Schwierigkeiten bei Implementierung und Evaluation von Datenstrukturen in Datenbanksystemen

Anton Rodenwald (19)

24. Januar 2024

Fachgebiet:	Mathematik/Informatik
Wettbewerbssparte:	Jugend Forscht
Bundesland:	Niedersachsen
Wettbewerbsjahr:	2024
Thema des Projektes:	In meinem Projekt wollte ich verschiedene in Datenbanksystemen genutzte Datenstrukturen implementieren und testen. Dabei betrachtete ich auch unterschiedliche Ansätze der Datenspeicherung und für den Datenzugriff. Bei den auftretenden Schwierigkeiten musste ich Lösungen finden und adaptieren.
Projektbetreuerin:	Birgit Ziegenmeyer
Institution:	Schillerschule Hannover

Kurzfassung

In meinem Projekt wollte ich erforschen, wie die in Datenbanksystemen genutzten Datenstrukturen die Zugriffsgeschwindigkeit beeinflussen und welche Datenstrukturen sich für welche Arten von Daten besonders gut eignen. Dazu wollte ich ein Datenbanksystem in C++ entwickeln und messen, wie schnell, je nach Datenstruktur, auf die Daten zugegriffen werden kann. Die Haupt-Schwierigkeit war dabei allerdings die hohe Komplexität, weshalb ich nur einige Datenstrukturen umsetzen und testen konnte. Insgesamt lässt sich feststellen, dass noch weitere Messungen nötig sind, um die Ergebnisse zu konsolidieren. Dies lässt noch viele weitere Möglichkeiten für weitere Forschungen zu.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Rahmen des Informatik-Leistungkurses meiner Schule haben wir (Mitschüler und ich) uns mit Datenbanken und Daten-Modellierung beschäftigt. Da jedoch auf die Funktionsweise einer Datenbank nicht weiter eingegangen wurde und ich auch schon von unterschiedlichen Datenbankansätzen, genauer Relationalen und Nicht-Relationalen, gehört hatte, wollte ich mich im Rahmen meines Projektes genauer damit beschäftigen und mehr darüber lernen. Nachdem ich einen spannenden Vortrag von Richard Hipp gesehen hatte, entschied ich mich, dass ich selbst eine Datenbank umsetzen wollte [Hip08].

1.2 Forschungsfrage

Meine Forschungsfrage ist, wie die Wahl der Datenstruktur die Geschwindigkeit einer Datenbank beeinflusst und inwiefern sich eine Datenstruktur für gewisse Daten besser oder schlechter eignet.

1.3 Ziel

Mein ursprüngliches Ziel war die Entwicklung einer Datenbank mit den Datenstrukturen R-Tree, B-Tree, Binary-Tree und Graphen. Davon konnte ich leider aufgrund der hohen Komplexität (stand jetzt) nur ein Binäres-Such-Array und eine Graph-Datenstruktur umsetzen.

2 Hintergrund

2.1 Arten von Daten und Datenstrukturen

Verschiedene Datenarten sind dabei räumliche, zeitliche, relationale (als Objekte modellierbare) und stark vernetzte Daten [Unk23] [JPA+12].

Räumliche Daten sind z. B. Kartendaten (Google Maps, OpenStreetMap). Bestimmte Landmarken oder Objekte (Bäume, Mulleimer) lassen sich beispielsweise als Punkte auf einer Fläche modellieren. Es müssen dann effizient Fragen wie z. B. “Wie viele Mülleimer gibt es in diesem Park?” oder “Was sind die 3 am besten bewerteten Restaurants in der Innenstadt?” beantwortet werden. Eine dafür ausgelegte Datenstruktur ist der R-Tree.

Zeitliche Daten können z. B. die von einem Sensor gemessenen Temperaturen sein. Wichtig wäre bei diesen, dass sich statistische Größen wie Durchschnittswerte einfach bilden lassen und Anomalien erkannt werden.

Daten, die klassischerweise in relationalens Datenbanken gespeichert werden, lassen sich als Objekte modellieren. Beispielsweise speichert ein Unternehmen über Kunden immer die gleichen Daten (Name, E-Mail, ...). Jeder Datensatz kann dann eindeutig z. B. über eine ID identifiziert werden. Eine dafür geeignete Datenstruktur ist der B-Tree, eine dem Binary-Tree ähnliche Datenstruktur.

Wenn zwischen einzelnen Daten von z. B. jedem Nutzer einer Social-Media-Plattform allerdings viele Beziehungen bestehen (Follower, Likes), so eignet sich eine Graph-Datenstruktur besser. Diese ermöglicht dann, schnell die Verbindungen zwischen Nutzern herausfinden, um z. B. Gruppen innerhalb des Netzwerks zu erkennen.

2.2 Arten von Datenbanksystemen

Moderne Datenbanksysteme ermöglichen meist, mit einem System verschiedene Speicherungsformen zu nutzen. Trotzdem muss man sich bei einem Multi-Modalen System natürlich überlegen, welche Daten man nun in welcher Art von Datenstruktur speichern möchte. Die Website db-engines.com stellt z. B. eine Übersicht über die aktuell beliebtesten Datenbanksysteme und deren Modalitäten dar.

3 Vorgehen

3.1 Schwierigkeiten bei der Implementierung

Mein geplantes Vorgehen bestand darin, ein Datenbanksystem mit diesen verschiedenen Datenstrukturen in C++ umzusetzen. Während des Entwicklungsprozesses und der Implementierung zeigte sich allerdings die zunehmende Komplexität, weswegen ich nur begrenzt die verschiedenen Datenstrukturen vergleichen konnte. Trotzdem werde ich die angestrebten Datenstrukturen zumindest in ihrer Funktion erläutern und meine Schwierigkeiten bei der Umsetzung beschreiben. Auch werde ich meine dann tatsächlich durchgeführten Tests beschreiben. Also ein ganz normaler Entwicklungsprozess nach dem Zitat:

> »we do these things not because they are easy,
> but because we thought they were going to be easy«

Zu Beginn meines Projektes setzte ich auch die Kommunikation über Linux Websockets um, da Datenbanksysteme eine Schnittstelle zu anderen Programmen benötigen. Da mein Datenbank-Backend allerdings nicht rechtzeitig ausgereift genug war, konnte ich diese Schnittstelle noch nicht in Kombination mit dem Rest testen. (Code auf Github)

3.2 Testdaten

Bei den Testdaten handelt es sich um eine fiktive Schülerschaft mit 400.000 Schülern, in der jeder dieser Freundschaften und Kursbelegungen hat. Diese Testdaten generierte ich mir mit einem Python Script und speicherte diese in einer CSV-Datei. Jeder Schüler hat dabei eine ID und einen Namen, der eine zufällige Zeichenkette ist. Anschließend sind bei jedem Schüler noch 100 Freundschaften und Kursbelegungen eingetragen. Jede dieser Beziehungen wird dabei durch die ID eines anderen Schülers bzw. Kurses beschrieben. Schüler 0 ist z. B. mit Schüler 255777 befreundet.

```
0, eszycidpyopumzgd, 255777-14862-..., 204372-226894-...
1, idixqgtnahamebxf, 233658-369416-..., 265451-355559-...
2, digztyrwpvlifrgj, 104446-129174-..., 188986-42660-...
...
```

3.3 Teststruktur

Bei meinen Performance Tests verglich ich die Suchgeschwindigkeit der Datenstrukturen. Die gleichen Testdaten wurden in beide Datenstrukturen eingelesen und anschlie-

ßend wurde gemessen, wie schnell auf einen bestimmten Teil der Daten jeweils zugegriffen werden kann. Im 1. Test sollten alle Freunde von Schüler Nr. 2 und im 2. Test alle Kursbelegungen von Schüler Nr. 1 gefunden werden.

3.4 Probleme bei B-Tree und Binary-Tree

Da der B-Tree eine in vielen Datenbanken verwendete Datenstruktur ist, wollte ich diesen zuerst implementieren. Als Grundlage nahm ich die erste B-Trees beschreibende Veröffentlichung von 2 Boeing-Entwicklern [BM70].

Der B-Tree ist dabei ähnlich zu einem binären Suchbaum. Der Hauptunterschied ist, dass jeder Node mehr als nur 2 Child-Nodes hat, wodurch der Baum weniger tief ist. Um Elemente zu suchen, zu löschen oder zu ändern folgt man, ähnlich wie beim Binary-Tree, der Baumstruktur vom Root Node ausgehend. Durch Vergleiche innerhalb der Nodes und das Traversieren der Äste findet man ein bestimmtes Element oder eine Position im Baum.

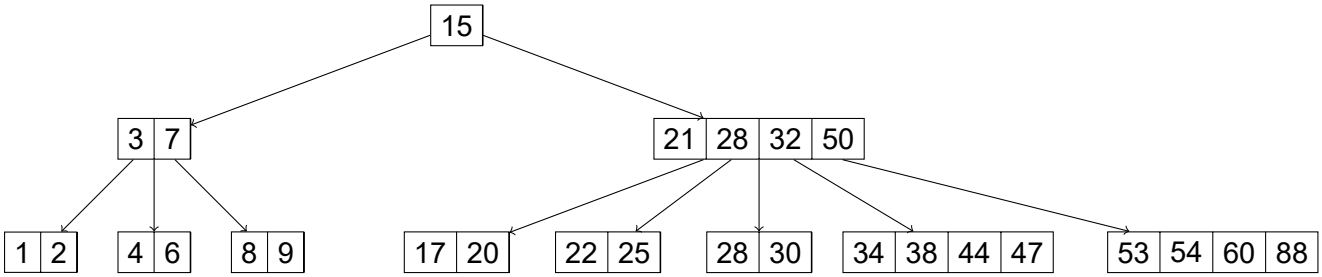

Abbildung 1: B-Tree der Tiefe 2

Bei der Implementierung des B-Tree traten allerdings einige Probleme auf. Das größte Problem war, dass die Ausbalancierung beim B-Tree relativ komplex ist. Deshalb entschied ich mich dafür, andere Datenstrukturen zu testen. Auch entschied ich mich, meine Tests ausschließlich mit im RAM liegenden Daten durchzuführen, da eine Speicherung auf einer Festplatte noch mehr Komplexität bedeutet hätte. Da der R-Tree eine modifizierte Version des B-Tree ist, konnte ich auch diesen somit nicht umsetzen.

Anschließend implementierte ich als Alternative dann einen AVL-Tree, eine sich selbst ausbalancierende Variante des binären Suchbaums. Beim anschließenden Konzeptionieren der Performance-Tests entdeckte ich allerdings noch einen beim Einfügen von Elementen auftretenden Fehler bei meinem AVL-Tree, weswegen ich mich spontan dazu entscheiden musste, auch diesen nicht zu testen.

3.5 Such-Array

Aufgrund der vorigen Schwierigkeiten testete ich nun nicht mit einem B-Tree oder Binary-Tree, sondern wandte die binäre Suche auf ein sortiertes Array an. Prinzipiell nutzen aber alle 3 Datenstrukturen die binäre Suche. Deswegen nahm ich an, dass alle 3 Datenstrukturen bei Daten im RAM ähnlich schnell Daten finden können. Die Testdaten sind in diesem Fall auf 3 Such-Arrays (vgl. Tabellen) aufgeteilt, jeweils eine für Schüler, Freundschaften und Kursbelegungen.

Schüler_1_ID	Schüler_2_ID
5	7
5	75
5	456
5	45

Schüler_1_ID	Kurs_ID
7	629
7	762
7	229
7	29435

Tabelle 1: Tabellenausschnitte Freundschaften/Kursbelegungen

Um alle Freunde/Kursbelegungen eines Schülers zu finden, müssen in den als Assoziationstabellen fungierenden, sortierten Such-Arrays alle Datensätze zwischen dem kleinstmöglichen und größtmöglichen Wert gefunden werten. Beispielsweise für den Schüler 5 alles zwischen (inklusiv) den Datensätzen $5|0$ und $5|m$ wenn m die Anzahl an Schülern ist. Da das Such-Array nach beiden Spalten sortiert ist, muss nur mit der binären Suche das linke und das rechte Ende des Bereichs bestimmt werden. Anschließend können in linearer Zeit alle gesuchten Datensätze ausgelesen werden.

```cpp
std::vector<void*> Sorted_Array::search_between_keys(void* key_left, void* key_right) {
    Search_Result res_l = this->search_for_key(key_left);
    Search_Result res_r = this->search_for_key(key_right);

    int left_start = res_l.index;
    int right_start = res_r.index;

    if (res_l.status == 0 &&
        compare_keys(this->elements.at(res_l.index).key, key_left, this->key_attribute_lengths) == 1)
        ++left_start;
    if (res_r.status == 0 &&
        compare_keys(this->elements.at(res_r.index).key, key_right, this->key_attribute_lengths) == -1)
        --right_start;

    std::vector<void*> data_ptrs;
    for (int i = left_start; i < right_start + 1; ++i) {
        data_ptrs.emplace_back(this->elements.at(i).data);
    }
    return data_ptrs;
}
```

Abbildung 2: C++ Code der Such-Array Suchfunktion

3.6 Graph-Datenstruktur

Bei meiner Implementierung einer Graph-Datenstruktur sind in jedem Node die Daten und die Verbindungen zu anderen Nodes gespeichert. Um alle Freundschaften oder Kursbelegungen zu erhalten, muss nur über diese Daten iteriert werden und man erhält schnell die Daten der befreundeten Schüler und der belegten Kurse. Die Art der Verbindung (Freund/Kursbelegung) kann dabei durch das $link\_schema$ festgelegt werden.

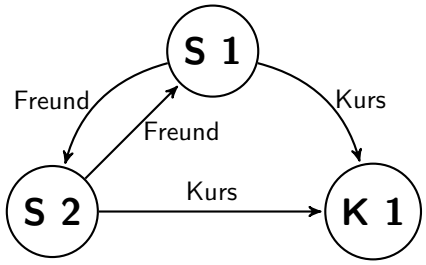

Abbildung 3: Graph mit Beziehungen zwischen Schülern

```
std::vector<Link> Graph::query_links(Node* node, int link_schema){
    std::vector<Link> matching_links;
    for(const auto link : node->links){
        if (link.link_schema == link_schema){
            matching_links.push_back(link);
        }
    }
    return matching_links;
}
```

Abbildung 4: C++ Code der Graph-Suchfunktion

3.7 Komplexitäts Analyse

$f_g =$ Anzahl Freundschaften $\quad K_g =$ Kursbelegungen

$f =$ Anzahl Freunde eines Schülers $\quad K =$ Anzahl Kurse eines Schülers

$s =$ Anzahl Schüler

Beim Such-Array muss mit der binären Suche jeweils die linke und rechte Begrenzung ermittelt werden. Anschließend müssen alle Freunde bzw. Kursbelegungen des Schülers durchlaufen werden.

$$O(2 \cdot (log_2(f_g) + 1) + f)$$

Bei der Graph-Datenstruktur müssen alle Verbindungen durchlaufen werden, also die Summe der Verbindungen aus Freundschaften und Kursbelegungen.

$$O(f + K)$$

Anhand der Komplexitäten lässt sich erwarten, dass die Graph-Datenstruktur deutlich schneller ist.

4 Messwerte

Meine Messungen führte ich auf meinem Desktop-PC und meinem Laptop (Plugged-In) durch. Auf jedem Computer nahm ich 2 Messreihen auf. Jede der 4 Anfragen an die 2 Datenstrukturen wurde jeweils 50-mal direkt hintereinander ausgeführt. SA beschreibt die Datenstruktur Such-Array. G steht für Graph-Datenstruktur. Das F beschreibt die Suche nach allen Freunden des Schülers Nr. 2. Das H beschreibt die Suche nach allen Kursbelegungen des Schülers Nr. 1.

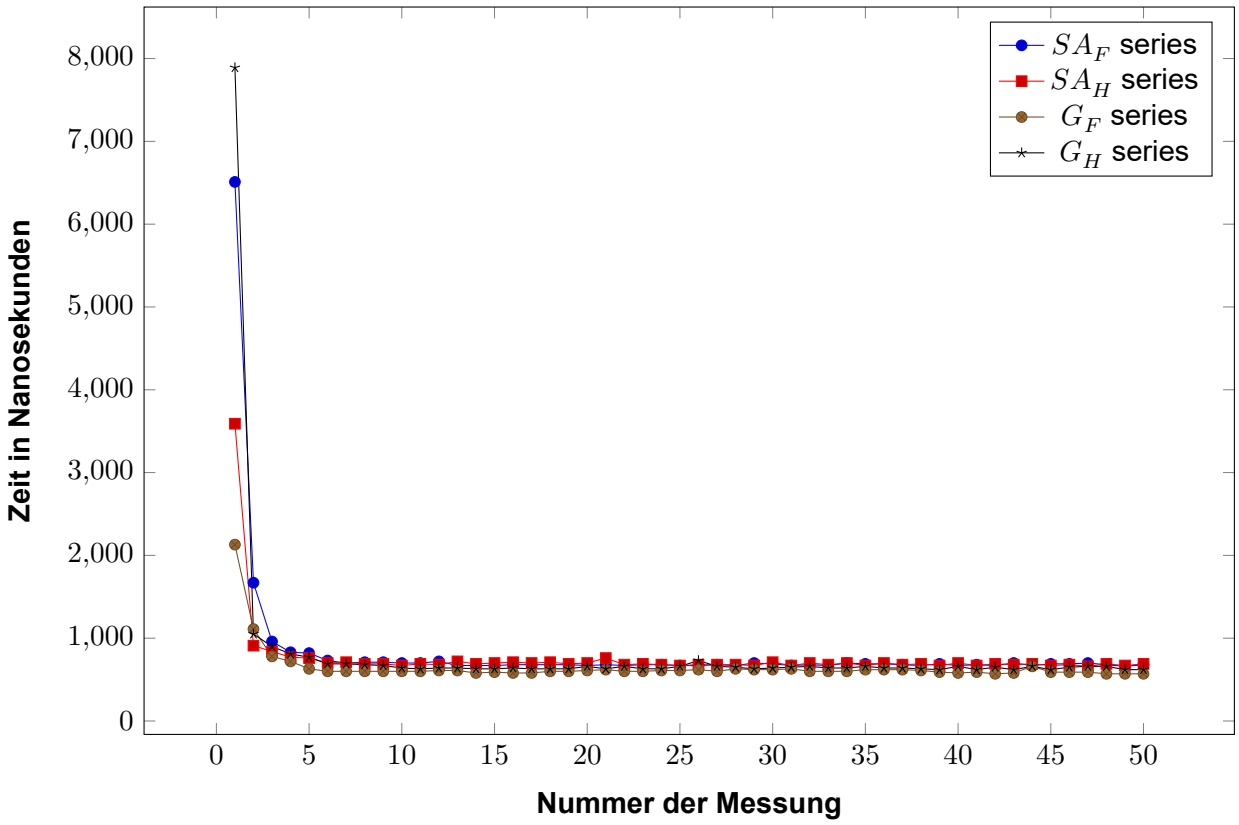

Abbildung 5: 1. Messreihe am PC (Ryzen 7 2700)

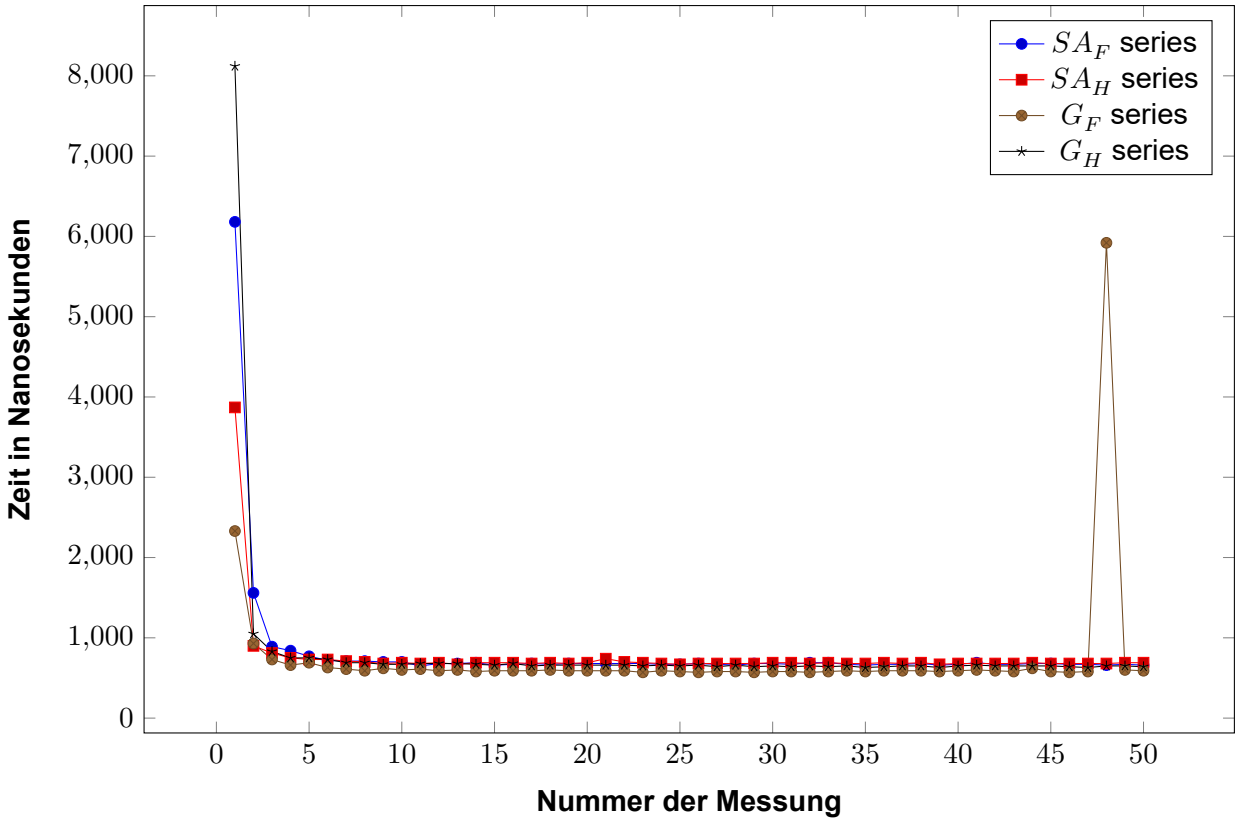

Abbildung 6: 2. Messreihe am PC (Ryzen 7 2700)

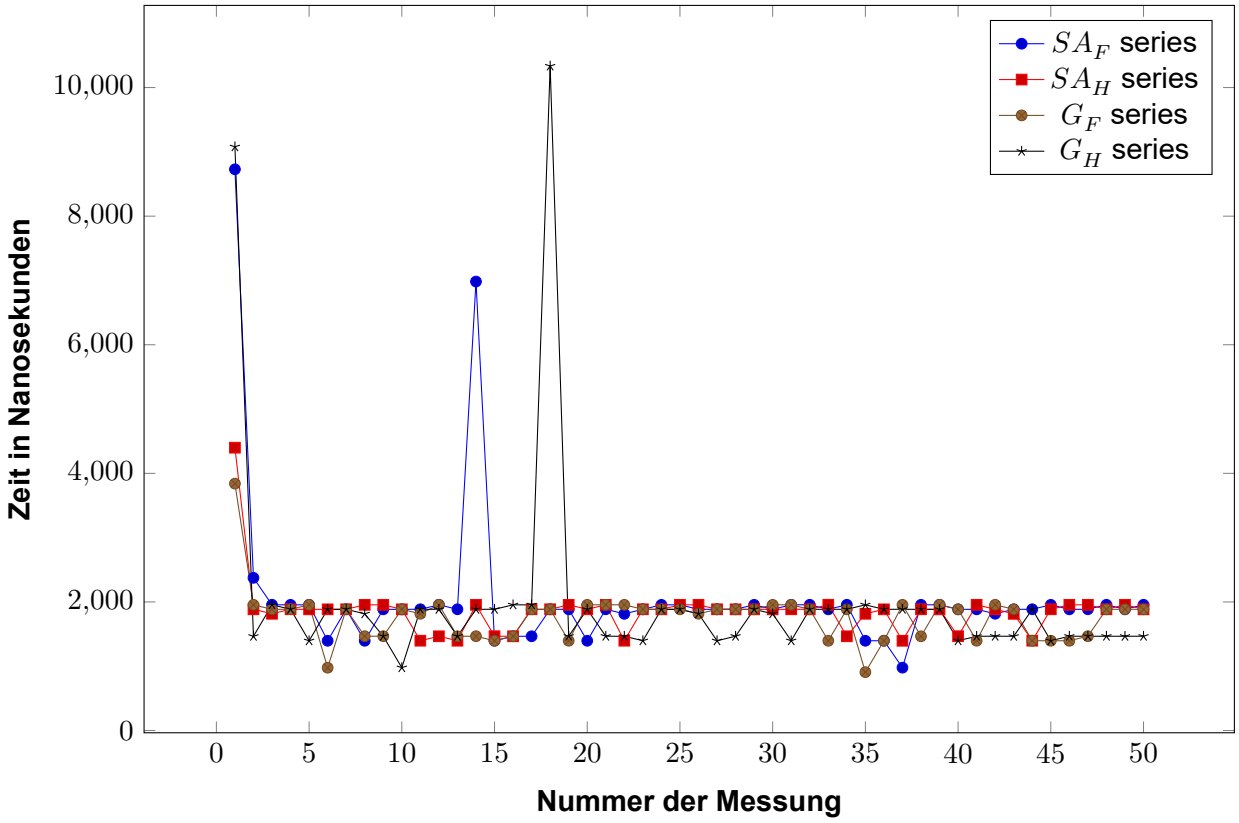

Abbildung 7: 1. Messreihe am Laptop (Ryzen 5 5500U)

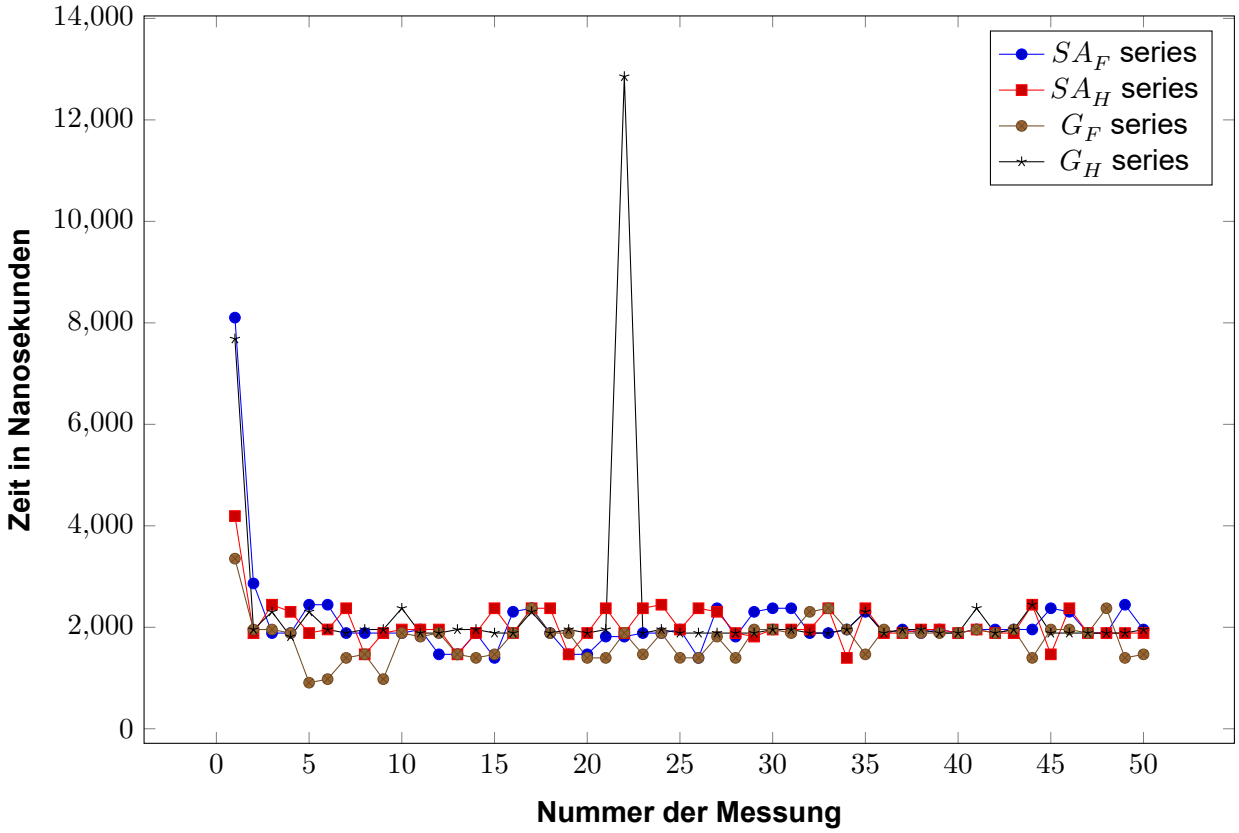

Abbildung 8: 2. Messreihe am Laptop (Ryzen 5 5500U)

5 Deutung der Messwerte

Bei den Messwerten am PC lässt sich erkennen, dass in den ersten paar Messungen die benötigte Zeit deutlich höher ist als bei den restlichen, nachfolgenden Messungen. Die benötigte Zeit aller Tests pendelt sich auf einem niedrigeren Niveau ein. Gleiches ist auch am Laptop zu beobachten, wo die Messpunkte jedoch unregelmäßiger sind. Dies lässt sich vermutlich auf eine Speicherung von Werten im CPU-Cache zurückführen, in dem nach den ersten Durchläufen Daten vorgehalten werden, was zu besseren Messergebnissen führt. Vereinzelt gibt es auch in diesen Bereichen bei PC und Laptop ausreißende Werte. Diese sind wahrscheinlich Mess-Anomalien, erzeugt durch einmalige Lastspitzen. Diese könnten durch andere Prozesse auf dem Testsystem oder durch hardwarebedingte Faktoren verursacht worden sein. Betrachtet man die 1. Messung, wo die Werte noch auseinanderliegen, so lässt sich nicht feststellen, dass keine der Datenstrukturen schneller ist als die anderen. Interessant ist bei diesen ersten Messwerten, dass die Messungen bei gleich strukturierten Daten (Freundschaften und Feindschaften) deutlich voneinander Abweichen. Auch für gleiche Datenstrukturen gibt es große Abweichungen der benötigten Zeit. Nicht überraschend sind die abweichenden Niveaus von PC und Laptop, da bei CPUs mit unterschiedlichem Leistungsniveau

logischerweise auch andere Performance-Werte erzeugt werden. Als dies lässt darauf schließen, dass es wahrscheinlich noch unbekannte Faktoren gibt, die die Performance oder Messung beeinflussen. Durch Identifizieren und Eliminieren dieser ließen sich vermutlich noch genauere und aussagekräftigere Messwerte gewinnen.

6 Beantwortung der Forschungsfrage

Abschließend lässt sich die Forschungsfrage insofern beantworten, als dass die Wahl der Datenstruktur in diesem Fall keinen Einfluss auf die Zugriffsgeschwindigkeit hat. Keine der Datenstrukturen scheint der anderen überlegen, zumindest auf Basis der Messwerte.

7 Schwierigkeiten und Limitationen

Bei der Umsetzung der Datenbank und den Tests hatte ich einige Probleme. Eine Sache, die ich nicht prüfen konnte ist, wie sich der Speicherort der Daten auswirkt. Also inwiefern Cache, RAM und Art des Massenspeichers (HDD, SSD) sich auf die Geschwindigkeit beim Datenzugriff auswirken. Dazu wollte ich eigentlich AMDuProf nutzen, eine von AMD für Ryzen CPUs bereitgestellte Software, mit der sich tiefgreifende Daten über die Arbeit der CPU sammeln lassen. Mit dieser hätte ich z. B. Cache-Misses der CPU analysieren können, doch leider lief diese nicht auf meinem System. Auch konnte ich nicht prüfen, welche Anfragen an eine Datenbank CPU limitiert und welche I/O bzw. Netzwerk limitiert sind. Dies war leider deswegen nicht möglich, weil ich noch kein vollständiges Datenbanksystem hatte.

8 Fazit

Insgesamt konnte ich nicht so viele relevante Daten sammeln, würde das Projekt aber trotzdem als lehrreich bewerten. Mir persönlich hat es einen guten Einblick in Datenbanken und andere interessante Themen wie Socket-Programmierung gegeben und ich konnte mehr über Datenstrukturen, C/C++ und gdb (GNU Debugger) lernen.

9 Quellenangaben

Links

https://github.com/Redstonerayy/light-db

https://beej.us/guide/bgnet/html/

https://db-engines.com/en/ranking

https://www.amd.com/en/developer/uprof.html

Literaturverzeichnis

[BM70] R. Bayer und E. McCreight. „Organization and Maintenance of large ordered Indices“. In: (1970). URL: https://dl.acm.org/doi/10.1145/1734663.1734671.

[Hip08] D. Richard Hipp. *How SQL Database Engines Work*. 2008. URL: https://www.youtube.com/watch?v=Z_cX3bzkExE (besucht am 28. 12. 2023).

[JPA+12] Nishtha Jatana u. a. „A Survey and Comparison of Relational and Non-Relational Database“. In: (2012).

[Unk23] Unknown. *NoSQL*. 2023. URL: https://de.wikipedia.org/wiki/NoSQL (besucht am 27. 12. 2023).