

Untersuchung von Datenbanksystemen zur Verwaltung von Provenance Graphen im Kontext der IT-Sicherheit

Sven-Ove Hänsel

Suggested citation:

Hänsel, Sven-Ove. 2024. "Untersuchung von Datenbanksystemen zur Verwaltung von Provenance Graphen im Kontext der IT-Sicherheit." Hannover: Hochschule Hannover. <https://doi.org/10.25968/opus-3232>.

Abstract

Im Kontext der IT-Sicherheit werden Provenance Graphen für die Beantwortung sicherheitsrelevanter Fragen wie „Welche Auswirkungen kann diese Sicherheitslücke haben?“ oder „Welcher Prozess hat meine Daten verändert oder gelöscht?“ verwendet. Weiterhin können Provenance Graphen für das Training von Machine Learning Methoden für die Erkennung von Angriffen auf Netzwerke und Betriebssysteme genutzt werden. Aufgrund der hohen Datenraten bei der Erzeugung von Provenance Graphen (mindestens 1,2 MB/s in realistischen Szenarien) ist eine effiziente Datenverwaltung für den praktischen Einsatz dieser Graphen notwendig.

In dieser Arbeit wird untersucht, welche Anforderungen und Herausforderungen für die Verwaltung von Provenance Graphen existieren und welche Vor- und Nachteile sich aus der Verwaltung mit den Datenbankmanagementsystemen (DBMS) PostgreSQL, Neo4j, ONgDB und Memgraph ergeben. Zur Untersuchung wurde ein Versuchsaufbau für reproduzierbare Experimente entwickelt, in dem ein Provenance Graph Datensatz in die untersuchten DBMS geschrieben wird und dabei Hardwaremetriken und Query-Antwortzeiten zur späteren Auswertung erhoben und gespeichert werden.

Die Ergebnisse der Arbeit zeigen, dass die hier betrachteten DBMS nur teilweise die recherchierten Anforderungen einer Datenverwaltung für Provenance Graphen im Kontext der IT-Sicherheit erfüllen können. Es konnte gezeigt werden, dass sich relationale Datenbanken durch sparsameren Ressourcenverbrauch gegenüber Graphdatenbanken auszeichnen, dafür aber Nachteile in den Abfragemöglichkeiten durch die Querysprache aufweisen.

Terms of use

CC BY 4.0

This document is made available under these conditions:
Creative Commons - CC BY - Namensnennung 4.0 International
For more information see:
<https://creativecommons.org/licenses/by/4.0/deed.de>



**HOCHSCHULE
HANNOVER**
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
*Fakultät IV
Wirtschaft und
Informatik*

Untersuchung von Datenbanksystemen zur Verwaltung von Provenance Graphen im Kontext der IT-Sicherheit

Sven-Ove Hänsel

Masterarbeit im Studiengang „Angewandte Informatik“

30. Juli 2024



Autor Sven-Ove Hänsel
Matrikelnr. 1661575
sven.haensel@schijo.de

Erstprüfer Prof. Dr. Carsten Kleiner
Abteilung Informatik, Fakultät IV
Hochschule Hannover
carsten.kleiner@hs-hannover.de

Zweitprüfer Kilian Dangendorf M. Sc.
Abteilung Informatik, Fakultät IV
Hochschule Hannover
kilian.dangendorf@hs-hannover.de

Soweit nicht anders gekennzeichnet, ist dieses Werk unter einem Creative-Commons-Lizenzvertrag Namensnennung 4.0 lizenziert. Dies gilt nicht für Zitate und Werke, die aufgrund einer anderen Erlaubnis genutzt werden. Um die Bedingungen der Lizenz einzusehen, folgen Sie bitte dem Hyperlink:

<https://creativecommons.org/licenses/by/4.0/deed.de>

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 30. Juli 2024

Unterschrift

Zusammenfassung

Im Kontext der IT-Sicherheit werden Provenance Graphen für die Beantwortung sicherheitsrelevanter Fragen wie *Welche Auswirkungen kann diese Sicherheitslücke haben?* oder *Welcher Prozess hat meine Daten verändert oder gelöscht?* verwendet. Weiterhin können Provenance Graphen für das Training von Machine Learning Methoden für die Erkennung von Angriffen auf Netzwerke und Betriebssysteme genutzt werden. Aufgrund der hohen Datenraten bei der Erzeugung von Provenance Graphen (mindestens 1,2 MB/s in realistischen Szenarien) ist eine effiziente Datenverwaltung für den praktischen Einsatz dieser Graphen notwendig.

In dieser Arbeit wird untersucht, welche Anforderungen und Herausforderungen für die Verwaltung von Provenance Graphen existieren und welche Vor- und Nachteile sich aus der Verwaltung mit den Datenbankmanagementsystemen (DBMS) PostgreSQL, Neo4j, ONgDB und Memgraph ergeben. Zur Untersuchung wurde ein Versuchsaufbau für reproduzierbare Experimente entwickelt, in dem ein Provenance Graph Datensatz in die untersuchten DBMS geschrieben wird und dabei Hardwaremetriken und Query-Antwortzeiten zur späteren Auswertung erhoben und gespeichert werden.

Die Ergebnisse der Arbeit zeigen, dass die hier betrachteten DBMS nur teilweise die recherchierten Anforderungen einer Datenverwaltung für Provenance Graphen im Kontext der IT-Sicherheit erfüllen können. Es konnte gezeigt werden, dass sich relationale Datenbanken durch sparsameren Ressourcenverbrauch gegenüber Graphdatenbanken auszeichnen, dafür aber Nachteile in den Abfragemöglichkeiten durch die Querysprache aufweisen.

Schlüsselwörter - IT-Sicherheit, Provenance Graphen, Provenance Information, Relationale Datenbanken, Graphdatenbanken, Untersuchung von Datenbanken, SQL, Cypher

Inhaltsverzeichnis

Abbildungsverzeichnis	5
Tabellenverzeichnis	6
1. Einleitung	10
2. Grundlagen zu Provenance Graphen und Datenbanksystemen	12
2.1. Provenance Information und Provenance Graphen	12
2.2. Vorstellung verschiedener DBMS	27
3. Verwandte Arbeiten	31
4. Methodik	35
4.1. Ziele und Anforderungen einer Datenverwaltung für Provenance Graphen	35
4.2. Bestimmung der Priorität der Anforderungen	46
4.3. Auswahl von DBMS	50
4.4. Der eingesetzte Provenance Graph Datensatz	52
5. Durchführung der Experimente	54
6. Ergebnisse der Experimente	58
6.1. Auswertung der Schreibgeschwindigkeiten	58
6.2. Auswertung der Experimente zum Ressourcenverbrauch	62
6.3. Beurteilung der Formulierung von Abfragen	68
6.4. Auswertung der Abfragelatenz	73
7. Diskussion der Experimentergebnisse	80
7.1. Einordnung der Ergebnisse ins Gesamtziel	80
7.2. Handlungsempfehlungen für ein DBMS zur Verwaltung von Provenance Graphen	85
7.3. Limitationen und Einschränkungen	86
8. Fazit	88
A. Anhang	93

Abbildungsverzeichnis

1.1. Referenzarchitektur für ein provenancegraphbasiertes Erkennungssystem aus [LCYC21].	10
2.1. Visualisierung des Big Data 3V Modells aus [SS13].	13
2.2. Konzeptioneller Aufbau des OPM - Knoten	15
2.3. Konzeptuelle Aufbau des OPM - Kanten	15
2.4. Konzeptuelle Aufbau des OPM - Ursache und Effekt vgl. [MCF ⁺ 11].	16
2.5. Konzeptuelle Aufbau des PROV-DM vgl. [MM13].	17
2.6. Exemplarische Abbildung eines Bearbeitungs Prozesses im PROV-DM [MM13].	18
2.7. Konzeptueller Aufbau des CDM aus [GKC ⁺ 19].	19
2.8. Konzeptioneller Aufbau der Kernelemente der <i>KRYSTAL</i> -Ontologie aus [KEK ⁺ 22].	20
2.9. Beispielhafter Provenance Graph nach PROV-DM.	21
2.10. Beispielhafter Provenance Graph nach CDM.	22
2.11. Beispielhafter Provenance Graph nach <i>KRYSTAL</i> -Ontologie.	22
2.12. Beispielhaftes Szenario für Verwendung mehrstelliger Beziehungen im PROV-DM.	23
2.13. Provenance Graph eines Angriffs aus <i>KRYSTAL</i> [KEK ⁺ 22].	24
4.1. Exemplarischer Graph zur Verdeutlichung des Forward Tracking.	41
4.2. Kano Modell vgl. [Poh21].	47
5.1. Konzeptionelle Veranschaulichung der Testumgebung.	54
6.1. Anzahl Knoten je DBMS über Experimentendauer.	59
6.2. DBMS Metriken Hardware Ressourcen - CPU - Batchgröße 500.	63
6.3. DBMS Metriken Hardware Ressourcen - RAM - Batchgröße 500.	64
6.4. DBMS Metriken Hardware Ressourcen - RAM Cached - Batchgröße 500.	65
6.5. DBMS Metriken Hardware Ressourcen - Lesezugriffe Festplatte - Batchgröße 500.	66
6.6. DBMS Metriken Hardware Ressourcen - Schreibzugriffe Festplatte - Batchgröße 500.	67
6.7. Latenzen der Anwendung je Abfrage bei Batchgröße von 500.	74
6.8. Latenzen der DBMS je Abfrage bei Batchgröße von 500.	75

6.9. Ergebnismenge je Abfrage bei Batchgröße von 500.	76
7.1. Auszug aus Grafana Dashboard - Vergleich CPU Metriken bei Absturz. . .	87
A.1. Architektur für einen Master Data Store für Provenance Graphen aus [GKC ⁺ 19]	96
A.2. DBMS Metriken Hardware Ressourcen - CPU - Batchgröße 1000.	101
A.3. DBMS Metriken Hardware Ressourcen - CPU - Batchgröße 1500.	101
A.4. DBMS Metriken Hardware Ressourcen - RAM - Batchgröße 1000.	102
A.5. DBMS Metriken Hardware Ressourcen - RAM - Batchgröße 1500.	102
A.6. DBMS Metriken Hardware Ressourcen - RAM Cache - Batchgröße 1000. .	103
A.7. DBMS Metriken Hardware Ressourcen - RAM Cache - Batchgröße 1500. .	103
A.8. DBMS Metriken Hardware Ressourcen - Lesezugriffe - Batchgröße 1000. .	104
A.9. DBMS Metriken Hardware Ressourcen - Lesezugriffe - Batchgröße 1500. .	104
A.10.DBMS Metriken Hardware Ressourcen - Schreibzugriffe - Batchgröße 1000.	105
A.11.DBMS Metriken Hardware Ressourcen - Schreibzugriffe - Batchgröße 1500.	105
A.12.Abfrage Latenzen DBMS je Query bei Batchgröße von 1000	106
A.13.Abfrage Latenzen DBMS je Query bei Batchgröße von 1500	107
A.14.Abfrage Latenzen Anwendung je Query bei Batchgröße von 1000	108
A.15.Abfrage Latenzen Anwendung je Query bei Batchgröße von 1500	109
A.16.Ergebnismenge je Query bei Batchgröße von 1500	110
A.17.Ergebnismenge je Query bei Batchgröße von 1500	111

Tabellenverzeichnis

4.1. Ziele einer Datenverwaltung für Provenance Graphen.	46
4.2. Übersicht der Basisfaktoren der Datenverwaltung.	48
4.3. Übersicht der Leistungsfaktoren der Datenverwaltung.	49
4.4. Übersicht der Begeisterungsfaktoren der Datenverwaltung.	49
6.1. Durchschnittliche Einfügeraten der DBMS.	61
6.2. Ergebnismengen der Queries bei gleichem Datenvolumen.	73
7.1. Zusammenfassung des Rankings der DBMS.	84
7.2. Metriken des Rankings.	85
A.1. Netzwerkverkehr der Publish-Subscriber Architektur.	97
A.2. Netzwerkverkehr der DBMS - empfangene Daten.	97
A.3. Hardware Vergleich der DBMS - CPU und Schreibzugriffe.	99
A.4. Hardware Vergleich der DBMS - Arbeitsspeicher.	100

Quellcodeverzeichnis

4.1. Intrusion Detection Abfrage in SPARQL aus [KEK ⁺ 22].	40
4.2. Abfrage zum Forward Tracking eines beliebigen Startknoten.	41
5.1. Cypher Einfügeanweisung zur Knoten- und Kantenerzeugung.	55
5.2. SQL Einfügeanweisung zur Knoten- und Kantenerzeugung.	55
6.1. Cypher Abfrage für das Finden der Nachfolger eines Knoten.	69
6.2. Cypher Abfrage für das Finden der Vorgänger eines Knoten.	69
6.3. Cypher Abfrage für das Finden des kürzesten Pfades zwischen zwei beliebigen Knoten.	70
6.4. Cypher Abfrage für das Finden der 2-Hop Nachbarschaft.	70
6.5. Cypher Abfrage für das Finden der 2-Hop Nachbarschaft mit Memgraph Prozedur.	71
6.6. SQL Abfrage für das Finden der Vorgänger eines Knoten.	71
A.1. SQL Query für das Finden der Nachfolger eines Knoten.	112
A.2. SQL Query für das Finden eines Pfades zwischen zwei beliebigen Knoten.	113
A.3. SQL Abfrage für das Finden der 2-Hop Nachbarschaft eines Knotens.	114

Abkürzungsverzeichnis

APT Advanced Persistent Threat.

CDM Common Data Model.

DBMS Datenbankmanagementsystem.

HDFS Hadoop Distributed File System.

OPM Open Provenance Model.

OWL W3C Web Ontology Language.

PIDS Provenance-Based Intrusion Detection System.

PROV-DM W3C PROV-DM: The PROV Data Model.

RBAC Role Based Access Control.

RDF Resource Description Framework.

SQL Structured Query Language.

STARC DARPA TC Scalable Transparency Architecture for Research Collaboration
(STARC)-DARPA Transparent Computing (TC) Program.

TSDB Time-Series Database.

LPM Linux Provenance Module.

1. Einleitung

Provenance wörtlich übersetzt aus dem Englischen bedeutet *Ursprung* oder *Herkunft*, oder aus dem lateinischen *provenire* übersetzt, *hervorkommen*, *entstehen aus*. Provenance Information beschreibt die Herkunft beliebiger Artefakte. Die Anwendungsgebiete für Provenance Information sind die Überwachung der Qualität beliebiger gesammelter Daten, die Reproduzierung wissenschaftlicher Versuche und Ergebnisse und in den letzten Jahren zunehmend IT-Sicherheit [ZGCD23]. Konkret in der IT-Sicherheit kann Provenance Information für die Beantwortung sicherheitsrelevanter Fragen wie *Welche Auswirkungen hat diese Sicherheitslücke?* oder *Welcher Prozess hat meine Daten verändert oder gelöscht?* genutzt werden. Mithilfe von Deep Learning kann Provenance Information außerdem für das Training von Graph Neuronalen Netzen für das Erkennen von Angriffen auf Netzwerke und Betriebssysteme genutzt werden.

In dieser Arbeit wird untersucht, welche Anforderungen und Herausforderungen die Verwaltung von Provenance Information mit sich bringt und welche Vor- und Nachteile sich aus der Verwaltung mit unterschiedlichen Datenbanksystemen ergeben. Zum besseren Verständnis der Problemstellung dient Abbildung 1.1.

In *Threat Detection and Investigation with System-level Provenance Graphs: A Survey* [LCYC21] stellen die Autoren Li et al. eine Referenzarchitektur für ein provenancegraphbasiertes Erkennungssystem vor. Die Referenzarchitektur ist in drei Module aufgeteilt, einem Data Collection Modul, einem Data Management Modul und einem Threat Detection Modul. Das Data Collection Modul befasst sich mit der Aufgabe aus Logdaten einen

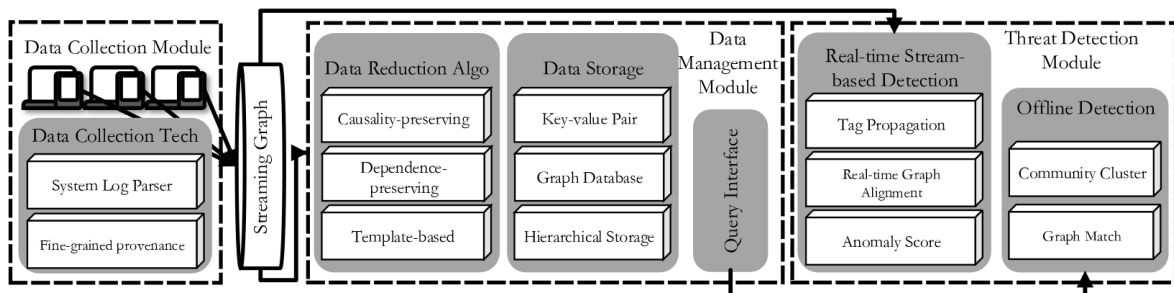


Abbildung 1.1.: Referenzarchitektur für ein provenancegraphbasiertes Erkennungssystem aus [LCYC21].

Provenance Graph in Echtzeit zu erzeugen, welcher hier als *Streaming Graph* bezeichnet wird. Das Data Management Modul befasst sich mit der Reduktion, der Speicherung der Echtzeit Graphdaten und der Bereitstellung eines Query Interface für die Threat Detection. Die Threat Detection übernimmt die Angriffserkennung zum einen in Real-Time durch Tag Propagation (Backtracking), Graph Alignment (informell Graph Matching), Anomalieerkennung (GNNs) und zum anderen offline durch Cluster Bildung oder Graph Matching.

Diese Arbeit befasst sich, bezogen auf die Referenzarchitektur, mit der Untersuchung welches Datenbankmanagementsystem (DBMS) sich als Data Storage und Data Management Module, sowie Query Interface eignet. Zusätzlich wird dabei untersucht, ob ein Query Interface die Anforderung einer Real-Time Detection erfüllen kann. Data Collection und Reduktionsalgorithmen werden in dieser Arbeit nicht betrachtet, da diese den Rahmen übersteigen würden.

Zuerst werden in Kapitel 2 die fachlichen Grundlagen und Grundbegriffe rund um Provenance Graphen, deren Verwendung im Kontext von IT-Sicherheit und kurz die Eigenschaften unterschiedlicher Datenbanksysteme und -produkte erläutert. Anschließend wird in Kapitel 3 der aktuelle Forschungsstand in Bezug auf die Datenverwaltung von Provenance Graphen betrachtet. In Kapitel 4 werden die Anforderungen an die Datenverwaltung von Provenance Graphen herausgearbeitet und mit verschiedenen Produkten von DBMS abgeglichen und bewertet, um die vielversprechendsten Kandidaten bzw. Produkte zu ermitteln. Die in Kapitel 4 ausgewählten DBMS Kandidaten werden dann als Prototyp implementiert, um mithilfe von Experimenten die Eignung der hier ausgewählten DBMS für die Verwaltung von Provenance Graphen festzustellen. Die Ergebnisse der Experimente werden in Kapitel 6 im Detail beschrieben und in Kapitel 7 zusammengefasst für eine abschließende Beurteilung.

Das Ergebnis der Arbeit ist, dass keines der hier betrachteten DBMS alle Anforderungen einer Datenverwaltung für Provenance Graphen erfüllen konnte. Es konnte gezeigt werden, dass sich relationale Datenbanken durch sparsameren Ressourcenverbrauch gegenüber Graphdatenbanken auszeichnen, jedoch dafür Nachteile in den Abfragemöglichkeiten durch die Querysprache SQL aufweisen. Dafür wurde ein Versuchsaufbau für reproduzierbare Experimente aufgestellt, der einen Provenance Graph Datensatz in die untersuchten DBMS schreibt und dabei Hardwaremetriken und Query-Antwortzeiten zur späteren Auswertung speichert.

2. Grundlagen zu Provenance Graphen und Datenbanksystemen

In diesem Kapitel werden Grundlagen zur Verwendung von Provenance Graphen in der IT-Sicherheit und verschiedene DBMS erläutert. Der Fokus liegt bei der Betrachtung der DBMS auf den jeweiligen Stärken nach Entwicklerangaben und auszeichnenden Eigenschaften. Dadurch soll hervorgehoben werden in welchem Anwendungsfall welches DBMS am besten geeignet ist.

2.1. Provenance Information und Provenance Graphen

Wie in der Einleitung erwähnt, bezeichnet Provenance Information bezeichnet all die Meta-Informationen zur Entstehung, Herkunft und Veränderung eines beliebigen Artefakts. Mit Hilfe von Provenance Information lassen sich kausale Zusammenhänge und Abhängigkeiten darstellen und Aussagen über Vertrauenswürdigkeit, Integrität und Qualität von Daten ableiten [HC17]. Zur Verdeutlichung ein Beispiel aus dem Kontext der IT-Sicherheit: Angenommen eine Datei *text.txt* wird durch einen Prozess *nginx* erzeugt und es ist bekannt, dass der Prozess *nginx* nicht vertrauenswürdig ist, da dieser von einem Angreifer übernommen wurde. Dann kann darauf geschlossen werden, dass auch die Datei *text.txt* nicht vertrauenswürdig ist.

Die Anwendungsgebiete beschränken sich nicht nur auf IT-Sicherheit, sondern Provenance Information wird u.a. für die Überwachung von Daten in DBMS, die Validierung und Reproduzierbarkeit von wissenschaftlichen Experimenten eingesetzt [PSR23].

Provenance Information weist charakteristische Eigenschaften von Big Data auf. Eine erste Definition zu den Charakteristiken für Big Data wurde in *3D Data Management: Controlling Data Volume, Velocity, and Variety* [Lan01] im Jahr 2001 mit dem *3V* Modell zur Beschreibung von Big Data präsentiert. *Volume* ist die erheblichste Eigenschaft von Big Data zur Abgrenzung gegenüber traditionellen Daten. Daten im Big Data Kontext erzeugen und benötigen ein erheblich höheres Datenvolumen. *Velocity* beschreibt die Geschwindigkeit mit der Daten erzeugt, verwaltet und verarbeitet werden. *Variety* beschreibt die heterogenen Datentypen, die im Big Data Kontext erzeugt werden. Dabei kann es sich um strukturierte, semi-strukturierte und unstrukturierte Daten handeln.

Mindestens die drei Eigenschaften *Volume*, *Velocity* und *Variety* sind charakteristisch für Provenance Information. *Volume* wird erfüllt durch die Menge an Provenance Information in Höhe von mehreren Terabytes, die auf einem Rechner entstehen kann. In *hreat Detection and Investigation with System-level Provenance Graphs: A Survey* von Li et al. [LCYC21] wird beschrieben, dass ein typisches System in einer Bank mit 20.000 Rechnern jährlich etwa 70 Petabyte an Logdaten erzeugt. *Velocity* wird ebenfalls erfüllt durch die Geschwindigkeit mit der die Daten erzeugt werden. In [BTBM15] wird in der Evaluation festgestellt, dass etwa zwei GB/Minute an Provenance Rohdaten erzeugt werden. *Variety* ist etwas geringer ausgeprägt, da die Daten strukturiert auftreten bzw. in eine strukturierte Form gebracht werden.

Die Abbildung 2.1 visualisiert das 3-V Modell aus [SS13] und die verschiedenen Ausprägungen die Daten in den jeweiligen Eigenschaften Variety (Strukturierte Daten - Unstrukturierte Daten), Velocity (Batch - Streaming) und Volume (Terrabyte - Zettabyte) haben können.

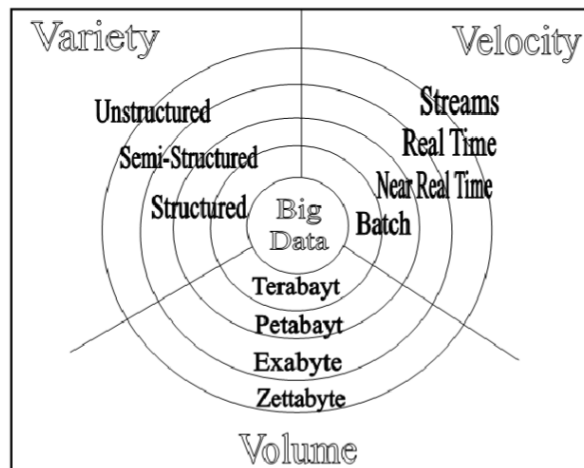


Abbildung 2.1.: Visualisierung des Big Data 3V Modells aus [SS13].

In aktuellen Forschungen und Artikeln wurde das ursprüngliche 3V Modell erweitert zum 4V Modell oder auch 5V Modell¹. Dabei wird zu den ursprünglichen 3 *V-Eigenschaften* noch *Value* bzw. *Veracity* als Eigenschaft hinzugefügt. *Value* beschreibt den großen Gesamtwert der Daten, also den Wert der Information, die aus der Gesamtheit der Daten abgeleitet werden können [GCD20]. *Veracity* beschreibt die Qualität und Genauigkeit der Daten, genauer das Vertrauen in die Richtigkeit und die Möglichkeit, Erkenntnisse aus den Daten zu gewinnen.

¹Link zu Artikel: <https://www.techtarget.com/searchdatamanagement/definition/5-Vs-of-big-data>

Modellierung von Provenance Graphen

Für die Modellierung der Provenance Information werden vor allem Graphenmodelle verwendet. Dieses ist vorteilhaft, da ein Graph die Beziehungen und Struktur zwischen Entitäten (Knoten) durch Kanten abbildet. Die Beziehungen zwischen Entitäten sind die Herkunftsbeziehungen z. B. *wurde erzeugt durch* oder *wurde verändert durch*, welche genau den Kern der Provenance Information widerspiegelt. Bei Provenance Information, die als Graph modelliert ist, spricht man von einem Provenance Graphen [MM13].

Bei einem Provenance Graphen handelt es sich um einen gerichteten Graphen. Je nach zu Grunde liegendem Modell ist der Graph zyklisch oder azyklisch. Es werden je nach Modell unterschiedliche Artefakte als Knoten und Kanten definiert. Generell repräsentiert jedoch jeder Knoten eine Entität und jede Kante die Beziehung zwischen den Entitäten [LD20].

Provenance Graphen haben einen starken temporalen Bezug, was sie von normalen Graphen unterscheidet. Ein Ereignis, das zu einer Veränderung von Knoten führt, wird zu einem gewissen Zeit ausgeführt. Ereignisse können außerdem auf bereits existierenden Knoten mehrfach vorkommen und zu unterschiedlichen Zustände der Knoten zu unterschiedlichen Zeitpunkten führen. In *Threat Detection and Investigation with System-level Provenance Graphs: A Survey* von Li et al. [LCYC21] wird dies als kausale Abhängigkeit bezeichnet. Angenommen, es existieren die Kanten k_1 und k_2 , welche die Knoten u_1 bzw. u_2 als Quelle und die Knoten v_1 bzw. v_2 als Ziel haben. Beide Kanten besitzen außerdem den Zeitstempel t . Kante $e_1 = (u_1, v_1, t_1)$ und Kante $e_2 = (u_2, v_2, t_2)$ sind kausal abhängig, wenn $v_1 = u_2 \wedge t_1 < t_2$. Kausale Abhängigkeit zeigt einen möglichen Daten- oder Kontrollfluss zwischen Knoten auf.

Um einen Provenance Graphen anhand von Provenance Information zu modellieren, existieren mehrere Ansätze. Zum einen gibt es mehrere Standards, die ein möglichst domänen-agnostisches Modell bieten, wie das Open Provenance Model (OPM) [MCF⁺11] oder das W3C PROV-DM: The PROV Data Model (PROV-DM) [MM13]. Weiterhin existieren domänen-spezifische Modelle, die sich auf das Erfassen von Herkunfts- bzw. Metainformationen einer Domäne spezialisieren, wie das Common Data Model (CDM) (IT-Sicherheit) [GKC⁺19], oder das standardisierte Modell ISO-19115 (Geospatial)². Zuletzt existieren Ansätze, um Provenance Graphen mithilfe einer Ontologie über Resource Description Framework (RDF) Triple als Knowledge Graphen zu modellieren. Für die Modellierung über RDF Triple existieren das W3C³ oder in der Forschung *KRYSTAL: Knowledge graph-based framework for tactical attack discovery, in audit data* [KEK⁺22]. Im Folgenden werden die Modelle OPM, PROV-DM, CDM und die Modellierung mithilfe von *KRYSTAL* als RDF-Triple genauer beschrieben.

²Link zu ISO-19915 - <https://www.iso.org/standard/53798.html>

³PROV-O Dokumentation - <https://www.w3.org/TR/2013/REC-prov-o-20130430/>

Im OPM, welches im Jahr 2011 veröffentlicht wurde [MCF⁺11], repräsentieren die Knoten ein Artefakt (ein unveränderlicher Zustand, der digital festgehalten ist), einen Prozess (die Aktionen und Kausalitäten eines Artefakts) oder einen Agenten (den Initiator des Prozesses), siehe Abbildung 2.2.

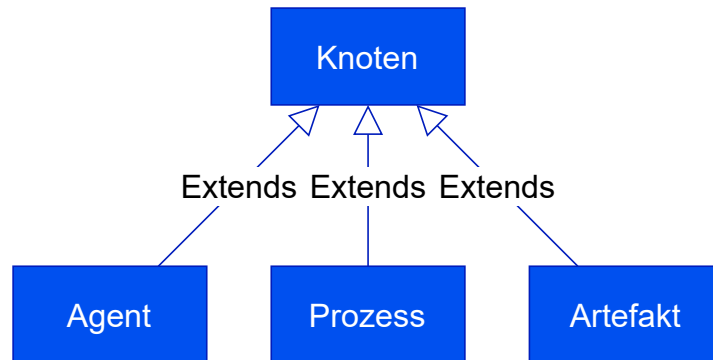


Abbildung 2.2.: Konzeptioneller Aufbau des OPM - Knoten
vgl. [MCF⁺11].

Die Kanten repräsentieren den Informationsfluss in Form von Eingaben und Ausgaben oder kausale Zusammenhänge wie, *wasControlledBy* (*WCB*), *wasDerivedFrom* (*WDF*), *used*, *wasGeneratedBy* (*WGB*) und *wasTriggeredBy* (*WTB*), wie in Abbildung 2.3 dargestellt [PSR23].

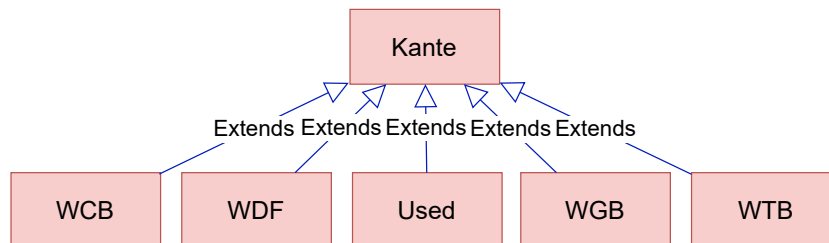


Abbildung 2.3.: Konzeptuelle Aufbau des OPM - Kanten
vgl. [MCF⁺11].

Im OPM haben Kanten spezifische Quellen (Effekt) und Ziele (Ursache). Die Abbildung 2.4 zeigt die möglichen Ursachen- und Effektbeziehungen der Kanten und Knoten in einem OPM Provenance Graphen. *Used* und *WGB* haben ein Artefakt als Effekt und einen Prozess als Ursache. *WDF* hat ein Artefakt als Effekt und Ursache. *WCB* hat einen Prozess als Effekt und einen Agenten als Ursache. *WTB* hat einen Prozess als Ursache und Effekt. Optional können die Kanten *WGB*, *WCB* und *used* einen Zeitbezug haben, sowie *WTB*, *WGB*, *WCB* und *Used* eine Rolle.

Das PROV-DM ist als Nachfolger zum OPM erschienen. Es wurde unter den Anforderungen entwickelt wie sein Vorgänger domänen-agnostisch anwendbar zu sein. Es

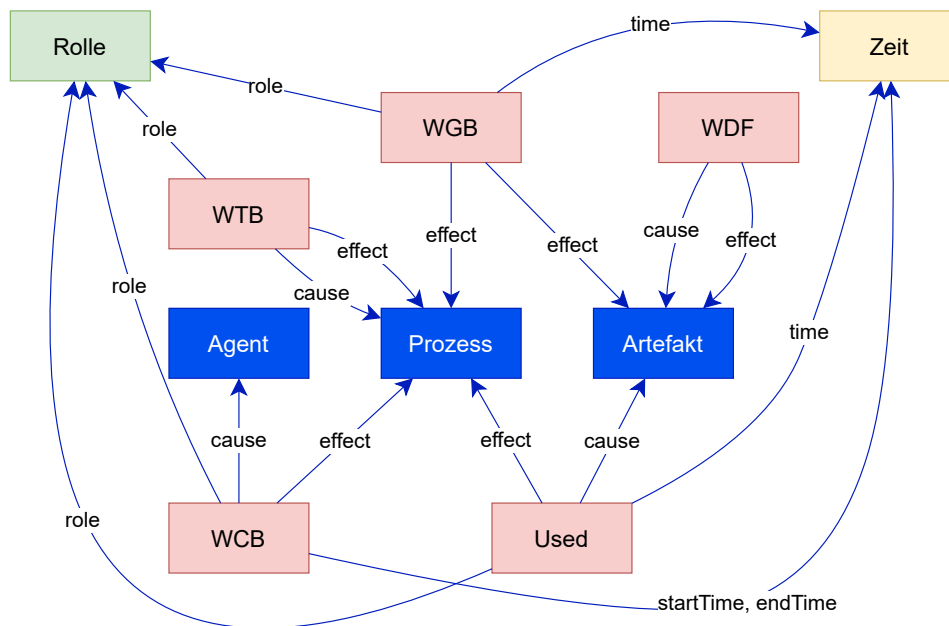


Abbildung 2.4.: Konzeptuelle Aufbau des OPM - Ursache und Effekt vgl. [MCF⁺11].

hat eine ähnliche Entity-Relationship-Struktur wie das OPM. Jedoch ermöglicht das PROV-DM die Verknüpfung von Entitäten, Aktivitäten und Agenten durch beliebige Beziehungen [PSR23].

In Abbildung 2.5 wird der konzeptuelle Aufbau der Kernelemente des PROV-DM dargestellt. Analog zum OPM sind im PROV-DM Entitäten (Entity) (OPM: Artefakt), Aktivitäten (Activity) (OPM: Prozess) und Agenten (Agent) (OPM: Agent) die Knoten im Provenance Graphen. Entitäten sind physische, digitale oder konzeptuelle Gegenstände mit fixen Eigenschaften. Aktivitäten geschehen über einen Zeitraum und interagieren mit oder aufgrund von Entitäten. Agenten sind etwas, das in irgendeiner einer Art und Weise Verantwortung für die Ausführung einer Aktivität, die Existenz einer Entität oder für die Aktivitäten eines anderen Agenten trägt. Die Beziehungen zwischen den Elementen werden durch die Konzepte, die auch im OPM definiert werden, wie *WasGeneratedBy*, *Used*, *WasDerivedFrom*. Zusätzlich zu den Konzepten des OPM werden im PROV-DM die Beziehungen *WasAttributedTo*, *WasAssociatedWith*, *WasInformedBy* und *ActedOn-BehalfOf* definiert. Diese Beziehungen sind binär, d.h. es werden je nach Beziehung jeweils zwei Entitäten, Aktivitäten oder Agenten miteinander verknüpft [MM13].

Erweiternd zu den bereits erwähnten grundlegenden Beziehungen existiert das Konzept der *Expanded Relations*. Dies ist gedacht zur Abbildung von komplexen oder kompositen Beziehungen, wie bspw. *Ableitungen*. Grundlegend, ohne zusätzliche Information, kann die Beziehung *WasDerivedFrom* binär sein (bspw. Dokument d1 wurde abgeleitet/als Kopie erzeugt von Dokument d2). Es kann jedoch spezifiziert werden, dass mehrere

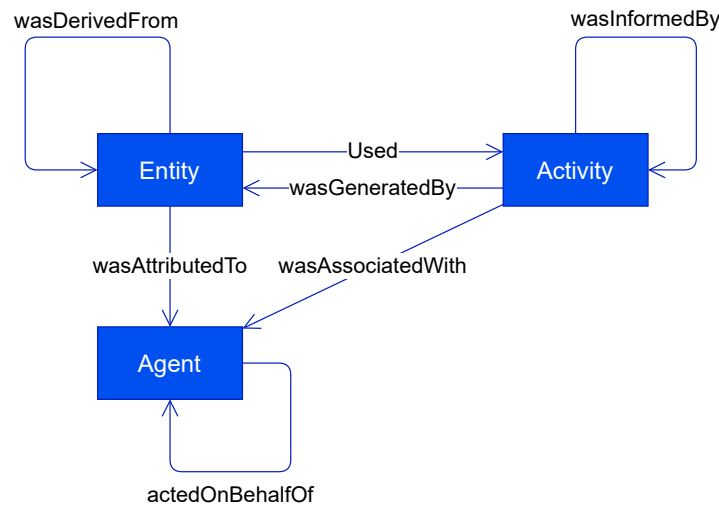


Abbildung 2.5.: Konzeptuelle Aufbau des PROV-DM vgl. [MM13].

Aktivitäten in der Ableitung beteiligt sind. Dann wird die binäre Beziehung erweitert zu einer Expanded Relation⁴.

Abbildung 2.6 zeigt einen exemplarisch aufgebauten Provenance Graphen nach dem PROV-DM für einen Bearbeitungsprozess eines Dokuments. Eine Entität mit dem Namen *WD-prov-dm-20111215* wurde durch eine Aktivität mit dem Namen *edit1* erzeugt. Die Herkunftsbeziehung zwischen den beiden Elementen lautet dementsprechend *wasGeneratedBy*. Die beiden Agenten mit dem Namen *Simon* und *Paolo* sind mit der Aktivität *edit1* über die Beziehung *wasAssociatedWith* mit unterschiedlichen Rollen *editor* bzw. *contributer* verknüpft.

Für die Modellierung der Provenance Graphen als Knowledge Graph bietet das W3C ergänzend zum PROV-DM eine Ontologie PROV-O⁵. In dieser Ontologie ist das Mapping für PROV-DM zur W3C Web Ontology Language (OWL) definiert.

Das in Abbildung 2.6 vorgestellte Beispiel lässt sich nach der PROV-O Ontologie folgendermaßen notieren: Die Notation der Entität *WD-prov-dm-20111215* (in Abbildung 2.6 in gelb) ist:

```
entity(tr:WD-prov-dm-20111215, [prov:type="document,ex:version="2"])
```

Für die Aktivität *edit1* (in der Abbildung in blau):

```
activity(ex:edit1, [ prov:type="editing" ])
```

Für die Beziehung zwischen Entität und Aktivität *wasGeneratedBy*:

```
wasGeneratedBy(tr:WD-prov-dm-20111215, ex:edit1, -)
```

⁴Link zu PROV-DM Dokumentation Expanded Relations: <https://www.w3.org/TR/prov-dm/>

⁵Link zu PROV-O Dokumentation: <https://www.w3.org/TR/2013/REC-prov-o-20130430/>

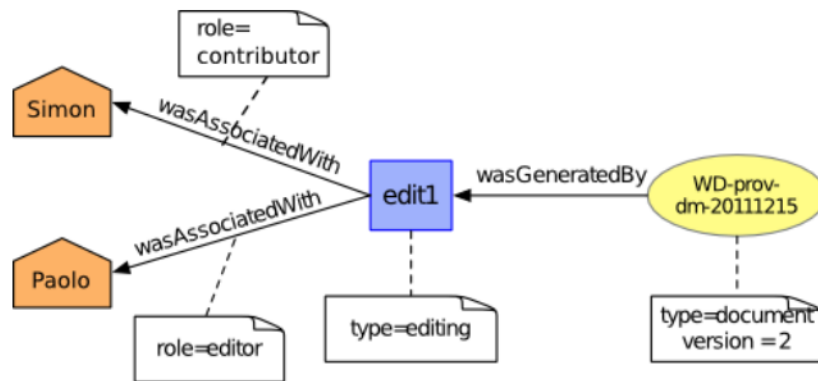


Abbildung 2.6.: Exemplarische Abbildung eines Bearbeitungs Prozesses im PROV-DM [MM13].

Für Agenten *Paolo* und *Simon* (in der Abbildung in orange):

```
agent(ex:Paolo, [ prov:type='prov:Person'])
```

```
agent(ex:Simon, [ prov:type='prov:Person'])
```

Für die unterschiedlichen Rollen der Agenten in Bezug auf die Editier-Aktivität:

```
wasAssociatedWith(ex:edit1, ex:Paolo, -, [ prov:role="editor"])
```

```
wasAssociatedWith(ex:edit1, ex:Simon, -, [ prov:role="contributor"])
```

Als weiteres Modell für Provenance Graphen wurde im Rahmen des Scalable Transparency Architecture for Research Collaboration (STARC)-DARPA Transparent Computing (TC) Program (STARC DARPA TC) das domänen-spezifische CDM für die IT-Sicherheit zur Modellierung von Betriebssystemen entwickelt [GKC⁺19]. In Abbildung 2.7 wird das konzeptuelle Modell des CDM dargestellt. Die Kernelemente des CDM sind Events, Subjekte und Objekte. Events repräsentieren Aktionen, die durch Subjekte des Systems ausgeführt werden. Sie repräsentieren den Informationsfluss zwischen Daten, Objekten und Subjekten und sind atomar, d.h. sie sind nicht direkt mit einem anderen Event verknüpft. Beispiele für Events sind Systemaufrufe oder Funktionsaufrufe. Subjekte repräsentieren Ausführungskontexte, also hauptsächlich Threads und Prozesse in einem Betriebssystem. Objekte sind Datenquellen wie bspw. Sockets, Dateien, Speicher, Variablen, Argumente von Events und generell Daten, die Eingabe oder Ausgabe für ein Event sein können.

Die drei Kernelemente stehen in den folgenden Beziehungen zueinander. Ein Event *isGeneratedBy* einem Subjekt und ein Event *affects* beliebig viele Subjekte oder Objekte. Ein Subjekt *hasParent* ein weiteres Subjekt (fork-Operation) und ein Subjekt *affects* beliebig viele Objekte. Objekte *affects* Events, wenn bspw. Objekte Eingabe-Argumente eines Events sind.

Weiterhin existieren Ansätze, um einen Provenance Graphen als Knowledge Graphen

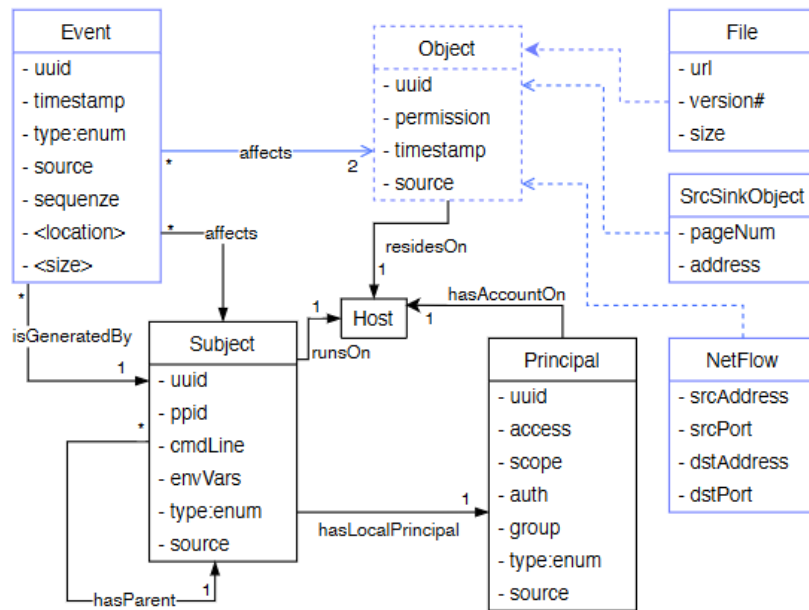


Abbildung 2.7.: Konzeptueller Aufbau des CDM aus [GKC⁺19].

zu modellieren. In [KEK⁺22] wird eine Ontologie definiert, um die Komponenten eines Betriebssystems mithilfe eines Knowledge Graphen entsprechend zur OWL2 Web Ontology Language⁶ abzubilden. Die Ontologie ermöglicht die Modellierung eines Provenance Graphen, der auf die Domäne IT-Sicherheit zugeschnitten ist. Ein RDF Graph nach OWL2 besteht aus einem Tripel *Subjekt, Prädikat, Objekt*. Subjekte und Objekte sind Knoten im Graphen und Prädikate sind die Beziehungen zwischen den Knoten.

Die Abbildung 2.8 aus [KEK⁺22] zeigt die Kernelemente der *KRYSTAL*-Ontologie. System Object ist die Abstraktion für die Elemente eines Betriebssystems wie Prozesse, Dateien und Sockets. Die konkreten Elemente sind Subklassen von System Object. Jedes System Object ist ein Knoten und jede Beziehung zwischen System Objects (*writes, deletes, isReadyBy, sends, isReceivedBy, isExecutedBy, mmap*) ist eine Kante im Provenance Graphen. *User* beschreibt die Benutzer eines Betriebssystems und hat ebenfalls drei Subklassen (*RootUser, SystemUser* und *LocalUser*). Zusätzlich gibt es einen *Host*, für die Maschine auf dem das System läuft. *User* und *Host* werden als Knoten und die Beziehungen zwischen *User* und System Object (*hasUser*) und zwischen System Object und *Host* (*originatesFrom*) als Kanten im Provenance Graphen abgebildet. Ergänzend zum *Host* wird auch die *IPAddress* als Knoten modelliert und besitzt die Beziehungen (*hasHostIp* und *HasSocketIp*) sowie Executables mit der Beziehung (*hasExe*). Um die Herkunftsbeziehung zwischen System Objects abzubilden (Socket zu Process, File zu Process) wird eine Provenance Relation als Beziehung hinzugefügt. Die Provenance

⁶Dokumentation OWL - <https://www.w3.org/OWL/>

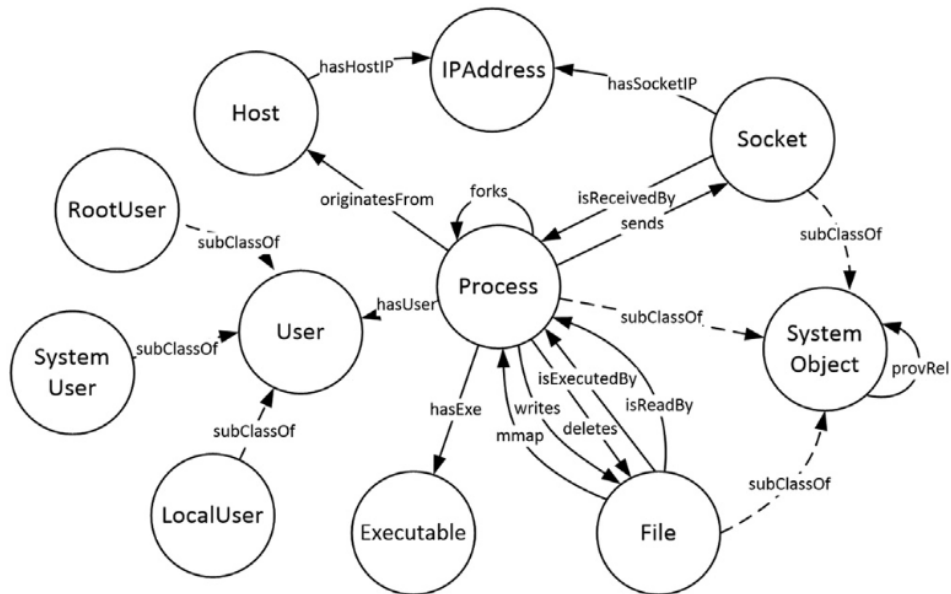


Abbildung 2.8.: Konzeptioneller Aufbau der Kernelemente der *KRYSTAL*-Ontologie aus [KEK⁺22].

Relation wird durch RDFS Entailment Rules und OWL Reasoning Rules inferiert⁷.

Gegenüberstellung der Modelle für Provenance Graphen

In diesem Abschnitt werden die vorgestellten Modellen für Provenance Graphen miteinander verglichen. Dabei wird auf die Konstruktion der Graphen und unterscheidende Merkmale der Modelle eingegangen.

Sind Provenance Graphen immer gerichtete, azyklische Graphen? Im Survey [PSR23] und UNICORN [HPB⁺20] wird als Definition für einen Provenance Graphen angegeben, dass es sich bei einem Provenance Graphen um einen gerichteten azyklischen Graphen - Directed Acyclic Graph (DAG) - handelt. In *KRYSTAL* [KEK⁺22] und Survey [LCYC21] wird angegeben, dass ein Provenance Graph ein gerichteter, gelabelter Graph ist.

Provenance Information sollte inhaltlich keine Zyklen beinhalten, da ein Provenance Information die Herkunft und Entstehung der betrachteten Daten beinhaltet. Ein Zyklus in den Provenance Daten würde bedeuten, dass Daten aus sich selbst ohne Einfluss von außen entstanden sind.

⁷Dokumentation der RDFS Entailment Rules - <https://www.w3.org/TR/rdf11-nt/#rdf-entailment>

PROV-DM und RDF erlauben vom Modell her Zyklen im Graphen. Da PROV-DM und *KRYSTAL* zur Modellierung RDF verwenden, sind Zyklen möglich. Um valide Provenance Graphen zu erzeugen, wurden für das PROV-DM Bedingungen⁸ formuliert. Dort wird eine PROV-Instanz als valide bezeichnet, wenn sich u.a. keine Zyklen der Beziehung *Vorgänger/strikter Vorgänger* im Graphen befinden. Das bedeutet, dass für manche Kanten Zyklen valide sind, jedoch nicht für die Herkunftsbeziehungen.

Wie unterscheiden sich die Datenmodelle OPM, Prov-DM, CDM, RDF? Die generellen Unterschiede zwischen den Modellen lassen sich am besten anhand eines Beispiels erläutern. Angenommen ist folgendes Szenario: Ein Benutzer *Alice* startet einen Prozess *change*. Der Prozess führt eine Schreiboperation *write* auf einer Datei *text.doc* aus und verändert diese. Anhand dieses Beispiels werden im folgenden Absatz drei Provenance Graphen jeweils nach dem PROV-DM, dem CDM und der *KRYSTAL*-Ontologie modelliert.

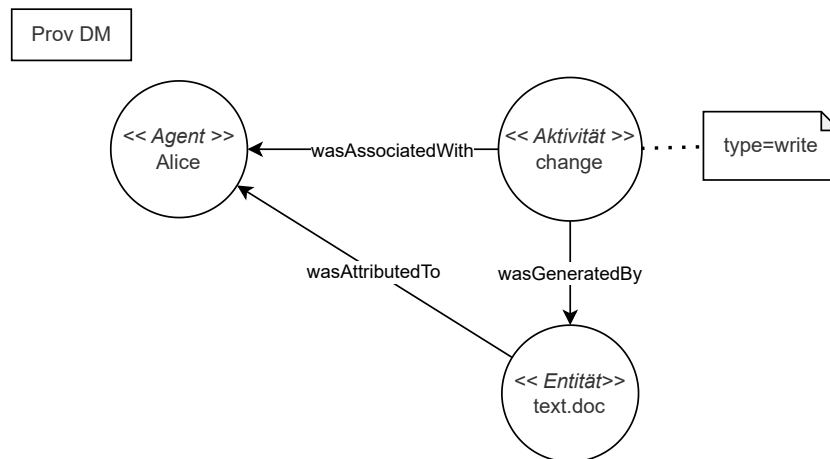


Abbildung 2.9.: Beispielhafter Provenance Graph nach PROV-DM.

Abbildung 2.9 zeigt den Provenance Graphen modelliert nach PROV-DM. Es fällt auf, dass der Typ der Aktivität (*write*) als Attribut definiert wird. Als domänen-agnostisches Modell sind außerdem die Beziehungen und Bezeichnungen der Knoten möglichst generisch gehalten.

Abbildung 2.10 zeigt den Provenance Graphen modelliert nach CDM. Im Gegensatz zum PROV-DM ist hier die Schreiboperation *write* explizit modelliert. Außerdem handelt es sich um ein Modell, das auf die Domäne der IT-Sicherheit spezialisiert ist. Kanten und Knoten wurden entsprechend des Domänenwissens bezeichnet.

⁸<https://www.w3.org/TR/2013/REC-prov-constraints-20130430/>

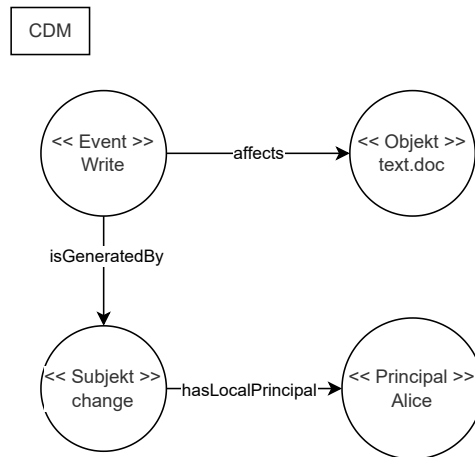


Abbildung 2.10.: Beispielhafter Provenance Graph nach CDM.

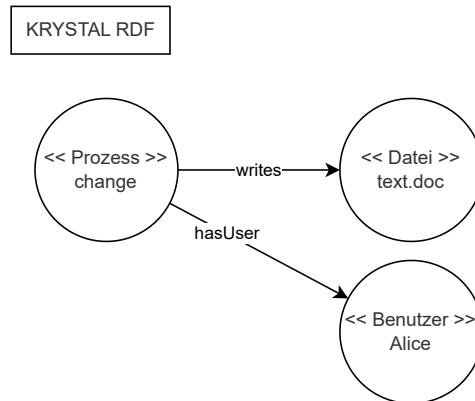


Abbildung 2.11.: Beispielhafter Provenance Graph nach *KRYSTAL*-Ontologie.

Abbildung 2.11 zeigt den Provenance Graphen modelliert nach der *KRYSTAL*-Ontologie. Die Schreiboperation *write* wird hier explizit als Kante definiert. Als domänen-spezifisches Modell sind Kanten und Knoten entsprechend der Domäne IT-Sicherheit vorgegeben.

Die größte Unterschied im Vergleich zwischen CDM gegenüber dem PROV-DM und OPM liegen laut [GKC⁺19] in der strikten Behandlung von Events (im Sinne des CDM) als *First-Class Entities*. Das bedeutet, dass im CDM Events Knoten im Graphen sind, während im Gegensatz dazu im Kern des OPM und PROV-DM *Events* als binäre Beziehungen zwischen Subjekten und Objekten verstanden werden. Die Modellierung von Events als Kanten anstelle von Knoten erhöht die Komplexität des Modells. Besonders im Fall, wenn Events mehr zwei Objekte und/oder Subjekte verbinden. Beziehungen wie Assoziation, Ableitung oder Delegation sind als binäre Beziehung nicht umfassend abbildbar. Zum Verständnis ist es hilfreich sich die Beziehung *Ableitung* genauer anzu-

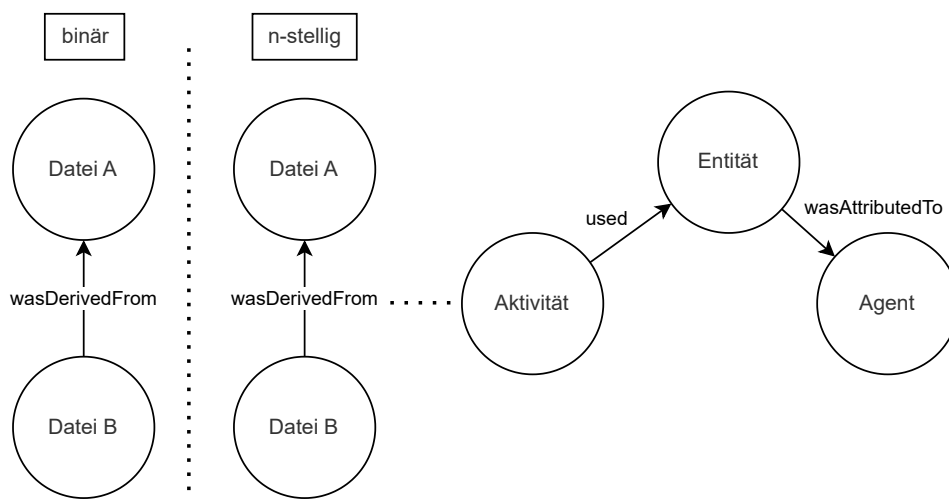


Abbildung 2.12.: Beispielhaftes Szenario für Verwendung mehrstelliger Beziehungen im PROV-DM.

schauen.

Ein beispielhaftes Szenario für eine Ableitungsbeziehung zwischen zwei Dateien ist in Abbildung 2.12 abgebildet. Eine Datei B kann beispielsweise von Datei A abgeleitet sein. Die Abbildung dieser Herkunftsbeziehung lässt sich einerseits *simpel* als binäre Kante ausdrücken (Datei B *wasDerivedFrom* Datei A). In diesem Fall sind jedoch keine weiteren Informationen über den Prozess der Ableitung im Provenance Graphen verfügbar. Andererseits lässt sich die Ableitung der Datei auch komplexer beschreiben, wenn bspw. in der Erzeugung der Ableitung weitere Aktivitäten und Entitäten involviert sind. Die Herkunftsinformationen, über die zur Ableitung verwendeten Aktivitäten und Entitäten, sind dann ebenfalls im Modell enthalten.

In [GKC⁺19] wird behauptet, dass das PROV-DM ausschließlich binäre Beziehungen unterstützt, jedoch bilden die binären Beziehungen nur die Kernelemente des PROV-DM. Es existieren im PROV-DM erweiterte Konzepte *Expanded Relations*⁹, um n-stellige Beziehungen abzubilden.

In [KEK⁺22] werden nur binäre Beziehungen betrachtet. Daher sind Beziehungen wie Ableitung, Assoziation oder Delegation nicht in der *KRYSTAL*-Ontologie enthalten. Dafür ermöglicht die Modellierung als RDF-Graph das Inferieren von Wissen über Entailment Rules. In *KRYSTAL* werden die Entailment Rules genutzt, um Wissen über die Herkunft von Knoten zu inferieren. Das Ableiten und Generieren von neuem Wissen mithilfe von Entailment Rules ist in PROV-DM und CDM nicht möglich.

⁹Link zu PROV-DM Doku: <https://www.w3.org/TR/prov-dm/#section-prov-extended-approach-expanded-relation>

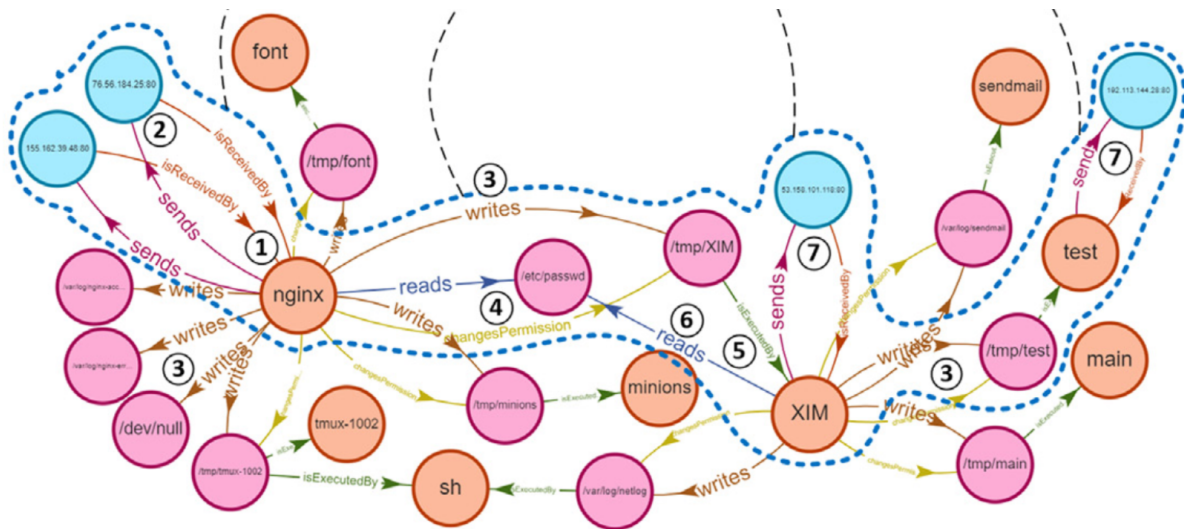


Abbildung 2.13.: Provenance Graph eines Angriffs aus *KRYSTAL* [KEK⁺22].

Verwendung von Provenance Information in der IT-Sicherheit

Provenance Information ermöglicht in der IT-Sicherheit die Analyse von historischen Daten z. B über die korrekte Veränderung von Daten oder forensische Analysen. Zudem ermöglicht Provenance Information das Aufspüren und die Detektion von Anomalien und Malware in einem laufenden Betriebssystem [PSR23] bspw. mithilfe neuronaler Netze [GKC⁺19] [CLL⁺23].

In der Domäne IT-Sicherheit wird der Ansatz verfolgt, mit Hilfe eines Provenance Graphen die Herkunft der Artefakte eines Betriebssystems zu modellieren. Die Knoten des Graphen beschreiben Artefakte wie Dateien, Prozesse oder Benutzer eines Betriebssystems. Die Kanten des Graphen beschreiben die Interaktionen und Ursprünge zwischen den Artefakten des Betriebssystems [CLL⁺23].

In Abbildung 2.13 ist ein Beispiel für einen Provenance Graphen aus [KEK⁺22] zu sehen, der einen Angriff beinhaltet. Die Angriffsaktivitäten sind die Knoten, die sich innerhalb der blau-gestrichelten Linie befinden. Die orangen Knoten repräsentieren Prozesse, die pinken Knoten repräsentieren Dateien und die blauen Knoten repräsentieren Hosts. Die Kanten zwischen den Knoten repräsentieren die Herkunftsbeziehungen und Interaktionen zwischen den Knoten. Der Ablauf des Angriffs ist nummeriert (1-7). Zuerst öffnet der Angreifer eine Shell-Verbindung zum Host des Opfers über einen vulnerablen Web Server (1,2). Danach schreibt eine ausführbare Datei eine Datei `/tmp/XIM` (3-5) und startet einen Prozess, der von der Datei `etc/passwd` liest (6). Zuletzt werden die gelesenen Daten an ein externes Netzwerk per HTTP gesendet (7).

Arten von Angriffen auf Systeme Für ein besseres Verständnis möglicher Angriffe werden im folgenden Abschnitt mit traditionellen und Advanced Persistent Threat (APT) zwei grundlegende Strategien von Angriffen auf Computersysteme vorgestellt. Die Merkmale von traditionellen Angriffen und APT stammen aus dem Survey *A Survey on Advanced Persistent Threats: Techniques, Solutions, Challenges, and Research Opportunities* [AMCH19].

Bei traditionellen Angriffen auf Computersysteme handelt es sich um Angriffe, die meistens von einer Person durchgeführt werden. Die Ziele solcher Angriffe sind im Regelfall beliebige individuelle Systeme. Traditionelle Angriffe werden meistens durchgeführt, damit sich Angreifer einen finanziellen Vorteil verschaffen oder auch aus Prestige Gründen. Der Ablauf des Angriffs ist eher kurzfristig angelegt und beschränkt sich auf einen kurzen Zeitraum (*smash and grab approach*).

Zu fortschrittlichen Angriffen, die auf Computersysteme ausgeführt werden, zählen Advanced Persistent Threat (APT). Typisch für einen APT-Angriff ist, dass der Angriff von einer Gruppe gut organisierter, ausgebildeter und finanzierter Angreifer durchgeführt wird. Die Ziele von APT-Angriffen sind in der Regel große Organisationen wie Unternehmen, staatliche Einrichtungen wie Sicherheitsdienste und Militär sowie öffentliche Infrastruktur.

APTs werden mit der Intention ausgeführt kritische Informationen über das Ziel zu erhalten und sich einen Vorteil gegenüber der betroffenen Organisation zu verschaffen.

Ein APT lässt sich treffend über die drei namensgebenden Bestandteile beschreiben:

- **Advanced:** APT-Angreifer sind gut organisiert, besitzen erhebliche Ressourcen, Werkzeuge und Methoden, um Angriffe durchzuführen.
- **Persistent:** Zum einen beschreibt *Persistent* die Angreifer, die mit großem Einsatz und Durchhaltevermögen einen Angriff durchführen. Zum anderen beschreibt es die Vorgehensweise über einen langen Zeitraum (von Monaten bis Jahren) unentdeckt ein Zielsystem zu infiltrieren (*low and slow approach*). Im Durchschnitt beträgt die Dauer eines APT-Angriffs 188 Tage [LCYC21].
- **Threat:** Die Gefahr, die von einem APT-Angriff ausgeht, liegt darin unentdeckt ein System zu kompromittieren und/oder zum Verlust von kritischen Daten und Bestandteilen eines Systems zu führen.

In [AMCH19] wird der Prozess der Ausführung eines APT-Angriffs in fünf Stufen unterteilt. Die erste Stufe ist *Reconnaissance*. Damit ist das Auskundschaften des Zielsystems gemeint, was generell den Beginn jedes erfolgreichen Angriffs markiert. Je mehr Angreifer über das Zielsystem erfahren, desto höher wird deren Erfolgsrate.

Die zweite Stufe ist *Establish Foothold*. Dies beschreibt den erfolgreichen und permanenten Zugang zum Zielsystem /-netz.

Die dritte Stufe *Lateral Movement/Staying Undetected* beschreibt das unentdeckte Ausspionieren des Systems nach kritischen Informationen und Komponenten.

Die vierte Stufe ist *Exfiltration/Impedient*. In diesem Schritt führen Angreifer die eigentliche schadhafte Aktivität aus (abhängig vom Ziel des Angriffs), wie das Senden von kritischen Informationen an von Angreifern kontrollierte Rechnern oder das Zerstören von kritischen Komponenten. In der fünften und letzten Stufe

Post-Exfiltration/Post-Impedient sind die Aktivitäten zusammengefasst, die nach dem erfolgreich ausgeführten Angriff folgen. Das sind im Regelfall das Löschen von Spuren und Beweisen, das Weiterführen von Angriffen oder ein sauberes Verlassen des Zielsystems.

Verwendung von Provenance Information zur Threat Detection Im Survey *Data Provenance in Security and Privacy* von Pan et al. [PSR23] werden vier Methoden zur Verwendung von Provenance Information zur Threat Detection festgehalten: *Generic Provenance Tracing*, *Malware Detection*, *Intrusion Detection* und *Analysis and Detection of Security Faults*. *Generic Provenance Tracing* wird typischerweise für die Analyse von bereits erfolgten Angriffen genutzt und wird eher nicht mit der Erkennung von Angriffen in Verbindung gebracht. Es ermöglicht die Analyse von Angriffen durch Erfassen und Sammeln von kausalen Zusammenhängen, um den Ursprung eines Angriffs zu erkennen, den Angriffsverlauf zu bestimmen und den Schaden abzuschätzen. *Malware Detection* bezeichnet Detection Systeme, die sich auf das Auffinden schadhafter Software fokussieren.

Im Gegensatz dazu wird bei *Intrusion Detection* versucht bösartige Aktivitäten und Angriffe zu identifizieren, unabhängig von der Präsenz schadhafter Software. Im Regelfall wird dabei ein Modell mit normalem bzw. harmlosem Systemverhalten genutzt, um Anomalien basierend auf den Abweichungen zum Normalverhalten zu erkennen. Zur *Intrusion Detection* zählt ebenfalls die Erkennung von Advanced Persistent Threat (APT). Für die *Intrusion Detection* werden Provenance Graphen für anomaliebasierte *Intrusion Detection* Systeme verwendet. Dabei wird *harmloses* Normalverhalten des Betriebssystems, das im Provenance Graphen modelliert ist, genutzt, um ein Modell zur Klassifikation von harmlosem Verhalten zu trainieren. Das trainierte Modell kann für neue Provenance Graphen Anomalien vom *gutartigen* Systemverhalten klassifizieren. Die Abweichungen stellen die Anomalien, bzw. das potenziell schadhafte Systemverhalten dar [CLL⁺23].

Analysis and Detection of Security Faults ist ein weiteres Einsatzgebiet von Provenance Information zur Threat Detection. Dabei wird versucht Sicherheitslücken wie Data Exfiltration oder Privacy Policy Violations zu entdecken.

Die Erzeugung von Provenance Information erfolgt durch Konvertierung der Logdaten von Systemen. Für die Erzeugung von Provenance Graphen für Betriebssysteme werden

in der Regel eigene Logging Mechanismen wie für Windows das Event Tracing for Windows (ETW) oder für Linux das Linux Security Module (LSM) [MG16]. Es existieren außerdem wissenschaftliche *Provenance Capture Mechanism* wie bspw. *CamFlow* von Pasquier et al. [PHG⁺17]. Hier werden die Logdaten und Events des LSM und NetFilter genutzt, um Provenance Information zu erzeugen.

Provenance Information kann in unterschiedlichen Granularitäten gesammelt werden. In 'Threat Detection and Investigation with System-level Provenance Graphs: A Survey' [LCYC21] werden unterschiedliche Granularitäten vorgestellt. *Grob-granulare Provenance Collection* beschreibt Provenance Information, die nur auf Systemebene erfasst wird, wie Dateizugriffe und Kommunikation zwischen Prozessen, bspw. modelliert durch das PROV-DM. *Fein-granulare Provenance Collection* beschreibt das Sammeln von Daten zur Nachverfolgung des Informationsflusses. Es abstrahiert von den konkreten Aktionen auf Systemebene, um eine Nachvollziehbarkeit des Informationsflusses ohne einzelne Aktionen darzustellen. Außerdem löst fein-granulare Provenance Collection das Problem der *Dependence Explosion*, welches beschreibt, dass viele harmlose Knoten in einem Provenance Graphen als potenziell gefährlich markiert werden können bei Knoten mit vielen eingehenden Kanten. Wenn ein Knoten n eingehende Kanten und m ausgehende Kanten besitzt, dann existieren potenziell $n \times m$ mögliche Informationsflüsse. Fein-granulare Provenance Collection zielt darauf ab, die ein- und ausgehenden Kanten genauer zuzuordnen und so den potenziellen Informationsfluss auf $n+m$ zu reduzieren.

Als angemessene Granularität gelten System-level Provenance Graphs [LCYC21]. In dieser Granularität ist ausreichend Daten- und Informationsfluss über das Betriebssystem für die Detektion enthalten und zusätzlich bietet diese Granularität genügend Kontext.

2.2. Vorstellung verschiedener DBMS

Im vorherigen Abschnitt wurde erläutert, dass Provenance Information zur Threat Detection und Angriffserkennung notwendig sind. Für Testszenarien mit kleinen Subgraphen genügt die Speicherung der Provenance Information im Hauptspeicher. In realistischen Szenarien, bspw. in einer großen Systemlandschaft in der riesige Datenmengen erzeugt werden, reicht der Hauptspeicher jedoch nicht mehr aus. In [LCYC21] von Li et al. wird beschrieben, dass ein typisches System in einer Bank mit 20.000 Rechnern jährlich etwa 70 Petabyte an Logdaten erzeugt werden. Daher wird ein DBMS benötigt, um Provenance Information zu verwalten. In diesem Abschnitt werden verschiedene Arten von relationalen und nicht-relationalen (NoSQL) DBMS hinsichtlich ihrer Vor- und Nachteile kurz vorgestellt und es erfolgt eine Beurteilung über die Tauglichkeit zur Verwaltung von Provenance Graphen.

Die Inhalte dieses Abschnitts zu NoSQL Datenbanken stammen aus *NoSQL Distilled - A Brief Guide to the Emerging World of Polyglot Persistence* [SF13]. Eine Zusammenfassung verschiedener DBMS findet sich auf der Webseite *Database of Databases*¹⁰.

Relationale Datenbanken Relationale DBMS (RDBMS) basieren auf Relationen und relationaler Algebra. Daten werden in Tabellen, Spalten und Zeilen nach einem festzulegenden Datenschema verwaltet. Im Datenschema wird definiert welche Daten in welchen Tabellen gespeichert werden, welche Beziehungen zwischen Daten existieren, welche Datentypen verwendet werden und es können Bedingungen (Constraints) formuliert werden. Eine Tabelle besitzt Spalten und Zeilen. In den Spalten werden die zu speichernden Attribute verwaltet. Ein Datensatz mit den jeweiligen Attributwerten wird in einer Zeile gespeichert. Identifiziert wird eine Zeile in einer Tabelle durch einen Primärschlüssel. Beziehungen zwischen Daten werden durch Referenzen (Foreign Keys) auf den jeweiligen Primärschlüssel eines Datensatzes umgesetzt. Durch Normalisierung werden Redundanz und Abhängigkeiten zwischen Daten eliminiert und Integrität der Daten gewährleistet. Relationale Datenbanken sind seit langem (erste Entwicklung etwa 1960-1970) im Einsatz und gelten als Standardmodell, sind also sehr ausgereift. Als Querysprache besitzen relationale Datenbanken Structured Query Language (SQL).

Bekannte Produkte von RDBMS sind Oracle DB¹¹, PostgreSQL¹² oder Microsoft SQL Server (MSSQL)¹³. Die Vorteile von RDBMS liegen in der hohen Konsistenz und Integrität der verwalteten Daten, die durch das Datenschema, Normalisierung und ACID-konforme Transaktionen gewährleistet wird. Weiter bieten RDBMS autorisierte Zugriff auf Daten durch Role Based Access Control (RBAC). Die Querysprache SQL ermöglicht die Formulierung von flexiblen Abfragen. Die Nachteile von RDBMS liegen zunächst in der Inflexibilität des Datenschemas. Änderungen am Schema sind nur aufwändig umzusetzen. RDBMS sind auf Redundanzfreiheit optimiert, da sie zu einer Zeit entwickelt wurden, in der Speicherplatz teuer war. Aus diesem Grund mangelt es den Produkten im Kern an Eigenschaften, die für Verteilte Systeme notwendig sind wie Redundanz oder einem loseren Datenschema. Dazu sei erwähnt, dass heutzutage Produkte relationaler Datenbanken Möglichkeiten zur Skalierung und Verteilung auf mehrere Maschinen bieten. Jedoch ist die Umsetzung häufig komplex, da RDBMS im Kern nicht für die Verteilte Systeme entwickelt wurden.

Relationale Datenbanken eignen sich zur Speicherung von Graphdaten nur bedingt. Sie sind darauf optimiert Daten strukturiert und tabellarische Daten zu verwalten und nicht darauf die Beziehung zwischen Entitäten abzufragen. Abfragen auf Grapheigenschaften,

¹⁰Übersicht verschiedener DBMS - <https://dbdb.io/>

¹¹Oracle Homepage - <https://www.oracle.com/de/database/>

¹²PostgreSQL Homepage - <https://www.postgresql.org/>

¹³MSSQL Homepage - <https://www.microsoft.com/de-de/sql-server/sql-server-2019>

besonders die für Provenance Information relevanten Eigenschaften, wie k-Hop Nachbarschaften und die Ermittlung aller Vorgänger bzw. Nachfolger eines Knoten in beliebiger Tiefe, werden durch relationale Datenbanken nicht direkt unterstützt, sondern müssen nachimplementiert werden. Es ist jedoch sinnvoll RDBMS mit in die Auswahl dieser Untersuchung zu nehmen, da sie als de-facto Standard zur Datenverwaltung gelten und damit eine Art Basis für den Vergleich verschiedener Systeme bilden. Außerdem existieren einige Erweiterungen bspw. von Microsoft *SQL Graph database*¹⁴, die Funktionalitäten für das Verwalten von Graphdaten liefert. Zusätzlich speichern RDBMS Daten auf der Festplatte und ermöglichen schnellen Zugriff durch Caching, was bei Graphdatenbanken nicht der Fall ist [LCYC21].

RDBMS eignen sich sehr gut für Anwendungsfälle in denen die Konsistenz der Daten als höchste Anforderung steht, wie bspw. bei Unternehmenssoftware oder Anwendungen im Finanzbereich. Außerdem sind RDBMS der de facto Standard bei der Wahl eines DBMS zur Verwaltung von Daten. Die Nachteile in der Skalierbarkeit und im Einsatz zur Graphverarbeitung motivieren einen Blick auf weitere nicht-relationale DBMS (NoSQL).

Graph Datenbanken Graphdatenbanken ermöglichen das Speichern von Entitäten und deren Beziehungen. Instanzen einer Entität werden als Knoten, Beziehungen zwischen den Entitäten als Kanten im Graphen modelliert. Die Kanten können gerichtet oder ungerichtet sein. Abfragen auf Graphdatenbanken sind letztendlich ein Traversieren des Graphen über Knoten und Kanten, und das Selektieren derjenigen Knoten, welche die gesuchten Attribute besitzen. Das Durchlaufen des Graphen geschieht sehr schnell, da die Beziehung eines Knoten nicht berechnet werden müssen, sondern persistiert werden. Ein bekanntes Produkt für Graphdatenbanken ist Neo4j¹⁵. Neo4j ist ACID-konform, besitzt also Transaktionen. Neo4j kann eine gewisse Verfügbarkeit sicherstellen durch Slave-Replikation des Master-Nodes. Als Querysprache besitzt Neo4j Cypher.

Gut geeignet sind Graphdatenbanken, um implizite Verknüpfungen zwischen Entitäten explizit darzustellen, bspw. für Daten sozialer Netzwerke und Knowledge Graphen. Weiterhin eignen sich Graphdatenbanken gut für Routing, Disposition und Ort-basierte Services. Jeder Ort oder Adresse kann bspw. als Knoten gespeichert werden, um so einen Graphen aus allen Lieferadressen zu haben. Die Beziehung zwischen den Knoten könnte die Eigenschaft Distanz beinhalten. Auf diese können Routing Algorithmen wie *Shortest-Path* verwendet werden, um die optimale Route von Lieferadressen zu ermitteln.

¹⁴Link zu MSSQL Erweiterungen - <https://learn.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-architecture?view=sql-server-ver16>

¹⁵Neo4j Homepage - <https://neo4j.com/>

Graphdatenbanken eignen sich generell nicht, wenn häufig Schreiboperationen auf allen oder eine Teilmenge der Entitäten durchgeführt werden müssen. Das Ändern der Eigenschaften von Knoten ist in Graphdatenbanken keine triviale Operation. Auch bei großen Datenmengen oder Operationen auf dem gesamten Graphen bieten Graphdatenbanken eher eine schlechte Performance.

Es erscheint naheliegend, die Daten eines Graphenmodells in einer Graphdatenbank zu verwalten. Zunächst ist die Speicherung der Daten als Knoten und Graphen passend für ein Graphenmodell. Auch das Modellieren der Herkunftsbeziehungen zwischen Knoten als Beziehung in Neo4j ist passend, da Neo4j darauf optimiert ist Beziehungen abzufragen. Für den Einsatz spricht, dass Neo4j ebenfalls auf Abfragen zur Graphverarbeitung wie z. B Graphtraversierung oder Shortest-Path optimiert ist.

Da Neo4j aber Schwächen in der Skalierung und Stabilität im Umgang mit großen Datenmengen aufweist, ist zu prüfen wie performant Neo4j in einem Echt-System, wie in Abbildung 1.1 dargestellt, wäre.

3. Verwandte Arbeiten

In diesem Kapitel werden wissenschaftliche Artikel und Projekte vorgestellt, die sich mit der Speicherung und Verwaltung von Provenance Graphen beschäftigt haben. Die vorgestellten Arbeiten sind chronologisch geordnet, um die Entwicklung der Forschung darzustellen.

A Comparison of a Graph Database and a Relational Database - A Data Provenance Perspective In [VMZ⁺10] aus dem Jahr 2010 wurde ein relationes DBMS (MySQL) mit einem NoSQL Graphdatenbank (Neo4j) für die Speicherung von Provenance Graphen verglichen.

Für die Durchführung von vergleichbaren Experimenten wurden objektive und subjektive Kriterien bestimmt. Als objektive Kriterien werden zwei Typen von Abfragen definiert. Zum einen strukturelle Abfragen, welche die Struktur des Graphen prüfen, zum anderen Datenabfragen, welche den Inhalt der Knoten und deren Attribute abfragen.

Für strukturelle Abfragen wurden die folgenden drei Abfragen formuliert:

- s1 Finde alle *Singleton*-Knoten im Graphen (alle isolierten Knoten).
- s2 Durchlaufe den Graphen bis zu einer Tiefe von 4 und zähle die erreichbaren Knoten.
- s3 Durchlaufe den Graphen bis zu einer Tiefe von 128 und zähle die erreichbaren Knoten.

Für Datenabfragen wurden die folgenden drei Abfragen formuliert:

- d1 Zähle die Knoten, bei denen ein Attributwert größer ist als Wert X .
- d2 Zähle die Knoten, bei denen ein Attributwert kleiner ist als Wert X .
- d3 Zähle die Anzahl, bei denen ein Attributwert einen Such-String S beinhaltet.

Als subjektive Kriterien werden die vier Kriterien *Ausgereiftheit und der Umfang an Support, die Komplexität im Umgang und der Entwicklung des DBMS, die Flexibilität und Sicherheit* bestimmt. Ausgereiftheit und Umfang an Support beschreibt die Stabilität des Systems, wie ausführlich das System getestet wurde und den Umfang an Support von Dokumentation bis zu Foren und einer großen Nutzerbasis. Die Komplexität im Umgang und Entwicklung des DBMS beschreibt wie komplex die Installation und Erlernbarkeit

des Systems und der Programmiersprachen und /-werkzeuge ist. Flexibilität meint in diesem Sinne, wie performant sich das DBMS außerhalb der Umgebung verhält, für das es entwickelt wurde. Dies bezieht sich z. B. auf die Anpassbarkeit des Datenschemas oder den Mehraufwand für die Verwaltung der Infrastruktur, auf der das DBMS in Betrieb ist. Die Sicherheit beschreibt die eingebauten Sicherheitskomponenten der DBMS wie RBAC.

Der Kern des Vergleichs des Papers ist, dass beide Systeme in den durchgeführten Versuchen ähnlich performant waren. Die Graphdatenbank schnitt bei strukturellen Abfragen wie Graphtraversierung und bei Abfragen auf die Payload von Knoten besser ab. Die Tatsache, dass Index Mechanismen in Neo4j auf Strings basieren, führt jedoch dazu, dass numerische Abfragen deutlich langsamer auf Graphdatenbanken ausgeführt werden, als auf relationalen. Begründet ist der Unterschied dadurch, dass Neo4j basierend auf Apache Lucene, einer Bibliothek zur Volltext-Suche in Java, implementiert ist.

Zum Zeitpunkt der Veröffentlichung in 2011 lautet das Ergebnis, dass der Einsatz von Neo4j zum Verwalten von Provenance Graphen in produktiven Umgebungen nicht geeignet ist. Begründet ist das zum einen durch die längeren Laufzeiten für numerische Abfragen und zum anderen an der fehlenden Möglichkeit zur Sicherung der Daten in Neo4j.

Aus diesem Paper sind einige Ansätze für die Evaluation inspiriert. Für die eigene Arbeit wird die Methodik des Vergleichs der Datenbanksysteme übernommen. Außerdem hat sich Neo4j grundlegend weiterentwickelt und bietet nun die Funktionalitäten, die in dem Paper bemängelt wurden wie fehlende Sicherheitsaspekte durch RBAC.

Applications of provenance in performance prediction and data storage optimisation In dieser Arbeit von Woodman et. al [WH17] aus dem Jahr 2017 beschäftigt sich mit der Erfassung von Provenance Information in medizinischen Anwendungen, der Verwaltung der Daten in einer Graphdatenbank und der Formulierung von Abfragen zur Unterstützung von Audit Fragestellungen.

In dem Paper werden vier Abfragen für die Analyse von Provenance Information definiert.

- Q1 Finde alle direkten und indirekten Abhängigkeiten für Knoten D .
- Q2 Finde alle Berechnungen und Knoten zur Erzeugung von Knoten D (Ermittlung der Vorgänger bzw. *Backward Tracking*).
- Q3 Finde alle Knoten, die von Knoten D abstammen (Ermittlung der Nachfolger bzw. *Forward Tracking*).
- Q4 Sind die Ergebnisse die gleichen, wenn die Berechnung der Herkunftskette wiederholt wird?

Q1 gilt als die grundlegende Abfrage, die Provenance Management Systeme unterstützen sollten. Die Abfrage selbst hat geringere Aussagekraft, als die folgenden Abfragen Q2 und Q3, wird jedoch für deren Formulierung benötigt. Q2 wird genutzt, um die Präsenz eines Knotens zu rechtfertigen, da die Vorgänger eines Knotens D im Provenance Graphen die kausalen Zusammenhänge zur Entstehung von D darstellen. Q3 wird genutzt, um die Auswirkungen eines Knotens zu analysieren, da die Nachfolger eines Knotens D im Provenance Graphen den Einfluss von D auf spätere Daten darstellen. Q4 beantwortet die Frage, welche Auswirkungen das Ändern von Diensten oder Versionen bei der Erzeugung der Daten hat.

Der Provenance Graph wird basierend auf dem OPM Version 1.1 modelliert. Zum Speichern und Verwalten der Provenance Information wird eine Neo4j Datenbank genutzt. Die Wahl für Neo4j wird durch das Datenmodell eines Graphen zum passenden Aufbau der Graph-Datenbank mit Knoten, Beziehungen und Eigenschaften begründet. Zusätzlich wird die Wahl für Neo4j durch die Performanz von Abfragen auf die Struktur des Graphen begründet.

Aus dieser Arbeit werden die Abfragen für die forensische Analyse inspiriert. Diese Abfragen werden später u.a. dafür genutzt, um die Formulierung der Abfragen sowie die Latenzen bei der Ausführung der Abfragen auf den zu untersuchenden DBMS zu erfassen.

Storage and Querying of Large Provenance Graphs Using NoSQL DSE In dieser Arbeit aus dem Jahr 2020 [Kas20] wird DataStax Enterprise¹, ein Column-Family Store basierend auf Cassandra, zur Verwaltung von Provenance Graphen verwendet. Um eine geringe AbfrageLatenz zu erreichen wird in [Kas20] ein Speicherschema und die Erzeugung von Indizes vorgeschlagen, was performante Abfragen auf Provenance Graphen ermöglicht.

Als Datenmodell wird das PROV-DM verwendet. Für performante Abfragen werden zwei Indizes vorgeschlagen, mithilfe derer die Knoten und Kanten des Provenance Graphen, welche einen kausalen Zusammenhang bilden, in den selben Blöcken auf der Festplatte gespeichert werden. Kausaler Zusammenhang meint die Herkunftsbeziehungen wie bspw. *wasGeneratedBy*. Das Speichern der Kanten und Knoten einer kausalen Kette in gleichen Blöcken ist sinnvoll, da diese im Regelfall gemeinsam abgefragt werden.

Für die Nutzung des Indizes werden entscheidende Merkmale von Column-Family Stores wie Cassandra genutzt. Cassandra speichert auf einem Node im Cluster eine Menge von Partitionen, welche auf weitere Nodes repliziert werden. Zunächst werden die Indizes, die Kopien aller Vorgänger und Nachfolger eines Knoten beinhalten, in einer Partition gespeichert. Da eine Partition immer komplett auf einem Knoten im Cluster gespeichert

¹Link zur DataStax Webseite - <https://www.datastax.com/de/products/datastax-enterprise>

wird, können alle Ursachen und Effekte (Vorgänger und Nachfolger im ProvGraphen) durch Zugriff auf eine einzelne Partition abgefragt werden.

Das Finden der Vorgänger bzw. Nachfolger mithilfe des Indexes geschieht dann über das Finden des gewünschten Partition Keys in der jeweiligen Tabelle. Wenn bspw. die Vorgänger des Knoten c gesucht sind genügt es den Partition Key c in der Tabelle I_A zu suchen. Die Abfragen nach dem Partition Key c in der Tabelle I_A würde die Knoten und Kanten $P_I = a, b, c$ ergeben und damit alle Vorgänger von c .

Für die Evaluation wurden zwei Datenzugriffsmuster für Provenance Daten verwendet. 1) Finde alle Nachfolger zu einem beliebigen Knoten. 2) Finde alle Vorgänger zu einem beliebigen Knoten. Die Zugriffsmuster sind referenziert aus *Applications of provenance in performance prediction and data storage optimisation* [WH17].

Diese Arbeit schlägt als weiteres DBMS Cassandra vor und zeigt, dass die optimale Speicherung von Provenance Daten aktuelles Forschungsgebiet ist. Die Suche nach Vorgängern und Nachfolgern ist für GNN Architekturen zur Intrusion Detection weniger relevant, als beispielsweise das Finden der k-Hop Nachbarschaft, oder das Graph Sampling. Im Fall von Threat Analyse ist die Suche nach Vorgängern und Nachfolgern jedoch interessant, wenn es darum ausgehend von einem potenziell schadhaften Knoten die Nachfolger zu durchlaufen.

Assessing the computational limits of GraphDBs' engines - A comparison study between Neo4j and Apache Spark In dieser Arbeit von Ballas et. al [BTPT21] aus dem Jahr 2021 wird Neo4j mit Apache Spark für die Verwaltung von Graphdaten verglichen. Das Ergebnis dieser Arbeit ist, dass die Leistung von Neo4j durch den physischen Arbeitsspeicher limitiert ist. Wenn Daten über die Grenzen des Arbeitsspeichers hinaus verwaltet werden müssen, ist eine skalierbare Lösung wie bspw. Apache Spark zu bevorzugen. In der Abfragegeschwindigkeit schneidet Apache Spark besser als Neo4j ab. Neo4j ist jedoch in der Verwaltung, Installation und der Abfrageschnittstelle einfacher zu bedienen.

4. Methodik

In diesem Kapitel wird die Methodik für die Untersuchung zur Auswahl eines DBMS zur Verwaltung von Provenance Graphen erläutert. Zunächst werden die Ziele einer solchen Datenverwaltung beschrieben. Zu den Zielen werden Szenarien definiert, aus denen sich Anforderungen ableiten. Zusätzlich werden Experimente für das Testen der Anforderungen formuliert. Die Experimente dienen der Evaluation, ob die geforderten Anforderungen durch die DBMS-Produkte erfüllt werden. Im darauf folgenden Abschnitt werden die zuvor definierten Anforderungen mit verschiedenen DBMS-Produkten abgeglichen, um eine Auswahl an DBMS für die Untersuchung in dieser Arbeit zu treffen. Im anschließenden Abschnitt wird auf Datensätze für Provenance Graphen eingegangen und die Auswahl für den in dieser Arbeit verwendeten Datensatz getroffen.

4.1. Ziele und Anforderungen einer Datenverwaltung für Provenance Graphen

In diesem Abschnitt werden die Anforderung einer Datenverwaltung für Provenance Graphen in einem PIDS erläutert. Zur Ermittlung von Anforderungen in der Anforderungsanalyse werden zunächst Ziele durch Interessenvertreter formuliert. Die in dieser Arbeit erarbeiteten Ziele resultieren jedoch aus den Ergebnissen der Recherche, da sich es sich bei der Datenverwaltung für Provenance Graphen, um ein aktuelles Thema der Forschung handelt und aufgrund des Fehlens *realer* Interessenvertreter aus der Industrie für diese Arbeit. Ein Ziel selbst beschreibt einen Aspekt des zu entwickelnden Systems und wird durch die Formulierung von Szenarien konkretisiert. Aus dem Szenario heraus werden Lösungsanforderungen abgeleitet, die zu erfüllende Eigenschaften der Datenverwaltung beschreiben. Das Szenario überbrückt damit die Lücke zwischen den abstrakteren Zielen und konkreten Lösungsanforderungen. Die einzelnen funktionalen und qualitativen Lösungsanforderungen werden als User Stories (Anwenderbericht) mithilfe von Requirements Template (Satzschablonen) in natürlicher Sprache formuliert. Im Zuge der Bestimmung der Lösungsanforderungen wird weiter darauf eingegangen, wie die geforderte Eigenschaft experimentell zwischen DBMS verglichen werden kann. Zuletzt werden die Anforderungen nach dem Kano-Modell priorisiert [Poh21].

Für die Formulierung der Ziele und Bestimmung der Funktionalitäten einer Datenverwaltung für Provenance Graphen dient die Referenz-Architektur aus [LCYC21] in Abbildung 1.1 (Seite 10) als Orientierung. Anhand der Referenzarchitektur lassen sich erste Ziele für die Datenverwaltung formulieren, indem der geplante Einsatz der Datenverwaltung betrachtet wird.

Der durch das *Data Collection Modul* (Datensammlung) aus Logdaten erzeugte *Streaming Graph* (Provenance Graph) ist die Eingabe für das *Data Management Modul* (Datenverwaltung) und das *Threat Detection Modul* (Gefahrenerkennung). Die Anwendung von *Data Reduction Algorithms* ist im Referenzframework die erste Komponente in der Datenverwaltung und notwendig, um das erzeugte Datenvolumen zu begrenzen. Auf diesen Teil wird in dieser Arbeit nicht weiter eingegangen, da er nicht Bestandteil der Aufgabenstellung ist. Nach der Reduktion der Datenmenge werden die Graphdaten gespeichert, verwaltet und über eine AbfrageSchnittstelle bereitgestellt. Bei der Speicherung und Verwaltung kann bspw. die Erzeugung eines Index für neu hinzugefügte Knoten angewendet werden. Die Gefahrenerkennung nutzt für *Real-Time Stream-based Detection* (Echtzeit-Detektion) den *Streaming Graph*. Methoden der Echtzeit-Detektion sind *Tag Propagation*, *Real-Time Graph Alignment* und *Anomaly Score*. Für Offline-Analysen und Forensik wird über eine Abfrageschnittstelle auf die verwalteten Daten zugegriffen. In der Forensik kommen Algorithmen der Graphverarbeitung zum Einsatz, wie das Bilden von Gruppierungen, das Finden der Vorgänger und Nachfolger eines Knotens oder das Matching von Subgraphen auf den gesamten Provenance Graphen.

In STARC DARPA TC [GKC⁺19] wird ein Szenario für einen *Master Data Store* zur Verwaltung eines Provenance Graphen zur forensischen Analyse beschrieben. Für diese Speicherung werden die folgenden vier Ziele definiert:

- Sequentielle Schreiboperationen ermöglichen, die Echtzeitverarbeitung ermöglichen.
- Reihenfolge der eingehenden Daten erhalten.
- Sehr selten Daten löschen.
- Horizontale Skalierbarkeit des Speichers, um Daten so lange wie möglich zu erhalten.

Die im Rahmen von [GKC⁺19] entwickelte Architektur in Abbildung A.1 hat konzeptionell den folgenden Aufbau. Provenance Daten werden durch verschiedene Teilnehmer (der Datenerzeugung) (*Cadets*, *ClearScope*, *FiveDirections*, *Trace*, *Marple*, *Theia*) aufgezeichnet und von jedem Teilnehmer an ein Kafka Cluster gesendet. Das Kafka Cluster dient als Messaging Instanz und liefert die Daten weiter an die Datenverwaltung. In der Architektur verfügt jeder Teilnehmer (der Threat Detection) über einen eigenen Master Data Store. Weiter wird beschrieben, dass der Master Data Store keine Strukturierung oder Indizierung der Daten vornehmen sollte, um Schreibperformance zu gewährleisten. Die Verarbeitung der gespeicherten Daten erfolgt dann über ein Verarbeitungsframework

in dem komplexe und zeitintensive Berechnungen ausgeführt werden. Das Ergebnis der Verarbeitung wird dann in *indizierten forensischen Sichten* dargestellt. In diesen Sichten wird die Ausgabe der Verarbeitung bereitgestellt, um darauf Abfragen auszuführen. Die in [GKC⁺19] von verschiedenen Performern generierte Provenance Information wurde jedoch nicht in einem Data Store verwaltet, sondern wurde nur an ein Kafka Cluster gesendet und dort belassen. Der geplante Master Data Store kam in den Engagements nicht zum Einsatz.

Eine weitere Eigenschaft der Datenverwaltung ergibt sich aus dem Bedarf die Provenance Information vor Manipulation zu schützen. Die Sicherheitsmaßnahmen zum Schutz der Provenance Information werden in [PSR23] unter dem Begriff *Secure Provenance* zusammengefasst. Bei einem Einsatz der Datenverwaltung zur Überwachung eines Netzwerks bestehend aus mehreren Maschinen ist es außerdem notwendig die Provenance Daten zentral an einem Ort zu speichern, um Analysen über das überwachte Gesamtsystem zu ermöglichen. Die Datenverwaltung muss Sicherheitsmaßnahmen implementieren, um *Secure Provenance* zu gewährleisten.

Unter Berücksichtigung der zuvor beschriebenen Einsatzgebiete für eine Datenverwaltung für Provenance Graphen ergeben sich die folgende Ziele.

Hoher Durchsatz

Der Durchsatz der durch Produzenten im Kafka Cluster in STARC DARPA TC [GKC⁺19] produzierten Daten ergab für eine Datengröße von 300 Byte, mit der Kafka Version 1.0 mit SSL Verschlüsselung mit Avro Serialisierung 15,29 MB/s und ohne Avro Serialisierung 23,45 MB/s. Der Durchsatz der durch Konsumenten im Kafka Cluster konsumierten Daten ergab für eine Datengröße von 300 Byte, mit der Kafka Version 1.0 mit SSL Verschlüsselung ohne Avro Serialisierung 31,02 MB/s, bei geringer Avro Serialisierung 22,4 MB/s und bei hoher Avro Serialisierung 5,17 MB/s. Durch das LPM werden 1.879 Events pro Sekunde protokolliert. Jedes Event erzeugt einen Knoten und eine Kante, was zu 3.758 Graphenelementen führt [MG16]. Dies beschreibt die Menge an Rohdaten, die direkt aus dem LPM übernommen werden. Zusammengeführt mit der Angabe, dass ein Datensatz für ein Graphenelement im CDM-Format durchschnittlich etwa 300 Byte beträgt, ergibt sich eine Schreibrate von 1,127 MB/s, um alle Events des LPM als Graphenelemente sequentiell schreiben zu können.

Anforderung Wenn Daten durch Produzenten von Provenance Graphen mit einer Geschwindigkeit von 1,127 MB/s erzeugt werden, dann soll die Datenverwaltung in der Lage sein die Daten echtzeitfähig zu verarbeiten.

Geplantes Experiment Für das Testen dieser Anforderung wird der Cadets Datensatz in die zu untersuchenden DBMS geschrieben. Dazu wird eine Publisher-Subscriber Architektur implementiert. Je zu evaluierendes DBMS wird ein Subscriber implementiert. Der Publisher liest sequentiell den Datensatz und sendet pro Intervall einen Batch an Zeilen des Datensatzes an den Broker der Architektur. Die Subscriber werden beim Broker angemeldet und haben die Aufgabe eingehende Nachrichten die Zeilen des Datensatzes in Einfügeanweisungen der jeweiligen Abfragesprache umzuwandeln und an die Instanz des DBMS zu senden. Der Durchsatz wird durch die Angabe der Größe des Batch im Publisher gesteuert. Ein Zeile des Datensatzes hat im JSON Format eine Größe zwischen 950 - 1400 Bytes. Um eine Schreibrate von etwa 1,127 MB/s zu imitieren, müssen pro Sekunde 1.500 Datensätze durch den Publisher veröffentlicht werden (bei einer geschätzten durchschnittlichen Dateigröße von $1.000 \text{ B} = 1 \text{ KB} = 0,001 \text{ MB}$ pro Zeile). Das Experiment wird so konstruiert, dass die Batchgröße pro Versuchsdurchlauf schrittweise erhöht wird. Die Batchgröße für das erste Experiment beträgt 500 und wird schrittweise um 500 erhöht, bis die zuvor erläuterte Anforderung erreicht ist.

Skalierbarer Speicher

In STARC DARPA TC [GKC⁺19] lag die Durchschnittsgröße eines Datensatzes im Engagement 3 für Cadets bei 252 Byte. Insgesamt wurden im Engagement 3 über einen Zeitraum von 11 Tagen durch Cadets 80.306.428 Datensätze mit einer Gesamtgröße von etwa 36 GB erzeugt. Im Engagement 5, welches über einen Zeitraum von drei Wochen ausgeführt wurde, wurden verteilt über 24 Rechner etwa 2 Terabyte an CDM Daten erzeugt [GKC⁺19]. Hochgerechnet auf ein Jahr ergibt das eine Datenmenge von etwa 35 Terabyte. In [GKC⁺19] wird, wie bereits erwähnt, angegeben, dass die Erzeugungsrate für Provenance Daten 2,5 MB pro Tag pro Host beträgt. Hochgerechnet auf ein Rechnernetz mit 1.000 Maschinen würde das über eine Dauer von einem halben Jahr 425 TB an rohen Speicherdaten erzeugen. Würden alleine die Rohdaten eines Hosts betrachtet werden, die durch das Linux Provenance Module (LPM) mit einer Rate von 1,27 MB/s Provenance Graphenelemente erzeugt werden, dann würde das für einen Host pro Jahr Rohdaten von 40 Terabyte ergeben, welche noch in Provenance Graphen umgewandelt werden müssen. Pessimistischere Schätzungen wie in *Threat Detection and Investigation with System-level Provenance Graphs: A Survey* von Li et al. [LCYC21] beschreiben, dass ein typisches Unternehmensnetzwerk in einer Bank mit 20.000 Rechnern jährlich etwa 70 Petabyte an Logdaten erzeugen würde.

Anforderung Ausgehend von der Datenwachstumsrate in [GKC⁺19], soll die Datenverwaltung für ein Rechnernetz von etwa 20 Rechnern pro Jahr Datenmengen von zu 50 Terabyte verwalten können.

Geplantes Experiment In dieser Masterarbeit ist diese Anforderung nicht testbar, da für die nötigen Experimente nicht die physischen Kapazitäten nicht zur Verfügung stehen. Um die geforderte Datenmenge von 2,5 Terabyte zu erreichen müsste bei einer Schreibrate von 1,27 MB/s ein Experiment mit einer Dauer von etwa 23 Tagen dauerhaft betrieben werden.

Geringe Latenzen auf Abfragen

In *Practical Whole-System Provenance Capture* [PHG⁺17] wird auf das Problem eingegangen, dass mit belegtem Speicher die Abfragelaufzeit proportional steigt. Als Lösung dafür wird angegeben, die zu speichernde Datenmenge einzugrenzen, indem konfiguriert werden kann, welche System- und Funktionsaufrufe als Knoten und Kanten im Provenance Graphen abgebildet werden. In *High-throughput Ingest of Data Provenance Records into Accumulo* von Moyer et al. aus dem Jahr 2016 [MG16], wird sich sich mit der Verwaltung von Provenance Graphen beschäftigt. Die Datenverwaltung von Provenance Graphen wird dort über *Apache Accumulo*¹ realisiert. Logdaten werden durch das LPM erzeugt und in PROV-DM Daten umgewandelt und anschließend mit dem D4M Schema in TSV Dateien umgewandelt, um Daten in Apache Accumulo zu speichern. Accumulo ist ein verteilter Key-Value Store basierend auf HDFS. Anschließend werden Graphanalyse-Frameworks wie Graphulo (oder GraphChi) genutzt.

Anforderung

- Die Beantwortung von Abfragen soll mit geringer Latenz erfolgen.
- Die Antwortzeiten der Abfragen sollen mit dem Datenvolumen skalieren.

Geplantes Experiment Die Abfragen, die zu den späteren Zielen *Forschungsabfragen* und *Forensische Abfragen* formuliert werden, starten in regelmäßigen Intervallen (z. B alle 5 Minuten). Da durch die Subscriber kontinuierlich Daten in die DBMS geschrieben werden, wächst die Datenmenge mit voranschreitender Zeit. Für die Beurteilung, ob die Abfragelatenz überproportional zum Datenvolumen steigt, wird die Antwortzeit je Abfragein Relation zum Zeitpunkt gesetzt, abhängig davon wann die Abfrage gestartet wird. Ein späterer Zeitpunkt ist dabei gleichzusetzen mit einer höheren Auslastung des Datenvolumens des DBMS.

¹Link zur Accumulo Webseite - <https://accumulo.apache.org/>

Intrusion Detektion

Eine Methode zur Angriffserkennung ist das *Provenance Graph Pattern Matching*, welches in [KEK⁺22] angewendet. Bekannte Angriffe werden als Graphmuster formuliert und als Abfrage an die Datenverwaltung gestellt. Die Antwort auf die Abfrage ergibt, ob ein Subgraph nach dem formulierten Muster existiert. Quellcode 4.1 zeigt eine beispielhafte SPARQL Abfrage für die Entdeckung von ausführbaren Dateien in verdächtigen Ordnern (Ausführung von Dateien an Orten, an denen keine Datei ausgeführt werden sollte). Der Angriff ist als *tactic, technique und procedure* (TTP) durch MITRE ATT&CK als Technik T1204.002² definiert. Die Abfrage sucht nach allen Elementen *s*, die ein Prozess sind und bei denen *s* die Eigenschaft `hasCmd` mit der Ausprägung `"/tmp/t/"`, `"/var/www/"`, `"/home/*/public_html/"` oder `"/usr/local/apache2/"` besitzt. Die Filtern definieren die verdächtigen Ordner, in denen ein beliebiger Prozess nicht ausgeführt werden sollte.

```

1 SELECT ?s ?cmd WHERE {
2   ?s a :Process . ?s :hasCmd ?cmd .
3 FILTER (regex(str(?cmd), "^/tmp/*")
4         || (regex(str(?cmd), "^/var/www/*")
5           || (regex(str(?cmd), "^/home/*/public_html/*")
6             || (regex(str(?cmd), "^/usr/local/apache2/*")
7           )
8         )
9       )

```

Quellcode 4.1: Intrusion Detection Abfrage in SPARQL aus [KEK⁺22].

Um Abfragen zur Echtzeit-Detektion beantworten zu können, darf außerdem keine außerordentlich hohe Latenz zwischen dem Schreiben der Daten und der Beantwortung der Anfrage liegen. Latenzen können z. B. durch die Verwaltung eines Indizes nach dem Schreiben der Daten entstehen.

Anforderung

- Die Abfrageschnittstelle soll Anfragen zum Graph Pattern Matching unterstützen.
- Die Datenverwaltung soll Abfragen zur Echtzeit-Detektion unterstützen.

Geplantes Experiment Diese Anforderungen wird lediglich theoretisch betrachtet. Die experimentelle Untersuchung übersteigt den Rahmen dieser Masterarbeit. Aus selbigen Grund wird die Formulierung der Abfragen nicht weiter betrachtet.

²Link zum MITRE Angriffsmuster - <https://attack.mitre.org/techniques/T1204/003/>

Forensische Abfragen

Gegeben ist ein Szenario, in dem ein Threat Analyst eine Benachrichtigung erhält, dass durch die Intrusion Detektion ein potenziell schadhafter Knoten entdeckt wurde. Um den potenziell schadhafte Knoten und dessen Herkunft und Auswirkungen zu untersuchen, werden Anfragen an die Datenbank bezüglich der Vorgänger (Back Tracking), der Nachfolger (Forward Tracking) oder auch das Finden eines Pfades zu einem beliebigen anderen Knoten gestellt. Das Forward und Backward Tracking gelten als grundlegende Abfragen, die auf Provenance Graphen ausgeführt werden [LCYC21]. Diese Abfragen basieren auf den azyklischen Herkunftsbeziehungen des Provenance Graphen.

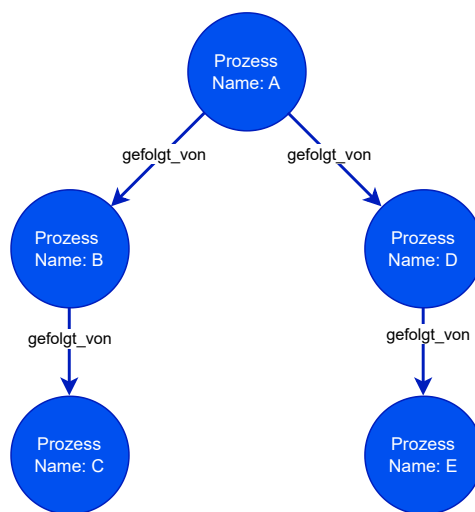


Abbildung 4.1.: Exemplarischer Graph zur Verdeutlichung des Forward Tracking.

Eine beispielhafte Abfrage für das Forward Tracking in der Neo4j Abfrage Sprache Cypher ist in Quellcode 4.2 formuliert. Die Abfrage soll alle Nachfolger eines beliebigen Startknotens bis zu einer Tiefe von zwei liefern. Für den Graphen in Abbildung 4.1 liefert die Abfrage als Ergebnis die Knoten *B, C, D, E*.

```
1 MATCH
2 (start: Prozess)
3 -[:gefolgt_von *1..2]->
4 (nachfolger: Prozess)
5 RETURN nachfolger.name;
```

Quellcode 4.2: Abfrage zum Forward Tracking eines beliebigen Startknoten.

Analog dazu lässt sich das Backward Tracking für alle Vorgänger eines Knotens definieren, mit dem Unterschied, dass in der Abfrage die Kante in umgekehrter Richtung formuliert wird. Die Anwendung von Forward und Backward Tracking ist eine Möglichkeit

mehrere Stufen in einem Angriff zu untersuchen, jedoch hat diese auch Nachteile. Zum einen ist es schwierig durch reines Backward Tracking den normalen vom potenziell schadhafte Datenfluss zu unterscheiden [LCYC21]. Die Ursache des Problem basiert auf der *Dependency Explosion*, was das Phänomen beschreibt, dass im Provenance Graphen viele Knoten mit einem hohen Grad existieren. Dies liegt an der Tatsache, dass viele Prozesse in einem Betriebssystem ablaufen und es initiale Prozesse gibt, die viele andere Prozesse starten, in UNIX Systemen als Daemon Prozesse bezeichnet. In [LCYC21] werden als Abfrageschnittstelle verschiedene Provenance Graph Abfrage Systeme vorgeschlagen, die eigene Abfragesprache - Attack Investigation Query Language (AIQL) [GXL⁺18b] und Stream-based Anomaly Query Language (SAQL) [GXL⁺18a] - zur Anomalieerkennung und Angriffsanalyse definieren.

Anforderung (als User Story) Als Threat Analyst möchte ich Abfragen zum Forward und Backward Tracking an die Abfrageschnittstelle der Datenverwaltung formulieren, die mir die Analyse von potenziellen oder erfolgten Angriffen ermöglichen.

Geplantes Experiment In dieser Masterarbeit wird die jeweilige Abfragesprache des DBMS als Abfrageschnittstelle verwendet. Das Forward und Backward Tracking sowie das Finden des kürzesten Pfades zwischen zwei beliebigen Knoten wird in dem jeweiligen DBMS als Abfrage formuliert. Dabei wird verglichen mit welcher Laufzeit die Abfrage beantwortet wird und wie komplex die Implementierung bzw. die Formulierung der Abfrage ist.

Die Abfragen lauten:

- Finde alle Vorgänger eines Knoten.
- Finde alle Nachfolger eines Knoten.
- Finde die kürzeste Verbindung von einem Startknoten zu einem beliebigen anderen Knoten.

Dauer der Aufbewahrung

Um das gesamte zu verwaltende Datenvolumen zu reduzieren, ist es sinnvoll nach einem gewissen Zeitraum Provenance Daten über harmloses Betriebssystemverhalten zu löschen. Das Löschen der Daten kann über *Retention Policies* (Aufbewahrungsregeln) umgesetzt werden, die festlegen nach welcher Zeit Daten gelöscht werden. In diesen Regeln wird festgelegt in welcher Art Daten aggregiert und/oder komprimiert werden, die ein bestimmtes Alter erreicht haben und damit historisch werden (z. B 30 Tage). Nach

der Aggregation werden die zugehörigen Rohdaten gelöscht und nur die Aggregation gespeichert. Um traditionelle Angriffe zu erkennen, ist es ausreichend Provenance Graphen über einen mittelfristigen Zeitraum von einigen Monaten bis zu einem Jahr zu speichern, da traditionelle Angriffe über einen eher kurzen Zeitraum ausgeführt werden. Im Fall von APT Angriffen, die sich über einen langen Zeitraum erstrecken, siehe Absatz 2.1, ist es notwendig Provenance Graphen über einen Zeitraum von bis zu mehreren Jahren zu speichern, um das Systemverhalten auf APT Bedrohungen überwachen zu können. Hier gilt es die entsprechende Abwägung zwischen höherer Sicherheit durch längere Aufbewahrung der Daten und Aufwand der Verwaltung und Performance-Einbußen zu finden.

Im Artikel *M-Trends 2021 Fireeye Mandiant Services - Special Report*³ wird eine *Dwell Time* angegeben. Sie beschreibt die Anzahl der Tage vom Zeitpunkt, als sich ein Angreifer Zugriff zu einem System verschafft, bis zur Entdeckung des Angriffs. Im globalen Durchschnitt betrug die Zeit 2020 bei interner Detektion des Angriffs 12 Tage und bei externer Benachrichtigung über einen Angriff (bspw. durch den Angreifer) 73 Tage. In 2018 betrug die *Dwell Time* für interne Detektion 50,5 Tage, bei externer Benachrichtigung 184 Tage. In [GKC⁺19] wird das Verhältnis zwischen Speicheranforderung und *Retention Policies* betrachtet. Dafür wird ein Szenario konstruiert, in dem 75 Rechner acht Stunden am Tag Rechner von Personen genutzt werden und den Rest der Zeit im Standby sind. Beachtet wurde auch, dass etwa 10 Personen die Rechner teilweise oder nicht benutzen, da diese zu der Zeit telefonieren. 60 Rechner werden mit Windows und 10 mit Linux Betriebssystemen betrieben. Der Zuwachs pro halbem Jahr beträgt in diesem Szenario linear 9 Terabyte. Innerhalb von drei Jahren ergibt sich damit eine zu speichernde Datenmenge von 54 Terabyte.

Anforderung Die Datenverwaltung soll Provenance Graphen über einen Zeitraum von bis zu drei Jahren verwalten.

Geplantes Experiment Diese Anforderung wird nur theoretisch betrachtet. Aufgrund der langfristigen Eigenschaft von drei Jahren nicht im Rahmen der sechsmonatigen Arbeit experimentell überprüft werden.

Forschungsabfragen

Die Datenverwaltung soll Abfragen zur Überprüfung und Weiterentwicklung von PIDS unterstützen. Häufige Abfragen, die Forscher bei der Entwicklung von Neuronalen Netzen PIDS benötigen, sind die 1-Hop und die 2-Hop Nachbarschaft von Knoten. Die 1-2 Hop Nachbarschaft wird genutzt, um Graphembeddings für Knoten zu berechnen. Die

³Link zum Artikel - <https://www.mandiant.com/resources/reports/m-trends-2021>

Graphembeddings wiederum bilden die numerische Grundlage für Graph Neuronale Netze. Um die Vorhersagen und Ergebnisse von GNNs nachzuvollziehen kann es hilfreich sein sich die Nachbarschaften des zum Training verwendeten Datensatzes anzuschauen.

Anforderung (als User Story) Als Entwickler oder Forscher von PIDS möchte ich Abfragen wie Graph Sampling oder 2-Hop Nachbarschaft formulieren, um Ergebnisse meiner Deep Learning Modelle nachzuvollziehen und zu verifizieren.

Geplantes Experiment Die 2-Hop Nachbarschaft zu einem beliebigen Knoten wird als Abfrage im jeweiligen DBMS formuliert. Dabei wird die Laufzeit zur Beantwortung der Abfrage und die Komplexität der Implementierung und Formulierung verglichen.

Secure Provenance

In [PSR23] werden verschiedene Angriffsszenarien auf Provenance Daten formuliert, wie das Erzeugen gefälschter Provenance Daten oder Ändern von bereits erfassten Daten. Weitere Angriffstaktiken beinhalten das Löschen der Provenance Daten oder die Kompromittierung des Datenschutzes in dem Sinne, dass sensitive Daten über die Provenance Capture Mechanismen preisgegeben werden. Die erfassten Provenance Daten müssen daher mindestens durch kryptographische Verfahren vor nicht autorisierten und nicht authentisierten Anfragen geschützt sein. In [PSR23] werden weiterhin *Blockchain-based Secure Provenance* Systeme vorgestellt.

Anforderung Die Datenverwaltung soll Operationen auf der Datenbank nur von authentifizierten und autorisierten Benutzern ausführen.

Geplantes Experiment Dieses Ziel wird in dieser Arbeit nicht durch Experimente untersucht, da deren Umfang den Rahmen dieser Arbeit übersteigen würde.

Zentrale Verwaltung

Die Datenverwaltung soll die Daten mehrerer Rechner an einem zentralen Ort speichern. Die zentrale Verwaltung vereinfacht den Umgang mit Provenance Graphen, die in einem Netz mehrerer Rechner entstehen. Außerdem ermöglicht eine zentrale Datenverwaltung, welche Provenance Graphen mehrerer Hosts vereint, Abfragen über mehrere Hostsysteme und dadurch umfangreichere Analysemöglichkeiten. [MG16].

Anforderung Die Datenverwaltung soll die Provenance Daten von verschiedenen Hosts zentral an einem Ort integrieren.

Geplantes Experiment Dieses Ziel wird nicht untersucht, da dessen Erfüllung den Rahmen dieser Arbeit übersteigen würde.

DSGVO

Die Datenverwaltung muss eine Nachvollziehbarkeit von Entscheidungen gewährleisten können. Gegen die Verwendung von ausschließlich maschinengestützten Entscheidungen zur Strafverfolgung gilt in Deutschland die Datenschutzgrundverordnung (DSGVO) Art. 22 Satz 1⁴ in der es heißt:

Die betroffene Person hat das Recht, nicht einer ausschließlich auf einer automatisierten Verarbeitung – einschließlich Profiling – beruhenden Entscheidung unterworfen zu werden, die ihr gegenüber rechtliche Wirkung entfaltet oder sie in ähnlicher Weise erheblich beeinträchtigt.

Das bedeutet, dass eine ausschließlich maschinengestützte Entscheidung vor Gericht nicht haltbar ist. Daher liegt ein Interesse darin die Nachvollziehbarkeit der Entscheidungen von PIDS zu gewährleisten. Eine Möglichkeit ist mit Hilfe von Provenance Graphen einer Person die Möglichkeit zu geben, einen Angriff auf ein System analysieren zu können und durch genügend Kontextinformationen aussagekräftige Schlüsse zu ziehen.

Anforderung Die Datenverwaltung soll im Fall eines als potenziell schadhaft identifizierten Knotens dessen Nachbarschaft und Kausalitätskette exportieren, um im Fall einer Strafverfolgung einen Nachweis über die Intrusion liefern zu können.

Geplantes Experiment Diese Anforderung wird in dieser Arbeit nicht überprüft, da dessen Erfüllung den Rahmen der Arbeit übersteigen würde.

Zusammengefasst ergeben sich somit für eine Datenverwaltung für Provenance Graphen die Ziele, die in Tabelle 4.1 aufgelistet sind.

⁴Link zu Auszug DSGVO - <https://dsgvo-gesetz.de/art-22-dsgvo/>

Tabelle 4.1: Ziele einer Datenverwaltung für Provenance Graphen.

Kurzbezeichnung	Beschreibung
Hoher Schreibdurchsatz	Die Datenverwaltung soll einen hohen Durchsatz an Schreiboperationen ermöglichen.
Skalierbarer Speicher	Die Datenverwaltung soll enorm große Datenmengen verwalten können.
Geringe Antwortzeiten auf Abfragen	Die Datenverwaltung darf bei wachsender Datenmenge nicht erheblich an Performance einbüßen und soll performant auf Abfragen antworten.
Intrusion Detektion	Die Datenverwaltung soll Abfragen zur Echtzeit-Detektion von Angriffen ermöglichen und in Echtzeit (weniger als einer Minute nach Erzeugung der Daten beantworten).
Forensische Abfragen	Die Datenverwaltung soll Abfragen zur Offline Analyse ermöglichen und innerhalb von einer Stunde beantworten können.
Dauer der Aufbewahrung	Die Datenverwaltung soll Daten für eine Dauer von drei Jahren speichern, um eine Nachvollziehbarkeit von Entscheidungen gewährleisten zu können.
Forschungsabfragen	Die Datenverwaltung soll Abfragen zur Überprüfung und Weiterentwicklung von PIDS unterstützen.
Secure Provenance	Die Datenverwaltung sollte Sicherheitsmaßnahmen gewährleisten, um <i>Provenance Poisoning</i> zu verhindern.
Zentrale Verwaltung	Die Datenverwaltung soll die Daten mehrerer Rechner an einem zentralen Ort speichern.
DSGVO	Bei Betrieb der Datenverwaltung sollen DSGVO Vorschriften eingehalten werden.

4.2. Bestimmung der Priorität der Anforderungen

Im folgenden Abschnitt wird die Priorität der zuvor formulierten Anforderungen bestimmt. Priorität lässt sich über mehrere Wege definieren. Als Ranking (Top Ten) nach einem beliebigen Kriterium, über Klassifizierung z. B. nach den Klassen des Kano Modells (Basisfaktor (essentiell), Leistungsfaktor (konditional), Begeisterungsfaktor (optional)), oder über Quantifizierung der Anforderungen z. B. nach der Wieger'schen Priorisierungsmatrix mit Gewichten für relative Kosten, Nutzen und Wert jeder Anforderung [Poh21]. Die Priorisierung der Anforderungen ist zum Großteil subjektiv, da die Priorität durch Interessen der Stakeholder eines zu entwickelnden Systems bestimmt wird. In dieser

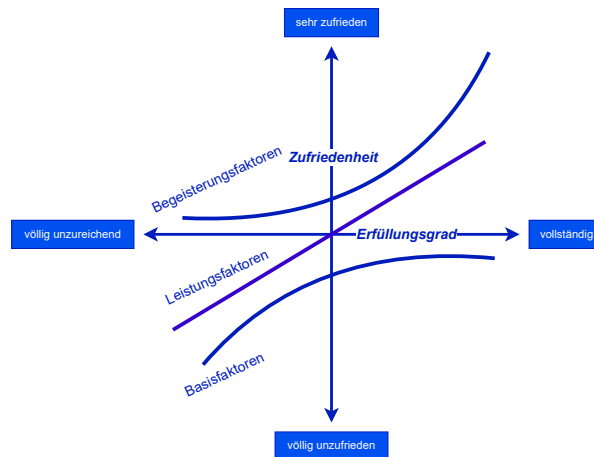


Abbildung 4.2.: Kano Modell vgl. [Poh21].

Arbeit wird als grundsätzliches Interesse aller Stakeholder angenommen, dass die Entwicklung eines performanten Prototyps das oberste Ziel ist.

Eine Priorisierung der Anforderungen durch Klassifizierung nach Basis-, Leistungs-, und Begeisterungsfaktoren bietet sich in dieser Arbeit an, da die Machbarkeit und die Eignung unterschiedlicher DBMS für die Speicherung von Provenance Graphen untersucht wird. Die essentiellen Basisfaktoren und die daraus resultierenden Eigenschaften des Systems dienen als Entscheidungsgrundlage für die Auswahl von möglichen DBMS für die Experimente in dieser Arbeit.

Das Kano-Modell Das Kano-Modell dient der Klassifizierung und Priorisierung von Anforderungen. Es setzt die Zufriedenheit ins Verhältnis zum Erfüllungsgrad einer Anforderung. Es können drei unterschiedliche Faktoren für Anforderungen bestimmt werden; Basis-, Leistungs- und Begeisterungsfaktoren. Basisfaktoren sind die grundsätzlichen Anforderungen, die als selbstverständlich vorausgesetzte Systemmerkmale unbedingt erfüllt werden müssen. Die Leistungsfaktoren sind explizit geforderte Systemmerkmale, deren Erfüllung die Zufriedenheit proportional erfüllt. Die Begeisterungsfaktoren beschreiben zusätzliche optionale Systemmerkmale, die während der Benutzung als angenehme und nützliche Überraschung entdeckt werden und die Zufriedenheit überproportional erfüllen.

Als Basisfaktoren werden die folgenden Anforderungen in Tabelle 4.2 identifiziert. Dies sind die Anforderungen, die von einer Datenverwaltung für Provenance Graphen unbedingt erfüllt werden müssen, um als performanter Prototyp zu gelten. Ein *hoher Schreibdurchsatz* ist Basis für die weiteren Abläufe der Datenverwaltung. Die hohe Rate mit der Provenance Daten erzeugt werden, macht diese Anforderung essentiell für ein performantes System. Die *geringe Latenz bei hohem Datenvolumen* ist ebenfalls eine Anforderung,

die unbedingt erfüllt werden muss und aus dem enormen Datenvolumen resultiert, dass bei der Modellierung von Betriebssystemen als Provenance Graphen entsteht. Die *geringe Latenz bei Schreiblast* ist ein weiteres essentielles Merkmal der Datenverwaltung. Die Datenverwaltung muss in der Lage sein auch bei sehr großen Datenmengen und hoher Auslastung performant Operationen durchzuführen. Die *Offline Analyse* ist die grundlegende Analyse, die auf Provenance Graphen ausgeführt wird, um erfolgte Angriffe oder Gefahren forensisch zu betrachten. Es wird von der Datenverwaltung erwartet mindestens die dafür nötigen Abfragen und Graphverarbeitungs-Algorithmen zu unterstützen. Die Möglichkeit zur Formulierung von *Abfragen zur Forschung* ist ein weiterer Basisfaktor. Für den Einsatz einer Datenverwaltung ist dieses Systemmerkmal erforderlich, um die Weiterentwicklung und Forschung von Intrusion Detektion Möglichkeiten durch bspw. Graph Neuronale Netze zu unterstützen.

Tabelle 4.2: Übersicht der Basisfaktoren der Datenverwaltung.

Bezeichnung Ziel	Anforderung	Kriterium
Hoher Durchsatz	Erzeugte Provenance Daten im Strom echtzeitfähig verarbeiten.	Geschwindigkeit von 1,5 MB/s.
Geringe Latenz bei Schreiblast	Geringen Latenz bei hoher Schreiblast.	Antworten auf Abfragen in weniger als zehn Minuten.
Geringe Latenz bei Datenvolumen	Latenz steigt nicht überproportional zum Datenvolumen.	Antworten auf Abfragen in weniger als zehn Minuten.
Offline Analyse	Abfragen zum Forward, Backward Tracking und Tiefensuche.	Unkomplizierte Formulierung und Implementierung der Abfragen.
Forschungsabfragen	Abfragen zur 2-Hop Nachbarschaft ermöglichen.	Unkomplizierte Formulierung und Implementierung der Abfragen.

Als Leistungsfaktoren werden die folgenden Anforderungen klassifiziert, die in Tabelle 4.3 zusammengefasst sind. Zu den Leistungsfaktoren zählen diejenigen Anforderungen, die von Stakeholdern gefordert werden, aber nicht als selbstverständlich angenommen werden. Die *Streaming*-Verarbeitung von Graphdaten ist die Basis für eine Echtzeit-Verarbeitung der Graphdaten. Durch die Verarbeitung der Daten als Strom kontinuierlicher Daten wird die Latenz, die bei der Zusammenfassung und Verarbeitung der Daten als Batch entsteht, vermieden. Da diese Anforderung die Grundlage für die Echtzeit-

Erkennung darstellt zählt diese zu den Leistungsfaktoren. Die *Intrusion Detektion* ist eine weitere Funktionalität, die von der Datenverwaltung gefordert wird. Aufgrund der Komplexität der Intrusion Detektion zählt diese Anforderung zu den Leistungsfaktoren einer Datenverwaltung. Die *Echtzeit-Detektion* ist ein weiterer Leistungsfaktor, da die Echtzeit-Verarbeitung der Daten die grundlegenden Eigenschaften der Datenverwaltung übersteigt. Die *Dauer der Aufbewahrung* beschreibt die Dauer, wie lange Provenance Daten verwaltet werden müssen, um zuverlässig auch langfristige Angriffe wie APTs zu erkennen. Die Dauer beeinflusst die insgesamt zu speichernde Datenmenge, daher zählt diese Anforderung zu den geforderten Merkmalen, anstelle der Basisfaktoren.

Tabelle 4.3: Übersicht der Leistungsfaktoren der Datenverwaltung.

Bezeichnung Ziel	Anforderung
Intrusion Detektion	Graph Pattern Matching ermöglichen.
Echtzeit-Detektion	Regelmäßige Ausführung von Abfragen.
Dauer Aufbewahrung	Daten sollen drei Jahre aufbewahrt werden.
Skalierbarer Speicher	50 TB an CDM Daten pro Jahr für 20 Hosts.

Als Begeisterungsfaktoren werden die folgenden Anforderungen bestimmt, die in Tabelle 4.4 aufgeführt sind. Diese Anforderungen sind für eine funktionsfähige Datenverwaltung nicht zwingend notwendig. Deren Erfüllung steigert jedoch erheblich die Zufriedenheit der Stakeholder des Systems. *Secure Provenance* beschreibt die Eigenschaft der Sicherheit der erzeugten und aufgezeichneten Provenance Daten. Aus der Sicht für den Erfolg eines performanten Prototypen ist dieses Systemmerkmal kein Basis- oder Leistungsfaktor. Dennoch besitzt diese Anforderung eine enorm hohe Bedeutung und könnte bei unterschiedlicher Betrachtungsweise anders priorisiert werden. Die Anforderung einer *Zentralen Verwaltung* und zur *DSGVO* ist für einen Prototypen ebenfalls von optionaler Bedeutung.

Tabelle 4.4: Übersicht der Begeisterungsfaktoren der Datenverwaltung.

Bezeichnung Ziel	Anforderung
Secure Provenance	Authentisierung und Autorisierung ermöglichen.
Zentrale Verwaltung	Integration Provenance Daten mehrerer Hosts.
DSGVO	Dokumentieren von schadhaften Knoten.

4.3. Auswahl von DBMS

Im folgenden Abschnitt werden die DBMS festgelegt, die im weiteren Verlauf dieser Arbeit auf die Eignung als Datenverwaltung für Provenance Graphen untersucht und verglichen werden. Die im vorherigen Abschnitt definierten Ziele und Anforderungen bilden die Grundlage für die Auswahl. Aus den Zielen wurden im vorherigen Abschnitt die Eigenschaften definiert, die durch ein DBMS erfüllt werden sollten.

In einem Popularitätsranking⁵ aus Dezember 2023 belegen relationale Datenbanken die ersten vier Plätze (Oracle, MySQL, MS-SQL Server und PostgreSQL). Dies bezeugt die Bedeutung und Verbreitung von RDBMS als de-facto Standard-Lösung in der Datenverwaltung. Für die Untersuchung in dieser Arbeit ist es daher sinnvoll RDBMS als *Baseline* für den Vergleich mit weiteren DBMS aufzunehmen. In dieser Arbeit wird PostgreSQL als Produkt für RDBMS verwendet, da PostgreSQL weit verbreitet ist und als Open-Source Lösung zur Verfügung steht.

Als zweites DBMS wird Neo4j ausgewählt. Die Nutzung von Neo4j für die Verwaltung von Provenance Graphen ist naheliegend, da es sich bei Provenance Graphen um Graphdaten handelt welche optimal zum Datenmodell der Datenbank passen. Die Eigenschaften von Neo4j sind in Abschnitt 2.2 beschrieben.

Als drittes DBMS wird in der Arbeit die Graphdatenbank Open Native Graph Database (ONgDB) betrachtet. Bei ONgDB handelt es sich um einen Neo4j Fork, welche im Kern die Enterprise-Version von Neo4j als Open-Source Lösung zur Verfügung stellt. Wie Neo4j bietet ONgDB Clustering, ACID Transaktionen, und eine auf Graphoperationen optimierte Abfragesprache⁶. Aufgrund der Verfügbarkeit als frei verfügbare Open Source Variante wird ONgDB als DBMS in dieser Untersuchung mit aufgenommen.

Als viertes DBMS wird Memgraph betrachtet. Memgraph ist ebenfalls eine Graphdatenbank, die im Gegensatz zu Neo4j und OngDB nicht in Java, sondern in C++ implementiert ist. Dadurch versprechen die Entwickler laut eigener Aussage bessere Geschwindigkeiten als Neo4j. Die Community Edition von Memgraph ist ebenfalls kostenlos zur Verfügung gestellt. Außerdem verfügt Memgraph über eine gut ausgearbeitete Dokumentation. Aufgrund des Versprechens deutlich schnellere Antwortzeiten als Neo4j bieten zu können, wird Memgraph in dieser Arbeit ebenfalls untersucht.

Weitere potenzielle Kandidaten In diesem Abschnitt werden weitere mögliche DBMS Kandidaten beschrieben, die jedoch in dieser Arbeit nicht untersucht werden. Diese Kandidaten ergeben sich aus der Literaturrecherche und bieten eine Grundlage für weitere Ansätze und Experimente.

⁵Popularitätsranking DBMS - https://db-engines.com/en/ranking_trend

⁶Link zur ONgDB Homepage - <https://graphfoundation.org/ongdb/>

In [HWdS⁺17] (2017) wurden drei DBMS, Cassandra (Column-Family), MongoDB (Dokumentenbasiert) und OrientDB (Graphdatenbank) implementiert und auf die Benutzbarkeit für die Verwaltung von Provenance Graphen für wissenschaftliche Workflows untersucht.

Die Verwendung von MongoDB bietet sich aufgrund des einfachen Mappings des Datenbankmodells zu den Datensätzen an. Jede Zeile im Datensatz ist ein eigenes Dokument. MongoDB verfügt über eine Graph-Lookup-Aggregation Pipeline Stage, welche das eigentlich dokumentenbasierte DBMS um die Funktionalitäten erweitert, die von Graphdatenbanken erwartet werden⁷. Durch die zusätzliche Graph-Lookup Aggregation Stage ist zu prüfen, ob sich MongoDB als Graphdatenbank einsetzen lässt. MongoDB fällt jedoch als Kandidat aus. Zwar ist das Importieren von Daten in MongoDB sehr komfortabel, da der für erste Versuche verwendete Datensatz in Dokumentenform ist. Allerdings erweist sich die versprochene Graphaggregation Pipeline in ersten Versuchen als unflexibel und ist nur für simple Graphtraversierung geeignet.

Wie bereits zuvor erwähnt wird in [HWdS⁺17] auch Cassandra (Column-Family) als DBMS für die Verwaltung von Provenance Graphen vorgeschlagen. In [Kas20] wird Cassandra (bereitgestellt über DataStax) genutzt und die Erzeugung eines Index vorgeschlagen. Cassandra hat den Vorteil eine hohe Schreibrate und geringe Antwortzeiten zu gewährleisten. Außerdem eignet sich Cassandra sehr gut für horizontale Skalierung. Cassandra ist daher ein weiterer interessanter Kandidat für die Verwaltung von Provenance Graphen. Aufgrund von Zeitmangel und Einschränkung der Aufgabenstellung wird Cassandra jedoch nicht als DBMS für Datenverwaltung von Provenance Graphen in dieser Arbeit untersucht.

In [GKC⁺19] wird HDFS als Datenverwaltung vorgeschlagen, wurde in [GKC⁺19] jedoch nicht eingesetzt. Die Gründe dafür sind, dass keine robuste Kafka und HDFS Implementierung rechtzeitig bereitgestellt wurde und dass die generierte Datenmenge klein genug war, um auf Kafka Servern verwaltet zu werden. (2 TB an Daten über drei Wochen, alles auf Kafka Servern). Eine Architektur für ein mögliches HDFS Cluster wird ebenfalls definiert. Der Einsatz von HDFS für die Datenverwaltung hat den Vorteil der Kompatibilität mit Verarbeitungsframeworks wie Apache Spark. Ebenfalls aufgrund von Zeitmangel und Einschränkung der Aufgabenstellung wird HDFS jedoch nicht als DBMS in dieser Arbeit untersucht.

In [GKC⁺19] wird zusätzlich erwähnt, dass die Implementierung einer TSDB als Langzeit Datenverwaltung zwar nicht beachtet wurde, jedoch aufgrund der Möglichkeiten zur Analyse und Visualisierung von Zeitreihendaten vorteilhaft gewesen wäre. Es wird angenommen, dass die Nutzung einer TSDB gegenüber den Lösungen verschiedener Teilnehmer von Vorteil gewesen wäre. Der Nachteil der TSDB ist, dass Zeitreihen sich

⁷Link zur MongoDB Graph Extension: <https://www.mongodb.com/databases/mongodb-graph-database>

weniger gut für die Modellierung von Graphdaten eignen. Aufgrund von Zeitmangel und Einschränkung der Aufgabenstellung wird TSDB nicht als DBMS in dieser Arbeit untersucht, jedoch sind TSDB für Untersuchungen in aufbauenden Arbeiten interessant.

4.4. Der eingesetzte Provenance Graph Datensatz

Es stehen eine Reihe von Datensätzen mit Provenance Daten zur Verfügung, die häufiger in der Wissenschaft verwendet werden. In *Provenance-based Intrusion Detection Systems: A Survey* [ZGCD23] werden einige öffentlich verfügbare Benchmark Datensätze aufgeführt. Darunter auch der STARC DARPA TC Engagement 3 Datensatz. Der gesamte Datensatz, welcher aus den aufgezeichneten Daten mehrerer Teilnehmer besteht, beinhaltet insgesamt über 2 Milliarden Events und wurde über 13 Tage aufgezeichnet. Die STARC DARPA TC Datensätze eignen sich gut, da das Verhalten eines Betriebssystems in einem möglichst realitätsnahem Szenario aufgezeichnet wurde. Konkret drückt sich das darin aus, dass innerhalb der Zeit in dem das Betriebssystem aufgezeichnet wurde, Angriffe auf das System durchgeführt wurden. Von diesen Angriffen sind konkret zwei Multi-Stage Angriffsszenarien im Datensatz enthalten. Außerdem handelt es sich bei dem *Cadets* Datensatz durch eine Ground Truth Datei, um einen gelabelten Datensatz, weshalb er sich für das Training mit Machine Learning und Deep Learning Modellen eignet. Nachteil beim Verwenden der Engagement 3 Datensätze ist jedoch, dass die Dokumentation ungenügend ist.

Generell besteht in der Forschung ein Mangel an öffentlich verfügbaren Datensätzen zur Evaluation von PIDS. In [ZGCD23] werden einige aktuelle Herausforderungen bei der Erstellung von Provenance Datensätzen vorgestellt. Zunächst besteht die Schwierigkeit darin einen Datensatz zu erzeugen, der unter nahezu realistische Bedingungen aufgezeichnet wurde. Häufige Veränderungen in Angriffen und Verteidigungstaktiken führen dazu, dass Datensätze in kurzer Zeit obsolet werden und nicht mehr aktuellen Angriffsmustern entsprechen. Ein weiteres Problem ist das Verhältnis zwischen harmlosen und schadhaftem Systemverhalten. Zum einen spiegelt das Verhältnis nicht die Realität wieder, da Benchmark Datensätze dazu tendieren einen höheren Anteil an schadhaftem Verhalten zu beinhalten. Zum anderen sind harmlose Daten häufig synthetisch erzeugt, was die Qualität der Daten mindert. Synthetische Daten sind allerdings notwendig, da aus Gründen der Privatsphäre und sensibler Daten das Sammeln von *Echt-Daten* problematisch ist. Eine weitere Herausforderung ist der Mangel an Dokumentationen und Werkzeuge, um aufgezeichnete Daten zu verwenden.

Von den am Engagement 3 involvierten Teilnehmern wurde der Datensatz des Teilnehmers *Cadets* ausgewählt. Dieser Datensatz wird in der Forschung häufig verwendet [HPB⁺20], [KEK⁺22], [CLL⁺23] und es existiert bereits Vorwissen über diesen Datensatz aufgrund der Verwendung im wissenschaftlichen Projekten *SecDER* der Hochschule

Hannover. Der Fokus in vorherigen Arbeiten lag jedoch auf dem Training von neuronalen Netzen zur Angriffserkennung und nicht auf der Verwaltung der Daten. Da der Datensatz dem STARC DARPA TC Projekt entstammt, ist er nach dem CDM modelliert, das in Abschnitt 2.1 beschrieben wurde. Der Cadets Datensatz beinhaltet die Knotentypen *Event*, *FileObject*, *Host*, *NetFlowObject*, *Principal*, *SrcSinkObject*, *Subject* und *UnnammedPipeObject*, welche ebenso im CDM definiert sind. Jede Zeile im Datensatz entspricht einem Knoten im Graphen. Der Datensatz besitzt insgesamt über 49 Millionen Zeilen. Das Verhältnis zwischen *harmlosen* und *schadhaften* Knoten beträgt 0,001% [YLJ19]. Dieser Datensatz wird für die in dieser Arbeit durchgeführten Experimente mit den zuvor ausgewählten DBMS verwendet. Im folgenden Kapitel wird auf die durchgeführten Versuche eingegangen.

5. Durchführung der Experimente

Dieses Kapitel beschreibt die Implementierung und Durchführung der in Abschnitt 4.1 definierten Experimente. Für die Entwicklung einer Testumgebung und die Durchführung der Experimente dient der Einsatz einer Datenverwaltung in einem realen Szenario aus Abbildung 1.1 (Seite 10) als Inspiration.

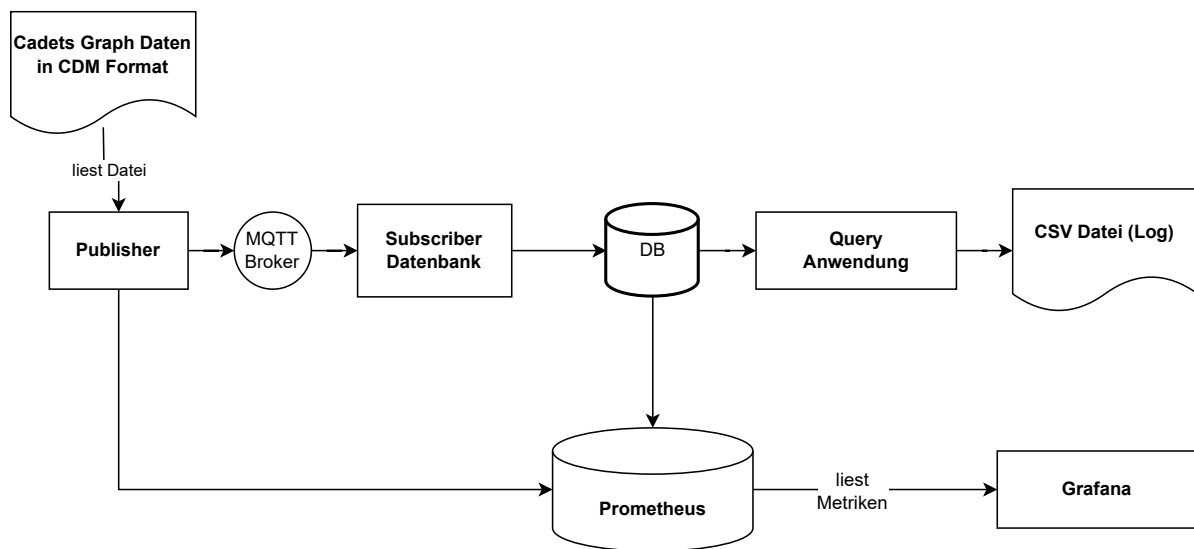


Abbildung 5.1.: Konzeptionelle Veranschaulichung der Testumgebung.

In Abbildung 5.1 wird die konzeptuelle Architektur, der in dieser Masterarbeit entwickelten Testumgebung, dargestellt. Jedes zu untersuchende DBMS wird auf einem Host installiert. Eine Publish-Subscriber Architektur wird implementiert, um die Erzeugung von Provenance Daten in einem realen Umfeld zu imitieren. Ein Publisher veröffentlicht kontinuierlich pro Intervall einen Batch des Cadets Datensatzes per MQTT-Nachricht an einen Broker. Die Veränderung der Größe des Batches sowie die Dauer eines Intervalls ermöglicht die Steuerung unterschiedlicher Erzeugungsraten von Graphdaten. Für jedes zu untersuchende DBMS wird ein Subscriber implementiert, der die Zeilen eines Batch vom Broker empfängt, in Einfügeanweisungen des jeweiligen DBMS umwandelt und ausführt. Für jede im Batch enthaltene Zeile erzeugt jeder Subscriber eine einzelne Einfügeanweisung und fügt die Daten ein. Bei einer Batchgröße von bspw. 500 werden so 500 Anweisungen erzeugt und an die Datenbank gesendet. Dies geschieht aus dem

folgenden Grund; während in PostgreSQL neue Knoten und Kanten jeweils einer Relation hinzugefügt werden können, ist dies bei Graphdatenbanken nicht möglich. Um einen zusammenhängenden Graphen zu gewährleisten, ist es notwendig die Zeilen des Datensatzes, die im Strom erzeugt werden, sequentiell zu verarbeiten. Ansonsten können für einen neu hinzuzufügenden Knoten keine Kanten zu existierenden Knoten erzeugt werden.

Der Quellcode 5.1 für das Einfügen neuer Knoten in Graphdatenbanken mithilfe der Abfragesprache Cypher und der Quellcode 5.2 für das Einfügen neuer Knoten und Kanten in einer relationalen Datenbank mithilfe von SQL veranschaulicht den Unterschied der Einfügeoperation für Graphdatenbanken. Für das Erzeugen einer Kante zu einem existierenden Knoten `existing_node` muss beim Einfügen eines neuen Knotens `new_node` zunächst über ein `MATCH` der existierende Knoten selektiert werden, um im `CREATE` verwendet werden zu können. Im SQL Statement hingegen werden die Knoten und Kanten als Zeile einer Relation gespeichert und benötigen keine zusätzliche Selektier-Operation.

```
1 WITH new_node
2 MATCH (existing_node) WHERE existing._uuid = <uuid>
3 CREATE (new_node) -[:<edge_type>]-> (existing_node)
```

Quellcode 5.1: Cypher Einfügeanweisung zur Knoten- und Kantenerzeugung.

```
1 -- Insert into node_list
2 INSERT INTO node_list (node_no, uuid, type)
3 VALUES ( nextval('node_number_seq'), <uuid_node>, <node_type> );
4
5 -- Insert into edge_list
6 INSERT INTO edge_list (edge_no, source, dest, edge_type )
7 VALUES nextval('edge_number_seq'), <uuid_source_node>,
   ↪ <uuid_dest_node>, <edge_type> ;
```

Quellcode 5.2: SQL Einfügeanweisung zur Knoten- und Kantenerzeugung.

Der Publisher und die Subscriber sind in Python implementiert. Die Publisher-Subscriber Architektur wurde mithilfe der Python Bibliothek *paho-mqtt*¹ realisiert, einer Implementierung des MQTT Protokolls. Für den MQTT Broker wurde eine Message Queue eingerichtet, die dafür sorgt, dass Nachrichten bis zur Verarbeitung zwischengespeichert werden. Die Größe der Queue beträgt ≈ 40 Millionen (die Größe des Datensatzes), um zu gewährleisten das keine Daten verloren gehen. Zusätzlich ist konfiguriert, dass der Broker jede Nachricht genau einmal sendet, was zusätzlich Datenverluste verhindern soll.

Um die Laufzeiten von Abfragen aus den Zielen *Forschungsabfragen* und *Forensische Abfragen* zu überwachen, sind pro DBMS jeweils Anwendungen implementiert. Die Anwendungen senden die Abfragen in regelmäßigen Abständen an ihre jeweilige Datenbank.

¹Link zur Bibliothek paho-mqtt - <https://pypi.org/project/paho-mqtt/>

Es werden dabei die internen Datenbank-Antwortzeiten und die Verarbeitungszeit der zurückgegebenen Daten jeder Abfrage in jeweiligen CSV Dateien gespeichert. Aus den CSV Dateien werden nach der Durchführung der Experimente Grafiken zur Auswertung und Interpretation der Antwortzeiten generiert. Die Abfragen werden in den Sprachen SQL und Cypher formuliert. Alle zu untersuchenden Graphdatenbanken unterstützen die Abfragesprache Cypher, so dass die Anwendungen zum Senden der Cypher-Abfragen wiederverwendet werden können.

Zur Verwaltung der erfassten Hardware-Metriken der Host-Systeme der DBMS und der Publish-Subscriber Architektur wird eine Zeitreihendatenbank verwendet. Zeitreihendatenbanken sind gut geeignet, da diese auf die Verwaltung gering-dimensionaler Daten, welche als Primärschlüssel einen Zeitstempel besitzen, optimiert sind². Als Zeitreihendatenbank wird in der Testumgebung *Prometheus* genutzt. Zur grafischen Darstellung der erfassten Metriken wird die Visualisierungssoftware *Grafana* verwendet. *Grafana* bietet eine Anbindung an *Prometheus* als Datenquelle und ermöglicht die Visualisierung der erfassten Metriken mit geringen Zusatzaufwand durch vorgefertigte Dashboards³.

Alle Komponenten der Testumgebung werden mithilfe von Docker aufgesetzt. Die Verwendung von Docker erlaubt den plattformunabhängigen Betrieb der Testumgebungen und ein horizontales⁴ bzw. vertikales Skalieren, durch *Umzug* der Testumgebung auf einem leistungsfähigerem Host, der Systemressourcen. Zusätzlich ermöglicht die Implementierung und Bereitstellung der Subscriber und Abfrageanwendungen über Docker-Images, dass weitere Datenbanken, die SQL oder Cypher unterstützen, der Testumgebung ohne Implementierung neuer Anwendungen hinzugefügt werden können. Die Anwendungen der Testumgebung werden in einer Docker-Compose Datei definiert. Die entwickelte Testumgebung wird in einem Git Repository zur Versionsverwaltung bereitgestellt⁵.

Die Experimente werden auf einem ASUS Notebook mit 24GiB RAM und einem Intel Core i7-7700HQ CPU @ 2,8GHz Prozessor durchgeführt. Die Ausführung eines Experimente erfolgt durch Start einer Docker-Compose Datei, welches die Docker-Dienste zur Durchführung des Experiments für ein DBMS startet. Dazu gehören Publisher, Broker, DBMS, Subscriber der DBMS, Abfrageanwendung des DBMS, Prometheus und Grafana. Nach Beendigung eines Experiments werden die Docker-Dienste heruntergefahren und eine neue Infrastruktur für das nächste DBMS gestartet. Die Experimente werden sequentiell durchgeführt, um Vergleichbarkeit zwischen den untersuchten DBMS herzustellen und Störungen untereinander bei bspw. gleichzeitigem Betrieb aller DBMS auszuschließen.

²Link zu Eigenschaften von TSDB - <https://dbdb.io/db/prometheus>

³Link zu Grafana Dashboards - <https://grafana.com/grafana/dashboards/>

⁴Horizontales Skalieren mit Docker - <https://docs.docker.com/reference/cli/docker/service/scale/>

⁵Git Repository zur Testumgebung - https://lab.it.hs-hannover.de/cwy-p8d-u1/ma_code

Ein Experiment läuft für zwei Stunden. Die Experimente lassen sich über die Größe des Batches steuern, das Intervall mit dem ein Batch veröffentlicht wird und das Intervall, in dem die Abfragen gestartet werden. Das erste Experiment für ein DBMS beginnt mit einer Batchgröße von 500. Für die nachfolgenden Experimente wird die Batchgröße jeweils um 500 erhöht, bis eine Batchgröße von 1.500 erreicht ist. Danach beginnen die Experimente für das nächste DBMS wieder mit der Batchgröße von 500. Das Intervall des Sendens eines Batches bleibt für alle Experimente konstant bei einer Sekunde. Das Intervall zum Absenden der Abfragen bleibt ebenfalls konstant bei fünf Minuten.

Es ergibt sich eine Gesamtzahl von 12 Experimenten (4 DBMS \times 3 Batchgrößen). Bei einer Laufzeit von zwei Stunden pro Experiment benötigt die Durchführung aller Experimente 24 Stunden. Eine weitere Variation des Sendeintervalls und der Batchgröße wurde aus Zeitgründen weggelassen. Mithilfe der Durchführung eines Experiments werden die Metriken zum Ressourcenverbrauch und Abfragemetriken für ein DBMS erzeugt. Die Ergebnisse der Experimente werden im folgenden Kapitel erläutert und diskutiert.

6. Ergebnisse der Experimente

In diesem Kapitel folgt die Beschreibung der Experimentenergebnisse. Zuerst wird überprüft ob die geforderten Schreibgeschwindigkeiten mithilfe der Testumgebung simuliert können. Danach werden Ergebnisse zur Untersuchung der Ressourcenauslastung der DBMS ausgewertet. Anschließend werden die Abfragen zu den Zielen in Abschnitt 4.1 untersucht und verglichen. Dabei wird vor allem betrachtet wie komplex die Formulierung ist. Zuletzt werden die Laufzeiten der Abfragen untersucht. Die Auswertung der Experimente zeigt, wie sich die Antwortzeiten der Abfragen entwickeln, wenn die Datenbank bereits mit Daten gefüllt ist.

6.1. Auswertung der Schreibgeschwindigkeiten

Aus dem Ziel *Hoher Durchsatz* stammt die Anforderung eine Schreibgeschwindigkeit von 1,12 MB/s zu erreichen. Wie in Abschnitt 4.1 ermittelt, wird erwartet, dass diese Schreibrate bei einer Batchgröße von etwa 1.500 und einem Sendeintervall von einer Sekunde erreicht wird. Die Tabelle A.1 (Seite 97) zeigt die gesendeten Netzwerkraten der Publish-Subscriber Architektur gruppiert nach Batchgröße. Es ist zu erkennen, dass der Publisher je nach Batchgröße Graphdaten in steigenden Erzeugungsraten generiert. Bei einer Batchgröße von 1.500 wird die geforderte Datenrate von 1,124 MB/s erreicht und überschritten. Es fällt jedoch auf, dass die übertragenen Daten nicht durch die Broker zu den jeweiligen Subscribern in der gleichen Geschwindigkeit weitergegeben werden. In Tabelle A.2 (Seite 97) ist zusätzlich der reduzierte empfangene Netzwerkverkehr der vier DBMS dargestellt.

Die Testumgebung weist an dieser Stelle einen Flaschenhals auf. Es fällt auf, dass besonders PostgreSQL eine höhere Datenmenge empfängt als die Graphdatenbanken. Wie in Kapitel 5 erläutert, lässt sich eine Begründung dafür in der konzeptuellen Art der Einfügeoperation finden, welche zu schnelleren Einfügeoperation bei PostgreSQL führt. Außerdem ist erkennbar, dass die Batchgröße nicht den erwarteten Einfluss auf die Menge der in den Datenbanken eingefügten Daten hat, sondern die empfangenen Datenraten pro Datenbank je nach Batchgröße nahezu ähnlich sind, wie in Tabelle A.1 (Seite 97) ersichtlich ist. Dies lässt sich auf die Art der Einfügeoperation durch die Subscriber

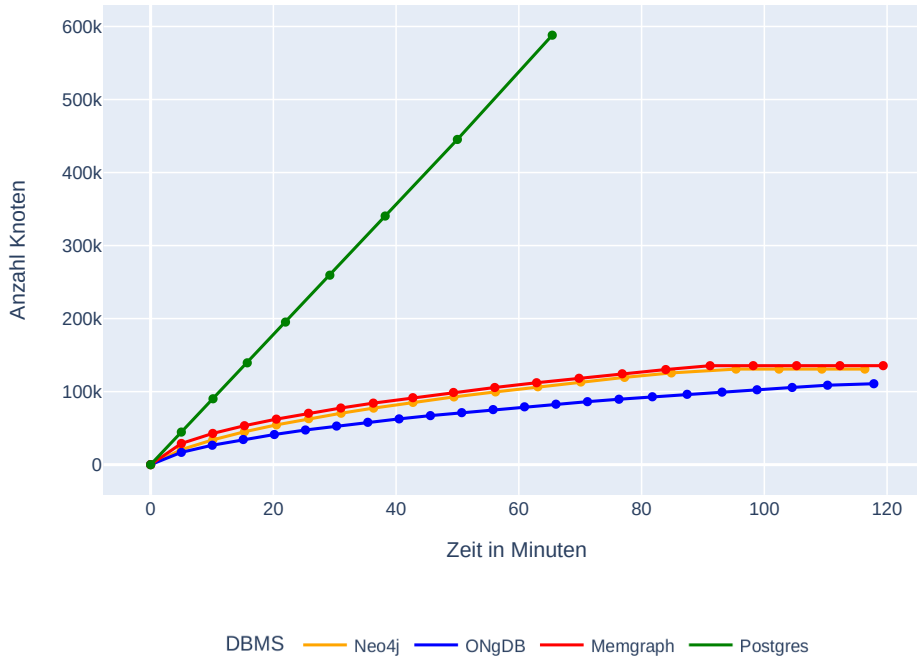


Abbildung 6.1.: Anzahl Knoten je DBMS über Experimentendauer.

zurückzuführen, bei der für jede Zeile eines Batches eine eigene Abfrage ausgeführt wird, anstelle alle Zeilen eines Batches in einer Einfügequery zusammenzuführen.

Weiterhin begründet die unterschiedliche Empfangsrate, die Unterschiede in der Ergebnismenge der Abfragen in Abschnitt 6.4. Bei Einfügeoperationen ohne Latenzen sollten sich rechnerisch nach einer Experimentendauer von zwei Stunden 3.600.000 Knoten in einer Datenbanken befinden, siehe Gleichung 6.1. Eine Zeile im Datensatz bedeutet einen eingefügten Knoten in der Datenbank.

$$(3600 \text{ Sekunden} \times 2) \times 500 \frac{\text{Zeilen}}{\text{Sekunde}} = 3.600.000 \text{ Zeilen} \quad (6.1)$$

In Abbildung 6.1 ist die tatsächliche Entwicklung der in den DBMS enthaltenen Knoten für die Experimente mit der Batchgröße 500 zu sehen. Auf der X-Achse ist die Zeit in Minuten ab Start eines Experiments abgebildet. Statt der theoretisch maximal 3.600.000 eingefügten (nach einer Stunde theoretisch 1.800.000) Knoten befinden sich in PostgreSQL nach 65 Minuten 588.099 Knoten. Das bedeutet, dass im Durchschnitt 150,79 Zeilen pro

Sekunde eingefügt wurden. In Tabelle 6.1 sind die durchschnittlich pro Sekunde eingefügten Knoten zusammengefasst. Für die Batchgröße 500 befinden sich in Memgraph nach 63 Minuten 112.163 Knoten was durchschnittlich etwa 29,67 Knoten pro Sekunde entspricht in Neo4j befinden sich nach 63 Minuten 106.661 Knoten, was durchschnittlich etwa 28,21 Knoten pro Sekunde entspricht, und in ONgDB befinden sich nach 66 Minuten 82.767 Knoten, was durchschnittlich etwa 21,22 Knoten entspricht. Ein weiteres Indiz für die in der Datenbank eingefügten Knoten sind die Senderaten der Broker an die Subscriber je Datenbank, die in Tabelle A.1 zusammengefasst sind. Der PostgreSQL Broker hat mit einer Rate von bspw. 0,155 MB/s gesendet, was bei 1 kB pro Knoten, 155 Knoten pro Sekunde entsprechen würde. Der Neo4j Broker sendet mit einer Rate von durchschnittlich 0,036 MB/s, was etwa 36 Knoten pro Sekunde entspricht. Memgraph liegt ebenfalls bei 0,036 MB/s und damit 36 Knoten pro Sekunde und ONgDB bei 0,029 MB/s was 29 Knoten pro Sekunde entspricht. Diese stimmen in etwa mit den tatsächlich in den DBMS enthaltenen Werten überein. Für die Batchgrößen 1.000 und 1.500 ist durchschnittliche Einfügerate ebenfalls in Tabelle 6.1 zusammengefasst. Es ist ersichtlich, dass bei einer Batchgröße von 1.000 nur die Raten für PostgreSQL und Memgraph leicht erhöht sind. Für Neo4j und ONgDB ist die Rate bereits kleiner als bei einer Batchgröße von 500. Bei einer Batchgröße von 1.500 sind die Einfügeraten aller DBMS kleiner bei einer Batchgröße von 500.

Die Entwicklung der Datenmenge ist in PostgreSQL linear. In den Graphdatenbanken hingegen nimmt die Anzahl der eingefügten Knoten pro Sekunde mit Dauer des Experiments ab. Zu Beginn des Experiments nach 5 Minuten befinden sich in PostgreSQL 44.589 Knoten, in Memgraph 29.001 Knoten, Neo4j 20.631 Knoten und ONgDB 16.845 Knoten. Nach einer Experimentendauer von 20 Minuten jedoch befinden sich in PostgreSQL bereits 195.228 Knoten, in Memgraph jedoch 62.309, in Neo4j 54.480 und in ONgDB 41.271 Knoten. Diese Entwicklung deutet darauf, dass die zusätzliche Suchoperation beim Einfügen der Knoten den Flaschenhals der Graphdatenbanken verursacht. Da die gesamte Datenmenge im Laufe des Experiments zunimmt, benötigt auch die Suchoperation mehr Zeit. Da die Datenbanken ohne Indizierung implementiert sind wird zum Finden eines bestimmten Knoten zu einem bestimmten Attribut in Neo4j, ONgDB und Memgraph ein kompletter Scan der Datenbank durchgeführt. Außerdem wird aus der Grafik ersichtlich, dass nach 91 Minuten bei Memgraph ein Plateau erreicht wird mit 135.486 eingefügten Knoten. Danach werden keine Knoten mehr eingefügt. Bei Neo4j ist das Plateau nach 96 Minuten mit 130.781 Knoten erreicht. Für das Erreichen des Plateaus zeigt sich der Absturz des Brokers verantwortlich.

Tabelle 6.1: Durchschnittliche Einfügeraten der DBMS.

Batchgröße	DBMS	eingefügte Knoten pro Sekunde
500	PostgreSQL	150,79
	Neo4j	28,21
	Memgraph	29,67
	ONgDB	21,22
1.000	PostgreSQL	155,86
	Neo4j	27,07
	Memgraph	30,15
	ONgDB	19,66
1.500	PostgreSQL	152,67
	Neo4j	22,60
	Memgraph	25,80
	ONgDB	17,31

Die in Kapitel 5 beschriebene Konfiguration des Brokers und der Message Queue resultiert darin, dass Daten weiter in die DBMS eingefügt werden, auch wenn der Publisher bereits aufhört Daten zu senden. Durch die Einrichtung der Message Queue kam es jedoch zum Absturz des Broker durch Überlaufen des Arbeitsspeichers. Für die Batchgröße von 500 kam es nach 90 Minuten zum Absturz des Brokers, für eine Batchgröße von 1.000 erfolgte der Absturz nach etwa 65 Minuten und bei einer Batchgröße von 1.500 kam es nach 45 Minuten zum Absturz des Brokers.

Die Abstürze der Broker bei allen Experimenten zeigen, dass die geringen Schreibraten der Subscriber dazu führen, dass durch die Experimente ein reales Szenario, in Bezug auf die erzeugte Datenrate, nur teilweise abbilden können. Durch Optimierung der Einfügeanweisungen in den Subscribern wird erwartet, dass eine höhere Schreibgeschwindigkeit erreicht werden kann. Im Vergleich der Graphdatenbanken zeigt sich, dass, gemessen an den eingefügten Knoten, Memgraph die schnellste Verarbeitungsgeschwindigkeit erreicht, gefolgt von Neo4j und ONgDB. Im folgenden Abschnitt wird geprüft, welchen Einfluss die Hardware-Ressourcen bei der Erzeugung der Latenzen spielen.

6.2. Auswertung der Experimente zum Ressourcenverbrauch

Für den Ressourcenverbrauch werden die CPU Auslastung, der RAM Verbrauch und die Festplatten Lese- bzw. Schreibzugriffe der DBMS Hostsysteme betrachtet. Die Hardware Metriken der DBMS werden unter Betrieb in der Testumgebung aufgezeichnet. Für die Batchgröße von 500 werden repräsentativ Grafiken erzeugt, um die Entwicklung des Ressourcenverbrauchs zu betrachten. Die Grafiken für die Batchgrößen 1.000 und 1.500 zeigen keine signifikanten Unterschiede und befinden sich daher im Anhang, siehe Abschnitt A (Seite 99). In Tabelle A.3 und Tabelle A.4 sind die Maximal-, Minimal- und Durchschnittswerte der Hardware-Metriken für die weiteren Batchgrößen 500, 1.000 und 1.500 zusammengefasst.

Die CPU Metrik beschreibt die Auslastung der zentralen Recheneinheit. Ein hoher Wert ist ein negativer Indikator und deutet auf eine hohe Arbeitslast und einen hohen Verbrauch der Rechenleistung hin¹. Die Memory Metrik (RAM) beschreibt den Arbeitsspeicher, der durch die Anwendung genutzt wird. Memory Cached beschreibt den Arbeitsspeicher, der durch eine Anwendung als Cache temporär genutzt wird bzw. wurde, jedoch wieder für andere Anwendungen freigegeben werden kann². Ein hoher Memory Verbrauch ist ein negativer Indikator und bedeutet, dass die Anwendung einen hohen Speicherbedarf hat. Ein hoher Memory Cached Wert wiederum ist ein positiver Indikator und bedeutet, dass die Anwendung vorher benötigten Arbeitsspeicher wieder für andere Anwendungen freigeben kann und diesen nicht weiter benötigt. Viele Festplatten Lesezugriffe sind ein Indikator dafür, dass eine Anwendung häufig Daten von der Festplatte lesen muss, was wiederum auf einen zu geringen Arbeitsspeicher deuten kann. Eine bereits hohe Anzahl an Schreibzugriffen kann dazu führen, dass zusätzliche Operationen nur eingeschränkt ausgeführt werden können, was auch als eine Sättigung des Systems beschrieben wird.

Abbildung 6.2 zeigt die CPU Auslastung über die Dauer des Experimentes der vier DBMS im Vergleich. Es ist zu erkennen, dass alle drei Graphdatenbanken im Durchschnitt mehr CPU Leistung benötigen als PostgreSQL. Der durchschnittliche CPU Verbrauch von PostgreSQL liegt bei etwa 70 %. In der Spitze besitzt PostgreSQL den höchsten CPU Verbrauch mit einem Maximum von 120 % während die Graphdatenbanken konstant bei etwa 100 % liegen. Dieses Verhalten zeigt, dass sich die Graphdatenbanken konstant rechenintensiver als PostgreSQL verhält. Die regelmäßig auftretenden Schwankungen bei PostgreSQL stimmen mit den zeitlichen Intervallen der Abfragen überein, die für die Bestimmung der Antwortzeiten alle 5 Minuten gestartet werden. Mit Verlauf des Experiments benötigt die Beantwortung mehr Rechenleistung, weswegen

¹<https://gitnux.org/cpu-metrics/>

²Hinweise zu RAM Cache Metrik - <https://www.linuxatemyram.com/>

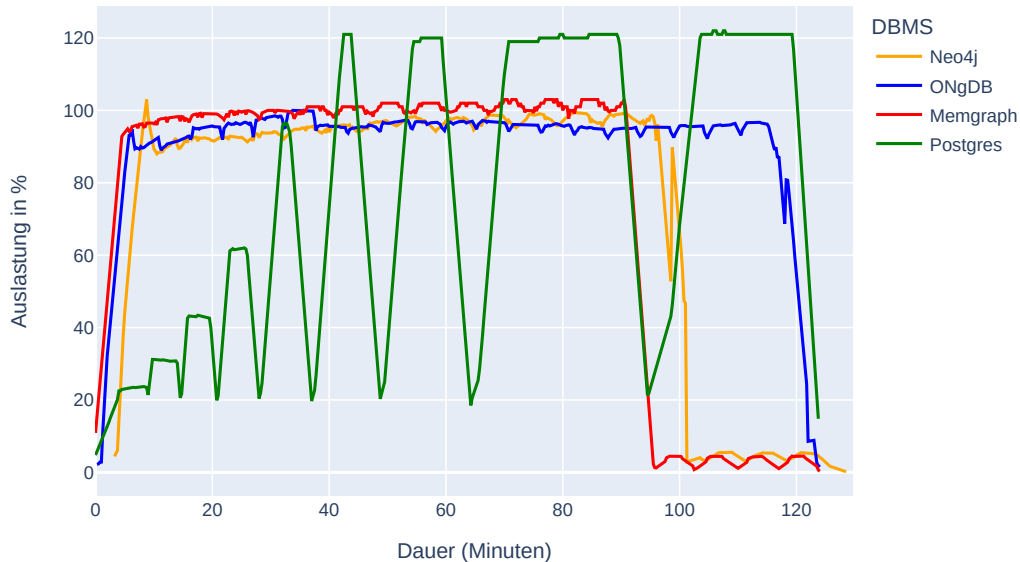


Abbildung 6.2.: DBMS Metriken Hardware Ressourcen - CPU - Batchgröße 500.

die Anstiege mit fortschreitendem Experiment höher werden. Die zeitlich zunehmenden Abstände zwischen den Anstiegen lassen sich durch die längere Verarbeitungszeit der Abfragen begründen. Bei den Graphdatenbanken lassen sich analog Anstiege zu den Zeitpunkten der Abfragen erkennen, jedoch sind die Anstiege deutlich geringer als bei PostgreSQL und im zeitlich korrekten Intervall von 5 Minuten. Dieses Verhalten deutet darauf, dass die Abfragen in den Graphdatenbanken eine geringere CPU Auslastung verursachen als in PostgreSQL. Das Maximum an zur Verfügung stehender Rechenleistung auf je Docker Container beträgt 120 % was die obere Beschränkung für PostgreSQL begründet. Nach etwa 90 Minuten kam es jeweils zum Absturz des Brokers, weswegen die CPU Auslastung ab diesem Zeitpunkt stark abfällt, besonders die der Graphdatenbanken. Aus Abbildung 6.2 geht hervor, dass zuerst Memgraph die CPU Auslastung verringert, danach Neo4j und zuletzt ONgDB. Dieses Verhalten spiegelt die Reihenfolge der Verarbeitungsgeschwindigkeit der DBMS wieder, die sich bereits in Abschnitt 6.1 angedeutet hat. Aufgrund der auf den DBMS ausgeführten Abfragen besteht weiterhin CPU Auslastung auf den DBMS.

Abbildung 6.3 zeigt den RAM Verbrauch der DBMS. Es ist zu erkennen, dass alle Graphdatenbanken mehr Arbeitsspeicher als PostgreSQL benötigen. Neo4j besitzt den höchsten Verbrauch mit 1,50 GB im Durchschnitt. Für den Neo4j-Klon ONgDB liegt

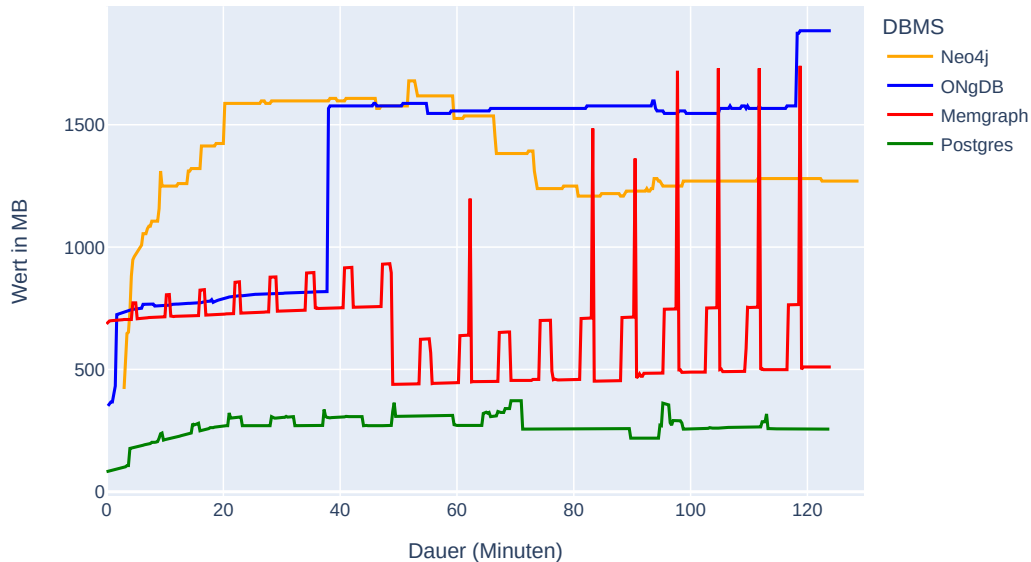


Abbildung 6.3.: DBMS Metriken Hardware Ressourcen - RAM - Batchgröße 500.

der Verbrauch im Durchschnitt bei 1,40 GB. Darauf folgt Memgraph mit 0,669 GB im Durchschnitt und PostgreSQL mit durchschnittlich 0,276 GB. Die regelmäßigen Anstiege im RAM Verbrauch, besonders bei Memgraph zu erkennen, korrelieren mit den Anstiegen im CPU Verbrauch zu den zeitlichen Intervallen der Abfragen.

PostgreSQL weist den geringsten RAM Verbrauch auf. Durch die regelmäßigen Abfragen sind verhältnismäßig geringe Erhöhungen in regelmäßigen Intervallen im Verbrauch festzustellen. Neo4j weist bereits nach 20 Minuten einen RAM Verbrauch von 1,52 GB auf. Nach etwa 65 Minuten nimmt der RAM Verbrauch jedoch ab. Aus den Schreibzugriffen in Abbildung 6.6 ist ersichtlich, dass ab 55 Minuten ein Anstieg in den geschriebenen Daten auf der Festplatte ersichtlich ist. Mit steigenden Festplattenschreibzugriffen nimmt der RAM Verbrauch ab. Bei ONgDB hingegen zeigt sich zu Beginn des Experiments ein geringerer RAM Verbrauch als bei Neo4j, jedoch nimmt der RAM Verbrauch ab Minute 36 stark zu und steigt auf ein ähnliches Niveau wie Neo4j an und bleibt bis zum Ende des Experiments dort. Zur gleichen Zeit ist im CPU Verbrauch ein plötzlicher kurzer Abfall der CPU Auslastung zu sehen. Eine genauere Erklärung lässt sich jedoch nicht anhand der erhobenen Metriken finden. Memgraph weist einen tendenziell steigenden RAM Verbrauch mit fortschreitendem Experiment auf. Nach etwa 45 Minuten kommt es zu einem plötzlichen Abfall des RAM Verbrauchs von 932 MB auf 440 MB. Es ist in

Abbildung 6.4 zu erkennen, dass zum selben Zeitpunkt der freigebbare RAM abnimmt. Ab Minute 60 nehmen die Spitzen im RAM Verbrauch enorm zu, was auf die steigende Datenmenge zurückzuführen ist, die durch die Abfragen zurückgeliefert wird. Es ist damit zu rechnen, dass bei steigender Knotenmenge diese Spitzen weiter zunehmen. In der Testumgebung übersteigen die Spitzen nicht den Wert von 1,74 GB.

Dass die Graphdatenbanken in der Speichernutzung deutlich höher als PostgreSQL liegen, ist vor allem durch die Tatsache begründet, dass Graphdatenbanken für Abfragen den kompletten Graphen von der Festplatte in den Hauptspeicher laden. Unter den Graphdatenbanken ist Memgraph besonders hervorzuheben. Im Maximum zeigt Memgraph zwar einen sehr hohen RAM-Verbrauch mit 1,73 GB jedoch ist der RAM-Verbrauch im Durchschnitt geringer als bei Neo4j und ONgDB. Die effizientere Speichernutzung von Memgraph ist vermutlich auf die Implementierung in C++ zurückzuführen, während Neo4j und ONgDB in Java implementiert sind. PostgreSQL erweist sich beim RAM Verbrauch effizienter als alle Graphdatenbanken.

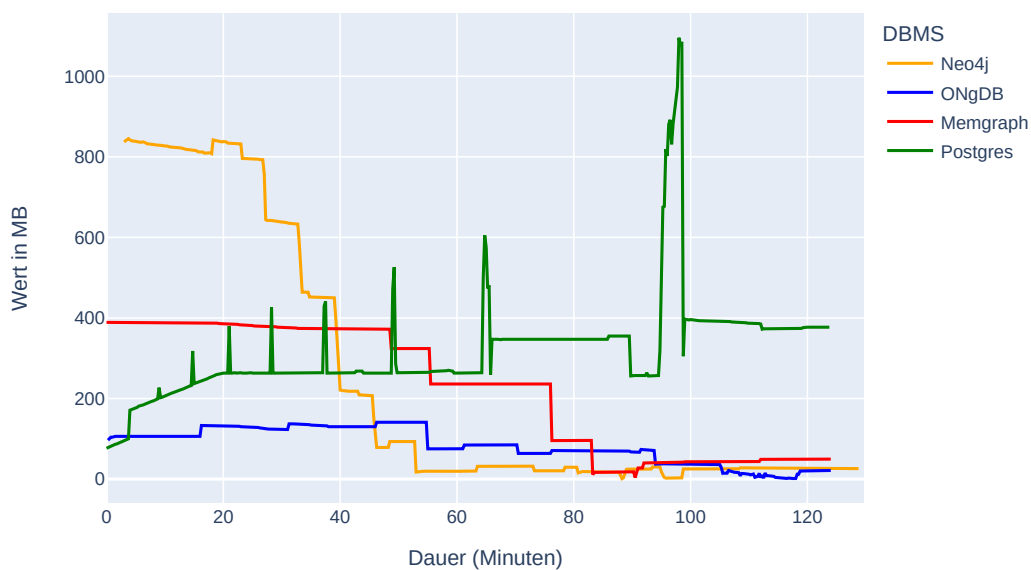


Abbildung 6.4.: DBMS Metriken Hardware Ressourcen - RAM Cached - Batchgröße 500.

Zusätzlich zum RAM-Verbrauch zeigt Abbildung 6.4 den freigebbaren Speicher (Memory Cached) als weiteres Indiz für eine effiziente Speichernutzung. Bei PostgreSQL beträgt der freigebbare RAM im Durchschnitt 0,328 GB. Für Neo4j beträgt der freigebbare Speicher im Durchschnitt 0,282 GB. Bei Memgraph liegt der freigebbare Speicher

im Durchschnitt bei 0,236 GB. Bei PostgreSQL ist der freigebbare RAM zu Beginn des Experiments gering, nimmt jedoch während der Experimentendauer zu. Dies deutet darauf hin, dass die PostgreSQL-Datenbank in der Lage benötigten Arbeitsspeicher nach Beantwortung einer Abfrage wieder freizugeben, während bei den Graphdatenbanken der RAM-Cached mit Dauer des Experiments abnimmt. Die Spitzen in PostgreSQL korrelieren mit dem Zeitpunkt der Ausführung der Queries. Die Tatsache, dass der freigebbare Speicher mit 1,09 GB höher liegt als der tatsächlich verbrauchte Speicher kann durch eine Überlappung des RAM und Memory Cache zustande gekommen sein. Indem PostgreSQL aufgrund einer Abfrage mehr RAM benötigt, wurde zusätzlicher Arbeitsspeicher aus dem Memory Cache allokiert. Diese Begründung würde ebenfalls die zunehmenden Spitzen von PostgreSQL im Verlauf des Experiments erklären. Bei Neo4j liegt der freigebbare Speicher zu Beginn bei über 800 MB, nimmt jedoch ab Minute 25 schnell ab. Bei Memgraph liegt der Wert zu Anfang geringer bei 395 MB, nimmt jedoch später und weniger schnell ab als im Vergleich bei Neo4j. Diese Entwicklung ist Indiz für eine effizientere Arbeitsspeichernutzung durch Memgraph. Bei ONgDB liegt der freigebbare Speicher unter 200 MB und nimmt nur verhältnismäßig wenig ab.

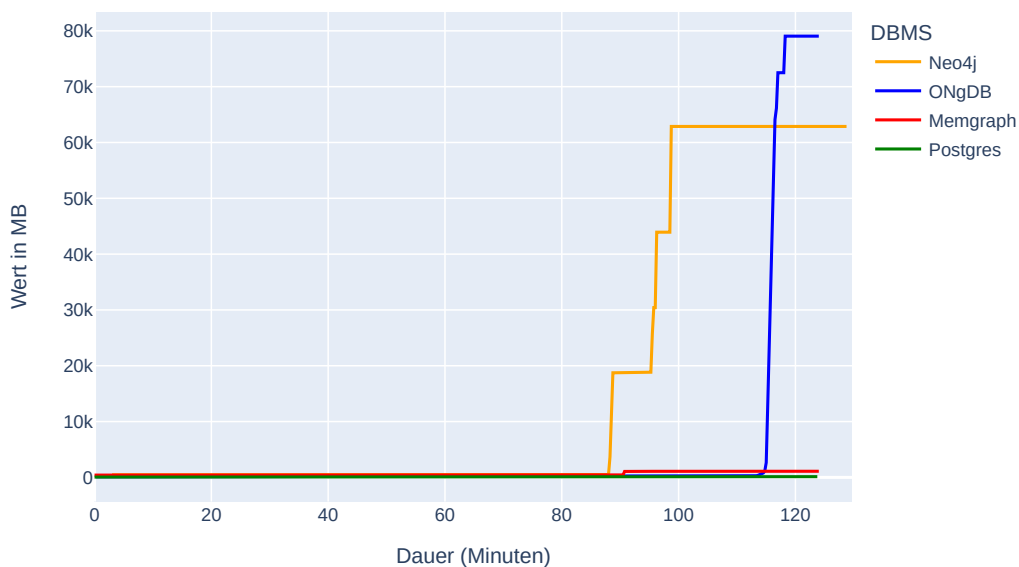


Abbildung 6.5.: DBMS Metriken Hardware Ressourcen - Lesezugriffe Festplatte - Batchgröße 500.

Bei den Festplatten-Lesezugriffen, siehe Abbildung 6.5, stechen besonders Neo4j und

ONgDB hervor mit durchschnittlich 4,9 GB bzw. 3,8 GB gelesenen Daten. Memgraph weist durchschnittlich 587 MB gelesene Daten auf und PostgreSQL hat durchschnittliche gelesene Daten von 123 MB. Bei Memgraph ist ein nahezu konstantes Niveau von 407 MB zu erkennen mit Ausnahme einer Zunahme nach etwa 90 Minuten auf 1,07 GB. Bei PostgreSQL nehmen die Leseoperationen nur sehr gering zu von initial 54 MB auf 130 MB zum Ende des Experiments. Es fällt auf, dass es bei Neo4j und zu einem plötzlichen Anstieg der gelesenen Daten nach 88 bzw. 113 Minuten kommt. Zeitlich korreliert dieser Anstieg mit dem Absturz des MQTT-Brokers. Ob ein Zusammenhang zwischen den Ereignissen besteht lässt sich jedoch nicht feststellen. Weiterhin zeigt sich unschlüssig, wieso ONgDB eine obere Grenze der gelesenen Daten bei 80 GB erreicht und Neo4j bei 62,8 GB. Der in der Dockerumgebung verfügbare Festplattenspeicher beträgt 67 GB. Bis zu dem Zeitpunkt des plötzlichen Anstiegs in Neo4j und ONgDB weist Neo4j Lesezugriffe von 474 MB auf, Memgraph von 407 MB und ONgDB von 110 MB.

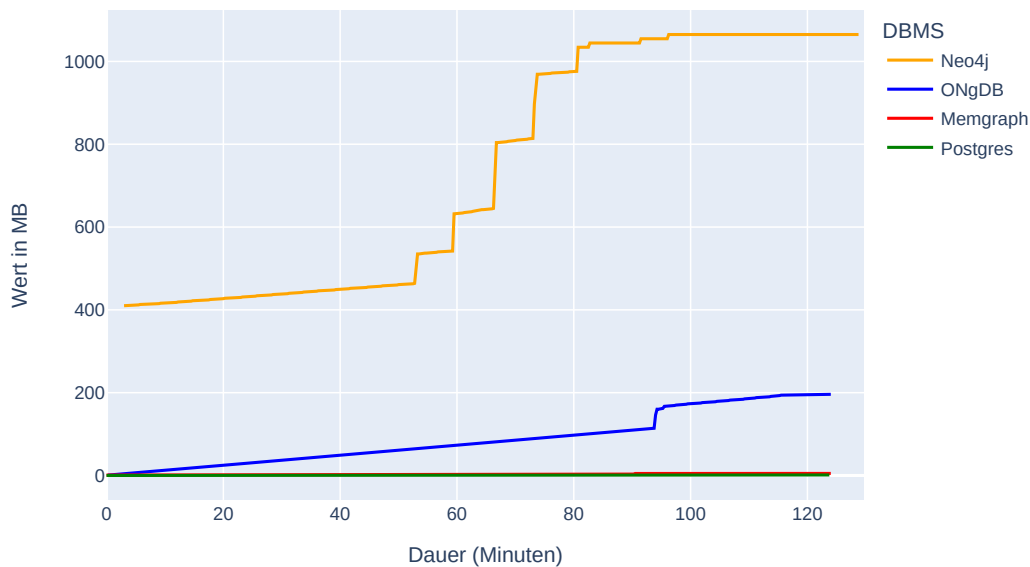


Abbildung 6.6.: DBMS Metriken Hardware Ressourcen - Schreibzugriffe Festplatte - Batchgröße 500.

Bei den Festplatten-Schreibzugriffen, zu sehen in Abbildung 6.6, ist zu erkennen, dass Neo4j die meisten Daten auf die Festplatte schreibt mit Durchschnittlich 657 MB. ONgDB liegt dahinter mit durchschnittlich 86,2 MB. Memgraph mit 2,27 MB und PostgreSQL mit 0,244 MB) benötigen deutlich weniger Schreibzugriffe. Es ist zu erkennen das in

Neo4j und ONgDB die geschriebene Datenmenge stetig zunimmt und nach 97 Minuten bei Neo4j ein Plateau erreicht. Dieser Zeitpunkt korreliert wieder mit dem Absturz des Brokers. Für PostgreSQL und Memgraph nehmen die geschriebenen Daten ebenfalls zu, jedoch ist der Anstieg im Verhältnis zu Neo4j wesentlich geringer. Für Neo4j lässt sich erkennen, dass nach 51 Minuten ein plötzlicher Anstieg der Schreiboperationen zu erkennen ist. Eine Begründung für den plötzlichen Anstieg der Schreiboperationen lässt sich im Erreichen des maximal verfügbaren Arbeitsspeichers der Hostmaschine von Neo4j finden. Da aufgrund des wachsenden Datenvolumens der gesamte Graph nicht mehr im Arbeitsspeicher vorgehalten werden kann, ist es notwendig Daten auf die Festplatte zu schreiben. Es ist zu erkennen, dass der RAM Verbrauch nach 51 Minuten eine Spitze mit 1,69 GB erreicht. Zusätzlich ist im Memory Cache zu erkennen, dass Neo4j dort seinen geringsten Wert aufweist mit 17,4 MB. Als Erkenntnis lässt sich hier ableiten, dass Neo4j in einer realen Anwendung als potenzieller Flaschenhals in Frage kommen würde, da mehr Schreiboperationen als bspw. bei Memgraph, PostgreSQL oder ONgDB benötigt werden.

In der Hardware Ressourcenausnutzung weist PostgreSQL die performantesten Werte auf. Unter den Graphdatenbanken zeigt Memgraph durch einen geringeren RAM Verbrauch Vorteile gegenüber Neo4j und ONgDB auf. Im folgenden Abschnitt werden zusätzlich zu Hardware Metriken die Abfragemöglichkeiten der DBMS verglichen.

6.3. Beurteilung der Formulierung von Abfragen

In diesem Abschnitt wird die Möglichkeit zur Formulierung *forensischer- Forschungs-Abfragen* mithilfe der DBMS untersucht. Dabei wird betrachtet wie komplex die Formulierung der Abfragen ist, welche Unterstützung das DBMS dabei anbietet und in welcher Zeit die Abfragen beantwortet werden. In einem späteren Abschnitt wird noch einmal gesondert auf die Laufzeit unter Auslastung der DBMS eingegangen. Für den Vergleich werden die Abfragesprachen SQL und Cypher betrachtet. Alle Graphdatenbanken, die in dieser Arbeit untersucht werden, unterstützen Cypher.

Es gilt die *forensischen* Abfragen für *Vorgänger eines Knoten*, *Nachfolger eines Knoten* und *kürzester Pfad zwischen zwei Knoten* zu formulieren, wie in Abschnitt 4.1 erläutert. Neben der Formulierung der forensischen Abfragen besteht auch die Anforderung komplexere Graph-theoretische Abfragen zu formulieren. Als beispielhafte Abfrage zur Forschung wird die 2-Hop Nachbarschaft gewählt. Das Finden der 2-Hop Nachbarschaft ist eine häufig gestellte Abfrage, die vor allem bei der Entwicklung von Graph Neuronalen Netzen von Interesse ist.

Abfragesprache Cypher Cypher Abfragen ermöglichen die Formulierung eines Musters, das im Graphen gefunden werden kann. Das Muster wird unmittelbar nach dem Schlüsselwort `MATCH` definiert. Ein Muster in Cypher entspricht der Form `()-()` und kann umgangssprachlich als *Knoten verbunden mit Knoten* ausgedrückt werden. Auch gerichtete Kanten zwischen Knoten lassen sich explizit über das Muster `() < -- ()` oder `() -- > ()` erfragen. Da Neo4j, ONgDB und Memgraph jeweils schemalose Datenbanken sind und damit nicht über Datentypen verfügen werden für die Identifikation von Knotentypen Label verwendet. Label werden auch eingesetzt, um Kantentypen zu definieren. Um in einer `MATCH`-Klausel ein Label für Knoten zu selektieren, wird das Label folgendermaßen definiert: `(a:Subject)-(b:Event)`, ausgedrückt als Knoten `a` mit dem Label `Subject` ist verbunden mit Knoten `b`, welcher das Label `Event` besitzt. Für Kanten erfolgt die Label-Selektion folgendermaßen `(a)-[:generatedBy]-(b)`. Knoten `a` ist über die Kante `generatedBy` mit dem Knoten `b` verbunden.

Quellcode 6.1 zeigt die Abfrage für das Finden aller Nachfolger eines Knotens und gibt die Entfernung zum Ursprungsknoten an. Als Startknoten werden alle Knoten mit Label `Host` festgelegt, die das Alias `node` besitzen und das Attribut `_uuid` (ein Attribut des `Cadets` Datensatz) mit dem Wert `'83C8ED1F-5045-DBCD-B39F-918F0DF4F851'` aufweisen. Es werden dann alle Knoten mit dem Alias `descendant` selektiert, die eine beliebige gerichtete Kante zu `node` haben. Zusätzlich wird das Ergebnis der `MATCH`-Klausel als Subgraph in der Variable `path` gespeichert³. Als Rückgabe liefert die Abfrage alle `descendant` Knoten, also alle Knoten die eine gerichtete Kante zu `node` besitzen. Da der `Host`-Node der Startknoten des Datensatzes ist, liefert diese Abfrage als Ergebnis alle Knoten und alle möglichen Pfade zum Startknoten.

```

1 MATCH path=(node:Host) <-[*]-(descendant)
2 WHERE node._uuid = '83C8ED1F-5045-DBCD-B39F-918F0DF4F851'
3 RETURN descendant, length(path) AS distance_downstream
4 ORDER BY distance_downstream;

```

Quellcode 6.1: Cypher Abfrage für das Finden der Nachfolger eines Knoten.

Quellcode 6.2 zeigt die Formulierung der Cypher Abfrage für das Finden aller Vorgänger eines Knotens und deren Entfernung zum Ursprungsknoten an. Diese Abfrage ist analog zur Nachfolger-Abfrage mit Ausnahme der Richtung der Kante in der `MATCH`-Klausel. Da hier der `Host` Node als Ursprung definiert ist und der Knoten als Startknoten im Graphen keine Vorgänger hat, wird diese Abfrage keine Ergebnisse liefern.

```

1 MATCH path=(node)-[*]->(ancestor)
2 WHERE node._uuid = '83C8ED1F-5045-DBCD-B39F-918F0DF4F851'
3 RETURN ancestor, length(path) AS distance_upstream
4 ORDER BY distance_upstream;

```

Quellcode 6.2: Cypher Abfrage für das Finden der Vorgänger eines Knoten.

³Blogbeitrag zu Path - <https://neo4j.com/developer-blog/the-power-of-the-path-1/>

Quellcode 6.3 zeigt die Formulierung der Abfrage für das Finden des kürzesten Pfades zwischen zwei beliebigen Knoten. Für das Finden des Pfades wird eine eingebaute Funktion `shortestPath` genutzt, die durch Neo4j und ONgDB unterstützt wird⁴. Wenn mehrere kürzeste Pfade gefunden werden, dann wird ein Pfad nicht-deterministisch ausgewählt. Zusätzlich wird eine `WHERE`-Bedingung verwendet, um Knoten über ein Attribut zu selektieren. Als Rückgabe auf diese Abfrage wird der kürzeste Pfad zwischen den Knoten ausgegeben.

```
1 MATCH path = shortestPath(  
2 (a WHERE a._uuid='A6A7C956-0132-5506-96D1-2A7DE97CB400')  
3 -[*]->  
4 (b WHERE b._uuid = '8DA367BF-36C2-11E8-BF66-D9AA8AFF4A69'))  
5 RETURN path;
```

Quellcode 6.3: Cypher Abfrage für das Finden des kürzesten Pfades zwischen zwei beliebigen Knoten.

Quellcode 6.4 zeigt die Formulierung der 2-Hop Nachbarschaft in Cypher. Für diese Untersuchung wird die 2-Hop Nachbarschaft abgefragt. In der Abfrage wird zuerst in der `MATCH`-Klausel ein Startknoten selektiert. Die 2-Hop Nachbarschaft wird in der Abfrage über den Ausdruck `(a) -[r*1..2]- (n)` definiert. Dieser Ausdruck, der als *Relationship Expansion* bezeichnet wird, bedeutet, dass alle Knoten `n` selektiert werden, die über ein oder zwei beliebige ungerichtete Kanten mit dem Ursprungsknoten `a` verbunden sind. Knoten können in dieser Abfrage mehrfach in der Ergebnismenge enthalten sein, wenn sie in Teilpfaden enthalten sind.

```
1 MATCH  
2 path=(startNode)-[r*1..2]-(neighborhood)  
3 WHERE startNode._uuid = '9FF334BB-9072-D756-B290-556656D73728'  
4 WITH neighborhood, COLLECT(DISTINCT r) AS uniqueRels  
5 RETURN DISTINCT neighborhood;
```

Quellcode 6.4: Cypher Abfrage für das Finden der 2-Hop Nachbarschaft.

Quellcode 6.5 zeigt die Formulierung der Cypher Abfrage für das Finden der 2-Hop Nachbarschaft in Memgraph mithilfe von built-in Prozeduren für Graphalgorithmen. Diese Abfrage unterscheidet sich von der Cypher Abfrage durch den Aufruf einer Prozedur (eingeleitet durch das Schlüsselwort `CALL`), welche die Nachbarschaft zurückgibt.

Abfragesprache SQL Die Formulierung der Abfragen für *Vorgänger*, *Nachfolger*, *kürzester Pfad zwischen zwei Knoten* und *2-Hop Nachbarschaft* ist in SQL optimal mithilfe von

⁴Link zur Dokumentation von `shortestPath` - <https://neo4j.com/docs/cypher-manual/current/patterns/concepts/>

```

1 MATCH (a:Subject)
2 WHERE a._uuid='0CF2BB3E-36B8-11E8-BF66-D9AA8AFF4A69'
3 CALL neighbors.by_hop(a, [""], 2)
4 YIELD nodes
5 RETURN nodes;

```

Quellcode 6.5: Cypher Abfrage für das Finden der 2-Hop Nachbarschaft mit Memgraph Prozedur.

Common Table Expression (CTE) umsetzbar. CTE ermöglichen hierarchische und rekursive Abfragen. Diese werden auf die Knoten- und Kantenliste, in denen der Graph enthalten ist, angewendet. Die CTE ist das Ergebnis einer Abfrage, die dann im selben Statement erneut verwendet werden kann. Die CTE wird durch das Schlüsselwort **WITH** eingeleitet und im Regelfall findet direkt danach ein **SELECT** auf die CTE statt.

Quellcode 6.6 zeigt die SQL Abfrage für das Finden der Vorgänger eines beliebigen Startknotens mit einer Tiefe von maximal 4. In der CTE wird als Start aus der Knotenliste der Startknoten selektiert. Dann wird die Knotenliste über **uuid** mit der **source** der Kantenliste verknüpft. Das Ergebnis der Startabfrage wird mit dem Ergebnis der rekursiven Abfrage vereinigt. In der rekursiven Abfrage wird ähnlich zur Startabfrage von der Knotenliste selektiert, jedoch wird die CTE selbst über das Feld **dest** (destination) verknüpft. Die Abbruchbedingung der Rekursion ist erreicht, wenn eine Tiefe von 4 erreicht wurde. Zuletzt wird das Ergebnis der CTE per **DISTINCT** selektiert, um doppelte Werte aus dem Ergebnis herauszufiltern.

```

1 WITH RECURSIVE AncestorCTE AS (
2     SELECT n.node_no, e.dest, n.type, 1 AS Level
3     FROM node_list n
4     JOIN edge_list e ON n.uuid = e.source
5     WHERE n.node_no = 1 -- starting node_no
6     UNION ALL
7     SELECT n.node_no, e.dest, n.type, d.Level + 1
8     FROM node_list n
9     JOIN edge_list e ON n.uuid = e.source
10    JOIN AncestorCTE d ON e.dest = n.uuid
11    WHERE d.Level < 5 -- desired level of ancestors
12 )
13 SELECT DISTINCT node_no, dest, type, Level FROM AncestorCTE;

```

Quellcode 6.6: SQL Abfrage für das Finden der Vorgänger eines Knoten.

Die SQL Abfrage für das Finden der Nachfolger eines beliebigen Startknotens mit einer Tiefe von maximal 4 ist nahezu analog. Die Abfrage unterscheidet sich lediglich durch die Verknüpfung der Tabellen in der Startabfrage über die **destination** zur Kantenliste

und über die Verknüpfung in der rekursiven Abfrage zur [source](#). Der entsprechende Quellcode 6.6 befindet sich im Anhang.

Die weiteren Abfragen für das Finden eines Pfades zwischen zwei beliebigen Knoten und dem Finden der 2-Hop Nachbarschaft sind in Quellcode A.2 (Seite 113) und Quellcode A.3 (Seite 114) dargestellt. Diese Abfragen sind ebenfalls über CTEs implementiert. Die Komplexität in der Formulierung und der Umfang der Abfragen haben zugenommen, so dass sich aus Platzgründen der Quellcode dazu im Anhang befindet.

Bewertung der Abfragesprachen Zur Formulierung der Abfragen lässt sich feststellen, dass Cypher eleganter und weniger komplex als SQL ist. Cypher ermöglicht eine fast sprechende Formulierung von Graphabfragen, da die Operatoren so implementiert und gewählt wurden Graphstrukturen möglichst konkret und simpel zu beschreiben. Hier zeigt sich deutlich der Unterschied einer (Graph)-Datenbank, die auf Graphabfragen optimiert ist, gegenüber einer relationalen Datenbank. Mithilfe der CTEs lassen sich rekursive Abfragen zur Graphtraversierung definieren. Es wird jedoch ersichtlich, dass die Komplexität in der Formulierung der Abfragen je nach Schwierigkeit und Graphoperation deutlich zunimmt. Beispiel dafür ist die Formulierung der Vorgänger bzw. Nachfolger eines Knotens in SQL, die als simplere Operation noch übersichtlich erscheint. Bei der Formulierung eines Pfades oder der 2-Hop Nachbarschaft steigt die Komplexität enorm.

Ein weiterer Unterschied zwischen Cypher und SQL findet sich auch konzeptionell bei der Formulierung der Abfrage der Vorgänger bzw. Nachfolger eines Knotens. Um eine endlose Rekursion zu verhindern muss bei einer rekursiven CTE in SQL die Tiefe der Rekursion mit angegeben werden. Dies ist ein struktureller Unterschied im Gegensatz zu Cypher Abfragen, bei denen bei der Formulierung der Abfrage keine Randbedingungen bezüglich der Graphstruktur gestellt werden muss.

Besonders positiv zu erwähnen sind die built-in Graphalgorithmen von Memgraph und Neo4j. Beide DBMS stellen fortgeschrittene Abfragen für verschiedenste Graphalgorithmen für Zentralität, PageRank und GNN Modelle zur Verfügung⁵⁶. Außerdem besteht für Memgraph die Möglichkeit *Custom Query Module* in C, C++ oder Python zu implementieren und als Prozeduren dem DBMS zur Verfügung zu stellen.

⁵Link zu Neo4j Graph Data Science - <https://neo4j.com/docs/graph-data-science/current/>

⁶Link zu Memgraph *Advanced Algorithms* - <https://memgraph.com/docs/advanced-algorithms/available-algorithms>

6.4. Auswertung der Abfragelatenz

Für die Bewertung dieses Ziels wird überprüft wie sich Abfrage-Antwortzeiten verhalten, wenn das DBMS unter Belastung steht. Die Experimente zum Überprüfen dieses Ziels sind folgendermaßen geplant; die zuvor formulierten Abfragen für Vorgänger, Nachfolger, Pfad zwischen zwei Knoten und 2-Hop Nachbarschaft werden, während der kontinuierlichen Schreiboperationen der Subscriber, alle 5 Minuten an das DBMS gesendet. Für jede Abfrage werden die Antwortzeiten zum Konsumieren der Daten in der Anwendung, die Datenbank-interne Verarbeitungszeit der Abfrage und die Zahl der zurückgegebenen Zeilen bzw. Knoten gespeichert. Ein Experiment wird für zwei Stunden mit steigenden Batchgrößen zur Steuerung der Datenmengen durchgeführt.

Zum Vergleich der Korrektheit der Abfragen, die zur Beurteilung der Performance der DBMS genutzt werden, wurde geprüft, dass die Abfragen bei identischen Daten in der Datenbank das gleiche Ergebnis liefern. Bei 9.500 Knoten und 17.774 Kanten liefern alle DBMS für die vier definierten Abfragen die gleichen Ergebnismengen zurück, siehe Tabelle 6.2. Dass die Anzahl der Nachfolger größer als die Anzahl der Knoten ist, liegt daran, dass auch Teilpfade enthalten sind und Knoten somit mehrfach in der Ergebnismenge vorkommen können.

Tabelle 6.2: Ergebnismengen der Queries bei gleichem Datenvolumen.

DBMS	Anzahl Knoten	Anzahl Kanten	Vorgänger	Nachfolger	Pfad	2-Hop
PostgreSQL	9.500	17.774	0	42.877	1	1.043
Neo4j	9.500	17.774	0	42.877	1	1.043
Memgraph	9.500	17.774	0	42.877	1	1.043
ONgDB	9.500	17.774	0	42.877	1	1.043

Aufgrund von technischen Schwierigkeiten fehlt die internen Datenbank-Beantwortungszeit der Memgraph Datenbank. Diese Metrik wird von der Datenbank nicht über den Treiber an die Anwendung weitergeben. Als Alternative zur internen Antwortzeit wird die Zeit zur Verarbeitung des Abfrage Resultats in der Anwendung gemessen und zum Vergleich der Abfragelatenzen genutzt. Diese Zeit ist zwar beeinflusst durch die Verarbeitungsgeschwindigkeit der Anwendung, ermöglicht dennoch eine Annäherung für den Vergleich der Antwortzeiten. Bei der Auswertung der Ergebnisse hat sich gezeigt, dass die Abfrage-Anwendung für PostgreSQL nach 65 Minuten abstürzt und keine weiteren Antwortzeiten aufzeichnet. Die Ursache für diese technische Schwierigkeit lässt sich in den benötigten Antwortzeiten in Abbildung 6.7 finden. Es stellt sich heraus, dass die Be-

6. Ergebnisse der Experimente

antwortung der Abfrage der 2-Hop Nachbarschaft in PostgreSQL und die Verarbeitung des Ergebnisses in der Abfrageanwendung den Absturz verursacht. Bis zum Zeitpunkt des Absturzes sind die Abfragelatenzen vergleichbar.

Abbildung 6.7 zeigt die Entwicklung der Abfragelatenzen der Anwendung über die Dauer eines Experiments von 120 Minuten. In der Abbildung sind vier Diagramme zu sehen in denen jeweils die Antwortzeiten der DBMS zu einer Abfrage enthalten sind. Diagramm zeigt A (Vorgänger), B (Nachfolger), C (Pfad) und D (2-Hop). Die Antwortzeiten in dieser Abbildung beziehen sich auf die Verarbeitungszeit der Abfrage in der Anwendung in Sekunden.

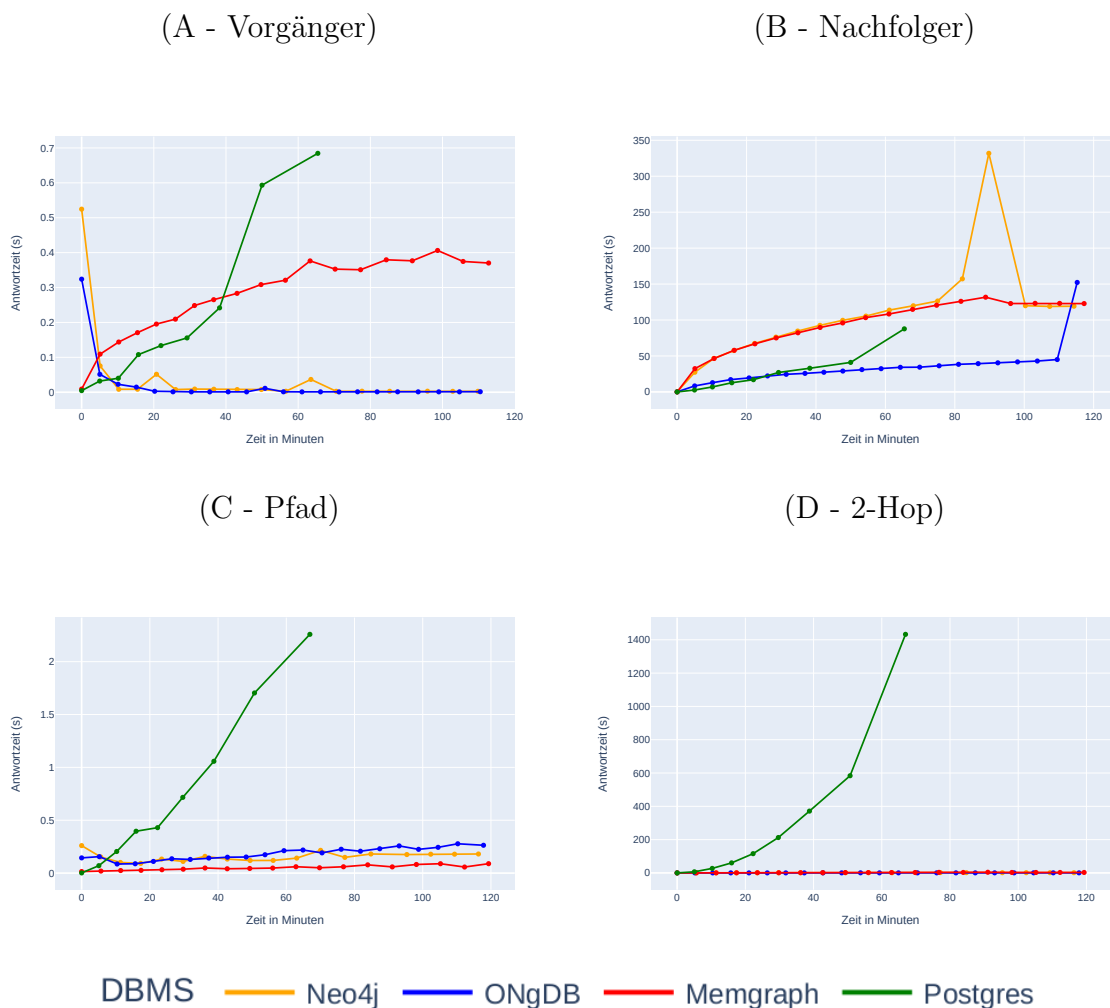
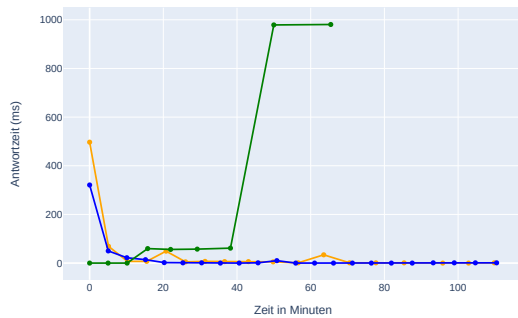


Abbildung 6.7.: Latenzen der Anwendung je Abfrage bei Batchgröße von 500.

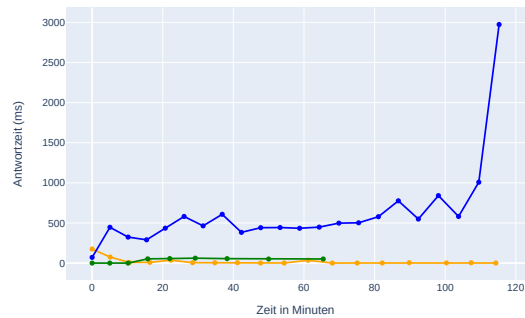
In Abbildung 6.8 ist ebenfalls die Entwicklung der Abfragelatenz je Abfrage abgebildet. Diese Abbildung ist analog zu Abbildung 6.7, jedoch ist auf der Y-Achse die interne

Antwortzeit des DBMS in Millisekunden dargestellt. Wie zuvor erläutert, fehlt aufgrund von technischen Schwierigkeiten Memgraph in den Diagrammen.

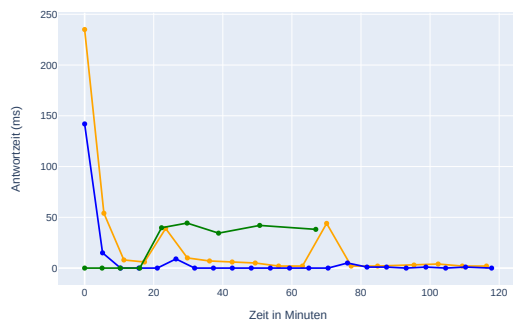
(A - Vorgänger)



(B - Nachfolger)



(C - Pfad)



(D - 2-Hop)



— Neo4j — ONgDB — Postgres

Abbildung 6.8.: Latenzen der DBMS je Abfrage bei Batchgröße von 500.

Abbildung 6.9 zeigt die Entwicklung der Ergebnismenge der Abfragen. Darin sind ebenfalls vier Diagramme abgebildet (ein Diagramm je Abfrage). Die X-Achse zeigt die Dauer des Experiments von 120 Minuten und die Y-Achse zeigt die Anzahl der zurückgegebenen Ergebnisse jedes DBMS. Für PostgreSQL bedeutet das die Anzahl der zurückgegebenen Zeilen der Knotenliste und für die Graphdatenbanken die Anzahl der Knoten in der Ergebnismenge. Dieses Diagramm wird hauptsächlich verwendet, um die Korrektheit der ausgeführten Abfragen zu prüfen.

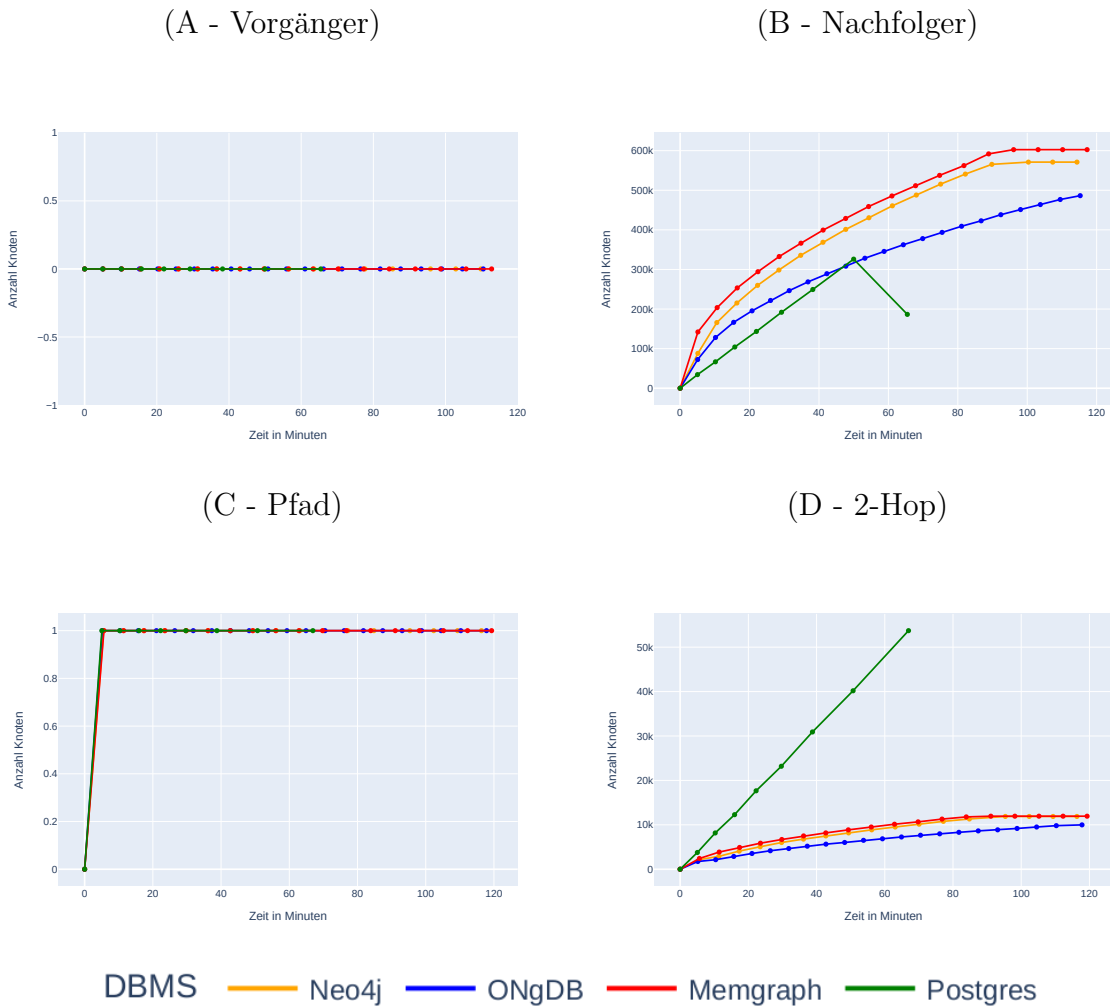


Abbildung 6.9.: Ergebnismenge je Abfrage bei Batchgröße von 500.

Abfrage Ancestors Die schnellste Abfrage in jedem DBMS ist die Abfrage der Vorgänger *Ancestors (anc)*, siehe dazu Abbildung 6.7 Diagramm A. Da diese Abfrage so gewählt wurde, dass kein Ergebnis resultiert (Vorgänger des Startknotens existieren nicht), entspricht dies auch dem erwarteten Ergebnis. Der Startknoten wurde gewählt, da dieser eine hohe Zentralität aufweist mit den meisten eingehenden Kanten (Nachfolgern), ohne eine ausgehende Kante (Vorgänger) zu besitzen. Die Abfrage wie viele Vorgänger dieser Knoten besitzt prüft somit gleich die Korrektheit der Einfügeoperation, ob fälschlicherweise eine Kante erzeugt. Alle Datenbanken können diese Abfrage schnell beantworten und erzeugen das erwartete Ergebnis von 0 Knoten, wie in Abbildung 6.1 Diagramm A zu sehen. Für das Beantworten der Vorgänger Abfrage benötigt von allen Datenbanken PostgreSQL die längste Zeit mit 0,7 Sekunden im Maximum. Die Graphdatenbanken sind performanter mit einer Antwortzeit zwischen 0,15 und 0,002 Sekunden.

Einzig zu Beginn des Experiments weisen Neo4j und ONgDB eine höhere Laufzeit auf von 0,52 bzw. 0,32 Sekunden. Dieses Verhalten zeigt sich unplausibel, da zu diesem Zeitpunkt sind noch keine Knoten in den Datenbanken enthalten sind. Die Laufzeit in Neo4j und ONgDB nimmt im Lauf des Experiments ab mit kurzen Ausreißen bei Neo4j nach 20 und 63 Minuten. Es ist im RAM Verbrauch in Abbildung 6.3 zu erkennen, dass Neo4j nach 20 Minuten ein Plateau im RAM Verbrauch erreicht, was zu der etwas längeren Antwortzeit führen kann. Nach 20 Minuten ist ebenfalls der Zeitpunkt ab dem der freigebbare Speicher abnimmt, siehe Abbildung 6.4. In Memgraph ist die Entwicklung der Antwortzeit steigend mit Dauer des Experiments. Hier lässt sich vermuten, dass durch die steigende Datenmenge in der Datenbank die Suchoperation mehr Zeit in Anspruch nimmt. In PostgreSQL steigt die Antwortzeit ebenfalls mit Experimentendauer. Ab Minute 40 benötigt PostgreSQL die längste Zeit zum Beantworten der Query. Es ist zu erwarten, dass die Antwortzeit in PostgreSQL im weiteren Verlauf schneller zugenommen hätte, als bei den Graphdatenbanken.

Abfrage Path Die Abfrage zum Finden eines kürzesten Pfades zwischen zwei beliebigen Knoten *path* ist die zweitschnellste Abfrage über alle DBMS hinweg. Die Antwortzeiten dieser Abfrage sind in den Abbildungen in Diagramm C dargestellt. Hierfür benötigt ebenfalls PostgreSQL die längste Zeit mit maximal 2,2 Sekunden nach 67 Minuten. Ähnlich wie bei den Vorgängern eines Knotens sind die Graphdatenbank schneller in der Beantwortung. Memgraph liegt bei 0,06 Sekunden während Neo4j mit 0,14 und ONgDB mit 0,22 Sekunden etwas länger sind. Auch bei dieser Abfrage nimmt die Antwortzeit in der Anwendung im Laufe des Experiments bei PostgreSQL schneller zu, als bei den Graphdatenbanken. Alle drei Graphdatenbanken weisen eine ähnliche Entwicklung der Antwortzeit auf. Bei Memgraph beträgt die Antwortzeit zu Beginn des Experiments 0,01 Sekunden und nach 120 Minuten 0,08 Sekunden. Die zunehmende Dauer im Laufe des Experiments lässt sich durch die wachsende Datenmenge gründen. Ab Minute 90 (dem Zeitpunkt des Plateaus der eingefügten Knoten), verändert sich die Laufzeit in Neo4j nicht mehr. In Memgraph und ONgDB schwankt die Laufzeit gering. Mit Blick auf die internen Antwortzeiten lässt sich feststellen, dass Neo4j nach 20 und 70 Minuten eine Spitze von knapp 50 Millisekunden aufweist. Dies ist ebenfalls der Zeitpunkt, an dem die Schreibzugriffe in Abbildung 6.5 für Neo4j stark zunehmen. Außerdem lässt sich in den internen Antwortzeiten erkennen, dass für die erste Ausführung der Abfrage zu Beginn des Experiments bei Neo4j und ONgDB eine lange Antwortzeit zu verzeichnen ist. Mit der folgenden Abfrage nimmt die Antwortzeit jedoch ab und nähert sich dem Durchschnittswert. Die Unterschiede in der Antwortzeit zwischen PostgreSQL und den Graphdatenbanken lassen auf eine performantere Implementierung der Graphdatenbank-internen Prozedur *shortestPath* schließen, die für Neo4j und ONgDB verwendet wurde. Dies könnte eine Begründung für die etwas höhere Antwortzeit gegenüber Neo4j und ONgDB sein. Ein Blick auf die Ergebnismenge der Abfragen zeigt, dass in allen DBMS nach 5 Minuten ein Pfad gefunden wurde. Die beiden gesuchten

Knoten wurden so gewählt, dass diese im ersten Batch enthalten sind, daher ist die Ergebnismenge zum Startzeitpunkt des Experiments 0 und erst nach der ersten Abfrage wird ein Pfad gefunden.

Abfrage 2-Hop Nachbarschaft Die Abfrage zum Finden der 2-Hop Nachbarschaft *2-hop* ist die dritt-schnellste Abfrage über alle DBMS. In den Abbildungen ist diese Abfrage in den Diagrammen D abgebildet. Für die Beantwortung dieser Abfrage benötigt PostgreSQL die längste Zeit mit 1400 Sekunden nach 66 Minuten. Es ist zu vermuten, dass die Verarbeitung des Ergebnisses dieser Abfrage zum Absturz der Abfrageanwendung geführt hat, da eine zu lange Zeit auf das Beantworten der Abfrage gewartet wurde. Für die Graphdatenbanken beträgt die Antwortzeit nach 63 Minuten für Memgraph 2,34 Sekunden, für Neo4j 1,35 Sekunden und für ONgDB 0,40 Sekunden. Damit weist ONgDB die schnellste Antwortzeit auf. Es ist jedoch zu berücksichtigen, dass sich zu diesem Zeitpunkt von den drei Graphdatenbanken die geringste Anzahl an Knoten befindet. ONgDB beinhaltet 79.183 Konten, Neo4j 106.661 Knoten und Memgraph 112.163 Knoten, siehe Abbildung 6.9. Die internen Antwortzeiten auf die Abfrage zeigen, dass Neo4j im Durchschnitt weniger Zeit für die Beantwortung der Query als ONgDB benötigt. Es ist zu erkennen, dass ONgDB mit Fortschreiten des Experiments länger für die Beantwortung benötigt, während bei Neo4j die Antwortzeit, mit Ausnahme der Spitzen aufgrund von Leseoperationen, ein ähnliches Niveau zwischen 2-10 Millisekunden aufweist. PostgreSQL benötigt weist bei den internen Antwortzeiten ebenfalls die längsten Latenzen auf. Es ist zu erkennen, dass zu Beginn des Experiments die Antwortzeit stark ansteigt und nach 38 Minuten im Maximum 905.463 Millisekunden (≈ 15 Minuten) benötigt. Diese Latenz spiegelt sich im Verzug bis zur Ausführung der nächsten Abfrage in PostgreSQL wider. Die Ergebnismenge der Abfrage zeigt eine analoge Entwicklung zu der insgesamt enthaltenen Datenmenge der DBMS in Abbildung 6.9.

Abfrage Descendants Das Finden der Nachfolger *Descendants (desc)* ist insgesamt die langsamste der vier Abfragen und ist in Diagramm C dargestellt. Nach 65 Minuten liegt die Antwortzeit der Anwendung bei ONgDB am niedrigsten mit 34,12 Sekunden, gefolgt von PostgreSQL mit 87,72 Sekunden. Am längsten benötigen Memgraph und Neo4j mit 108,38 bzw. 113,79 Sekunden. Die Antwortzeiten der Anwendung zeigen, dass Neo4j und Memgraph bis Minute 68 eine nahezu identische Entwicklung vornehmen. Ab Minute 89 weist Neo4j jedoch eine Spitze in den Antwortzeiten mit 331,82 Sekunden auf. Dieser Zeitpunkt korreliert mit den stark steigenden Lesezugriffen für Neo4j in Abbildung 6.5. Es ist daher zu vermuten, dass die nötigen Lesezugriffe die Querylatenz erhöhen. Die internen Antwortzeiten zeigen, dass ONgDB die längste Antwortzeit aufweist mit 448 Millisekunden nach 64 Minuten. PostgreSQL benötigt 51,01 Millisekunden und Neo4j 35 Millisekunden. Hier zeigt sich, dass die größere Datenmenge in Neo4j nicht

dazu führt, dass die Abfrage langsamer beantwortet wird, sondern das ONgDB mit weniger Knoten eine längere Antwortzeit benötigt. Der Anstieg der Antwortzeiten nach 103 Minuten bei ONgDB korreliert zeitlich mit dem Anstieg der Lesezugriffe durch ONgDB in Abbildung 6.5. Es fällt auf, dass die Spitze in Neo4j nach 89 Minuten nicht in den internen Antwortzeiten zu sehen ist. Dies lässt darauf schließen, dass die Latenz durch die Verarbeitung der Daten in der Anwendung entstanden ist. Bei der Betrachtung der Ergebnismenge der Nachfolgerabfrage in Abbildung 6.9 fällt auf, dass für PostgreSQL nach etwa 50 Minuten die Ergebnismenge von 326.072 auf 186.529 abnimmt. Dies ist jedoch dem Absturz der PostgreSQL Abfrageanwendung geschuldet. Bis zur Minute 50 weist PostgreSQL ein lineares Wachstum der Nachfolger auf. Die Graphdatenbanken weisen in der Ergebnismenge zu Beginn eine schnellere Zunahme als PostgreSQL auf, obwohl weniger Knoten in der DB enthalten sind.

Bewertung der Abfrage Antwortzeiten Die Auswertung der Abfrage Antwortzeiten zeigt, dass Graphoperationen wie das Durchlaufen bzw. Traversieren eines Graphen über Kanten (Ancestors und Descendants) auch in relationalen Datenbanken ohne weitere Mittel performant implementiert werden können und dabei sogar bessere Antwortzeiten als Graphdatenbanken liefern können. Für komplexere Abfragen, wie das Finden eines kürzesten Pfades oder das Finden der 2-Hop Nachbarschaft, zeigt sich das PostgreSQL höhere Antwortzeiten als Graphdatenbanken aufweist.

Nach der detaillierten Beschreibung der Experimente in diesem Kapitel folgt die Diskussion der Ergebnisse und die Ermittlung des DBMS, das für die Verwaltung von Provenance Graphen am besten geeignet ist im folgenden Kapitel.

7. Diskussion der Experimentergebnisse

In diesem Kapitel werden die detaillierten Ergebnisse der Experimente zusammengefasst und bewertet, um eine Einschätzung über die Verwendung der betrachteten DBMS zur Verwaltung von Provenance Graphen zu treffen. Zuerst werden die Ergebnisse aus den Experimenten in Kapitel 6 mit den priorisierten Anforderungen aus Kapitel 4 abgeglichen. Danach werden einige subjektive Kriterien diskutiert. Anschließend wird eine Empfehlung für die Wahl eines DBMS zur Verwaltung für Provenance Graphen formuliert. Zum Schluss wird noch auf die Limitationen und Problemstellungen der Testumgebung eingegangen.

7.1. Einordnung der Ergebnisse ins Gesamtziel

Im folgenden Abschnitt werden die Anforderungen, die in Abschnitt 4.2 zu den Basisanforderungen priorisiert wurden, mit den Ergebnissen der Evaluation abgeglichen und ein Ranking formuliert, welches die untersuchten DBMS entsprechend der Ergebnisse einteilt.

Hoher Durchsatz Diese Ziel konnte durch die technischen Einschränkungen der Testumgebung durch die Experimente nicht ausreichend überprüft werden, daher lässt sich keine verlässliche Aussage zu diesem Ziel treffen. Zwar ist es möglich mithilfe des Publishers Daten in ausreichender Geschwindigkeit zu veröffentlichen, aber die Subscriber verarbeiten die Daten mit zu geringer Geschwindigkeit, so dass die Daten nicht mit der geforderten Geschwindigkeit in die DBMS eingefügt werden konnten. Dennoch lässt sich die Erkenntnis ziehen, dass aufgrund der konzeptionellen Unterschiede der Einfügeoperationen zwischen den DBMS relationale Datenbanken beim Einfügen wesentlich performanter abschneiden. Für weitere Experimente bietet sich hier an die Einfügeoperationen zu optimieren. Für relationale Datenbanken wäre die erste Optimierung diese als Batch zu verarbeiten, durch das Erzeugen einer einzigen Einfügeoperation

für alle Zeilen anstelle des Erzeugens einer Einfügeoperation für jede Zeile einer Nachricht. Bei den Graphdatenbanken zeigt sich, dass Memgraph die schnellsten Einfügeoperationen aufweist und damit performanter als Neo4j und ONgDB gesehen werden kann.

Ranking der DBMS bezogen auf das Ziel *Hoher Durchsatz* unter Berücksichtigung der technischen Schwierigkeiten:

1. PostgreSQL
2. Memgraph
3. Neo4j
4. ONgDB

Latenz bei Schreiblast Die Ergebnisse der Experimente haben gezeigt, dass für *simple* Graphoperationen PostgreSQL ähnliche Antwortzeiten wie Graphdatenbanken während kontinuierlicher Schreiboperationen liefert. Bei komplexen Graphoperationen weisen die Graphdatenbanken jedoch bessere Antwortzeiten auf. Von den untersuchten Graphdatenbanken weisen Neo4j und Memgraph ähnliche Antwortzeiten auf, während ONgDB etwas langsamer antwortet. Memgraph weist jedoch eine bessere Ressourcennutzung als Neo4j auf.

Ranking der DBMS bezogen auf das Ziel *Latenz bei Schreiblast*:

1. Memgraph
2. Neo4j
3. PostgreSQL
4. ONgDB

Latenz bei Datenvolumen Da sich mit steigender Experimentendauer das Datenvolumen kontinuierlich erhöht, ist zu erwarten, dass auch die Abfragelatenzen mit der Dauer des Experiments zunehmen. Tatsächlich ist zu sehen, dass in den Graphdatenbanken die Laufzeit bei steigender Experimentdauer geringer ansteigt, als bei der relationalen Datenbank. Daher schneiden die Graphdatenbanken bei der Erfüllung dieses Ziels etwas besser ab. Zusätzlich konnte gezeigt werden, dass bei PostgreSQL und Neo4j Leseoperationen auf der Festplatte zu höheren Abfragelatenzen führen. Es ist zu erwarten, dass sich mit steigendem Datenvolumen die Abfragelatenz aufgrund der notwendigen Leseoperationen weiter erhöht. Bei den Graphdatenbanken erweist sich der Arbeitsspeicher als möglicher Flaschenhals. Neo4j und Memgraph weisen eine ähnliche Entwicklung der Antwortzeiten auf bei steigendem Datenvolumen auf. Jedoch zeigt sich im Ressourcenverbrauch, dass Memgraph im Vergleich zu Neo4j etwas besser abschneidet.

Ranking der DBMS bezogen auf das Ziel *Latenz bei Datenvolumen*:

1. Memgraph
2. Neo4j
3. ONgDB
4. PostgreSQL

Abfragen zur Offline Analyse In der Formulierung der Abfragen bieten Graphdatenbanken deutliche Vorteile in Bezug auf die Komplexität und auch den Umfang der Abfragen. Besonders durch die auf Graphoperationen optimierte Abfragesprache lassen sich Graphabfragen komfortabel formulieren. Da die hier untersuchten Datenbanken die gleiche Abfragesprache unterstützen, liegen die Graphdatenbank im Ranking auf dem selben Platz. Auch in relationalen Datenbanken lassen sich simple Graphoperationen mithilfe von SQL ohne erhebliche Komplexität formulieren, jedoch nicht im Ausmaß einer Graphdatenbank.

Ranking der DBMS bezogen auf das Ziel *Abfragen zur Offline Analyse*:

1. Memgraph
1. Neo4j
1. ONgDB
2. PostgreSQL

Forschungsabfragen Bei den komplexeren Graphoperationen haben die Graphdatenbanken deutliche Vorteile gegenüber den relationalen Datenbanken. Besonders die built-in Operationen für Breiten- und Tiefensuche sowie die für Graph Data Science bereitgestellten Bibliotheken in Neo4j und Memgraph lassen diese beiden DBMS im Ranking auf den oberen Plätzen landen. Die Möglichkeit Memgraph durch eigene C++ oder Python-Bibliotheken zu erweitern, was die Möglichkeit zur Bildung einer Community darstellt, bietet noch einen Vorteil gegenüber Neo4j, wodurch Memgraph auf dem ersten Platz abschneidet.

Ranking der DBMS bezogen auf das Ziel *Forschungsabfragen*:

1. Memgraph
2. Neo4j
3. ONgDB
4. PostgreSQL

Wahl des DBMS anhand subjektiver Kriterien Neben den technischen messbaren Eigenschaften spielen auch subjektive Kriterien eine Rolle bei der Auswahl eines DBMS. In [VMZ⁺10], siehe Kapitel 3, wird auf subjektive Kriterien bei der Auswahl von DBMS eingegangen. Dazu zählen die vier Kriterien *Ausgereiftheit und Support*, *Komplexität der Entwicklung*, *Flexibilität* und *Sicherheit*. Generelle Informationen über die PostgreSQL, Neo4j und Memgraph sind auf dieser Webseite zusammengefasst¹.

In ersten Punkt *Ausgereiftheit und Support* weist PostgreSQL Vorteile auf, durch langjährigen Betrieb als de-facto Standard für DBMS. Neo4j und Memgraph bieten beide eine ausführliche Dokumentation und und Support durch eine große Anwenderbasis. Neo4j kann jedoch als etwas ausgereifter betrachtet werden aufgrund der Veröffentlichung in 2007, während Memgraph 2017 veröffentlicht wurde. ONgDB wurde in der Alpha-Version 2021 veröffentlicht².

1. PostgreSQL
2. Neo4j
3. Memgraph
4. ONgDB

Im zweiten Punkt *Komplexität der Entwicklung* beruht das Ranking auf den Erfahrungen, die im Umgang mit den DBMS bei der Entwicklung der Testumgebung dieser Arbeit entstanden sind. Mit Memgraph und ONgDB kam es bei der Bereitstellung der internen DBMS Metriken und Monitoring zu Schwierigkeiten. ONgDB zeigt sich etwas komplexer, durch die Notwendigkeit eine veraltete Version des DBMS-Treibers *Neo4j*³ für die Implementierung des Subscribers und der Abfrageanwendung einzusetzen. Postgres zeigte sich von den untersuchten Datenbanken am unkompliziertesten. Die Bereitstellung der Systeme als Docker Container war für jedes DBMS problemlos.

1. Postgres
2. Neo4j
3. ONgDB
4. Memgraph

Im dritten Punkt *Flexibilität* können die schemalosen Graphdatenbanken gegenüber PostgreSQL punkten und zeichnen sich dadurch als flexibler aus.

1. Memgraph
1. Neo4j

¹Vergleich DB Featues - <https://db-engines.com/en/system/Memgraph%3BNeo4j%3BPostgreSQL>

²Homepage ONgDB - <https://graphfoundation.org/ongdb/1.0.0-alpha01/>

³Link zum DBMS Treiber - <https://neo4j.com/docs/api/python-driver/current/>

1. ONgDB
4. PostgreSQL

Im vierten Punkt *Sicherheit* liegt PostgreSQL im Vergleich vor den Graphdatenbanken. PostgreSQL bietet fein-granulares RBAC und Zugangsrechte. In Neo4j Benutzer, Rollen und Rechte und Unterstützung von Standards wie LDAP, Active Directory, Kerberos. In Memgraph lassen sich in Punkten Sicherheit nur Benutzer, Rollen, Rechte einsetzen⁴.

1. PostgreSQL
2. Neo4j
3. ONgDB
4. Memgraph

Tabelle 7.1: Zusammenfassung des Rankings der DBMS.

	Ziele und Eigenschaften								
	Objektive Kriterien					Subjektive Kriterien			
DBMS	Hoher Durchsatz	Latenz Schreiblast	Latenz Datenvolumen	Offline Analyse	Forschungsabfragen	Ausgereiftheit	Komplexität	Flexibilität	Sicherheit
PostgreSQL	1	3	4	2	4	1	1	4	1
Neo4j	3	2	2	1	2	2	2	1	2
Memgraph	2	1	1	1	1	3	4	1	4
ONgDB	4	4	3	1	3	4	3	1	3

Aus der Zusammenfassung des Ranking aller objektiven und subjektiven Kriterien in Tabelle 7.1 ergeben sich die in Tabelle 7.2 ermittelten Werte für das arithmetische Mittel und den Median für das Ranking. Zwischen den objektiven und subjektiven Kriterien wurde eine Gewichtung vorgenommen, so dass objektive Kriterien mit dem 1,5-fachen Wert in die Berechnung des arith. Mittel und Medians einfließen. Dadurch soll die Bedeutung der objektiven Kriterien, die in dieser Arbeit gegenüber den subjektiven Kriterien eine wichtigere Rolle einnehmen, in den statistischen Mitteln zum Tragen kommen. Nach diesen Ergebnissen erweist sich von den vier betrachteten DBMS Memgraph als am besten geeignete für die Verwaltung von Provenance Graphen.

⁴Vergleich DB Features - <https://db-engines.com/en/system/Memgraph%3BNeo4j%3BPostgreSQL>

Tabelle 7.2: Metriken des Rankings.

DBMS	Arithm. Mittel	Median
Memgraph	1,89	2,22
Neo4j	1,89	2,44
PostgreSQL	3,11	3
ONgDB	3,72	4

7.2. Handlungsempfehlungen für ein DBMS zur Verwaltung von Provenance Graphen

Aus den Ergebnissen der Experimente lassen sich noch weitere Handlungsempfehlungen für die Wahl eines DBMS ableiten. Für simple Graphoperationen wie Graph-Traversierung ist eine relationale Datenbank performant einsetzbar. Für komplexen Graphoperationen empfiehlt sich die Verwendung von Graphdatenbanken und wenn möglich die Nutzung der built-in Bibliotheken. Die Auswertung des Hardware Ressourcenverbrauchs zeigt, dass PostgreSQL sich gut eignet, um die hier verwendete Datenmenge zu bewältigen, während Graphdatenbanken einen hohen Ressourcenverbrauch haben. Die Ausnahme bei den Graphdatenbanken ist Memgraph im Verbrauch des Arbeitsspeichers. Die ausschließliche Verwendung von Graphdatenbanken zur Verwaltung von Provenance Graphen birgt das Risiko eines Flaschenhalses aufgrund der Latenz der Schreiboperationen. Hier eignet sich der Einsatz von DBMS, die auf performante Schreiboperationen optimiert sind.

Für zusammengefasste Abfragen eignen sich relationale Datenbanken, da alle Knoten und alle Kanten jeweils in einer Tabelle gespeichert werden. Mehrere Einfügeanweisungen können als einzige Operation zusammengefasst werden, da keine weiteren Tabellen oder Referenzierungen beachtet werden müssen [HC17]. Nachteile weisen relationale Datenbanken bei Abfragen auf Graphdaten auf. Graph Abfragen wie *kürzester Pfad*, die Traversierung des Graphen oder Finden der Nachbarschaft eines Knotens, die in Graphdatenbanken bereits implementiert sind, müssen für relationale Datenbanken nachimplementiert werden [HC17]. Es sei an dieser Stelle erwähnt, dass die Nachteile von *reinen* relationalen DBMS durch Erweiterungen abgemindert werden. Für PostgreSQL existieren Graph-Erweiterungen wie Citus⁵ oder AgensGraph⁶.

Bereits in [BTPT21] auf die Grenzen von Graphdatenbanken im Umgang von Big Data eingegangen. Es wird als Vertreter für Graphdatenbanken Neo4j mit dem Verarbeitungs-

⁵Git Repository Citus - <https://github.com/citusdata/citus>

⁶Git Repository AgensGraph - <https://github.com/bitnine-oss/agensgraph>

framework Apache Spark hinsichtlich der Performance und Grenzen der Verarbeitung verglichen. Als Ergebnis zeigt sich, dass für Neo4j die Grenzen durch den Arbeitsspeicher gesetzt werden, da Neo4j Graphdaten zur Verarbeitung vollständig in den Arbeitsspeicher lädt. Solange jedoch diese Grenze noch nicht erreicht ist, schneidet Neo4j performanter als Apache Spark ab. In dieser Arbeit deutet sich für Neo4j und ONgDB eine ähnliche Limitation ab, wobei in den Experimenten nicht die Grenzen des Arbeitsspeichers erreicht werden. Memgraph stellt sich als Ausnahme heraus, da hier der durchschnittliche Arbeitsspeicherverbrauch wesentlich geringer ist, als bei Neo4j und ONgDB.

Aufgrund der hier genannten Erkenntnisse erscheint ein Einsatz von hybriden bzw. polygloter Persistenz sinnvoll, bei dem bspw. ein DBMS als *Langzeitspeicher* mit hoher Schreibgeschwindigkeit (von den untersuchten DBMS bspw. PostgreSQL) Provenance Daten speichert und ein weiteres DBMS, welches die Daten aus dem *Langzeitspeicher* liest, visualisiert und komplexe Abfragen und Auswertungen ermöglicht (von den untersuchten DBMS bspw. Memgraph).

7.3. Limitationen und Einschränkungen

Im folgenden Abschnitt werden Limitationen und Einschränkungen vorgestellt, die während der Entwicklung der Testumgebung und Durchführung der Experimente aufgetreten sind. Die hier aufgeführten Punkte können als Anknüpfungspunkte für weitere Evaluationen und Forschung betrachtet werden.

Die folgenschwerste Limitation in der Testumgebung ist die Verzögerung bei der Einfügeoperation der Subscriber, wodurch die gewünschten Schreibraten nicht konkret überprüft bzw. erreicht werden konnten. Hier bedarf es Optimierung der Schreiboperationen, um die geforderten Schreibgeschwindigkeiten zu erreichen.

Während der Experimente fiel auf, dass beim geplanten gleichzeitigen Betrieb aller DBMS in einem Docker-Stack Abstürze den Ressourcenverbrauch und Betrieb anderer Systeme beeinflussen. Für eine bessere Vergleichbarkeit, wurden daher die DBMS einzeln getestet, um gegenseitige Einflüsse der Datenbanken aus den Ergebnissen auszuschließen. Abbildung 7.1 zeigt die Auswirkungen eines Systemabsturzes auf die weiteren Systeme. In der Abbildung ist die CPU Auslastung der DBMS ONgDB (lila), memgraph (orange), Neo4j (blau), PostgreSQL (grün) dargestellt. Kurz vor 03:10h kam es zu einem Absturz der Neo4j Datenbanken (in der Grafik der blaue Graph). Zeitgleich sind die restlichen drei DBMS vom Absturz betroffen. Kurz nach 03:10h starten die restlichen DBMS wieder während Neo4j weiterhin offline bleibt.

Ein weiteres Manko der Testumgebung ist, dass das Senden der Abfragen zum Aufzeichnen der Latenzen gleichzeitig mit den Schreiboperationen der Subscriber geschieht.

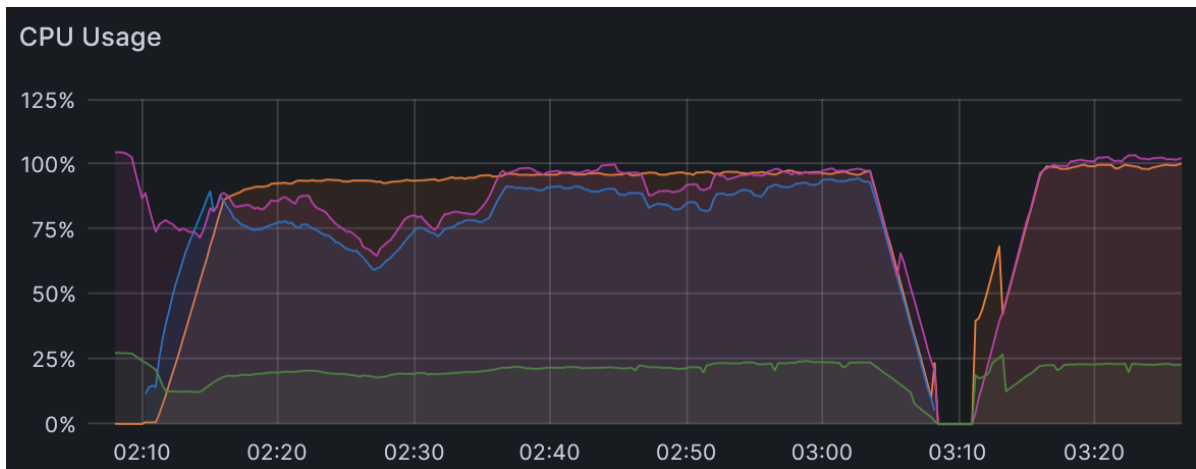


Abbildung 7.1.: Auszug aus Grafana Dashboard - Vergleich CPU Metriken bei Absturz.

Zum einen kann hier argumentiert werden, dass dies den realen Betrieb einer Provenance Verwaltung darstellt, in dem neben Einfügeoperationen auch regelmäßig Abfragen ausgeführt werden würden. Auf der anderen Seite erzeugen die regelmäßigen Einfügeoperationen auch Last auf den Datenbanken, welche die reine Antwortzeit der DBMS auf die Abfragen beeinflussen können.

8. Fazit

In dieser Arbeit konnte gezeigt werden welche DBMS sich zur Verwaltung von Provenance Graphen eignen. Dafür wurden die Anforderungen an eine Datenverwaltung für Provenance Graphen recherchiert, Basisanforderungen identifiziert und anschließend durch Experimente untersucht welches DBMS die Basisanforderungen erfüllt. Von den hier betrachteten DBMS sticht Memgraph als am besten geeignete Datenbank heraus. Es konnte gezeigt werden, dass Graphdatenbanken sowie relationale Datenbanken für den Einsatz als alleinige Datenverwaltung für Provenance Graphen nicht geeignet sind, da nicht sie alle Basisanforderungen erfüllen können. Die Ergebnisse der Arbeit zeigen, dass relationale Datenbanken gut für das Speichern der Daten geeignet sind, aber Limitationen in den Abfragemöglichkeiten und Antwortzeiten aufzeigen. Graphdatenbanken eignen sich dafür diese Limitationen zu mitigieren, bedürfen aber dafür eines höheren Ressourcenverbrauchs.

Die hier angewendete Methodik zur Wahl eines DBMS kann für weitere Anwendungsgebiete eingesetzt werden. Zur Erfassung der Anforderungen an DBMS sind in Abschnitt A Fragen formuliert, deren Beantwortung Einsichten über die Eigenschaften des benötigten DBMS liefert.

Für weitere Forschung und Untersuchung von DBMS zur Verwaltung von Provenance Graphen bietet sich die Anbindung weiterer DBMS an die bestehende Testumgebung an. Wie in Kapitel 3 beschrieben existieren Ansätze, die eine Indizierung für Provenance Graphen mit Cassandra vorschlagen. Die Ergebnisse der Arbeit zeigen, dass komplexe Graphoperationen mit PostgreSQL unperformant sind. Die Verwendung eines auf Provenance Graphen optimierten Index lässt die Verwendung von Cassandra vielversprechend erscheinen. Die Testumgebung wurde so entwickelt, dass die Anbindung weiterer DBMS durch die Implementierung entsprechender Subscriber und Erweiterung der Docker Infrastruktur möglich ist. In der entwickelten Testumgebung wird in keiner Datenbank ein Index verwendet. Während ein Index die Leseoperationen deutlich beschleunigen kann, bedeutet die Verwaltung eines Index potenziell einen Einfluss auf die Performance von Schreiboperationen und erhöhten Speicherverbrauch. Zusätzliche Experimente mit der Verwendung von Indizes wären der nächste Schritt in dieser Arbeit gewesen, um den Trade-Off zwischen Schreibgeschwindigkeit und Lesegeschwindigkeit zu beurteilen. Außerdem wäre bei einer folgenden Untersuchung die Verwendung weiterer Datensätze neben dem Cadets Datensatz interessant.

Literaturverzeichnis

- [AMCH19] Adel Alshamrani, Sowmya Myneni, Ankur Chowdhary, and Dijiang Huang. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Communications Surveys & Tutorials*, 21(2):1851–1877, 2019.
- [BTBM15] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium, SEC’15*, pages 319–334, USA, August 2015. USENIX Association.
- [BTPT21] Ioannis Ballas, Vassilios Tsakanikas, Evaggelos Pefanis, and Vassilios Tampakas. Assessing the computational limits of GraphDBs’ engines - A comparison study between Neo4j and Apache Spark. In *Proceedings of the 24th Pan-Hellenic Conference on Informatics, PCI ’20*, pages 428–433, New York, NY, USA, March 2021. Association for Computing Machinery.
- [CLL⁺23] Zijun Cheng, Qiujuan Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. Kairos: Practical Intrusion Detection and Investigation using Whole-system Provenance, August 2023. arXiv:2308.05034 [cs].
- [EFHB11] Prof. Dr. Stefan Edlich, Achim Friedland, Benjamin Hampe, Jens und Brauer, and Markus Brückner. *NoSQL – Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken - 2. Auflage*. Carl-Hanser Verlag, 2011.
- [GCD20] Yuanzhao Gao, Xingyuan Chen, and Xuehui Du. A Big Data Provenance Model for Data Security Supervision Based on PROV-DM Model. *IEEE Access*, 8:38742–38752, 2020.
- [GKC⁺19] John Griffith, Derrick Kong, Amando Caro, Joud Benyo, Brettand Khoury, Timothy Upthegrove, Timothy Christovich, Stanislav Ponomorov, Ali Sydney, Arjun Saini, Vladimir Shurbanov, Christopher Willig, David Levin, and Jack Dietz. Scalable Transparency Architecture for Research Collaboration (STARC)-DARPA Transparent Computing (TC) Program. In *2017 Communication and Information Technologies (KIT)*, Nov 2019.

- [GXL⁺18a] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. SAQL: a stream-based query system for real-time abnormal system behavior detection. In *Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18*, pages 639–656, USA, August 2018. USENIX Association.
- [GXL⁺18b] Peng Gao, Xusheng Xiao, Zhichun Li, Kangkook Jee, Fengyuan Xu, Sanjeev R. Kulkarni, and Prateek Mittal. AIQL: Enabling Efficient Attack Investigation from System Monitoring Data, June 2018. arXiv:1806.02290 [cs].
- [HC17] Thomas Heinis and Adriane Chapman. Provenance Storage. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1–4. Springer New York, New York, NY, 2017.
- [HPB⁺20] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. arXiv:2001.01525 [cs].
- [HWdS⁺17] Fernanda Hondo, Polyane Werceles, Waldeyr da Silva, Klayton Castro, Ingrid Santana, Maria Emília Walter, Aletéia Araújo, Maristela Holanda, and Sergio Lifschitz. Data provenance management for bioinformatics workflows using NoSQL database systems in a cloud computing environment. In *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1929–1934, November 2017.
- [Kas20] Andrii Kashliev. Storage and Querying of Large Provenance Graphs Using NoSQL DSE. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 260–262, May 2020.
- [KEK⁺22] Kabul Kurniawan, Andreas Ekelhart, Elmar Kiesling, Gerald Quirchmayr, and A Min Tjoa. KRYSTAL: Knowledge graph-based framework for tactical attack discovery in audit data. *Computers & Security*, 121:102828, October 2022.
- [Lan01] Doug Laney. 3d data management: Controlling data volume, velocity, and variety. techreport, META Group, 2001.
- [LCYC21] Zhenyuan Li, Qi Alfred Chen, Runqing Yang, and Yan Chen. Threat Detection and Investigation with System-level Provenance Graphs: A Survey, December 2021. arXiv:2006.01722 [cs].

- [LD20] Changhong Liu and Hancong Duan. A Graph Database Storage Engine for Provenance Graphs. In *DBKDA 2020, The Twelfth International Conference on Advances in Databases, Knowledge, and Data Applications*, Lisbon, Portugal, 2020.
- [MCF⁺11] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van Den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011.
- [MG16] Thomas Moyer and Vijay Gadepally. High-throughput ingest of data provenance records into Accumulo. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, September 2016.
- [MM13] Luc Moreau and Paolo Missier. PROV-DM: The PROV Data Model, April 2013.
- [PHG⁺17] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, Santa Clara California, September 2017. ACM.
- [Poh21] Klaus Pohl. *Basiswissen Requirements Engineering: Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level*. dpunkt.verlag, Heidelberg, 5., überarbeitete und aktualisierte auflage edition, 2021.
- [PSR23] Bofeng Pan, Natalia Stakhanova, and Suprio Ray. Data Provenance in Security and Privacy. *ACM Computing Surveys*, 55(14s):1–35, December 2023.
- [SF13] Pramod J. Sadlage and Martin Fowler. *NoSQL Distilled - A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, Jan 2013.
- [SS13] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, San Diego, CA, USA, May 2013. IEEE.
- [VMZ⁺10] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference, ACM SE '10*, pages 1–6, New York, NY, USA, April 2010. Association for Computing Machinery.

- [WH17] Simon Woodman and Hugo Hiden. Applications of provenance in performance prediction and data storage optimisation. *Future Generation Computer Systems*, 75, January 2017.
- [YLJ19] Han Yu, Aiping Li, and Rong Jiang. Needle in a Haystack: Attack Detection from Large-Scale System Audit. In *2019 IEEE 19th International Conference on Communication Technology (ICCT)*, pages 1418–1426, Xi’an, China, October 2019. IEEE.
- [ZGCD23] Michael Zipperle, Florian Gottwalt, Elizabeth Chang, and Tharam Dillon. Provenance-based Intrusion Detection Systems: A Survey. *ACM Computing Surveys*, 55(7):1–36, July 2023.

A. Anhang

Generischer Fragebogen zur Erfassung der Anforderungen von DBMS

online

Klassische Fragen (hier inspiriert ¹) um die Anforderungen an ein DBMS zu bestimmen:
Allgemein:

- Wie viele Nutzer gibt es, die Zugriff auf die Datenbank haben?
- Hat man strukturierte oder unstrukturierte Daten?
- Ändert sich die Datenstruktur / Flexibilität der Datenstruktur notwendig?
- Welcher Wert wird auf die Konsistenz der Daten gelegt?
- Welche Sicherheitsanforderungen stehen an das DBMS?

Über die Daten:

- Gibt es simultanen Zugriff auf die Daten?
- Sind die Daten statisch oder dynamisch?
- Welcher Teil der Daten soll organisiert sein?
- Nach welchen Kriterien soll auf die Daten zugegriffen werden?
- Wie und in welchem Umfang sollen Daten geschützt werden?
- Wer hat Zugriff auf die Daten?

Weiterhin sollte man betrachten:

- Welche zusätzlichen Features bietet ein Produkt? (z.B. Volltextsuche)

¹<https://www.digitalbusiness-cloud.de/intelligentes-datenmanagement-erfolgsgarant-fuer-die-digitalisierung/>

Lehrbuch

Fragen zur Auswahl eines geeigneten Datenbanksystems aus 'NoSQL – Einstieg in die Welt nichtrelationaler Datenbanken' [EFHB11].

Grundsätzliche Anforderungen

- Datenmodell benötigt?
- Schema benötigt?
- Flexible Daten benötigt?

Anforderungen an die Daten

- Welche Art von Daten werden gespeichert? (Log Daten, Geodaten, Metadaten...)
- Was ist das optimale Speichermodell?
- Datenkonsistenz und Typsicherheit benötigt?
- Wie hängen Daten zusammen? Verknüpft, Verlinkt, unabhängig? Viele Joins oder Navigation benötigt?
- Wenn Datenschema benötigt wird, wie häufig wird es geändert?

Anforderungen an Transaktionen

- ACID oder BASE Transaktionen benötigt?
- CAP Theorem beachten (Partitionierung (P) benötigt wird, Abwägen zwischen Konsistenz (C) und Verfügbarkeit (A))

Anforderungen Performance

- Wie hoch ist das Datenvolumen?
- Wie schnell müssen Queries beantwortet werden?
- Ist mit Latenzen zu rechnen?
- Wie viele Queries werden erzeugt pro ms, Sekunde...?
- Muss das DBMS skalierbar sein?
- Wie gut lassen sich weitere Systeme anbinden (z.B. zur Datenverarbeitung)

Anforderungen Queries

- Wie komplex werden Queries aussehen?
- Sind Queries heterogen?
- Wie flexibel können Queries formuliert werden?
- Welche Funktionalitäten liefert das DBMS?

Anforderungen an Systemarchitektur

- Verteiltes System oder Monolith?
- Wie stabil ist das DBMS (Fehlertoleranz, Backup, Recovery und Crash Resistenz)?

Weitere Aspekte

- Lohnt sich der Aufwand für die Implementierung des DBMS?
- Gibt es Support und Beratung?
- Gibt es erfahrene Entwickler?
- Wie komplex, zeitaufwändig ist die Entwicklung des Systems (hinsichtlich Skalierung und Schemaänderungen)?
- Wie komplex ist die Anbindung an die Anwendung?
- Wie hoch sind die Kosten für Software, Support, Hardware, Administration?
- Gibt es Security Features?
- Integration in bestehende/existierende Systeme?

HDFS Architektur Darpa

In Abbildung A.1 ist die vorgeschlagene Architektur für eine Datenverwaltung von Provenance Daten dargestellt.

Netzwerkverkehr der Broker und DBMS

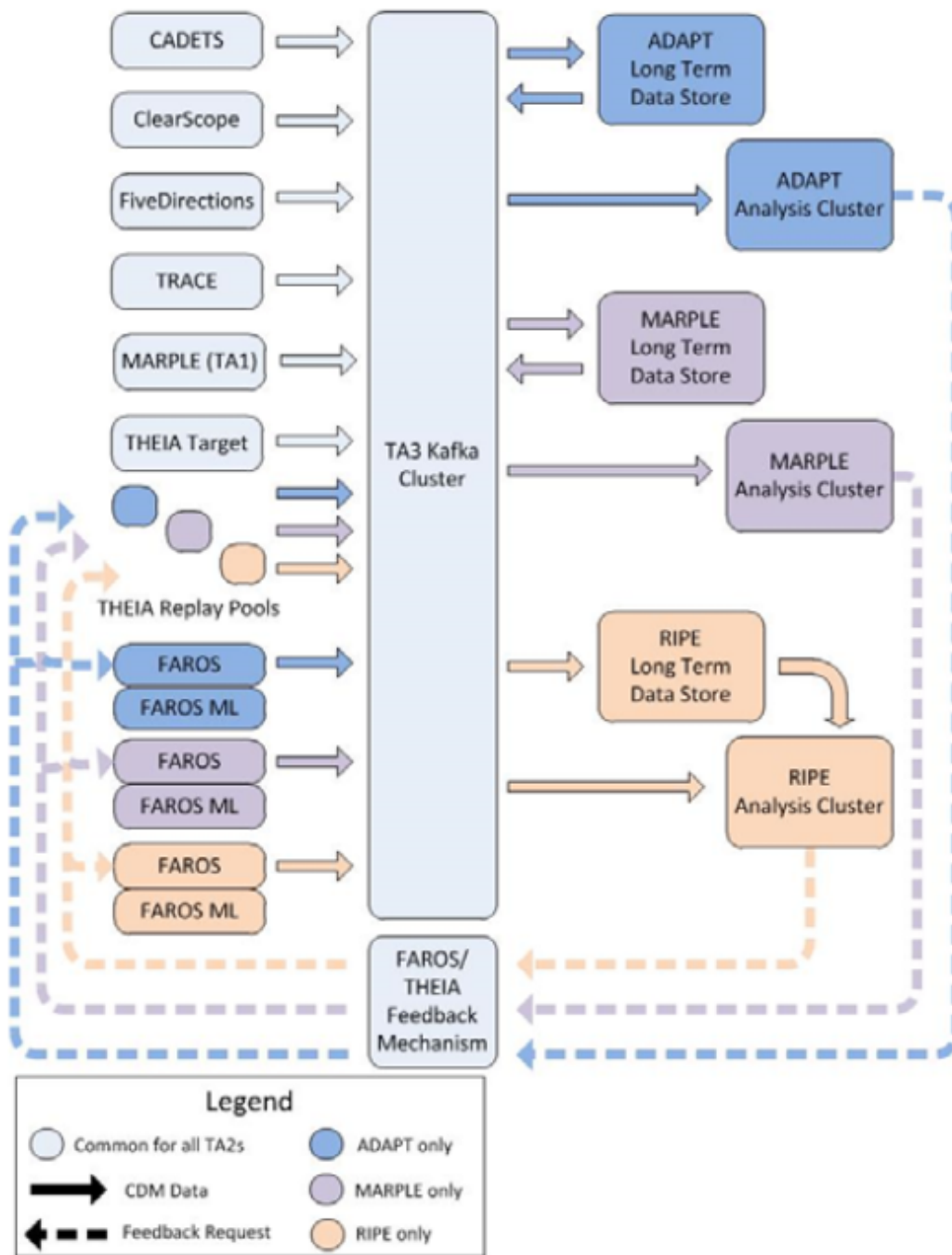


Abbildung A.1.: Architektur für einen Master Data Store für Provenance Graphen aus [GKC⁺19]

Tabelle A.1: Netzwerkverkehr der Publish-Subscriber Architektur.

Komponente	Batch- größe	max. empfan- gen in MB/s	avg. empfan- gen in MB/s	max. gesendet in MB/s	avg. gesendet in MB/s
Publisher	500	0,016	0,012	0,510	0,373
Broker Postgres	500	0,506	0,461	0,173	0,155
Broker Neo4j	500	0,511	0,460	0,079	0,036
Broker Memgraph	500	0,511	0,450	0,106	0,036
Broker ONgDB	500	0,511	0,453	0,069	0,029
Publisher	1000	0,032	0,027	0,993	0,823
Broker Postgres	1000	0,997	0,857	0,193	0,166
Broker Neo4j	1000	0,971	0,834	0,093	0,051
Broker Memgraph	1000	0,979	0,838	0,120	0,052
Broker ONgDB	1000	0,980	0,866	0,081	0,043
Publisher	1500	0,045	0,025	1,410	0,776
Broker Postgres	1500	1,41	1,28	0,195	0,179
Broker Neo4j	1500	1,39	1,12	0,116	0,065
Broker Memgraph	1500	1,40	1,22	0,129	0,065
Broker ONgDB	1500	1,38	1,18	0,091	0,054

Tabelle A.2: Netzwerkverkehr der DBMS - empfangene Daten.

DBMS	Batch- größe	max. empfangen in MB/s	avg. empfangen in MB/s
Postgres	500	0,286	0,254
Neo4j	500	0,085	0,032
Memgraph	500	0,123	0,029

Fortsetzung auf der nächsten Seite

Tabelle A.2: Netzwerkverkehr der DBMS - empfangene Daten. (Fortgesetzt von der vorherigen Seite)

ONgDB	500	0,067	0,020
Postgres	1000	0,299	0,243
Neo4j	1000	0,085	0,022
Memgraph	1000	0,124	0,026
ONgDB	1000	0,067	0,015
Postgres	1500	0,279	0,159
Neo4j	1500	0,085	0,017
Memgraph	1500	0,120	0,022
ONgDB	1500	0,065	0,012

Hardware Metriken

Tabelle A.3: Hardware Vergleich der DBMS - CPU und Schreibzugriffe.

Batchgröße	DBMS	CPU		I/O	
		max. Auslastung in %	avg. Auslastung in %	avg. Schreibzugr. in MB	avg. Lesezugr. in MB
500	Postgres	122	75,3	0,244	123
	Neo4j	103	93,9	657	4.910
	Memgraph	103	73,8	2,27	587
	ONgDB	100	90,2	86,2	3.830
1000	Postgres	119	70,2	9,100	2.890
	Neo4j	99,8	52,1 (bis Absturz Broker 89,4)	601	19.900
	Memgraph	102	52,6 (bis Absturz Broker 91,5)	118	651
	ONgDB	97,6	57,4 (bis Absturz Broker 89,4)	76,1	12.400
1500	Postgres	119	67,0	0,007	10,04
	Neo4j	101	35,5 (bis Absturz Broker 87,0)	624	8.280
	Memgraph	101	42,0 (bis Absturz Broker 89,9)	5,96	12.000
	ONgDB	97,3	39,5 (bis Absturz Broker 87,4)	0,162	27.900

Tabelle A.4: Hardware Vergleich der DBMS - Arbeitsspeicher.

DBMS	Batch- größe	RAM Verbrauch			RAM Cached		
		max. Ver- brauch in GB	min. Ver- brauch in GB	avg. Ver- brauch in GB	max. freige- bar in GB	min. freige- bar in GB	avg. freige- bar in GB
Postgres	500	0,390	0,072	0,276	1,150	0,067	0,328
Neo4j	500	1,76	1,12	1,50	0,883	0,001	0,292
Memgraph	500	1,820	0,460	0,669	0,408	0,003	0,236
ONgDB	500	1,980	0,368	1,400	0,148	0,001	0,087
Postgres	1000	0,393	0,072	0,244	0,719	0,067	0,290
Neo4j	1000	1,95	0,656	1,24	0,886	0,002	0,208
Memgraph	1000	1,31	0,275	0,519	0,409	0,013	0,161
ONgDB	1000	1,28	0,425	1,10	0,149	0,001	0,067
Postgres	1500	0,379	0,075	0,236	0,773	0,067	0,271
Neo4j	1500	2,280	0,976	1,300	0,894	0,001	0,151
Memgraph	1500	1,06	0,419	0,548	0,289	0,001	0,096
ONgDB	1500	1,76	0,620	1,48	0,145	0,001	0,049

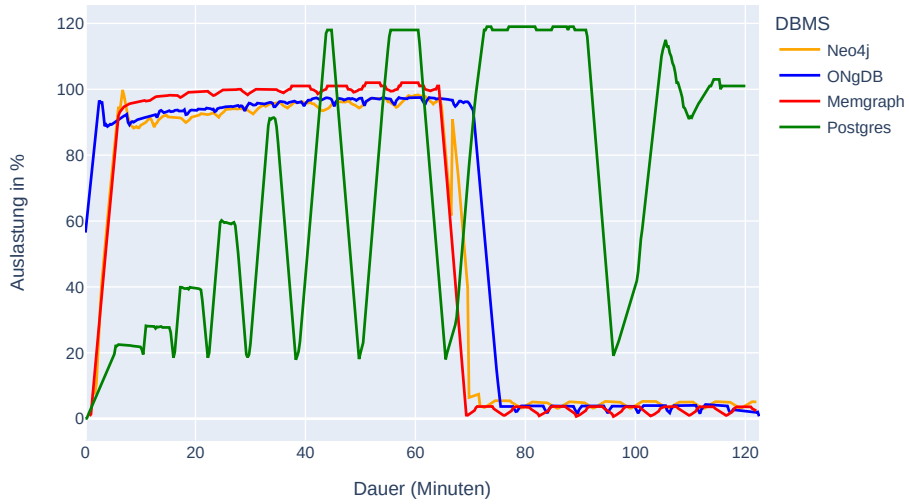


Abbildung A.2.: DBMS Metriken Hardware Ressourcen - CPU - Batchgröße 1000.

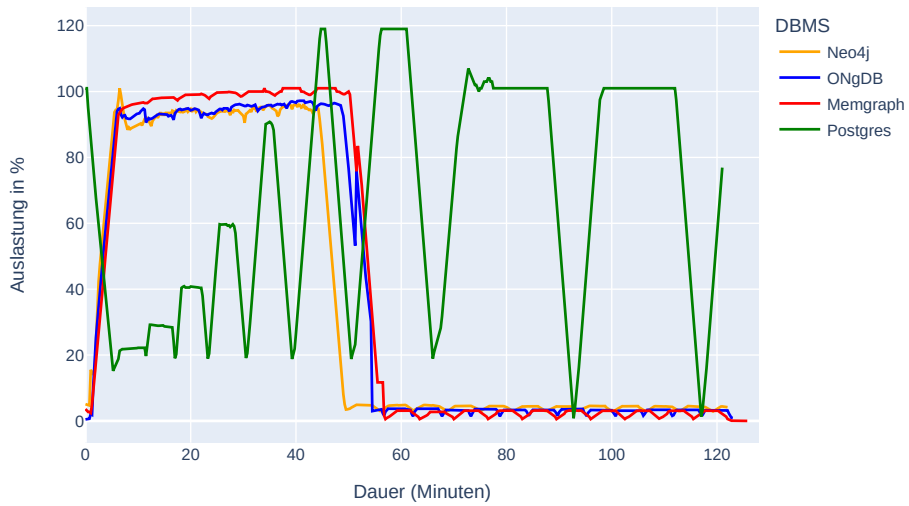


Abbildung A.3.: DBMS Metriken Hardware Ressourcen - CPU - Batchgröße 1500.

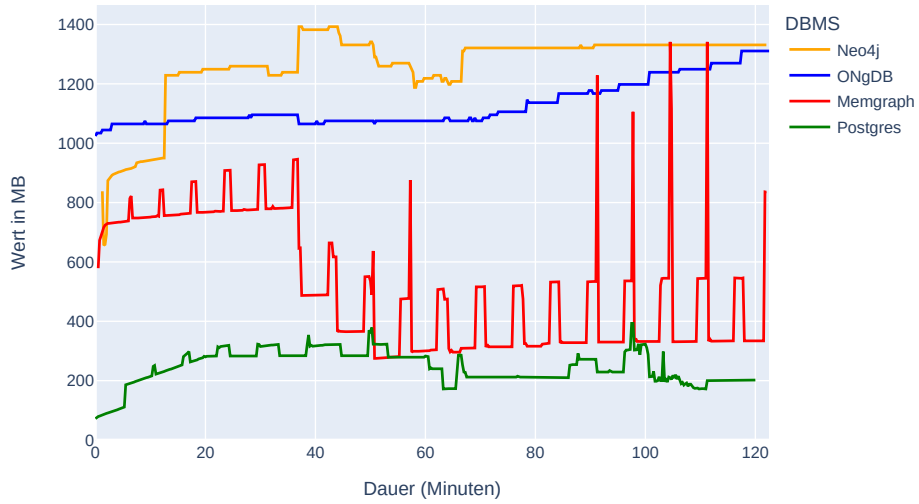


Abbildung A.4.: DBMS Metriken Hardware Ressourcen - RAM - Batchgröße 1000.

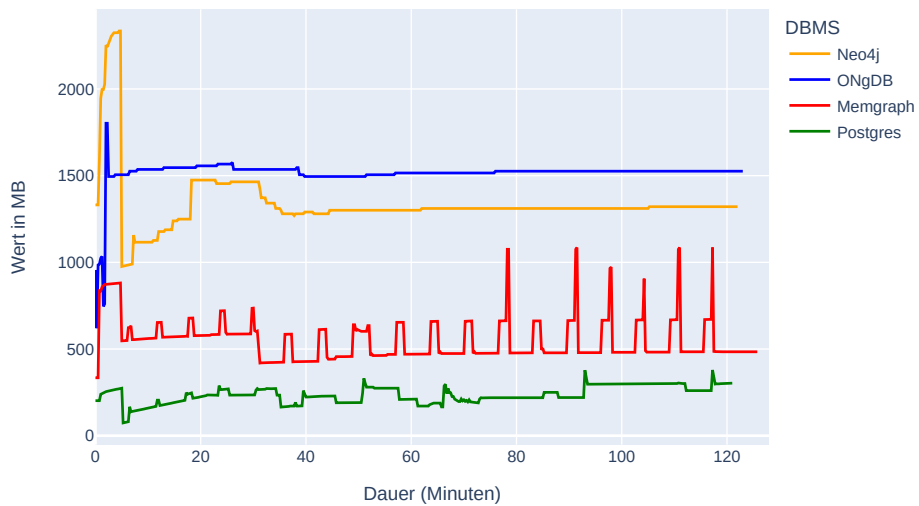


Abbildung A.5.: DBMS Metriken Hardware Ressourcen - RAM - Batchgröße 1500.

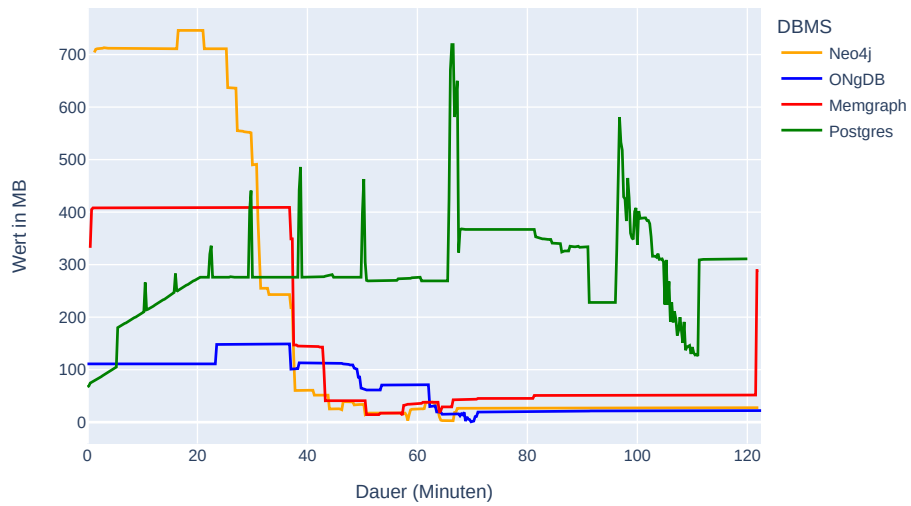


Abbildung A.6.: DBMS Metriken Hardware Ressourcen - RAM Cache - Batchgröße 1000.

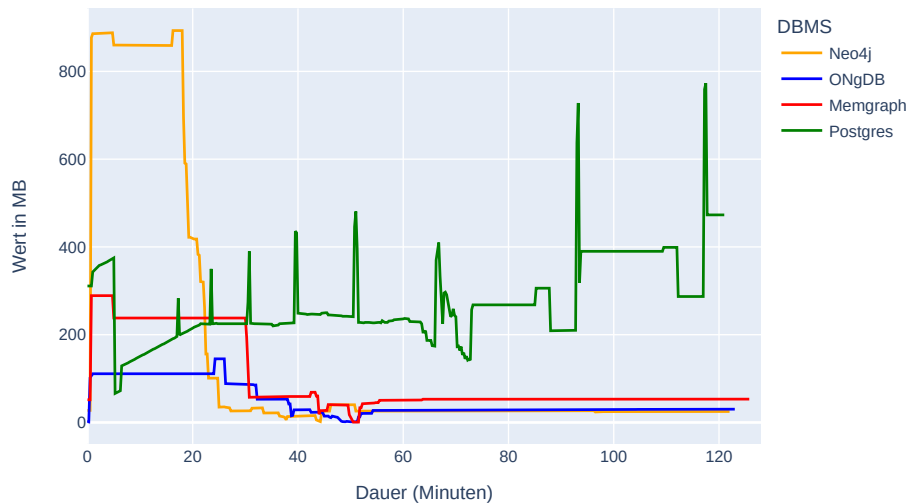


Abbildung A.7.: DBMS Metriken Hardware Ressourcen - RAM Cache - Batchgröße 1500.

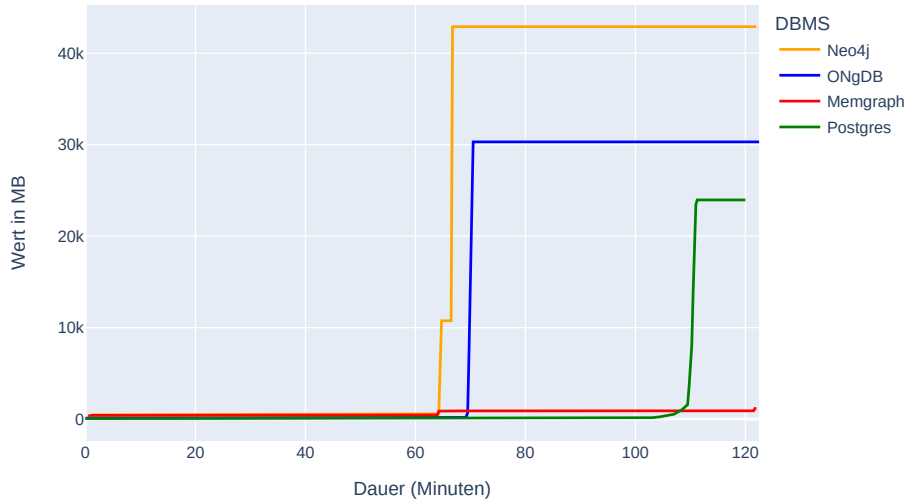


Abbildung A.8.: DBMS Metriken Hardware Ressourcen - Lesezugriffe - Batchgröße 1000.

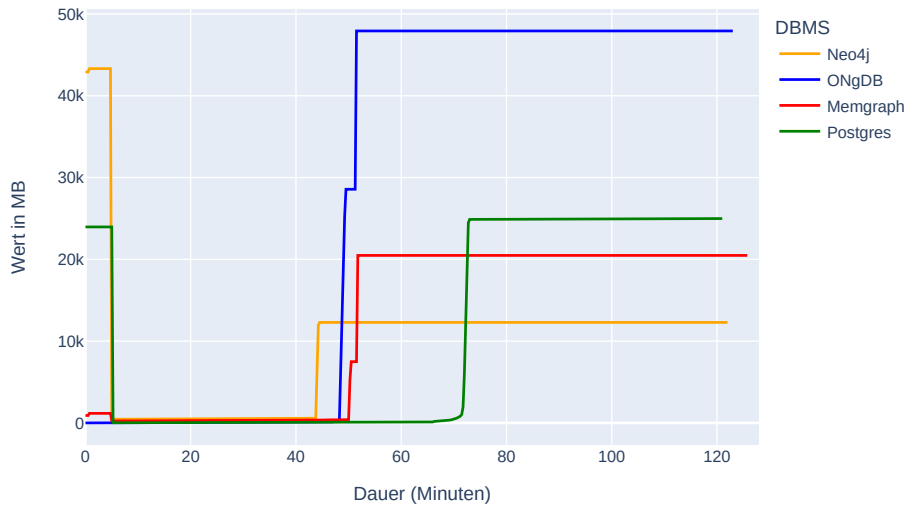


Abbildung A.9.: DBMS Metriken Hardware Ressourcen - Lesezugriffe - Batchgröße 1500.

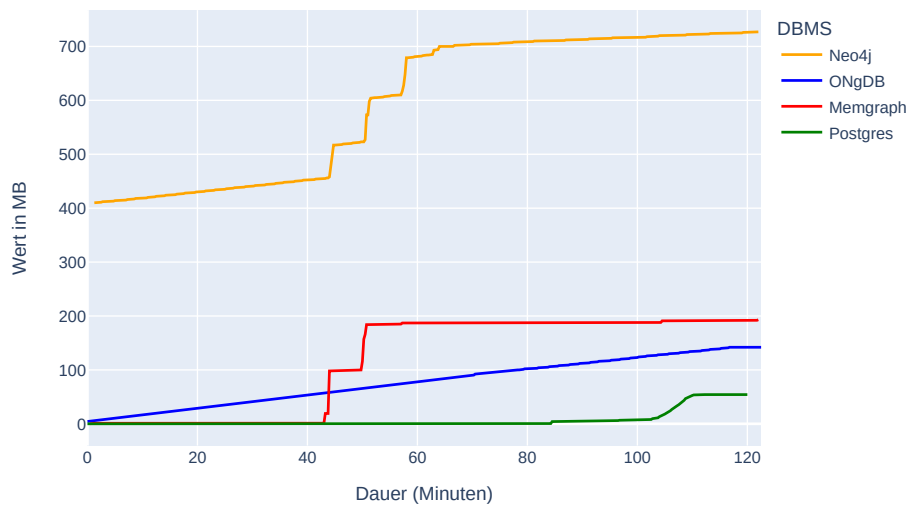


Abbildung A.10.: DBMS Metriken Hardware Ressourcen - Schreibzugriffe - Batchgröße 1000.

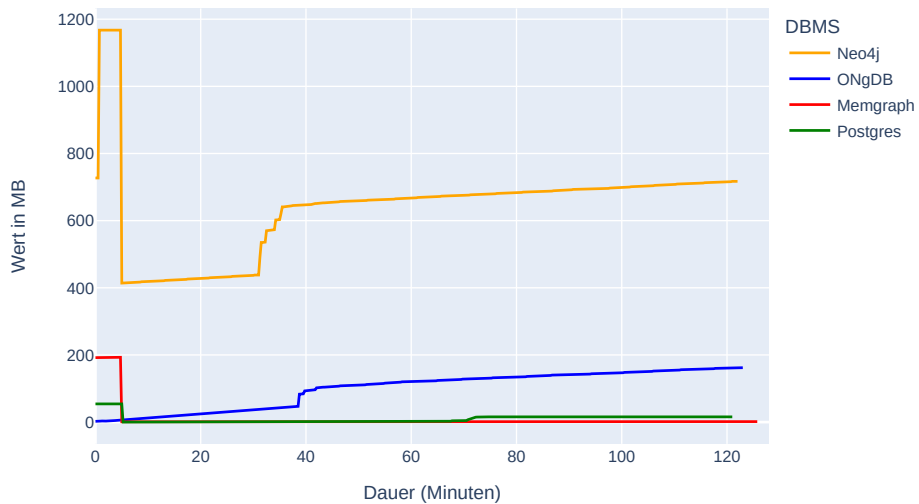
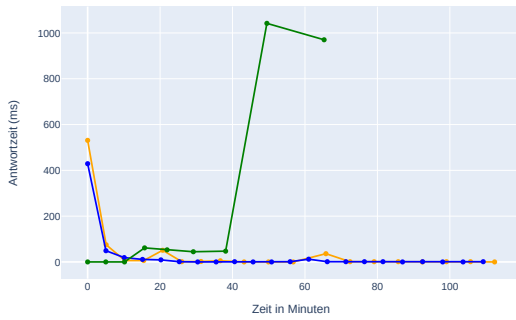


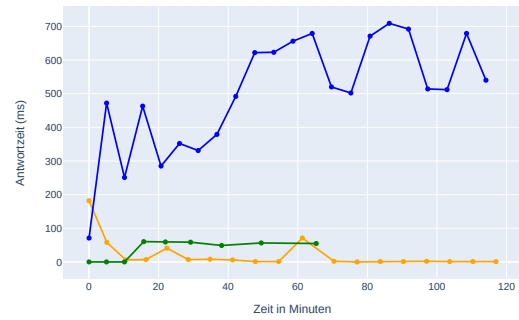
Abbildung A.11.: DBMS Metriken Hardware Ressourcen - Schreibzugriffe - Batchgröße 1500.

Ergebnisse der Abfragen

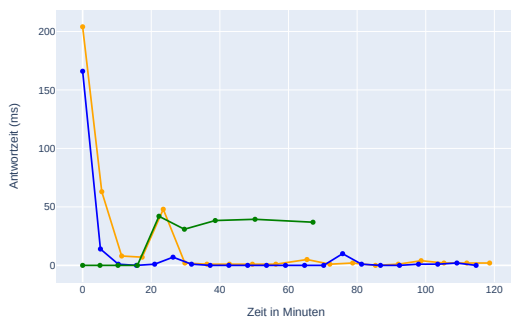
(A - Vorgänger)



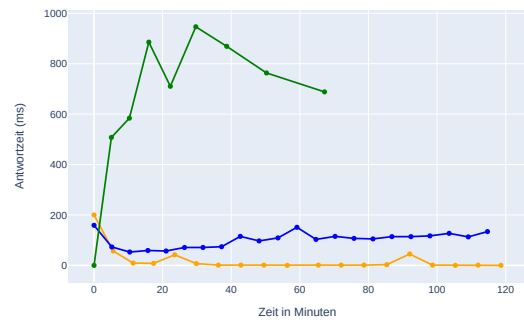
(B - Nachfolger)



(C - Pfad)



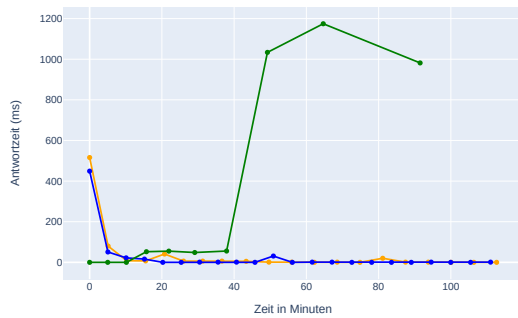
(D - 2-Hop)



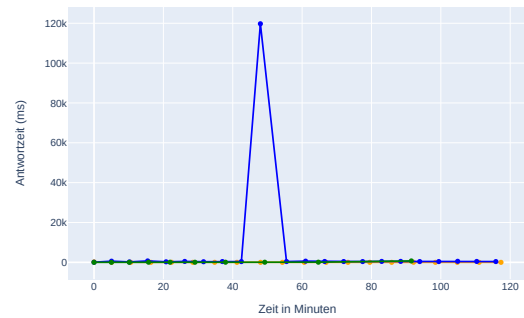
—●— Neo4j —●— ONgDB —●— Postgres

Abbildung A.12.: Abfrage Latenzen DBMS je Query bei Batchgröße von 1000

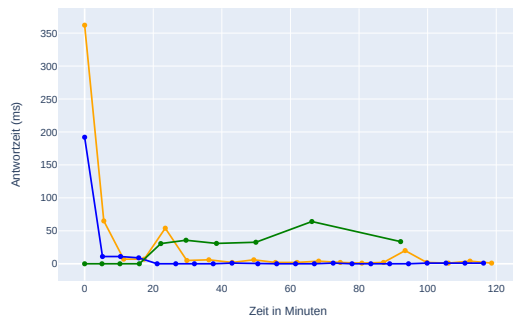
(A - Vorgänger)



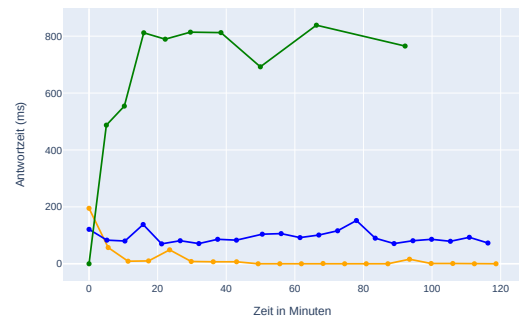
(B - Nachfolger)



(C - Pfad)



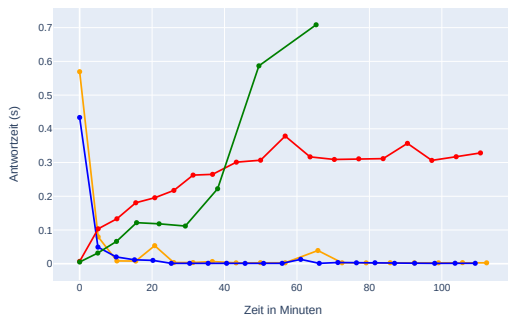
(D - 2-Hop)



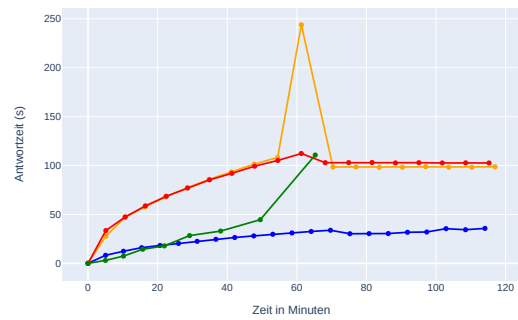
— Neo4j — ONgDB — Postgres

Abbildung A.13.: Abfrage Latenzen DBMS je Query bei Batchgröße von 1500

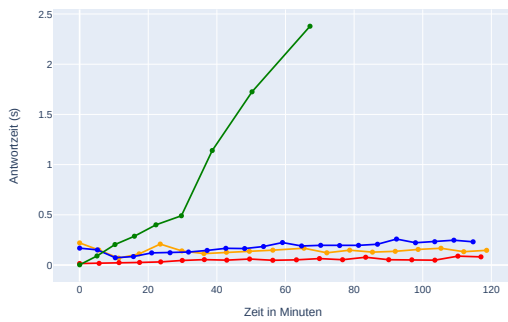
(A - Vorgänger)



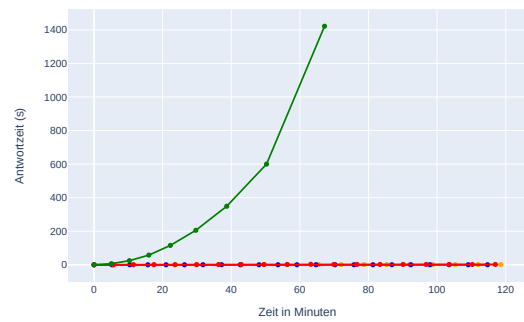
(B - Nachfolger)



(C - Pfad)



(D - 2-Hop)



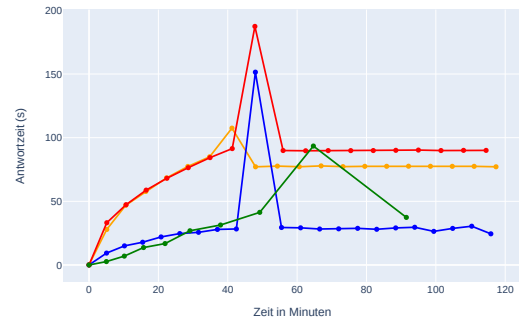
DBMS — Neo4j — ONgDB — Memgraph — Postgres

Abbildung A.14.: Abfrage Latenzen Anwendung je Query bei Batchgröße von 1000

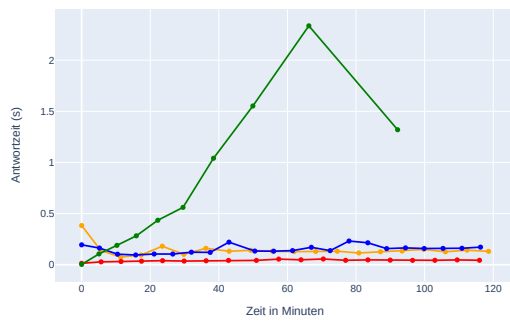
(A - Vorgänger)



(B - Nachfolger)



(C - Pfad)



(D - 2-Hop)

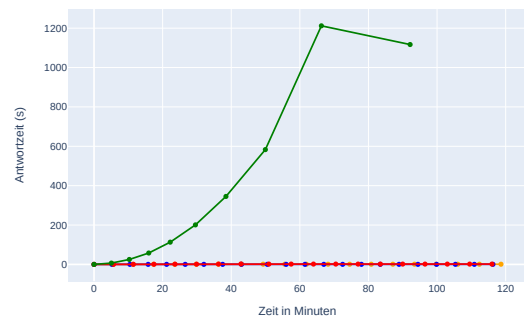
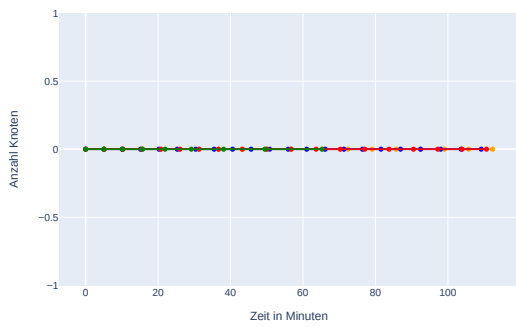
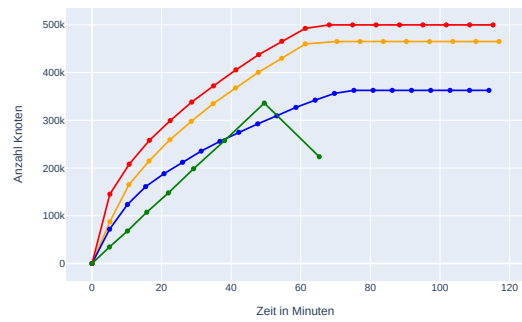


Abbildung A.15.: Abfrage Latenzen Anwendung je Query bei Batchgröße von 1500

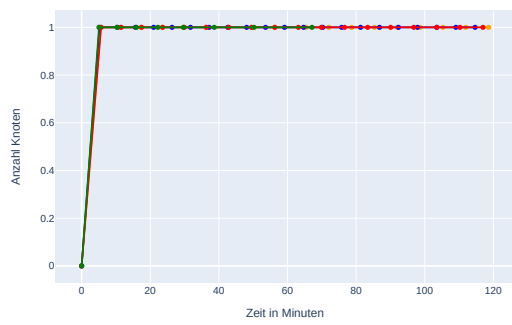
(A - Vorgänger)



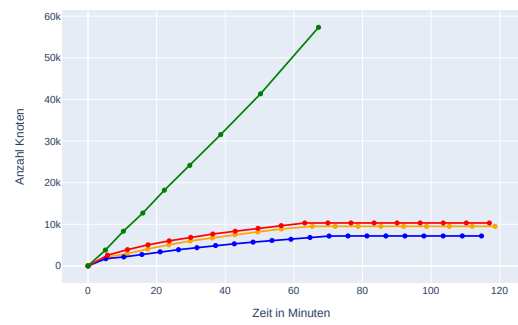
(B - Nachfolger)



(C - Pfad)



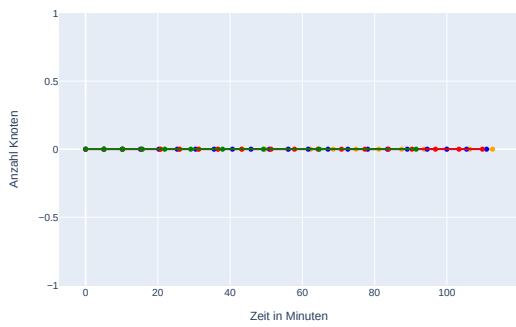
(D - 2-Hop)



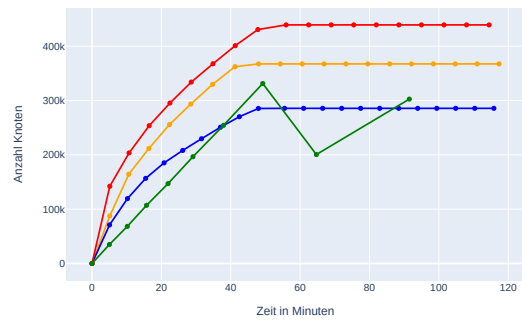
— Neo4j — ONGDB — Memgraph — Postgres

Abbildung A.16.: Ergebnismenge je Query bei Batchgröße von 1500

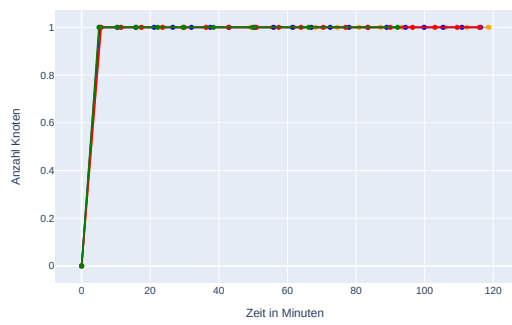
(A - Vorgänger)



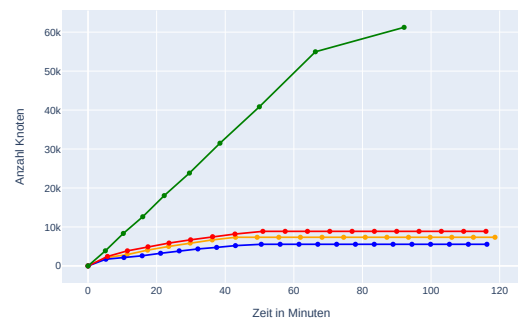
(B - Nachfolger)



(C - Pfad)



(D - 2-Hop)



— Neo4j — ONGDB — Memgraph — Postgres

Abbildung A.17.: Ergebnismenge je Query bei Batchgröße von 1500

Abfragen SQL und Cypher

```
1 WITH RECURSIVE NodeCTE AS (  
2     SELECT  
3         n.node_no ,  
4         e.source ,  
5         e.dest ,  
6         1 AS Level  
7     FROM  
8         node_list n  
9     JOIN  
10        edge_list e ON n.uuid = e.dest  
11    WHERE  
12        n.node_no = 1 -- starting node_no  
13  
14    UNION ALL  
15  
16    -- Recursive selection  
17    SELECT  
18        n.node_no ,  
19        e.source ,  
20        e.dest ,  
21        c.Level + 1 AS Level  
22    FROM  
23        edge_list e  
24    JOIN  
25        NodeCTE c ON e.dest = c.source  
26    JOIN  
27        node_list n ON e.dest = n.uuid  
28 )  
29 SELECT * FROM NodeCTE;
```

Quellcode A.1: SQL Query für das Finden der Nachfolger eines Knoten.

```

1 WITH RECURSIVE search_path(edge_no, path, dest, visited, depth)
  ↪ AS (
2   SELECT
3     e.edge_no,
4     ARRAY[e.source, e.dest]::VARCHAR[] AS path,
5     e.dest,
6     ARRAY[e.source]::VARCHAR[] AS visited,
7     1 AS depth
8   FROM
9     edge_list e
10  WHERE
11    e.source = 'A6A7C956-0132-5506-96D1-2A7DE97CB400' --
      ↪ Starting Node
12 UNION ALL
13
14  SELECT
15    e.edge_no,
16    sp.path || e.dest,
17    e.dest,
18    visited || e.dest::VARCHAR,
19    sp.depth + 1
20  FROM
21    edge_list e, search_path sp
22  WHERE
23    e.source = sp.dest
24    AND NOT (e.dest = ANY(sp.visited))
25 )
26 , shortest_paths AS (
27  SELECT
28    path,
29    depth
30  FROM
31    search_path
32  WHERE
33    dest = '8DA367BF-36C2-11E8-BF66-D9AA8AFF4A69' --
      ↪ Destination Node 458F029B-36C2-11E8-BF66-D9AA8AFF4A69
34  ORDER BY
35    depth ASC
36  LIMIT 1
37 )
38 SELECT
39   path
40 FROM
41   shortest_paths;

```

Quellcode A.2: SQL Query für das Finden eines Pfades zwischen zwei beliebigen Knoten.

```
1 WITH hop1 AS (  
2     SELECT DISTINCT e.source, e.dest  
3     FROM edge_list e  
4     WHERE e.source = '9FF334BB-9072-D756-B290-556656D73728'  
5           OR e.dest = '9FF334BB-9072-D756-B290-556656D73728'  
6 ),  
7 combined AS (  
8     SELECT source AS node FROM hop1  
9     UNION  
10    SELECT dest FROM hop1  
11 ),  
12 hop2 AS (  
13    SELECT e.source, e.dest  
14    FROM edge_list e  
15    JOIN combined c ON e.source = c.node OR e.dest = c.node  
16 )  
17 SELECT DISTINCT node  
18 FROM (  
19     SELECT source AS node FROM hop2  
20     UNION ALL  
21     SELECT dest FROM hop2  
22 ) AS all_nodes  
23 WHERE node <> '9FF334BB-9072-D756-B290-556656D73728'
```

Quellcode A.3: SQL Abfrage für das Finden der 2-Hop Nachbarschaft eines Knotens.