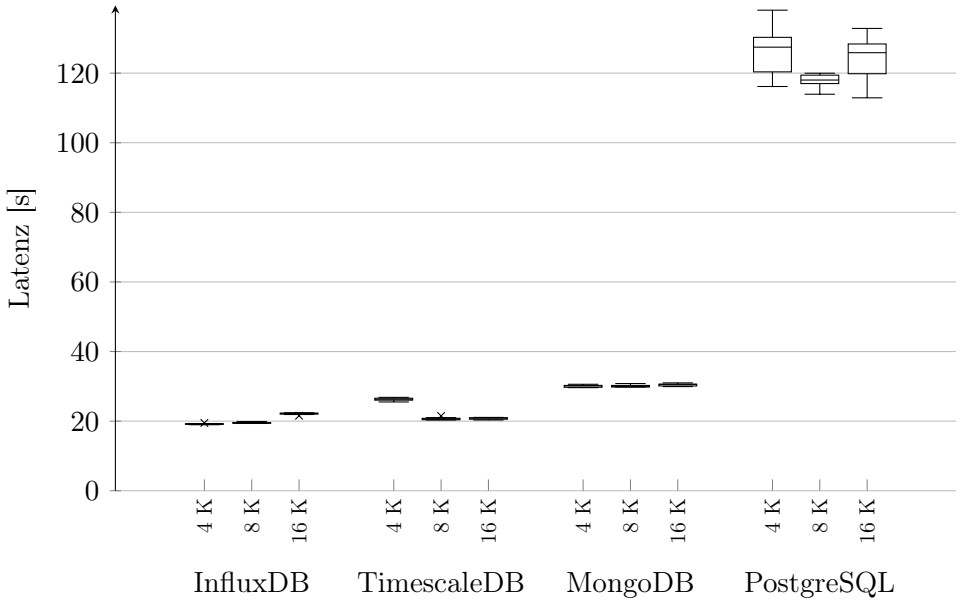


Abbildung 6.19: Latenzen der ersten Anfrage von Szenario B (K ≙ Prozessorkerne)

Abbildung 6.19 zeigt die Latenzen der ersten Anfrage von Szenario B in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *InfluxDB*: ca. 19,1 und 22,2 Sekunden
2. *TimescaleDB*: ca. 20,6 bis 26,3 Sekunden
3. *MongoDB*: ca. 29,9 bis 30,6 Sekunden
4. *PostgreSQL*: ca. 118,0 bis 127,5 Sekunden

Abbildung 6.20 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

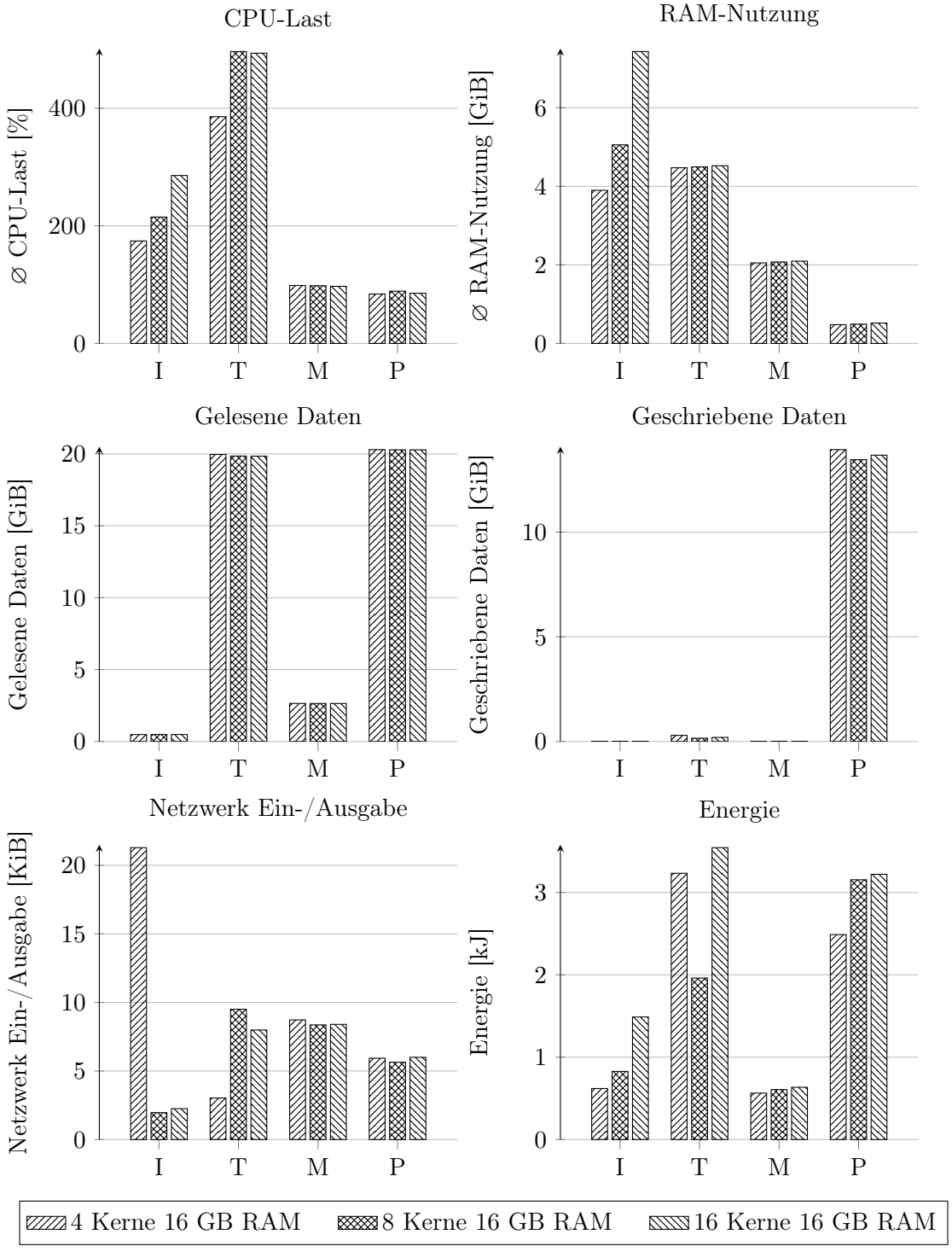


Abbildung 6.20: System- und Energiemetriken der ersten Anfrage von Szenario B (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 2 – Abfrage aller Fahrgastzahlen der ersten Jahreshälfte von 2023

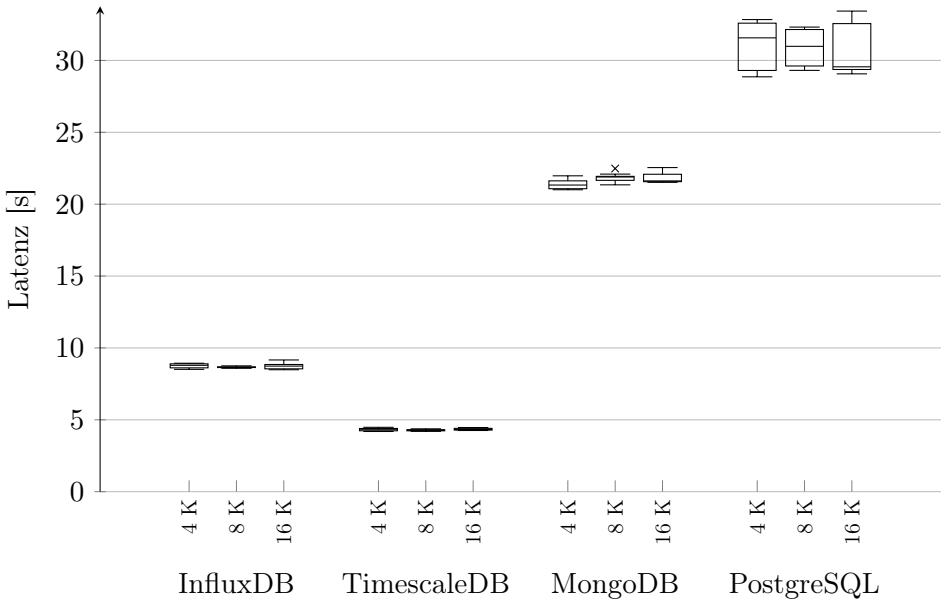


Abbildung 6.21: Latenzen der zweiten Anfrage von Szenario B (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.21 zeigt die Latenzen der zweiten Anfrage von Szenario B in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 4,3 Sekunden
2. *InfluxDB*: ca. 8,6 bis 8,8 Sekunden
3. *MongoDB*: ca. 21,3 bis 21,9 Sekunden
4. *PostgreSQL*: ca. 29,6 bis 31,6 Sekunden

Abbildung 6.22 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

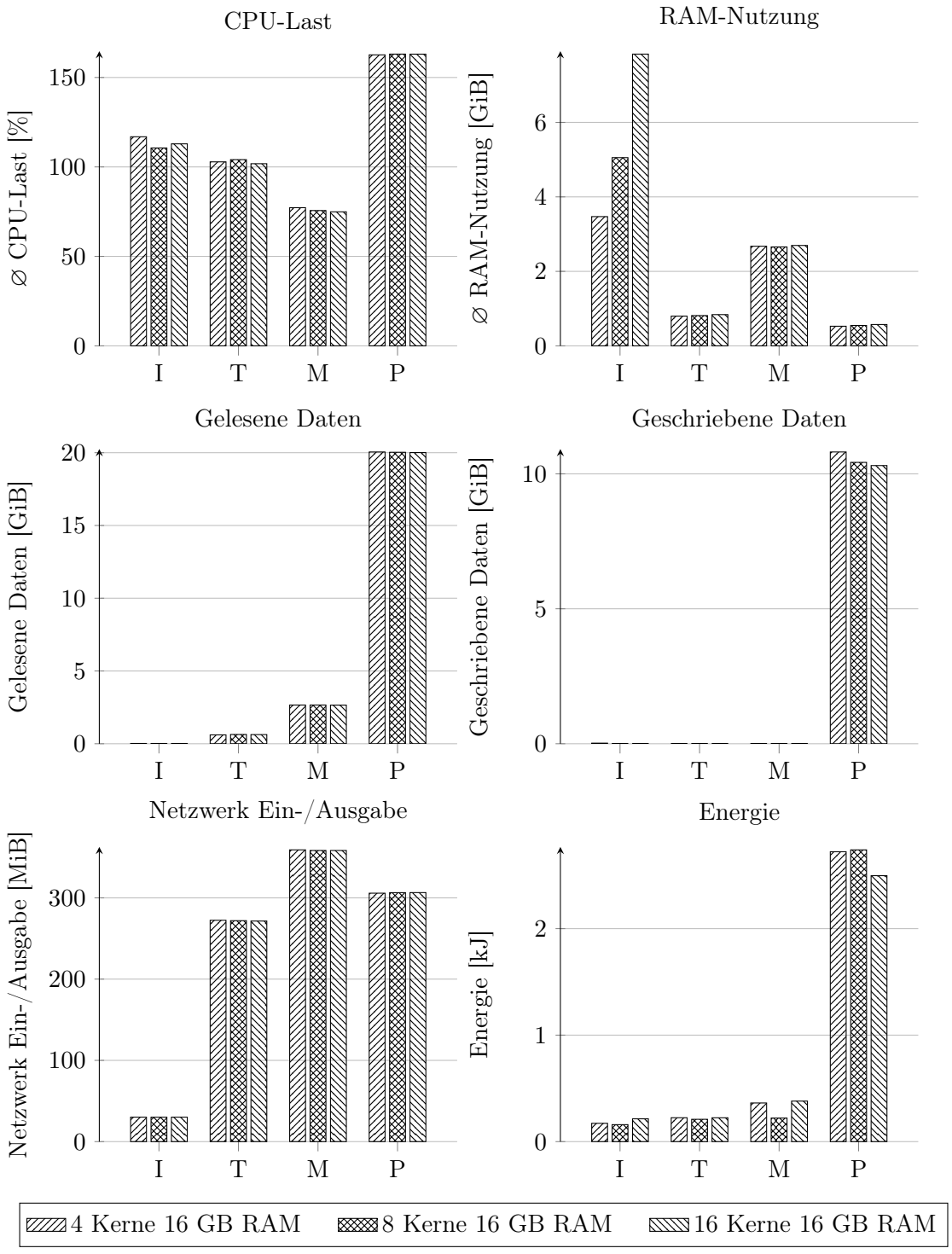


Abbildung 6.22: System- und Energiemetriken der zweiten Anfrage von Szenario B (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 3 – Abfrage aller Fahrten, deren Strecke größer als x ist

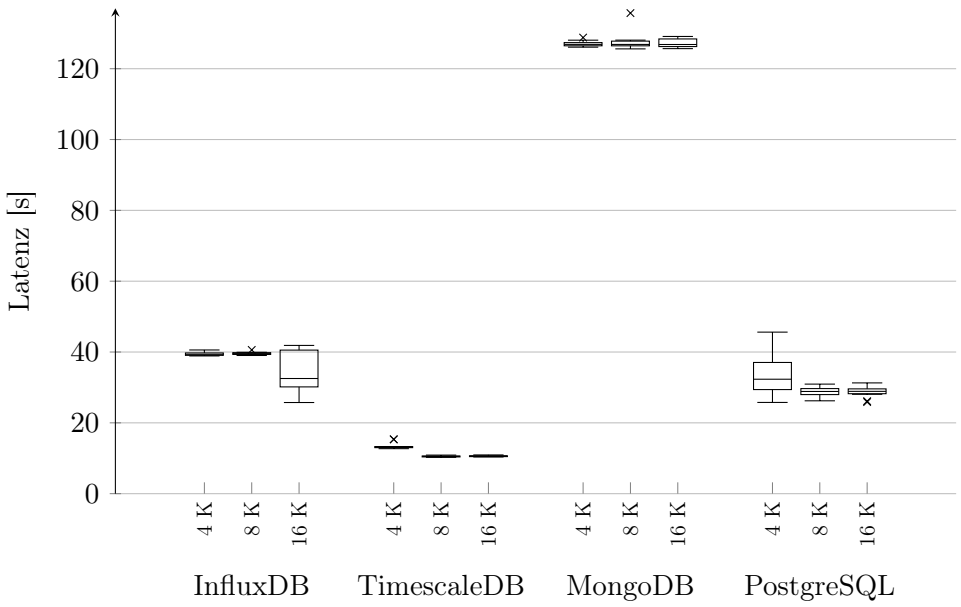


Abbildung 6.23: Latenzen der dritten Anfrage von Szenario B (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.23 zeigt die Latenzen der dritten Anfrage von Szenario B in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 10,6 bis 13,1 Sekunden
2. *PostgreSQL*: ca. 28,9 bis 32,3 Sekunden
3. *InfluxDB*: ca. 32,6 bis 39,6 Sekunden
4. *MongoDB*: ca. 126,9 Sekunden

Abbildung 6.24 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

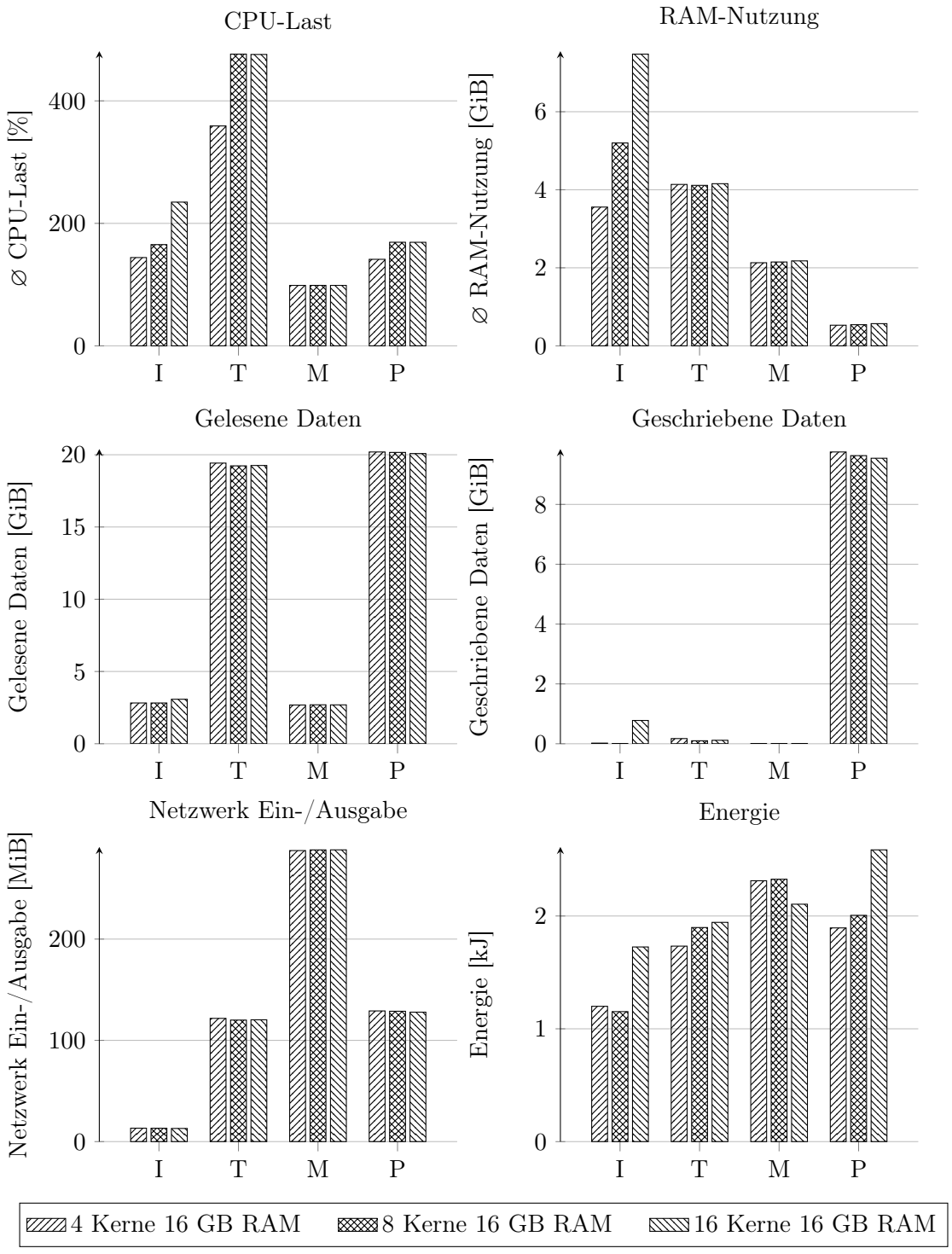


Abbildung 6.24: System- und Energiemetriken der dritten Anfrage von Szenario B (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 4 – Stunde mit den meisten Fahrten

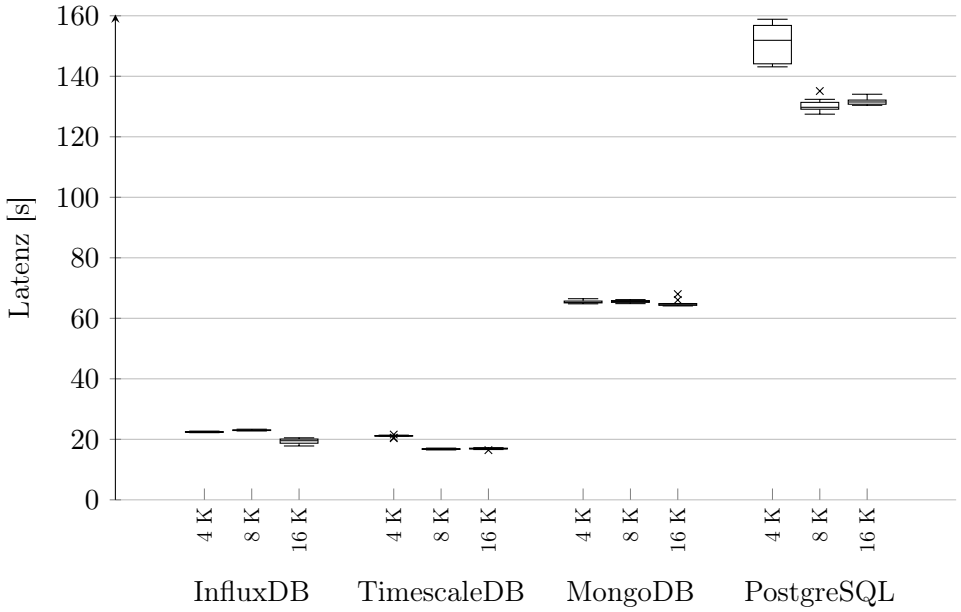


Abbildung 6.25: Latenzen der vierten Anfrage von Szenario B (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.25 zeigt die Latenzen der vierten Anfrage von Szenario B in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 16,8 und 21,1 Sekunden
2. *InfluxDB*: ca. 19,5 bis 23,0 Sekunden
3. *MongoDB*: ca. 64,7 bis 65,5 Sekunden
4. *PostgreSQL*: ca. 129,7 bis 151,9 Sekunden

Abbildung 6.26 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

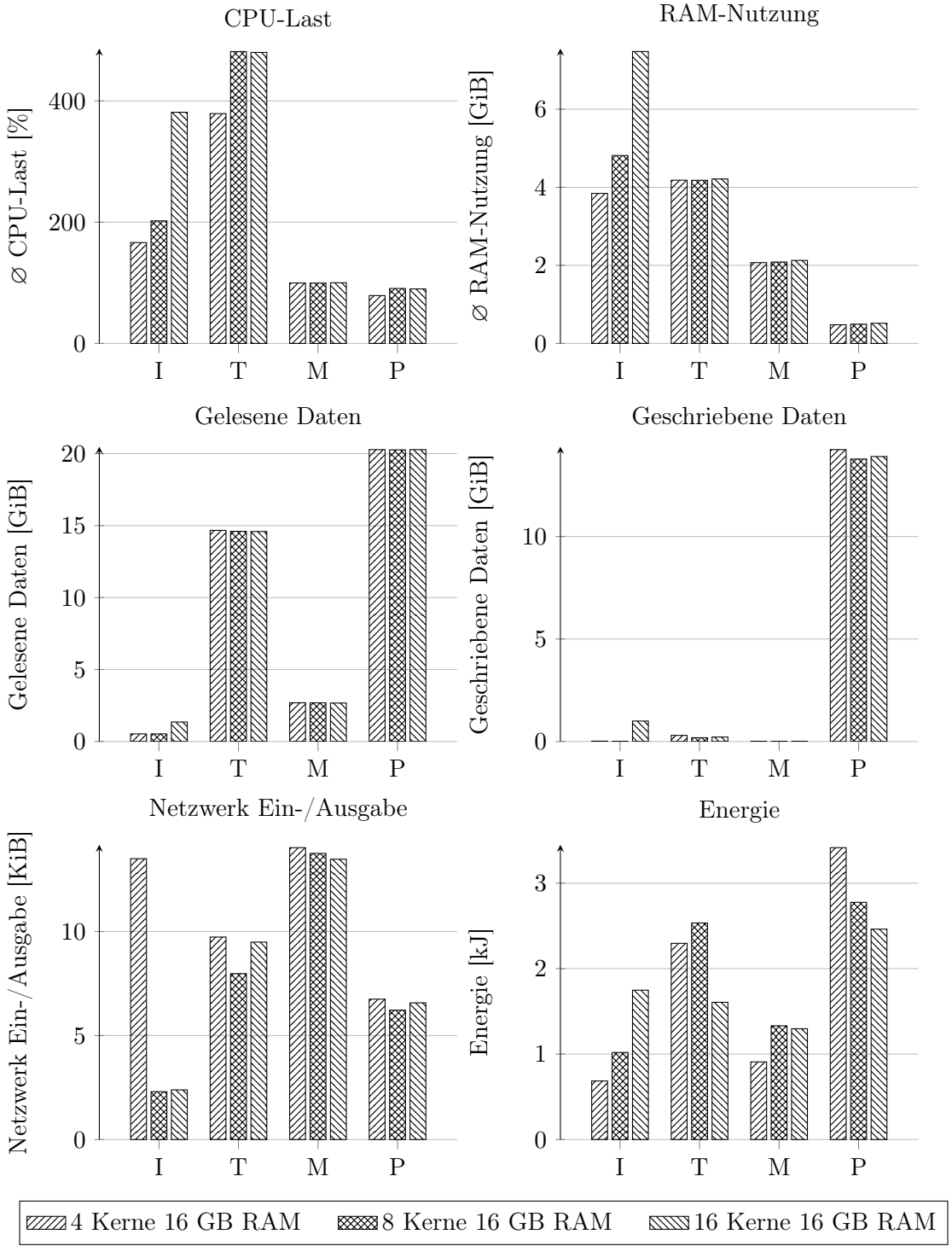


Abbildung 6.26: System- und Energiemetriken der vierten Anfrage von Szenario B (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

6.5.3 Szenario C

Tabelle 6.4 zeigt die genutzte Festplattenkapazität der DBS:

Tabelle 6.4: Genutzte Festplattenkapazitäten pro DBS

	InfluxDB	TimescaleDB	MongoDB	PostgreSQL
Festplattennutzung	6,34 GB	15,94 GB	7,20 GB	14,19 GB

Anfrage 1 – kombinierte Prozessorlast pro Minute der zu dem jeweiligen Zeitpunkt laufenden Instanzen

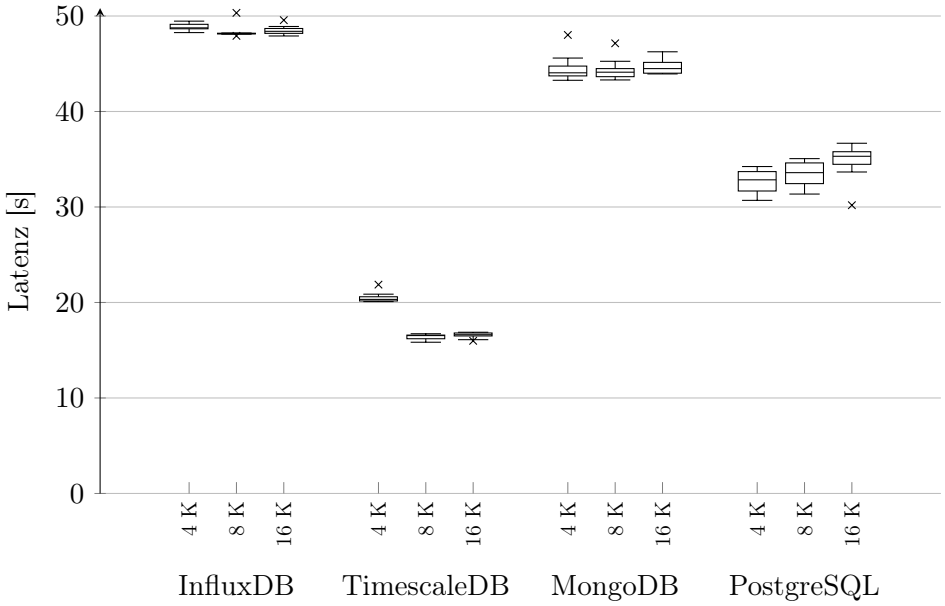


Abbildung 6.27: Latenzen der ersten Anfrage von Szenario C (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.27 zeigt die Latenzen der ersten Anfrage von Szenario C in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 16,5 bis 20,3 Sekunden
2. *PostgreSQL*: ca. 32,8 bis 35,3 Sekunden
3. *MongoDB*: ca. 44,1 bis 44,5 Sekunden
4. *InfluxDB*: ca. 48,2 bis 48,8 Sekunden

Abbildung 6.28 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

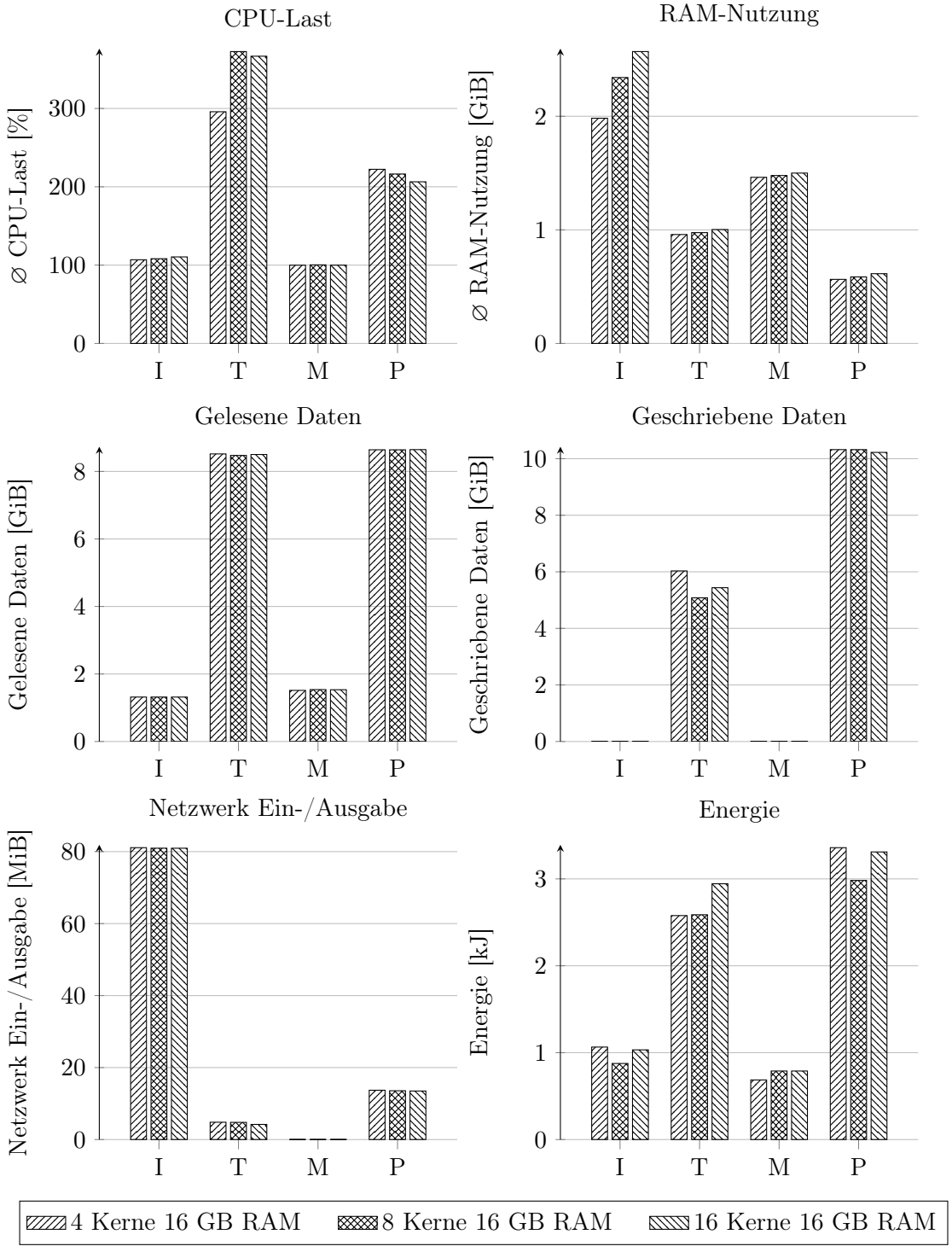


Abbildung 6.28: System- und Energiemetriken der ersten Anfrage von Szenario C (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 2 – Abfrage aller Instanzmetriken in einer bestimmten Stunde

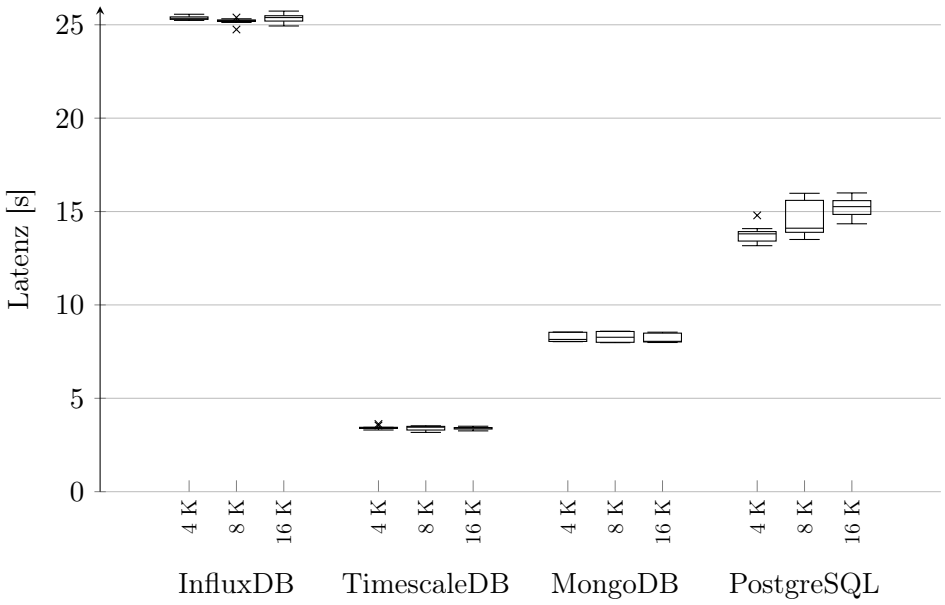


Abbildung 6.29: Latenzen der zweiten Anfrage von Szenario C (K $\widehat{=}$ Prozessorkerne)

Abbildung 6.29 zeigt die Latenzen der zweiten Anfrage von Szenario C in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 3,4 Sekunden
2. *MongoDB*: ca. 8,1 bis 8,3 Sekunden
3. *PostgreSQL*: ca. 13,8 bis 15,3 Sekunden
4. *InfluxDB*: ca. 25,2 bis 25,4 Sekunden

Abbildung 6.30 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

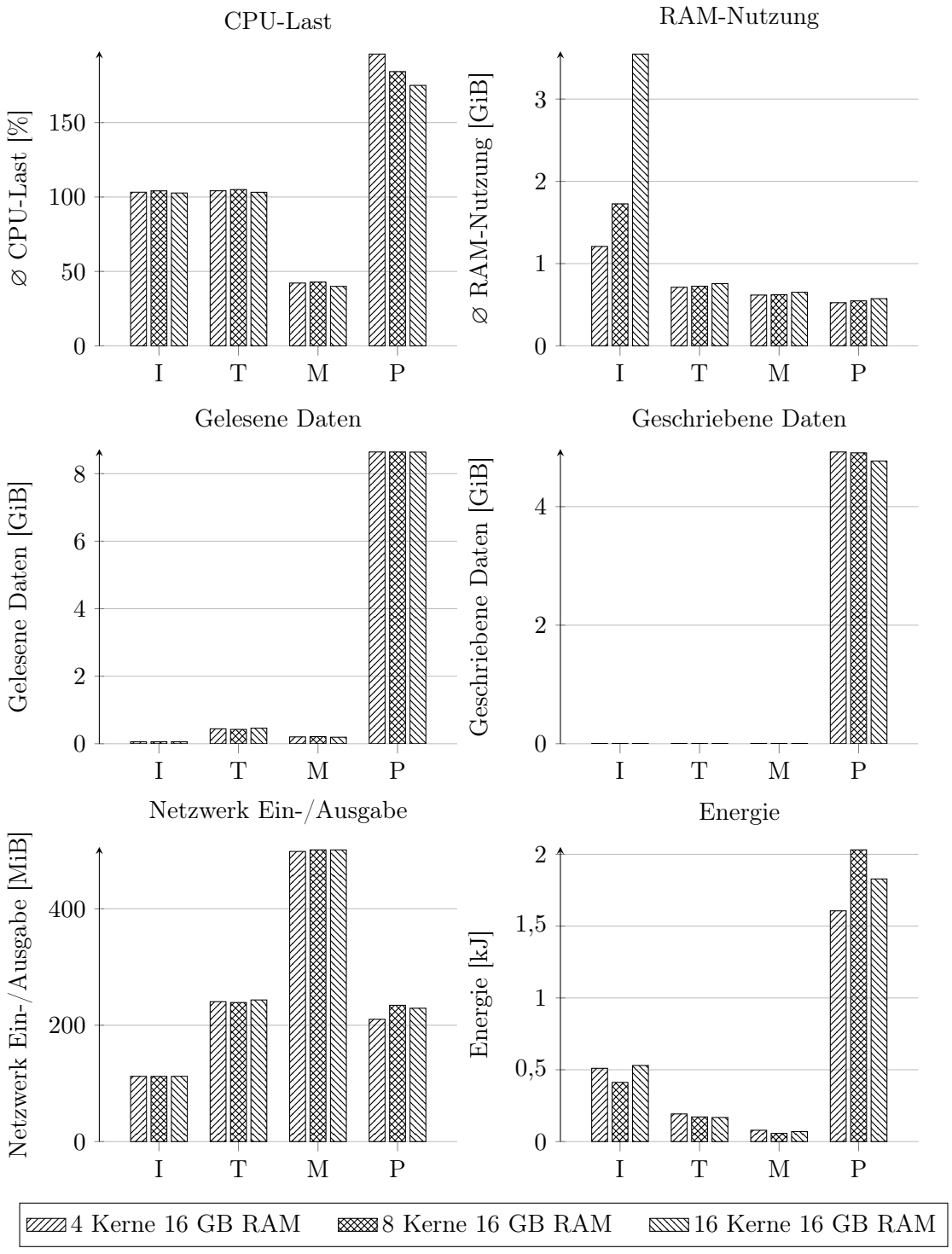


Abbildung 6.30: System- und Energiemetriken der zweiten Anfrage von Szenario C (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

Anfrage 3 – Abfrage der Instanzen, bei denen Prozessorauslastung größer x und Arbeitsspeicherauslastung größer y sind

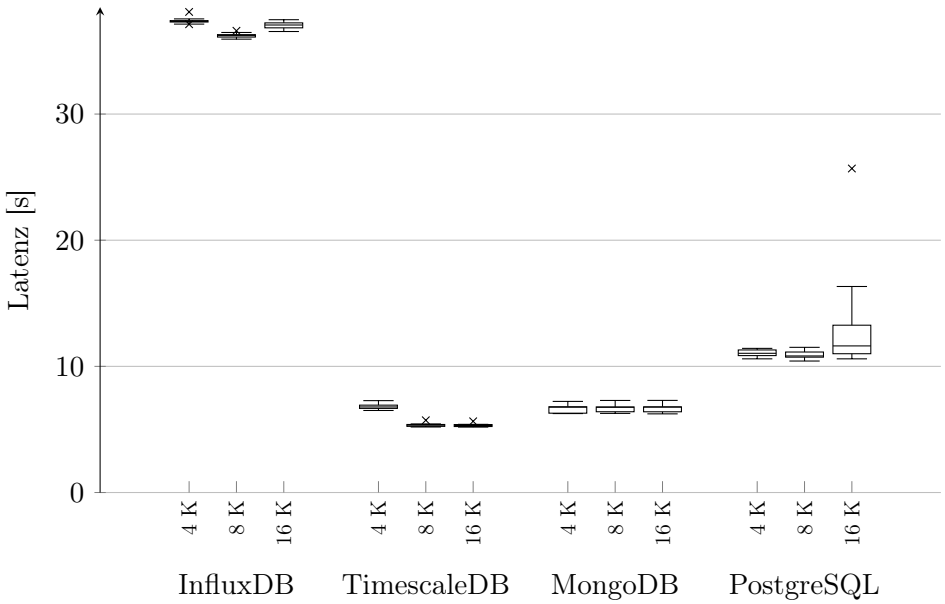


Abbildung 6.31: Latenzen der dritten Anfrage von Szenario C (K ≙ Prozessorkerne)

Abbildung 6.31 zeigt die Latenzen der dritten Anfrage von Szenario C in Abhängigkeit vom jeweiligen DBS und der entsprechenden Systemkonfiguration. Die aufsteigend sortierten Medianwerte der Latenzen sind dabei wie folgt:

1. *TimescaleDB*: ca. 5,3 bis 6,8 Sekunden
2. *MongoDB*: ca. 6,8 Sekunden
3. *PostgreSQL*: ca. 10,8 bis 11,6 Sekunden
4. *InfluxDB*: ca. 36,2 bis 37,3 Sekunden

Abbildung 6.32 stellt wie eingangs beschrieben (siehe Abschnitt 6.5) die System- und Energiemetriken der Testumgebung dar.

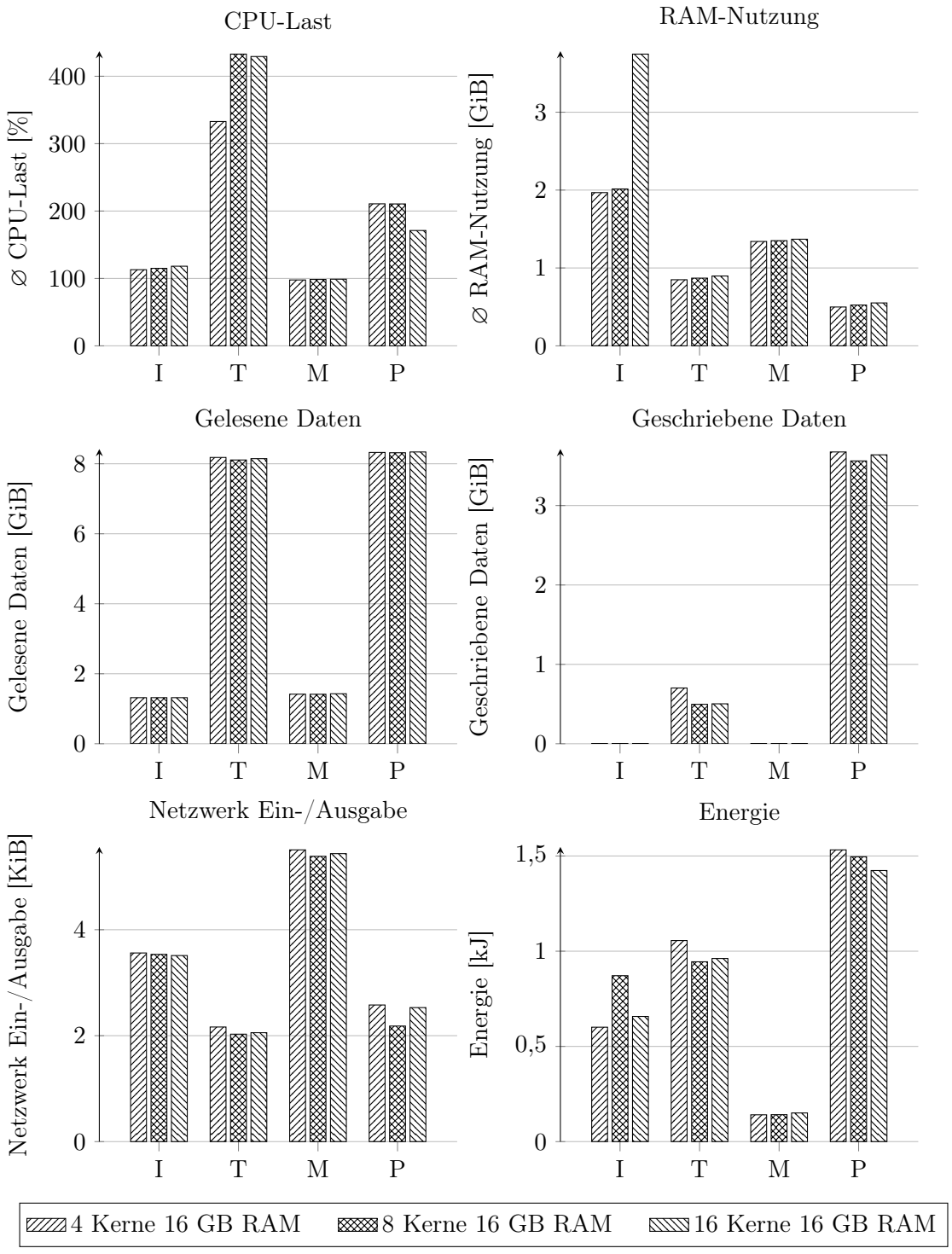


Abbildung 6.32: System- und Energiemetriken der dritten Anfrage von Szenario C (I ≙ InfluxDB, T ≙ TimescaleDB, M ≙ MongoDB, P ≙ PostgreSQL)

7 Diskussion und Empfehlungen

Dieses Kapitel geht zunächst auf Beobachtungen ein, die während der Ausführung der Experimente getätigt wurden, und diskutiert danach die Ergebnisse des konzeptionellen (Kapitel 5) und des experimentellen Vergleichs (Kapitel 6). Im Anschluss werden aus den Ergebnissen dieser Arbeit Nutzungsempfehlungen abgeleitet.

7.1 Beobachtungen während der Experimente

Es wurde beobachtet, dass InfluxDB häufig so viel Arbeitsspeicher genutzt hat, dass das Betriebssystem der Testumgebung das DBS aufgrund von Speicherknappheit beendet hat. Dies geschah reproduzierbar bei der Initialisierung der Szenarios, bei der VMs mit 16 GB Arbeitsspeicher verwendet wurden, und teilweise bei der Durchführung der Anfragen. Die Lösung für dieses Problem, die sich in der Dokumentation von Version 1 von InfluxDB [30] befindet, liegt darin, die Konfigurationsoption `storage-cache-snapshot-write-cold-duration` auf 2 Sekunden zu setzten, wodurch kürzlich erstellte Shards (siehe Abschnitt 3.2.3) nach dieser Zeit auf die Festplatte geschrieben werden und somit wieder Arbeitsspeicher freigegeben wird.

Des Weiteren ist aufgefallen, dass PostgreSQL während der Ausführung der Anfragen von Szenario B und C große Datenmengen geschrieben hat (siehe entsprechende Diagramme in Kapitel 6.5.2 und 6.5.3), obwohl diese Operationen nur lesende Zugriffe durchgeführt haben. Bei den anderen DBS war ein solches Verhalten hingegen nicht zu beobachten.

7.2 Diskussion der Ergebnisse

Nachfolgend werden die Ergebnisse der Vergleiche diskutiert. Dabei ist zu beachten, dass es nur schwer möglich ist, die DBS szenarioübergreifend zu bewerten, da sich diese in

den Szenarios unterschiedlich verhalten haben und somit jeweils andere Schlüsse gezogen werden müssen. Generell kann jedoch festgestellt werden, dass die Anzahl der Prozessoren nur geringe Auswirkungen auf die Latenzen der Anfragen haben.

7.2.1 Szenario A

Anhand der Diagramme der Einfügelatenzen (siehe Abb. 6.12) lässt sich erkennen, dass die Werte InfluxDB und MongoDB nahezu linear mit der Anzahl der Generatoren ansteigen. Zudem ist hier zu sehen, dass mehr Prozessorkerne und/oder mehr Arbeitsspeicher jeweils zu geringeren Latenzen führt. Bei TimescaleDB und PostgreSQL kann dieses Verhalten nicht beobachtet werden. Hier lässt sich jedoch feststellen, dass bei steigenden Größe des Arbeitsspeichers eine größere Anzahl von Verbindungen möglich ist. Bei der genutzten Festplattenkapazität sind im Gegensatz dazu nur vernachlässigbare Unterschiede zwischen den DBS zu erkennen (siehe Tabelle 6.2).

Bei den Anfragen aus Szenario A wird deutlich, dass für einfache Abfragen wie Zähloperationen (Anfrage 1) nicht zwingend ein Zeitreihen-Datenbanksystem erforderlich ist. PostgreSQL als reguläres Datenbanksystem weist hier nur geringfügig höhere Latenzen sowie minimale Unterschiede bei System- und Energiemetriken im Vergleich zum besten Zeitreihen-Datenbanksystem TimescaleDB auf. Je komplexer die Anfragen zu Zeitreihenoperationen werden, desto sinnvoller ist der Einsatz eines Datenbanksystems mit expliziter Zeitreihenunterstützung. Dies zeigt sich deutlich bei Anfrage 3 (Aggregationsoperation über Zeitfenster), wo die Zeitreihen-Datenbanksysteme MongoDB und InfluxDB erheblich geringere Latenzen als PostgreSQL aufweisen und bei vergleichbarer Prozessor- und Arbeitsspeicherauslastung weniger Energie verbrauchen. Anfrage 2 (Aggregation über alle Datenpunkte einer Zeitreihenkategorie) stellt eine Zwischenstufe zwischen den Anfragen 1 und 3 dar.

Wie bereits in Kapitel 4.1 erwähnt, sind eine einfache Installation und Bedienung der DBS für den Heimanwendungskontext besonders wichtig. Daher wurde bei der Einrichtung der Testumgebung besonderes Augenmerk auf diese Aspekte gelegt. Die Installation aller DBS verlief problemlos gemäß den jeweiligen Anleitungen. Lediglich bei TimescaleDB und PostgreSQL musste die maximale Anzahl an Verbindungen durch die Konfigurationsoption `max_connections` erhöht werden.

Die Anfragesprachen der DBS unterscheiden sich hingegen deutlich, wie aus Kapitel 5.2 und 6.3.5 ersichtlich wird. Nutzer mit SQL-Erfahrung können TimescaleDB problemlos

verwenden, da sich lediglich die Funktionen für Zeitreihenoperationen vom SQL-Standard unterscheiden. InfluxQL weist zwar Ähnlichkeiten zu SQL auf, unterscheidet sich jedoch leicht in der Syntax und bietet einen stark eingeschränkten Funktionsumfang, sodass vorhandene SQL-Kenntnisse nur begrenzt hilfreich sind. Die Anfragesprachen Flux und MQL hingegen weichen grundsätzlich von SQL ab, lassen sich aber für lernwillige Anwender dennoch leicht erlernen.

7.2.2 Szenario B

In Szenario B wird deutlich, dass sich die Effizienz bei der Datenspeicherung in den DBS stark unterscheidet. MongoDB und InfluxDB nutzen dabei etwa gleich viel Festplattenkapazität. Im Gegensatz dazu benötigen TimescaleDB und PostgreSQL deutlich mehr Speicherplatz. Während PostgreSQL etwa das Dreifache der Kapazität von InfluxDB und MongoDB beansprucht, liegt der Speicherverbrauch von TimescaleDB beim Vierfachen (siehe Tabelle 6.3).

Anhand der Anfragen lässt sich erkennen, dass Zeitreihen-Datenbanksysteme stets bessere Latenzen erzielen als das Nicht-Zeitreihen-Datenbanksystem PostgreSQL – mit Ausnahme von Anfrage 3 (Filteroperation), bei der PostgreSQL besser abschneidet als InfluxDB und MongoDB. TimescaleDB ist jedoch weiterhin etwa dreimal schneller, verursacht aber doppelt so viel Prozessorauslastung. Hinsichtlich des Energieverbrauchs lässt sich bei Anfrage 3 kein signifikanter Unterschied zwischen den Datenbanksystemen feststellen. Bei Anfrage 1 (Aggregationsoperation über Zeitfenster) und Anfrage 4 (komplexe Operationen, d. h. Kombinationen verschiedener Grundoperationen) zeigen InfluxDB und TimescaleDB in Bezug auf die Latenzen sehr ähnliche Ergebnisse. Allerdings hat InfluxDB bei Systemkonfigurationen mit acht oder mehr Prozessorkernen mehr Arbeitsspeicher genutzt als TimescaleDB. Im Gegensatz dazu führt TimescaleDB zu einer doppelt so hohen Prozessorauslastung und einem höheren Energieverbrauch. Bei Anfrage 2 schneidet TimescaleDB deutlich besser ab als die anderen Datenbanksysteme, da es die niedrigsten Latenzen aufweist. Zudem verursacht es eine geringere Prozessorauslastung und verbraucht weniger Arbeitsspeicher als InfluxDB, das die zweitbesten Latenzen hat. Wie bei Anfrage 3 ist auch hier kein wesentlicher Unterschied im Energieverbrauch zu beobachten.

Zudem wurde festgestellt, dass InfluxDB mit einer steigenden Anzahl an Prozessorkernen sowohl deren Auslastung als auch die Arbeitsspeicherauslastung erhöht, ohne dass sich

die Latenzen signifikant verbessern. Darüber hinaus zeigt sich, dass PostgreSQL stets die geringste Menge an Arbeitsspeicher nutzt.

7.2.3 Szenario C

Ähnlich wie bei Szenario B gibt es auch hier einen Unterschied in der Speichereffizienz der DBS: So nutzen TimescaleDB und PostgreSQL etwa doppelt so viel Speicherplatz wie InfluxDB und MongoDB (siehe Tabelle 6.4).

In den Anfragen von Szenario C zeigt TimescaleDB durchgehend die niedrigsten Latenzen, während InfluxDB die höchsten aufweist und somit stets schlechter als das Baseline-DBS PostgreSQL ist. Dies deutet darauf hin, dass InfluxDB weniger gut für Anwendungen mit einer großen Anzahl von Zeitreihen geeignet ist – obwohl laut Hersteller durch die Einführung des Time Series Index (siehe Kapitel 3.2.4) bereits Verbesserungen in diesem Bereich vorgenommen wurden. MongoDB zeigt bei den Anfragen unterschiedliches Verhalten: Bei Anfrage 1 (Aggregationsoperation über Zeitfenster) ähnelt es InfluxDB hinsichtlich Latenzen sowie System- und Energie-Metriken. Bei Anfrage 2 (Abfrage aller Daten) und Anfrage 3 (Filteroperation) verhält es sich hingegen eher wie TimescaleDB, jedoch mit einer geringeren Prozessorauslastung und einem niedrigeren Energieverbrauch. Wie auch bei Szenario B nutzt PostgreSQL stets am wenigsten Arbeitsspeicher.

Das oben bereits erwähnte Verhalten von InfluxDB, bei dem die Anzahl der Prozessorkerne Auswirkungen auf die System- und Energienutzung, jedoch nur geringe Auswirkungen auf die Latenzen hat, lässt sich hier ebenfalls beobachten, wobei sich in diesem Fall ausschließlich die Nutzung des Arbeitsspeichers vergrößert.

7.3 Empfehlungen

Aus den Ergebnissen der Vergleiche werden im Folgenden spezifische Nutzungsempfehlungen für die in dieser Arbeit verglichenen Datenbanksysteme abgeleitet. Es ist jedoch zu beachten, dass keine allgemeingültige Empfehlung für ein bestimmtes DBS ausgesprochen werden kann, da die Wahl des optimalen Systems maßgeblich von den individuellen Anforderungen und dem jeweiligen Anwendungsfall abhängt. Der experimentelle Vergleich hat allerdings gezeigt, dass es durchaus sinnvoll ist, ein Datenbanksystem mit expliziter Zeitreihenunterstützung zu wählen, wenn Zeitreihen gespeichert und verarbeitet werden

sollen. Lediglich bei simplen Operationen auf einem kleinen Bestand von Zeitreihendaten kann es vorteilhaft sein, auf ein klassisches DBS zurückzugreifen, da dieses in der Regel weniger Systemressourcen beansprucht. Die nachfolgenden Empfehlungen berücksichtigen daher relevante Faktoren, die eine fundierte und individuelle Entscheidungsfindung unterstützen:

- *Anzahl der Zeitreihen*: Wenn eine große Anzahl von Zeitreihen gespeichert und verarbeitet werden soll, liefert TimescaleDB hier konsistent gute Ergebnisse, wobei MongoDB in den häufig ebenfalls eine solide Leistung zeigt. Die Verwendung von InfluxDB hier hingegen nicht empfehlenswert.

- *Speichereffizienz*: Für Anwendungen, bei denen Speicherplatz eine entscheidende Rolle spielt, sind InfluxDB und MongoDB die besseren Optionen, da sie deutlich weniger Speicherplatz benötigen als TimescaleDB und PostgreSQL. Wenn jedoch nur eine geringe Menge an Daten gespeichert werden muss, ist der Unterschied in der Speichereffizienz weniger relevant.

- *Komplexität der Anfragen*: Die Experimente haben gezeigt, dass mit zunehmender Komplexität der Zeitreihenoperationen InfluxDB und TimescaleDB die besten Ergebnisse erzielen. MongoDB schnitt in diesen Fällen weniger gut ab und PostgreSQL zeigte häufig die schwächste Performance.

- *Zeitreihenanalysen*: Falls Zeitreihenanalysen bereits im DBS ausgeführt werden sollen, bietet InfluxDB den größten Funktionsumfang mit der Sprache Flux an. Jedoch muss hier berücksichtigt werden, dass die Entwicklung dieser Sprache mit der kommenden Version 3 von InfluxDB eingestellt wird. Auch TimescaleDB bietet spezielle Funktionen zur Verarbeitung von Zeitreihen an. Insbesondere der Funktionsumfang kann hier durch das Timescale Toolkit erweitert werden.

- *Energieverbrauch*: Für Anwendungen, bei denen der Energieverbrauch eine entscheidende Rolle spielt, zeigt sich, dass MongoDB tendenziell den geringsten Energieverbrauch aufweist. Allerdings muss berücksichtigt werden, dass MongoDB häufig deutlich längere Verarbeitungszeiten für Anfragen benötigt. Bei InfluxDB und TimescaleDB variiert der Energieverbrauch hingegen stark je nach Art der Anfrage. Daher ist es bei der Wahl dieser DBS ratsam, individuelle Tests durchzuführen.

- *Integration in bereits bestehende DBS*: Wenn bereits PostgreSQL im Einsatz ist und im selben DBS auch Zeitreihendaten gespeichert werden sollen, bietet sich an,

die Erweiterung TimescaleDB zu nutzen. Diese ermöglicht zudem die Durchführung von Abfragen über Zeitreihen- und Nicht-Zeitreihendaten innerhalb desselben Systems, ohne dass der Administrationsaufwand erheblich steigt. Allerdings muss beachtet werden, dass die Systemanforderungen durch die Verwendung von TimescaleDB zunehmen. Alternativ kann auch ein existierendes MongoDB mit seiner integrierten Zeitreihenfunktionalität genutzt werden. Die Performance ist hier im Vergleich zu anderen Zeitreihen-Datenbanksystemen teilweise jedoch deutlich schlechter.

- *Hardwarespezifikationen*: Für Anwendungen mit geringen Anforderungen, wie etwa in einem Smart-Home-Umfeld, sind Systeme mit ein bis zwei Prozessorkernen und 1 GB Arbeitsspeicher in der Regel ausreichend. Für größere Installationen sollten nicht weniger als vier Prozessorkerne genutzt werden, wobei acht Kerne empfohlen werden. Der Arbeitsspeicher sollte mindestens 8 GB betragen – allerdings können größere Kapazitäten insbesondere durch Cachingmechanismen von Vorteil sein.

Generell sollten vor der endgültigen Wahl eines Systems immer Tests der auszuführenden Abfragen durchgeführt werden. Dies wird insbesondere durch die teils stark abweichenden Ergebnisse der Experimente verdeutlicht.

8 Fazit

Dieses Kapitel fasst den Inhalt dieser Arbeit zusammen und gibt einen Ausblick auf zukünftige Forschungen.

8.1 Zusammenfassung

In dieser Arbeit wurden vier Datenbanksysteme sowohl konzeptionell als auch experimentell miteinander verglichen – darunter drei Zeitreihen-Datenbanksysteme (InfluxDB, TimescaleDB und MongoDB) und ein klassisches relationales DBS (PostgreSQL), das als Baseline diente. Zunächst wurden in Kapitel 2 die grundlegenden Konzepte im Bereich der Zeitreihen und Datenbanksysteme erläutert. Anschließend wurde in Kapitel 3 eine Kategorisierung von Zeitreihen-Datenbanksystemen in drei Gruppen vorgenommen. Darauf basierend wurden repräsentative Systeme für den Test ausgewählt und deren interne Funktionsweise erläutert. Für den experimentellen Vergleich wurden daraufhin drei Szenarios mit jeweils spezifischen Anfragen entwickelt (siehe Kapitel 4), die typische Anwendungsfälle von Zeitreihen-Datenbanksystemen abbilden. In Kapitel 5 wurden die gewählten DBS anschließend anhand ihres Funktionsumfangs konzeptionell gegenübergestellt. Zur Durchführung des experimentellen Vergleichs wurde in Kapitel 6 ein Testsystem entwickelt, das eine automatisierte Testumgebung für die definierten Szenarios einrichtet. Innerhalb dieser Umgebung wurden die entsprechenden Anfragen vollautomatisch ausgeführt, während gleichzeitig Latenzen sowie System- und Energiemetriken (siehe Kapitel 6.1) vom Testsystem erfasst wurden. Zum Schluss wurden in Kapitel 7 die Ergebnisse der Vergleiche diskutiert und davon ausgehend Nutzungsempfehlungen abgeleitet. Dabei wurde festgestellt, dass auf Zeitreihen optimierte Datenbanksysteme in in den meisten Fällen klare Vorteile bieten, jedoch kein einzelnes System universell empfohlen werden kann.

8.2 Ausblick

Die durchgeführten Vergleiche liefern zahlreiche Erkenntnisse über die Leistungsfähigkeit und Eignung der getesteten Zeitreihen-Datenbanksysteme. Dennoch bleiben einige offene Forschungsfragen, die in zukünftigen Arbeiten betrachtet werden können.

Die ausgewählten Szenarien decken bereits große Teile der typischen Einsatzgebiete von Zeitreihen-Datenbanksystemen ab. Dennoch gibt es sicherlich weitere Anwendungsfälle, die bislang nicht berücksichtigt wurden und deren Untersuchung weitere Erkenntnisse liefern könnte. Zukünftige Arbeiten könnten daher das bestehende Testsystem erweitern, indem neue Szenarios entwickelt und getestet werden, um die Leistungsfähigkeit der Systeme unter weiteren realistischen Bedingungen zu evaluieren.

In den durchgeführten Experimenten wurde primär die sequenzielle Verarbeitung einzelner großer Anfragen untersucht. In realen Anwendungsgebieten treten jedoch auch häufig parallele, dafür aber kleinere Anfragen auf, insbesondere in Anwendungen mit hoher Nutzerzahl. Zukünftige Untersuchungen könnten daher die Skalierbarkeit der getesteten Systeme unter Lastbedingungen mit mehreren gleichzeitigen Anfragen analysieren. Dabei wäre es interessant zu untersuchen, wie sich die Latenzen verändern und ob es zu Ressourcenengpässen bei der Parallelverarbeitung kommt.

Literaturverzeichnis

[1] Abadi, Daniel: Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. In: *Computer* 45 (2012), Februar, Nr. 2, S. 37–42. – URL http://ieeexplore.ieee.org/document/6127847/. – ISSN 0018-9162

[2] Alpine Linux Development Team: *index | Alpine Linux*. – URL https://www.alpinelinux.org/. – Zugriffsdatum: 2025-02-17

[3] Bader, Andreas: *Comparison of Time Series Databases*. Januar 2016. – URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3729&engl=0. – Diplomarbeit

[4] Benoit, Petit: *scaphandre*. Februar 2025. – URL https://github.com/hubblo-org/scaphandre/. – Zugriffsdatum: 2025-02-27

[5] Brewer, Eric: *Towards Robust Distributed Systems*. Proceedings of the Annual ACM Symposium on Principles of Distributed Computing. Juli 2000. – URL https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[6] Broadcom: *Desktop Hypervisor Solutions | VMware*. – URL https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion. – Zugriffsdatum: 2025-02-17

[7] Brockwell, Peter J. ; Davis, Richard A.: *Introduction to Time Series and Forecasting*. Cham : Springer International Publishing, 2016 (Springer Texts in Statistics). – URL http://link.springer.com/10.1007/978-3-319-29854-2. – ISBN 978-3-319-29854-2

[8] Canonical: *cloud-init - The standard for customising cloud instances*. – URL https://cloud-init.io/. – Zugriffsdatum: 2025-02-17

[9] Chatfield, C.: *The Analysis of Time Series: Theory and Practice*. Boston, MA : Springer US, 1975. – URL http://link.springer.com/10.1007/978-1-4899-2925-9. – ISBN 978-1-4899-2925-9

[10] ClickHouse, Inc.: *Fast Open-Source OLAP DBMS - ClickHouse*. – URL https://clickhouse.com/. – Zugriffsdatum: 2024-12-17

[11] Codd, E. F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Juni, Nr. 6, S. 377–387. – URL https://dl.acm.org/doi/10.1145/362384.362685. – ISSN 0001-0782, 1557-7317

[12] Codd, E. F.: Relational Completeness of Data Base Sublanguages. In: *Research Report / RJ / IBM / San Jose, California* RJ987 (1972), März. – URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6a048dc38250ffce49c5e6a5040b4c91ca05e83d

[13] Couchbase, Inc.: *Couchbase: Best Free NoSQL Cloud Database Platform*. – URL https://www.couchbase.com/. – Zugriffsdatum: 2024-12-17

[14] Deutsche Börse: *DAX Index | Kurs | Charts | DE0008469008 | Börse Frankfurt*. November 2024. – URL https://www.boerse-frankfurt.de/index/dax/charts. – Zugriffsdatum: 2024-11-09

[15] Dunning, Ted ; Friedman, B. E.: *Time Series Databases: New Ways to Store and Access data*. 1. Sebastopol, CA : O'Reilly Media, Inc, 2014. – OCLC: ocn896860337. – ISBN 978-1-4919-1472-4

[16] Edlich, Stefan ; Friedland, Achim ; Hampel, Jens ; Brauer, Benjamin: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. München : Hanser, 2010. – ISBN 978-3-446-42355-8

[17] Fasel, Daniel ; Meier, Andreas (Hrsg.): *Big Data: Grundlagen, Systeme und Nutzungspotenziale*. Wiesbaden : Springer Fachmedien Wiesbaden, 2016 (Edition HMD). – URL http://link.springer.com/10.1007/978-3-658-11589-0. – ISBN 978-3-658-11589-0

[18] Gessert, Felix ; Wingerath, Wolfram ; Ritter, Norbert: *Schnelles und skalierbares Cloud-Datenmanagement*. Cham : Springer International Publishing, 2024. – URL https://link.springer.com/10.1007/978-3-031-54388-3. – ISBN 978-3-031-54388-3

[19] Gilbert, Seth ; Lynch, Nancy: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *ACM SIGACT News* 33 (2002), Juni, Nr. 2, S. 51–59. – URL https://dl.acm.org/doi/10.1145/564585.564601. – ISSN 0163-5700

[20] Gomez, Guillaume: *sysinfo*. Dezember 2024. – URL https://github.com/GuillaumeGomez/sysinfo. – Zugriffsdatum: 2025-01-08

[21] Graphite Project: *Graphite*. – URL https://graphiteapp.org/. – Zugriffsdatum: 2024-12-17

[22] Haerder, Theo ; Reuter, Andreas: Principles of transaction-oriented database recovery. In: *ACM Computing Surveys* 15 (1983), Dezember, Nr. 4, S. 287–317. – URL https://dl.acm.org/doi/10.1145/289.291. – ISSN 0360-0300, 1557-7341

[23] Hao, Yuanzhe ; Qin, Xiongpai ; Chen, Yueguo ; Li, Yaru ; Sun, Xiaoguang ; Tao, Yu ; Zhang, Xiao ; Du, Xiaoyong: TS-Benchmark: A Benchmark for Time Series Databases. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. Chania, Greece : IEEE, April 2021, S. 588–599. – URL https://ieeexplore.ieee.org/document/9458659/. – ISBN 978-1-7281-9184-3

[24] Hibernating Rhinos: *Life is an Adventure — Your Database Shouldn't Be - RavenDB NoSQL Database*. – URL https://ravendb.net/. – Zugriffsdatum: 2024-12-17

[25] IBM: *IBM Informix*. – URL https://www.ibm.com/products/informix. – Zugriffsdatum: 2024-12-17

[26] InfluxData Inc.: *Complete list of Flux functions | Flux Documentation*. – URL https://docs.influxdata.com/flux/v0/stdlib/all-functions/. – Zugriffsdatum: 2025-03-08

[27] InfluxData Inc.: *Flux language specification | Flux Documentation*. – URL https://docs.influxdata.com/flux/v0/spec/. – Zugriffsdatum: 2025-03-01

[28] InfluxData Inc.: *In-memory indexing and the Time-Structured Merge Tree (TSM) | InfluxDB OSS v1 Documentation*. – URL https://docs.influxdata.com/influxdb/v1/concepts/storage_engine/. – Zugriffsdatum: 2024-11-22

[29] InfluxData Inc.: *InfluxDB design principles | InfluxDB OSS v2 Documentation.* – URL https://docs.influxdata.com/influxdb/v2/reference/key-concepts/design-principles/. – Zugriffsdatum: 2025-03-08

[30] InfluxData Inc.: *InfluxDB frequently asked questions | InfluxDB OSS v1 Documentation.* – URL https://docs.influxdata.com/influxdb/v1/troubleshooting/frequently-asked-questions/#what-are-the-configuration-recommendations-and-schema-guidelines-for-writing-sparse-historical-data. – Zugriffsdatum: 2025-02-26

[31] InfluxData Inc.: *InfluxDB HTTP API | InfluxDB OSS v2 Documentation.* – URL https://docs.influxdata.com/influxdb/v2/reference/api/. – Zugriffsdatum: 2025-03-09

[32] InfluxData Inc.: *Join data in InfluxDB with Flux | InfluxDB OSS v2 Documentation.* – URL https://docs.influxdata.com/influxdb/v2/query-data/flux/join/. – Zugriffsdatum: 2025-03-09

[33] InfluxData Inc.: *Line protocol | InfluxDB OSS v2 Documentation.* – URL https://docs.influxdata.com/influxdb/v2/reference/syntax/line-protocol/. – Zugriffsdatum: 2025-02-28

[34] InfluxData Inc.: *pivot() function | Flux Documentation.* – URL https://docs.influxdata.com/flux/v0/stdlib/universe/pivot/. – Zugriffsdatum: 2025-03-09

[35] InfluxData Inc.: *Query data with InfluxQL | InfluxDB OSS v2 Documentation.* – URL https://docs.influxdata.com/influxdb/v2/query-data/influxql/. – Zugriffsdatum: 2025-03-01

[36] InfluxData Inc.: *Time Series Index (TSI) details | InfluxDB OSS v1 Documentation.* – URL https://docs.influxdata.com/influxdb/v1/concepts/tsi-details/. – Zugriffsdatum: 2024-12-09

[37] InfluxData Inc.: *Time Series Index (TSI) overview | InfluxDB OSS v1 Documentation.* – URL https://docs.influxdata.com/influxdb/v1/concepts/time-series-index/. – Zugriffsdatum: 2024-12-09

[38] InfluxData Inc.: *union() function | Flux Documentation.* – URL https://docs.influxdata.com/flux/v0/stdlib/universe/union/. – Zugriffsdatum: 2025-03-09

[39] InfluxData Inc.: *Flux vs InfluxQL | InfluxDB OSS v2 Documentation*. 2024. – URL https://docs.influxdata.com/influxdb/v2/reference/syntax/flux/flux-vs-influxql/. – Zugriffsdatum: 2024-11-24

[40] InfluxData Inc.: *The future of Flux | Flux Documentation*. 2024. – URL https://docs.influxdata.com/flux/v0/future-of-flux/. – Zugriffsdatum: 2024-11-24

[41] InfluxData Inc.: *InfluxDB data elements | InfluxDB OSS v2 Documentation*. 2024. – URL https://docs.influxdata.com/influxdb/v2/reference/key-concepts/data-elements/. – Zugriffsdatum: 2024-11-24

[42] InfluxData Inc.: *InfluxDB Open Source | InfluxData*. 2024. – URL https://www.influxdata.com/products/influxdb/. – Zugriffsdatum: 2024-11-21

[43] Khelifati, Abdelouahab ; Khayati, Mourad ; Dignös, Anton ; Difallah, Djellel ; Cudré-Mauroux, Philippe: TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications. In: *Proceedings of the VLDB Endowment* 16 (2023), Juli, Nr. 11, S. 3363–3376. – URL https://dl.acm.org/doi/10.14778/3611479.3611532. – ISSN 2150-8097

[44] Knuth, Donald E.: *The art of computer programming*. Bd. 3. 2. Reading (Mass.) London Manila [etc.] : Addison-Wesley publ, 1997. – ISBN 0-201-89685-0

[45] Kreiss, Jens-Peter ; Neuhaus, Georg: *Einführung in die Zeitreihenanalyse*. Berlin/Heidelberg : Springer-Verlag, 2006 (Statistik und ihre Anwendungen). – URL http://link.springer.com/10.1007/3-540-33571-4. – ISBN 978-3-540-25628-1

[46] launchbadge: *SQLx - The Rust SQL Toolkit*. Januar 2025. – URL https://github.com/launchbadge/sqlx. – Zugriffsdatum: 2025-03-05

[47] Meier, Andreas ; Kaufmann, Michael: *SQL- & NoSQL-Datenbanken*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2016 (eXamen.press). – URL http://link.springer.com/10.1007/978-3-662-47664-2. – ISBN 978-3-662-47664-2

[48] Menage, Paul ; Jackson, Paul ; Lameter, Christoph: *Control Groups — The Linux Kernel documentation*. – URL https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html. – Zugriffsdatum: 2025-01-07

[49] Microsoft: *Entdecken Sie die Leistungsfähigkeit der KI mit dem Betriebssystem Windows 11, Computern und Apps | Microsoft.* – URL https://www.microsoft.com/de-de/windows. – Zugriffsdatum: 2025-02-17

[50] MongoDB, Inc.: *About Time Series Data - MongoDB Manual v8.0.* – URL https://www.mongodb.com/docs/manual/core/timeseries/timeseries-bucketing/. – Zugriffsdatum: 2024-12-12

[51] MongoDB, Inc.: *Create and Query a Time Series Collection - MongoDB Manual v8.0.* – URL https://www.mongodb.com/docs/manual/core/timeseries/timeseries-procedures/. – Zugriffsdatum: 2024-12-12

[52] MongoDB, Inc.: *Insert Documents - Rust Driver v3.2.* – URL https://www.mongodb.com/docs/drivers/rust/current/fundamentals/crud/write-operations/insert/. – Zugriffsdatum: 2025-02-28

[53] MongoDB, Inc.: *$lookup (aggregation) - MongoDB Manual v8.0 - MongoDB Docs.* – URL https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/. – Zugriffsdatum: 2025-03-09

[54] MongoDB, Inc.: *MongoDB Wire Protocol - MongoDB Manual v8.0 - MongoDB Docs.* – URL https://www.mongodb.com/docs/manual/reference/mongodb-wire-protocol/. – Zugriffsdatum: 2025-03-09

[55] MongoDB, Inc.: *Read Concern - MongoDB Manual v8.0 - MongoDB Docs.* – URL https://www.mongodb.com/docs/manual/reference/read-concern/. – Zugriffsdatum: 2025-03-09

[56] MongoDB, Inc.: *Set Granularity for Time Series Data - MongoDB Manual v8.0.* – URL https://www.mongodb.com/docs/manual/core/timeseries/timeseries-granularity/. – Zugriffsdatum: 2024-12-13

[57] MongoDB, Inc.: *Time Series - MongoDB Manual v8.0.* – URL https://www.mongodb.com/docs/manual/core/timeseries-collections/. – Zugriffsdatum: 2024-12-12

[58] MongoDB, Inc.: *Time Series Collections Considerations - MongoDB Manual v8.0.* – URL https://www.mongodb.com/docs/manual/core/timeseries/timeseries-considerations/. – Zugriffsdatum: 2024-12-12

[59] MongoDB, Inc.: *Transactions - MongoDB Manual v8.0 - MongoDB Docs.* – URL https://www.mongodb.com/docs/manual/core/transactions/. – Zugriffsdatum: 2025-03-09

[60] MongoDB, Inc.: *Write Concern - MongoDB Manual v8.0 - MongoDB Docs.* – URL https://www.mongodb.com/docs/manual/reference/write-concern/. – Zugriffsdatum: 2025-03-09

[61] MongoDB, Inc.: *MongoDB: Die Datenplattform für Entwickler | MongoDB.* 2024. – URL https://www.mongodb.com/de-de. – Zugriffsdatum: 2024-11-16

[62] MongoDB, Inc.: *Read Concern - MongoDB Manual v8.0.* 2024. – URL https://www.mongodb.com/docs/manual/reference/read-concern/. – Zugriffsdatum: 2024-11-14

[63] Monroe, Robert: *American Chemical Society Honors Keeling Curve and NOAA Observatory.* April 2015. – URL https://scripps.ucsd.edu/news/american-chemical-society-honors-keeling-curve-and-noaa-observatory. – Zugriffsdatum: 2024-10-07

[64] Murugesan, San: Harnessing Green IT: Principles and Practices. In: *IT Professional* 10 (2008), Nr. 1, S. 24–33. – URL http://ieeexplore.ieee.org/document/4446673/. – ISSN 1520-9202

[65] Neo4j, Inc.: *Neo4j Graph Database & Analytics | Graph Database Management System.* 2024. – URL https://neo4j.com/. – Zugriffsdatum: 2024-11-16

[66] NYC Taxi & Limousine Commission: *TLC Trip Record Data - TLC.* – URL https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. – Zugriffsdatum: 2025-01-10

[67] Oracle: *Oracle VirtualBox.* – URL https://www.virtualbox.org/. – Zugriffsdatum: 2025-02-17

[68] O'Neil, Patrick ; Cheng, Edward ; Gawlick, Dieter ; O'Neil, Elizabeth: The log-structured merge-tree (LSM-tree). In: *Acta Informatica* 33 (1996), Juni, Nr. 4, S. 351–385. – URL http://link.springer.com/10.1007/s002360050048. – ISSN 0001-5903, 1432-0525

[69] Praschl, Christoph ; Pritz, Sebastian ; Krauss, Oliver ; Harrer, Martin: A Comparison Of Relational, NoSQL and NewSQL Database Management Systems For The Persistence Of Time Series Data. In: *2022 International Conference on*

Electrical, Computer, Communications and Mechatronics Engineering (ICECCME). Maldives, Maldives : IEEE, November 2022, S. 1–6. – URL https://ieeexplore.ieee.org/document/9988333/. – ISBN 978-1-6654-7095-7

[70] Prometheus Authors: *Prometheus - Monitoring system & time series database*. – URL https://prometheus.io/. – Zugriffsdatum: 2024-12-17

[71] Red Gate Software Ltd: *DB-Engines Ranking - die Rangliste der populärsten Time Series DBMS*. November 2024. – URL https://db-engines.com/de/ranking/time+series+dbms/all. – Zugriffsdatum: 2024-11-19

[72] Red Hat: *KVM*. – URL https://linux-kvm.org/page/Main_Page. – Zugriffsdatum: 2025-02-17

[73] Red Hat: *libvirt: The virtualization API*. – URL https://libvirt.org/. – Zugriffsdatum: 2025-02-17

[74] Redis Ltd.: *Redis - The Real-time Data Platform*. – URL https://redis.io/. – Zugriffsdatum: 2024-11-15

[75] Rust Team: *Rust Programming Language*. – URL https://www.rust-lang.org/. – Zugriffsdatum: 2025-02-17

[76] Sadalage, Pramod J. ; Fowler, Martin: *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Upper Saddle River, NJ : Addison-Wesley, 2013. – OCLC: 1065788907. – ISBN 978-0-13-301800-4

[77] Schicker, Edwin: *Datenbanken und SQL*. Wiesbaden : Springer Fachmedien Wiesbaden, 2017 (Informatik & Praxis). – URL http://link.springer.com/10.1007/978-3-658-16129-3. – ISBN 978-3-658-16129-3

[78] Schlittgen, Rainer ; Streitberg, Bernd H. J.: *Zeitreihenanalyse*. 9., unwesentlich veränd. Aufl. München : Oldenbourg, 2001 (Lehr- und Handbücher der Statistik). – ISBN 978-3-486-71096-0

[79] Shumway, Robert H. ; Stoffer, David S.: *Time Series Analysis and Its Applications: With R Examples*. Cham : Springer International Publishing, 2017 (Springer Texts in Statistics). – URL https://link.springer.com/10.1007/978-3-319-52452-8. – ISBN 978-3-319-52452-8

[80] SingleStore, Inc.: *SingleStore | The Real-Time Data Platform for Intelligent Applications.* – URL https://www.singlestore.com/. – Zugriffsdatum: 2024-12-17

[81] Software in the Public Interest, Inc.: *Debian – Das universelle Betriebssystem.* – URL https://www.debian.org/index.de.html. – Zugriffsdatum: 2025-02-17

[82] Studer, Thomas: *Relationale Datenbanken: Von den theoretischen Grundlagen zu Anwendungen mit PostgreSQL.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2019. – URL http://link.springer.com/10.1007/978-3-662-58976-2. – ISBN 978-3-662-58976-2

[83] The Apache Software Foundation: *Apache Cassandra | Apache Cassandra Documentation.* 2024. – URL https://cassandra.apache.org/_/index.html. – Zugriffsdatum: 2024-11-16

[84] The OpenTSDB Authors: *OpenTSDB - A Distributed, Scalable Monitoring System.* – URL https://opentsdb.net/. – Zugriffsdatum: 2024-12-17

[85] The PostgreSQL Global Development Group: *PostgreSQL: Documentation: 17: Chapter 53. Frontend/Backend Protocol.* – URL https://www.postgresql.org/docs/17/protocol.html. – Zugriffsdatum: 2025-03-09

[86] The PostgreSQL Global Development Group: *PostgreSQL: Documentation: 17: COPY.* – URL https://www.postgresql.org/docs/17/sql-copy.html. – Zugriffsdatum: 2025-02-28

[87] The PostgreSQL Global Development Group: *PostgreSQL: Documentation: 17: 13.2. Transaction Isolation.* September 2024. – URL https://www.postgresql.org/docs/17/transaction-iso.html. – Zugriffsdatum: 2024-11-08

[88] The PostgreSQL Global Development Group: *PostgreSQL: The world's most advanced open source database.* 2024. – URL https://www.postgresql.org/

[89] Timescale Inc.: *Timescale Documentation | About compression.* – URL https://docs.timescale.com/use-timescale/latest/compression/about-compression/. – Zugriffsdatum: 2024-12-06

[90] Timescale Inc.: *Timescale Documentation | Designing your database for compression.* – URL https://docs.timescale.com/use-timescale/latest/compression/compression-design/. – Zugriffsdatum: 2024-12-06

[91] Timescale Inc.: *Timescale Documentation | Hyperfunctions.* – URL https://docs.timescale.com/api/latest/hyperfunctions/. – Zugriffsdatum: 2025-03-08

[92] Timescale Inc.: *Understanding ACID Compliance in PostgreSQL | Timescale.* – URL https://www.timescale.com/learn/understanding-acid-compliance. – Zugriffsdatum: 2025-03-09

[93] Timescale Inc.: *Data Model.* Januar 2020. – URL https://github.com/timescale/docs.timescale.com-content/blob/master/introduction/data-model.md. – Zugriffsdatum: 2024-11-24

[94] Timescale Inc.: *Architecture & Concepts.* Februar 2021. – URL https://github.com/timescale/docs.timescale.com-content/blob/master/introduction/architecture.md. – Zugriffsdatum: 2024-11-24

[95] Timescale Inc.: *PostgreSQL ++ for time series and events | Timescale.* 2024. – URL https://www.timescale.com/. – Zugriffsdatum: 2024-11-21

[96] Timescale Inc.: *Timescale Documentation | About hypertables.* 2024. – URL https://docs.timescale.com/use-timescale/latest/hypertables/about-hypertables/. – Zugriffsdatum: 2024-11-24

[97] Winters, Peter R.: Forecasting Sales by Exponentially Weighted Moving Averages. In: *Management Science* 6 (1960), April, Nr. 3, S. 324–342. – URL https://pubsonline.informs.org/doi/10.1287/mnsc.6.3.324. – ISSN 0025-1909, 1526-5501

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
LaTeX	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments
ChatGPT	Sprachmodell (LLM) teilweise genutzt als Formulierungshilfe
IntelliJ IDEA	Programmierumgebung verwendet für die Entwicklung des Testsystems

A.2 Einrichtung des Testsystems

Zur Einrichtung des Testsystems müssen die folgenden Schritte durchgeführt werden:

1. *Abhängigkeiten installieren*:

   Für die Ausführung des Testsystems müssen folgende Pakete installiert werden[1]:

   - `libvirt`
   - `qemu-full`
   - `rust`
   - `python-pandas`

[1] Zur Ausführung des Testsystems wurde in dieser Arbeit Arch Linux genutzt. Auf anderen Distributionen können die Paketnamen abweichen.

- `python-pyarrow`
- `virt-manager` (optional)

Außerdem muss entweder das `kvm_amd` oder `kvm_intel` Kernelmodul entsprechend des genutzten Prozessors geladen werden.

2. *Projekt herunterladen*:

   Die Projektdateien und auch die Rohdaten der Ergebnisse können über Git unter der Adresse `https://github.com/Malex14/time-series-database-system-tester` heruntergeladen werden:
   `git clone https://github.com/Malex14/time-series-database-system-tester.git`

   Alternativ befinden sich die Dateien auch auf der beigefügten CD.

3. *Datensatz für Szenario B herunterladen*:

   Für Szenario B muss der verwendete Datensatz bei [66] im parquet-Format heruntergeladen[2] und mit dem Python-Skript `convert_dataset_scenario_b.py` vorverarbeitet werden. Das Skript nutzt dabei die Dateien im anzugebenden Ordner und schreibt die verarbeiteten Dateien in den Ordner `./dataset/scenario_b`.

4. *Debian Festplattenabbild herunterladen*:

   Von der Debian Internetseite muss im Anschluss das grundlegende Festplattenabbild unter der Adresse `https://cloud.debian.org/images/cloud/bookworm/latest/` heruntergeladen werden. Der Dateiname lautet dabei `debian-12-genericcloud-amd64.qcow2`

   Alternativ befindet sich die für die Experimente verwendete Datei auch auf der beigefügten CD.

5. *Projekt konfigurieren*:

   In den Projektdateien befindet sich die Konfigurationsdatei `vm_config.toml`, in der die Option `temp_dir` auf einen existierenden Ordner gesetzt werden muss – in diesem Ordner werden temporäre Dateien des Testsystems abgelegt. Der Ordner

[2] Für die in dieser Arbeit durchgeführten Experimente wurden die Daten von Januar 2011 bis November 2024 genutzt.

sollte sich dabei auf einem schnellen Speichermedium befinden, da u. a. die Festplattenabbilder der VMs dort abgelegt werden. Zudem muss die Option `base_disk_image` auf den Pfad des zuvor heruntergeladenen Debian-Abbilds zeigen.

6. *Projekt kompilieren*:

   Mit dem folgenden Befehl kann das Projekt kompiliert werden:
   `cargo build --release --workspace`

   Danach wird sich im Ordner `./target/release` die ausführbare Datei `instrumentation_gatherer` befinden. Diese stellt die Metrikerfassung in der Testumgebung dar und muss in den `bin`-Ordner des Projektverzeichnis kopiert werden.

7. *Dateiberechtigungen korrigieren*:

   Für die Energiemessung müssen die folgenden Befehle ausgeführt werden, die die Berechtigungen der entsprechenden Dateien anpassen:
   `sudo chmod +r /sys/class/powercap/intel-rapl:0/energy_uj`
   `sudo chmod +r /sys/class/powercap/intel-rapl:0:0/energy_uj`

8. *Testsystem ausführen*

   Um das Testsystem zu starten, muss folgendes Programm ausgeführt werden:
   `./target/release/database_tester -t [Anzahl von Ausführungen jeder Anfrage] -s [Szenario]`

   Beispiel:
   `./target/release/database_tester -t 10 -s ScenarioB`

   Im Ordner `out` befinden sich im Anschluss die Ergebnis-CSV-Dateien.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original