

**BACHELORARBEIT**

# **Performance – Optimierung von Datenbanken**

---

vorgelegt am 26. März 2022  
Daniel Freire Mendes

Erstprüferin: Prof. Dr. Stefan Sarstedt  
Zweitprüfer: Prof. Dr. Olaf Zukunft

---

**HOCHSCHULE FÜR ANGEWANDTE  
WISSENSCHAFTEN HAMBURG**

Department Informatik  
Berliner Tor 7  
20099 Hamburg

## **Zusammenfassung**

Der Arbeit beginnt mit einer kurzen Beschreibung ihrer zentralen Inhalte, in der die Thematik und die wesentlichen Resultate skizziert werden. Diese Beschreibung muss sowohl in deutscher als auch in englischer Sprache vorliegen und sollte eine Länge von etwa 150 bis 250 Wörtern haben. Beide Versionen zusammen sollten nicht mehr als eine Seite umfassen. Die Zusammenfassung dient u. a. der inhaltlichen Verortung im Bibliothekskatalog.

## **Abstract**

The thesis begins with a brief summary of its main contents, outlining the subject matter and the essential findings. This summary must be provided in German and in English and should range from 150 to 250 words in length. Both versions combined should not comprise more than one page. Among other things, the abstract is used for library classification.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>II</b>
<b>Tabellenverzeichnis</b>	<b>III</b>
<b>1 Überblick</b>	<b>1</b>
1.1 Einführung in Benchmarks . . . . .	1
1.2 Measures . . . . .	2
<b>2 Tools</b>	<b>4</b>
2.1 Auswahl der Tools . . . . .	4
2.2 Einführung in die Tools . . . . .	5
2.3 Genauere Darstellung anhand eines Beispiels . . . . .	10
<b>3 Projektdurchführung</b>	<b>22</b>
3.1 GitHub Action . . . . .	22
<b>4 Optimierungen von Datentypen</b>	<b>27</b>
4.1 Allgemeine Faktoren . . . . .	27
4.2 Einzelne Datentypen und weitere Faktoren . . . . .	28
4.3 Analyse der Benchmarks . . . . .	31
<b>5 Indexierung und Einfluss auf die Performance</b>	<b>34</b>
5.1 Grundlagen der Indexierung . . . . .	34
5.2 B-Baum-Index . . . . .	35
5.3 Hash - Index . . . . .	38
<b>Literatur</b>	<b>42</b>
<b>Anhang</b>	<b>43</b>

# Abbildungsverzeichnis

2.1	Demo-Graphs: Gnuplot vs. Pandas . . . . .	10
2.2	Join-Typ: Skriptvergleich . . . . .	20
2.3	Join-Typ: Metrikvergleich . . . . .	20
2.4	Join-Typ: Hexagon-Diagramm . . . . .	21
4.1	Vergleich von NULL und NOT NULL . . . . .	31
4.2	Vergleich von INT und CHAR . . . . .	32
4.3	Vergleich von unterschiedlichen Select-Abfragen . . . . .	32
4.4	Vergleiche von unterschiedlichen Write-Abfragen . . . . .	33
5.1	Binärbaum - Grafik . . . . .	36
5.2	High-Counts: Reads und Writes . . . . .	37
5.3	Low-Counts: Reads und Writes . . . . .	38
5.4	B - Tree - Selects - Ergebnis . . . . .	38
5.5	Hash-Kollisionen: Reads und Writes . . . . .	40
5.6	Hash - Selects - Ergebnis . . . . .	41

# Tabellenverzeichnis

4.1	Ergebnis der SQL-Abfrage aus 4.2	29
-----	----------------------------------	----

# 1 Überblick

## 1.1 Einführung in Benchmarks

Benchmarks dienen dazu, praktisch und effektiv zu untersuchen, wie sich ein System unter Last verhält. Die wichtigsten Erkenntnisse, die man aus Benchmarks gewinnen kann, sind die Probleme und Fehler, die man systematisch dokumentieren und nach Priorität abarbeiten sollte. Das Ziel von Benchmarks ist die Reduzierung und Bewertung von unerwünschtem Verhalten sowie die Analyse, wie sich das System derzeit und unter simulierten, zukünftigen, anspruchsvolleren Bedingungen verhalten könnte.

Es gibt zwei verschiedene Techniken für Benchmarks. Die erste zielt darauf ab, die Applikation als Ganzes zu testen (full-stack). Dabei wird nicht nur die Datenbank getestet, sondern die gesamte Applikation, einschließlich des Webserver, des Netzwerks und des Applikationscodes. Der Ansatz dahinter ist, dass ein Nutzer genauso lange auf eine Abfrage warten muss, wie das gesamte System benötigt. Daher sollte diese Wartezeit so gering wie möglich sein. Es kann dabei vorkommen, dass MySQL nicht immer das Bottleneck ist.<sup>1</sup>

Full-Stack-Benchmarks haben jedoch auch Nachteile. Sie sind schwieriger zu erstellen und insbesondere schwieriger korrekt einzurichten. Wenn man lediglich verschiedene Schemas und Abfragen in MySQL auf ihre Performance testen möchte, gibt es sogenannte Single-Component-Benchmarks. Diese analysieren ein spezifisches Problem in der Applikation und sind deutlich einfacher zu erstellen. Ein weiterer Vorteil besteht darin, dass nur ein Teil des gesamten Systems getestet wird, wodurch die Antwortzeiten kürzer sind und man schneller Ergebnisse erhält.

Wenn bei Benchmarks schlechte Designentscheidungen getroffen werden, kann dies zu einer falschen Interpretation des Systems führen, da die Ergebnisse nicht die Realität widerspiegeln. Die Größe des Datensatzes und des Workloads muss realistisch sein. Idealerweise verwendet

---

<sup>1</sup>Gemeint ist ein Engpass beim Transport von Daten oder Waren, der maßgeblichen Einfluss auf die Abarbeitungsgeschwindigkeit hat. Optimierungsversuche an anderer Stelle führen oft nur zu geringen oder gar keinen messbaren Verbesserungen der Gesamtsituation. ([1])

man einen Snapshot<sup>2</sup> des tatsächlichen produktiven Datensatzes. Gibt es keine Produktionsdaten, sollten die Daten und der Workload simuliert werden, da realistische Benchmarks komplex und zeitaufwendig sein können.

Häufige Fehler beim Durchführen von Benchmarks sind unter anderem, dass nur ein kleiner Teil der tatsächlichen Datensatzgröße verwendet wird und die Datensätze unkorrekt gleichmäßig verteilt sind. In der Realität können Hotspots auftreten. Bei zufällig generierten Werten kommt es hingegen häufig zu unrealistisch gleichmäßig verteilten Datensätzen. Ein weiterer Fehler besteht darin, dass man beim Testen einer Anwendung nicht das tatsächliche Benutzerverhalten nachstellt. Wenn gleiche Abfragen in einer Schleife ausgeführt werden, muss man außerdem auf das Caching achten, da sonst falsche Annahmen über die Performance getroffen werden können. Zudem wird oft die Warmmachphase des Systems vollständig ignoriert. Kurze Benchmarks können schnell zu falschen Annahmen über die Performance des Systems führen.

Um verlässliche Ergebnisse zu erhalten, sollte ein Benchmark ausreichend lange laufen, um den stabilen Zustand des Systems zu beobachten, insbesondere bei Servern mit großen Datenmengen und viel Speicher. Dabei ist es wichtig, so viele Informationen wie möglich zu erfassen und sicherzustellen, dass der Benchmark wiederholbar ist, da unzureichende oder fehlerhafte Tests wertlos sind. Außerdem ist es wichtig, die Ergebnisse in einem Diagramm darzustellen, da auftretende Phänomene sonst anhand einer tabellarischen Darstellung nicht erkannt werden können.

## 1.2 Measures

- **Durchsatz (Throughput):** Der Durchsatz ist die Anzahl an Transaktionen pro Zeiteinheit. Er ist standardisiert, und Datenbankanbieter versuchen, diesen zu optimieren. Meistens werden Transaktionen pro Sekunde (oder manchmal pro Minute) als Einheit verwendet.
- **Antwortzeiten (Latenz):** Die Antwortzeit misst die gesamte Zeit, die für eine Abfrage benötigt wird. Diese kann, abhängig von der Applikation, in Mikrosekunden ( $\mu$ s), Millisekunden (ms), Sekunden oder Minuten angegeben werden. Von dieser Zeit können aggregierte Antwortzeiten wie Durchschnitt, Maximum, Minimum und Perzentile abgeleitet werden. Das Maximum ist oft eine weniger sinnvolle Metrik, da es sich nicht gut wiederholen lässt. Daher nutzt man eher Perzentile bei den Antwortzeiten. Wenn

---

<sup>2</sup>Snapshots bestehen größtenteils aus Metadaten, die den Zustand Ihrer Daten definieren, und sind keine vollständige Duplikation der Daten auf Ihrer Festplatte. Snapshots werden häufig für Test-/Entwicklungsaufgaben verwendet. ([2])

beispielsweise das 95. Perzentil der Antwortzeit bei 5 ms liegt, bedeutet dies, dass mit einer Wahrscheinlichkeit von 95 % die Abfrage in weniger als 5 ms abgeschlossen ist.

- **Nebenläufigkeit (Concurrency):** Die Nebenläufigkeit auf dem Webserver lässt sich nicht zwangsläufig auf den Datenbankserver übertragen. Eine genauere Messung der Gleichzeitigkeit auf dem Webserver besteht darin, zu bestimmen, wie viele gleichzeitige Anfragen zu einem bestimmten Zeitpunkt ausgeführt werden. Es kann auch geprüft werden, ob der Durchsatz sinkt oder die Antwortzeiten steigen, wenn die Gleichzeitigkeit zunimmt. Beispielsweise benötigt eine Website mit „50.000 Benutzern gleichzeitig“ vielleicht nur 10 oder 15 gleichzeitig laufende Abfragen.
- **Skalierbarkeit (Scalability):** Skalierbarkeit ist wichtig für Systeme, die ihre Performance unter unterschiedlich starken Workloads beibehalten müssen. Ein ideales System würde doppelt so viele Abfragen beantworten (Throughput), wenn doppelt so viele „Arbeiter“ versuchen, die Aufgaben zu erfüllen. Die meisten Systeme sind jedoch nicht linear skalierbar und zeigen Leistungseinbußen, wenn die Parameter variieren.



## 2 Tools

### 2.1 Auswahl der Tools

Die Grundlage für diese Bachelorarbeit ist das Verhalten der MySQL – Datenbank [3] in Bezug auf die unterschiedlichen Aspekte, die im Rahmen dieser Arbeit behandelt werden. Der folgende Abschnitt beschäftigt sich mit Umsetzung, um dieses Verhalten messbar und veranschaulicht mithilfe von Grafiken zu machen. Damit wir die Kennzahlen für bestimmte Abfragen an die MySQL – Datenbank bestimmen können, brauchen wir ein zentrales Tool. Dieses Tool ist dafür verantwortlich ist die Benchmark - Tests durchzuführen.

Meine Entscheidung ist dabei schlussendlich auf Sysbench [4] gefallen. Sysbench ist ein Open-Source-Tool, das ein skriptfähiges, multi-threaded Benchmark-Tool ist, das auf LuaJIT basiert. Es wird hauptsächlich für Datenbankbenchmarks verwendet, kann jedoch auch dazu eingesetzt werden, beliebig komplexe Arbeitslasten zu erstellen, die keinen Datenbankserver erfordern. Sysbench analysiert dabei Metriken, wie unter anderem Transaktionen pro Sekunde, Latenz und Anzahl an Threads. Dabei kann man genauer spezifizieren, wie oft diese Metriken geloggt werden sollen. Sysbench ist dabei nicht auf ein einziges Datenbanksystem eingeschränkt, sondern man kann sich zwischen vielen unterschiedlichen Systemen entscheiden.

Im Zuge der Wahl des Benchmark – Tools habe ich auch andere Benchmarking-Tools betrachtet, wie beispielsweise Benchbase [5] oder mybench [6]. Im Vergleich zu diesen Tools bietet Sysbench jedoch die Vorteile der höheren Skriptfähigkeit und Flexibilität. Damit ist gemeint, dass bei Sysbench das erste Projekt mit mehr Aufwand verbunden ist als bei den Alternativen. Wenn man aber ein Projekt erstmal erstellt hat, dann ist es sehr individuell und man kann schnelle Änderungen hervorheben. In dem Kapitel (TODO (Daniel): Kapitel mit Join Typ) betrachten wir ein beispielhaftes Projekt mit Sysbench, bei dem der Einfluss von unterschiedlichen Datentypen als Join-Operator zwischen zwei Tabellen verglichen wird. Wenn wir später die Performance von unterschiedliche Index-Typen betrachten, dann müssen wir nur an wenigen Stellen Veränderungen durchführen, die in dem Kapitel genauer besprochen werden.

Ein weiterer Vorteil von Sysbench ist, dass es als de facto Standard im Bereich der Datenbankbenchmarks angesehen wird [7]. Durch diese Position gibt es viele aktive Nutzer und

dadurch bedingt viele verfügbaren Ressourcen. Vorteile der anderen Tools sind jedoch die weniger präzise Steuerung der Ergebnissraten und der Transaktionen von Sysbench. Zudem ist Sysbench auf das Minimale beschränkt, was den Output angeht, da es, wie schon erwähnt, nur eine Reihe von Log-Dateien gibt und die Visualisierung der Ergebnisse muss vom Benutzer selbst mithilfe von anderen Tools umgesetzt werden. Anders sieht dies bei dem Tool mybench aus, da es dort die Möglichkeit gibt in Echtzeit umfassende Abbildungen zu betrachten [8]. Obwohl dieses Feature sehr hilfreich ist, bin ich nach Abwägung der Vor- und Nachteile zu dem Entschluss gekommen, dass die einfachere Bedienung und die Tatsache, dass Sysbench der de facto Standard ist, für mich überwiegen, weshalb ich mich für Sysbench entschieden habe.

Nichtsdestotrotz kann nicht komplett auf Graphen verzichtet werden, da Entwicklungen im Laufe einer Zeitmessung in einem Kurvenverlauf deutlich besser zu erkennen sind als in einer Log - Datei. Anhand der reinen Zahlen lassen sich unter Umständen Trends von zwei oder etwas mehr unterschiedliche Messungen erkennen, aber besonders wiederkehrende Trends werden aus der schriftlichen Form nicht schnell ersichtbar. Ganz anders sieht dies bei Graphen mit einer Zeitachse aus. Dort werden sofort Trends ersichtlich und auch der Vergleich zwischen den unterschiedlichen Messungen erfolgt deutlich besser.

Um die Kennzahlen, die mithilfe von Sysbench ermittelt worden sind, in eine grafische Darstellung umzuwandeln, gibt es unterschiedliche Tools, die wiederum einige Vor - und Nachteile mit sich bringen. Das erste mögliche Tool stellt Gnuplot [9] dar, mit dem sich CSV - Dateien sehr gut darstellen lassen. Wenn man aber nur bestimmte Spalten aus der Tabelle darstellen lassen, dann kommt man schnell an seine Grenzen. Deshalb habe ich mich für eine anpassungsfähigere Alternative entschieden, denn die Transformationen und die grafische Darstellung wird mithilfe eines Python Scripts umgesetzt. Für die grafische Darstellung sind dabei die Libraries pandas [10] und matplotlib.pyplot [11] verantwortlich.

## 2.2 Einführung in die Tools

Als allererstes muss der MySQL - Server gestartet sein. Dabei ist es egal, ob dies lokal auf dem Rechner oder über einen Docker in eines GitHub CI/CD-Workflows erfolgt. Das Wichtigste dabei ist es, dass man sich die Zugangsdaten, bestehend aus User - und Passwortdaten, zwischen speichert, da diese gebraucht werden, um den Benchmarktest mit Sysbench zu starten. Nachdem das RDBMS gestartet worden ist, muss zudem eine Datenbank erstellt werden. Dies könnte unter anderem so aussehen:

### Codeblock 2.1: Create Database

```
1 CREATE DATABASE sbtest;
```

Nach der erfolgreichen Erstellung der Datenbank muss das Tool Sysbench installiert werden. Um sich mit dem Tool Sysbench vertraut zu machen, gehen wir die verschiedenen Argumente, die beim Aufruf mitgegeben können oder müssen, durch. Darunter gehören:

- `--db-driver`: Gibt den Treiber für die Datenbank an, die Sysbench verwenden soll. In diesem Fall `mysql`, um die MySQL-Datenbank zu testen.
- `--mysql-host`: Der Hostname oder die IP-Adresse des MySQL-Servers. Standardmäßig wird `localhost` verwendet, wenn nichts angegeben wird.
- `--mysql-user`: Der Benutzername, mit dem Sysbench auf die MySQL-Datenbank zugreift.
- `--mysql-password`: Das Passwort für den MySQL-Benutzer. Falls der Benutzer kein Passwort hat oder der Zugriff über eine andere Authentifizierungsmethode erfolgt, kann dieses Argument weggelassen werden.
- `--mysql-db`: Der Name der MySQL-Datenbank, auf die zugegriffen wird. In diesem Beispiel `sbtest`.
- `--time`: Gibt die Laufzeit des Benchmarks in Sekunden an und muss immer mit angegeben werden.
- `--report-interval`: Gibt das Intervall in Sekunden an, in dem Zwischenergebnisse während des Tests ausgegeben werden. Sofern `--report-interval` nicht gesetzt wird, werden die Ergebnisse nur am Ende des Tests angezeigt.
- `--tables`: Die Anzahl der Tabellen, die für den Test erstellt werden sollen. Standardmäßig wird nur eine Tabelle erstellt.
- `--table-size`: Die Anzahl der Datensätze (Zeilen) pro Tabelle. Muss auch nicht zwingend angegeben werden.

Neben den sieben aufgelisteten Argumenten gibt es zwei weitere wichtige Optionen:

1. Wie im Abschnitt (2.3) erwähnt, kann ein Lua-Skript angegeben werden, um eigene Tabellen zu erstellen, Beispieldaten einzufügen und bestimmte Abfragen durchzuführen. Dazu muss am Ende der Sysbench-Befehlszeile lediglich der Pfad zur Lua-Datei hinzugefügt werden. Ein erklärendes Beispiel dazu folgt weiter unten in diesem Abschnitt.
2. Die Methode, den Sysbench ausführen soll, muss ebenfalls spezifiziert werden. Auch dieser wird am Ende der Sysbench-Befehlszeile angehängt.

Zunächst schauen wir ein kurzes Demo-Beispiel, denn es gibt die Möglichkeit die Datenbank auf Performance zu testen, ohne selbst eigene SQL-Befehle zu schreiben. Dafür gibt es vordefinierte Testtypen von Sysbench. Auf diese Weise kann man schnell die Korrektheit

der Einrichtung des Tools überprüfen, bevor man Lua-Scripts für die eigenen Bedürfnisse schreibt.

Man kann unter anderen zwischen diesen Testtypen wählen:

- **oltp\_insert**: Prüft die Fähigkeit der Datenbank, Daten schnell und effizient einzufügen und simuliert eine Umgebung, in der viele Schreiboperationen ausgeführt werden.
- **oltp\_read\_only**: Fokussiert sich auf die Performance bei Leseoperationen und eignet sich, um die Leistung bei einer rein lesenden Arbeitslast zu testen.
- **oltp\_read\_write**: Simuliert eine realistische Arbeitslast, bei der sowohl Lese- als auch Schreiboperationen gleichzeitig durchgeführt werden.

Des Weiteren gibt es auch unterschiedliche Methoden, die mit den Testtypen kombiniert werden können.

- **prepare**: Bereitet die Datenbank für den Test vor, u.a. das Erstellen von benötigten Tabellen.
- **run**: Ist die Ausführungsphase des Tests. Je nach Testtyp führt diese Methode die spezifizierten Operationen aus, wie etwa das Einfügen von Daten (oltp\_insert), das Abfragen von Daten (oltp\_read\_only) oder beides (oltp\_read\_write). Dabei wird die Performance der Datenbank unter der angegebenen Arbeitslast gemessen.
- **cleanup**: Diese Methode sorgt dafür, dass nach Abschluss des Tests alle Testdaten entfernt werden. Sie stellt die Datenbank in ihren ursprünglichen Zustand zurück und stellt sicher, dass keine Testdaten zurückbleiben, die eine mögliche produktive Umgebung beeinträchtigen könnten.

Für das Demo-Beispiel wählen wir den Testtypen **oltp\_read\_write** und allen Methoden aus. Für die Methode run würde unsere Query so aussehen, wobei YOUR\_USER und YOUR\_PASSWORD entsprechend ersetzt werden müssten:

```
1 sysbench oltp_read_write \  
2   --db-driver=mysql \  
3   --mysql-user=YOUR_USER \  
4   --mysql-password=YOUR_PASSWORD \  
5   --mysql-db="sbtest" \  
6   --time=10 \  
7   --report-interval=1 \  
8   run
```

Wenn man nur diese Query ausführt, fällt er auf, dass die Query scheitert. Die Fehlermeldung lautet dabei wie folgt:

```
1 FATAL: MySQL error: 1146 "Table 'sbtest.sbtest1' doesn't exist"
```

Der entstandene Fehler wird offensichtlich dadurch verursacht, dass die Tabelle nicht erstellt worden ist. Die Lösung für dieses Problem ist die Ausführung der prepare - Methode vor der oltp\_read\_write - Methode. Damit sich die Datenbank wieder im Anfangszustand befindet noch anschließend an die oltp\_read\_write - Methode noch die cleanup aufrufen werden. Um sich die manuelle Ausführung dieser drei Befehl in der korrekten Reihenfolge zu sparen, bietet es sich an, ein Shell-Script zu schreiben, indem zuerst die Methoden nacheinander aufgerufen werden.

### Codeblock 2.2: Process of Sysbench commands

```
1 # Define necessary variables
2 DB_HOST="127.0.0.1"
3 DB_USER="root"
4 DB_PASS="password"
5 DB_NAME="sbtest"
6 TABLES=10
7 TABLE_SIZE=10000
8 DURATION=10
9 RAW_RESULTS_FILE="output/sysbench.log"
10
11 SYSBENCH_OPTS="--db-driver=mysql --mysql-host=$DB_HOST --mysql-user=$DB_USER --mysql-password=$DB_PASS --mysql-db=
    $DB_NAME --tables=$TABLES --table-size=$TABLE_SIZE"
12
13 # Create output directory
14 rm -rf "output"
15 mkdir -p "output"
16
17 # Prepare the database
18 echo "Preparing the database...";
19 sysbench oltp_read_write $SYSBENCH_OPTS prepare >> "$RAW_RESULTS_FILE" 2>&1
20 echo "Database prepared."
21
22 # Run the benchmark
23 echo "Running benchmark...";
24 sysbench oltp_read_write $SYSBENCH_OPTS --time=$DURATION --threads=1 --report-interval=1 run >> "$RAW_RESULTS_FILE"
    2>&1
25 echo "Benchmark complete."
26
27 # Cleanup the database
28 echo "Cleaning up...";
29 sysbench oltp_read_write $SYSBENCH_OPTS cleanup >> "$RAW_RESULTS_FILE" 2>&1
30 echo "Database cleanup complete."
```

Die Ergebnisse werden nun der Log-Datei (unter output/sysbench.log) gespeichert. Damit fehlt uns nur noch die Erstellung der Graphen. Um uns diese Erstellung zu vereinfachen, bietet es sich an, dass aus der Log - Datei die entsprechenden Kennzahlen extrahiert und die Werte mit korrekten Spaltenüberschriften in einer CSV-Datei speichert. Dies geht mit dem Shell - Kommando grep:

### Codeblock 2.3: Extraction from log to CSV

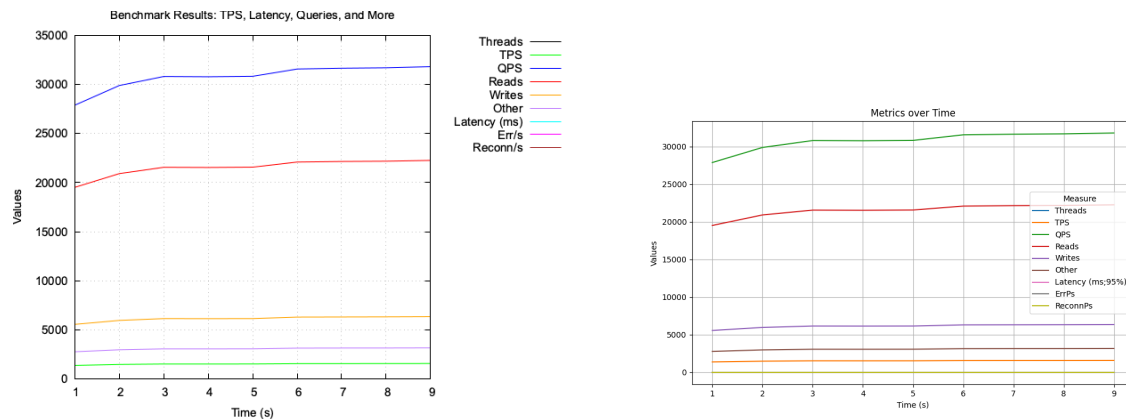
```
1 # Define necessary variables
2 RAW_RESULTS_FILE="output/sysbench.log"
3 OUTPUT_FILE="output/sysbench_output.csv"
4
5 echo "Script,Time (s),Threads,TPS,QPS,Reads,Writes,Other,Latency (ms;95%),ErrPs,ReconnPs" > "$OUTPUT_FILE"
6 grep '^[' ' $RAW_RESULTS_FILE | while read -r line; do
7     time=$(echo $line | awk '{print $2}' | sed 's/s//')
8     threads=$(echo $line | awk -F 'thds: ' '{print $2}' | awk '{print $1}')
9     tps=$(echo $line | awk -F 'tps: ' '{print $2}' | awk '{print $1}')
10    qps=$(echo $line | awk -F 'qps: ' '{print $2}' | awk '{print $1}')
11    read_write_other=$(echo $line | sed -E 's/.*(r/w/o: ([0-9.]+)\|([0-9.]+)\|([0-9.]+)\).*\/1,2,3/')
12    reads=$(echo $read_write_other | cut -d',' -f1)
13    writes=$(echo $read_write_other | cut -d',' -f2)
14    other=$(echo $read_write_other | cut -d',' -f3)
15    latency=$(echo $line | awk -F 'lat \\\(ms,95%\): ' '{print $2}' | awk '{print $1}')
16    err_per_sec=$(echo $line | awk -F 'err/s: ' '{print $2}' | awk '{print $1}')
17    reconn_per_sec=$(echo $line | awk -F 'reconn/s: ' '{print $2}' | awk '{print $1}')
18
19    echo "demo,$time,$threads,$tps,$qps,$reads,$writes,$other,$latency,$err_per_sec,$reconn_per_sec" >> "
20    $OUTPUT_FILE"
21
22 echo "Results saved to $OUTPUT_FILE."
```

Als letzten Schritt gibt es die Erstellung der Graphen mithilfe von der Tools Gnuplot oder der Python - Library Pandas. Die kompletten Scripts `plot_sysbench.gp` und `generatePlot.py` befinden sich am Ende dieser Bachelorarbeit. Das Python-Skript, das zuständig ist für die Graphgenerierung muss als Argument zum einen die CSV-Datei übermittelt bekommen und zum anderen kann es nur eine bestimmte Auswahl an Messwerten übergeben, damit nur für diese die Graphen erzeugt werden.

### Codeblock 2.4: Generation of graphs

```
1 OUTPUT_FILE="$OUTPUT_DIR/sysbench_output.csv"
2
3 # Gnuplot
4 GNUPLOT_SCRIPT="YOUR_PATH_TO_PROJECT/plot_sysbench.gp"
5 rm -rf "$OUTPUT_DIR/gnuplot"
6 mkdir -p "$OUTPUT_DIR/gnuplot"
7 echo "Generating plot with gnuplot..."
8 gnuplot $GNUPLOT_SCRIPT
9 echo "Plots generated with gnuplot"
10
11 # Python with Library Pandas
12 PYTHON_SCRIPT="YOUR_PATH_TO_PROJECT/generatePlot.py"
13 echo "Generating plots with pandas..."
14 python3 "$PYTHON_SCRIPT" "$OUTPUT_FILE"
15 echo "Plots generated with pandas"
```

- **Threads:** Die Anzahl der gleichzeitig verwendeten Threads. Mehr erhöhen die Parallelität, jedoch zu viele können zur Überlastung des Systems führen.



**Abbildung 2.1:** Die Grafik zeigt die erstellten Graphen mit Gnuplot (links) und Pandas (rechts)

- **TPS (Transactions Per Second):** Die Anzahl der Transaktionen pro Sekunde. Ein höherer Wert deutet auf eine bessere Datenbankleistung hin.
- **QPS (Queries Per Second):** Die Anzahl der Abfragen pro Sekunde. Ein höherer Wert ist besser und zeigt die Effizienz bei der Verarbeitung von Abfragen.
- **Reads:** Die Anzahl der Leseoperationen.
- **Writes:** Die Anzahl der Schreiboperationen.
- **Other:** Bezieht sich auf andere Arten von Operationen, die weder als Reads noch als Writes kategorisiert werden.
- **Latency (ms; 95%):** Die durchschnittliche Zeit in Millisekunden, die benötigt wird, um Anfragen zu bearbeiten, wobei der Wert im 95. Perzentil betrachtet wird. Niedrigere Werte sind besser, da sie auf schnellere Reaktionszeiten hinweisen.
- **ErrPs (Errors Per Second):** Die Anzahl der Fehler pro Sekunde. Ein niedriger Wert weist auf höhere Stabilität und Zuverlässigkeit des Systems hin.
- **ReconnPs (Reconnects Per Second):** Die Anzahl der Wiederverbindungen pro Sekunde. Häufige Wiederverbindungen können auch auf Stabilitätsprobleme hindeuten.

## 2.3 Genauere Darstellung anhand eines Beispiels

In dem vorausgegangenen Kapitel [Einführung in die Tools](#) wurde das Tool Sysbench und seine Funktionsweise anhand eines Demo - Projekts näher erläutert. Damit die Reihenfolge

und die Bedeutungen der unterschiedlichen Methoden (prepare → run → cleanup) sowie die Vorgehensweise zur Erstellung unserer Grafiken deutlich geworden. Das bisherige Problem ist aber, dass wir bei dem dargelegten Beispiel keine Kontrolle über die getesteten Daten haben. Wenn man sich die Logs genauer anschaut, dann sieht man, dass man über die Parameter an den Sysbench – Befehl die Anzahl der erstellten Tabellen und eingefügten Datensätze von außen steuern kann, aber die genaue Implementierung können wir auf diese Weise nicht steuern. Genau für diese Anwendungsfälle gibt es die Möglichkeit ein Lua - Skript als Parameter beim Sysbench - Aufruf mit anzugeben. In diesen Lua - Dateien können die Implementierungen der einzelnen Methoden selbstständig gewählt werden.

Um das Vorgehen besser erklären zu können, schauen wir uns dafür ein Beispiel an. Für das Beispiel wollen wir zwei Tabellen erstellen und anschließend mit zufälligen Testdaten befüllen. Die Abfrage, die wir auf Performance testen wollen, ist das Verbinden (Joinen) dieser beiden Tabellen. In unserem Fall wollen wir eine Kundentabelle erstellen, die Informationen wie Name, Geburtstag und Adresse enthält, sowie eine Bestelltabelle, die Details wie Artikelnummer, Bestelldatum usw. speichert und einen Bezug zu dem Kunden herstellt, der die Bestellung aufgegeben hat. Damit wir aber nicht nur ein Beispiel haben, das dargestellt wird, brauchen wir einen Vergleich zwischen zwei verschiedenen Implementierungen. Dieser Unterschied zwischen den beiden Implementierungen besteht darin, dass in der einen Version die Tabelle eine Kundennummer vom Typ Int enthält, während in der anderen keine Kundennummer vorhanden ist. Stattdessen wird in der Bestelltabelle auf den Namen (Typ Varchar) verwiesen. Da Verbundoperationen zu den aufwendigsten Operationen gehören, gehen wir davon aus, dass der kleine Typ Int Performancevorteile gegenüber der anderen Version hat. Dies gilt es nun mit den Benchmarktest genauer zu untersuchen.

Für die Durchführen der Benchmarks beginnen wir zunächst unabhängig von Sysbench und den Lua – Skripten mit der Spezifizierung der Tabellen, die erstellt werden sollen. Dies müssen wir einmal mit der Kundennummer und einmal mit dem Namen als Fremdschlüssel der Bestelltabelle machen. Damit müssen insgesamt vier unterschiedliche Create Table - Befehle umgesetzt werden. So sehen die Create Table für den Fall mit der Kundennummer aus:

#### **Codeblock 2.5:** Create Table - Befehl für Tabelle Kunden

```
1 CREATE TABLE KUNDEN (  
2     KUNDEN_ID      INT PRIMARY KEY,  
3     NAME           VARCHAR(255),  
4     GEBURTSTAG     DATE,  
5     ADRESSE        VARCHAR(255),  
6     STADT          VARCHAR(100),  
7     POSTLEITZAHL   VARCHAR(10),  
8     LAND           VARCHAR(100),
```



```

9     EMAIL          VARCHAR(255) UNIQUE,
10    TELEFONNUMMER  VARCHAR(20)
11 );

```

### Codeblock 2.6: Create Table - Befehl für Tabelle Bestellung

```

1 CREATE TABLE BESTELLUNG (
2     BESTELLUNG_ID INT PRIMARY KEY,
3     BESTELLDATUM DATE,
4     ARTIKEL_ID INT,
5     UMSATZ INT,
6     FK_KUNDEN INT NOT NULL,
7     FOREIGN KEY (FK_KUNDEN) REFERENCES KUNDEN (KUNDEN_ID)
8 );

```

Anschließend müssen wir diese Befehle in `prepare()` - Funktion miteinbinden. Dafür müssen wir einfach die Create Table - Befehle an die Datenbank senden. Wenn wir bestimmte Indexe oder andere Datenbankstrukturen erstellen wollen würden, dann müssten wir dies ebenfalls in dieser Funktion machen. Dies ist ein Auszug aus der Prepare - Funktion:

### Codeblock 2.7: Lua - Script für die Erstellung der Tabellen

```

1 function prepare()
2     local create_kunden_query = [[
3         CREATE TABLE KUNDENMITID (
4             ...
5         );
6     ]]
7     local create_bestellung_query = [[
8         CREATE TABLE BESTELLUNGMITID (
9             ...
10        );
11    ]]
12
13    db_query(create_kunden_query)
14    db_query(create_bestellung_query)
15    print("Tables KUNDENMITID and BESTELLUNGMITID have been successfully created.")
16 end

```

Wenn die Datenbank beispielsweise in einer Produktivumgebung läuft, dann wollen wir, dass die Benchmarks möglichst wenig Einfluss auf sie haben. Damit ist es das Ziel, dass die Datenbank möglichst wieder in ihrem Anfangszustand ist. Außerdem sollte der Benchmark beliebig oft nacheinander ausgeführt werden können, ohne zu Problemen zu führen. Wenn wir aber eine Tabelle erstellen und nicht wieder löschen, dann würde im nächsten Durchlauf der Create Table - Befehl scheitern. Lösen könnte man dies über Klausel „IF NOT EXISTS“

bei der Erstellung der Tabelle hinzufügen oder noch es besser ist es die Tabelle am Ende des Benchmarks einfach zu löschen. Dafür ist die `cleanup()` – Funktion vorgesehen:

### Codeblock 2.8: Lua - Script für das Aufräumen

```
1 function cleanup()
2     local drop_kunden_query = "DROP TABLE IF EXISTS KUNDENMITID;"
3     local drop_bestellung_query = "DROP TABLE IF EXISTS BESTELLUNGMITID;"
4
5     db_query(drop_bestellung_query)
6     db_query(drop_kunden_query)
7     print("Cleanup successfully done")
8 end
```

Wichtig ist dabei, dass man keine Schlüsselintegritäten verletzt. Da in diesem Fall die Tabelle `BESTELLUNGMITID` eine Referenz auf die Tabelle `KUNDENMITID` hat, muss zuerst die Bestelltabelle und danach erst die Kundentabelle entfernt werden.

Jetzt haben wir das Gerüst für die eigentlichen Insert - und Select - Befehle geschaffen. Bei den Insert - Befehlen können wir entweder mit zufälligen Zahlen die Werte generieren oder wir setzen Listen von Namen fest, aus denen zufällig gewählt werden kann. Da wir jedoch keine Kontrolle über diese zufällig erstellten Werte haben, müssen wir beim Insert - Befehl die Bedingung „Insert Ignore“ hinzufügen, damit doppelte Schlüsselwerte ignoriert werden und keine Fehler verursachen. Wir müssen hier auch festlegen, wie viele Datensätze für die Kunden erstellt werden und wie viele Bestellungen pro Kunden es geben soll. Später werden wir noch eine Möglichkeit kennen lernen, um diese Werte von außen zusteuern. Um sicherzustellen, dass keine Werte in den Tabellen enthalten sind, können wir alle Datensätze aus den Tabellen entfernen, bevor wir sie hinzufügen. Damit die Performance der Insert – Query auch gemessen wird, ist es wichtig, dass die `insert()` - Funktion in der `event()` – Funktion aufgerufen wird. Sonst kommt es zu diesem Fehler:

```
1 FATAL: cannot find the event() function in Join.lua
```

### Codeblock 2.9: Lua - Script für das Einfügen von Daten

```
1 local num_rows = 1000
2 local bestellungProKunde = 4
3
4 function delete_data()
5     local delete_bestellung_query = "DELETE FROM BESTELLUNGMITID;"
6     local delete_kunden_query = "DELETE FROM KUNDENMITID;"
7     db_query("START TRANSACTION")
8     db_query(delete_bestellung_query)
9     db_query(delete_kunden_query)
10    db_query("COMMIT")
11 end
12
13 function insert_data()
```

```

14 delete_data()
15 for i = 1, num_rows do
16     local kunden_id = i
17     local name = string.format("Kunde_%d", i)
18     local geburstag = string.format("19%02d-%02d-%02d", math.random(50, 99), math.random(1, 12), math.random
(1, 28))
19     -- define adresse, stadt, postleitzahl, land, email, telefonnummer
20
21     local kunden_query = string.format([[
22         INSERT IGNORE INTO KUNDENMITID
23         (KUNDEN_ID, NAME, GEBURSTAG, ADRESSE, STADT, POSTLEITZAHL, LAND, EMAIL, TELEFONNUMMER)
24         VALUES (%d, '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s');
25     ]], kunden_id, name, geburstag, adresse, stadt, postleitzahl, land, email, telefonnummer)
26
27     db_query(kunden_query)
28
29     for j = 1, bestellungProKunde do
30         local bestellung_id = (i-1) * bestellungProKunde + j
31         -- define bestelldatum, artikel_id, umsatz
32
33         local bestellung_query = string.format([[
34             INSERT IGNORE INTO BESTELLUNGMITID
35             (BESTELLUNG_ID, BESTELLDATUM, ARTIKEL_ID, UMSATZ, FK_KUNDEN)
36             VALUES (%d, '%s', %d, %d, %d);
37         ]], bestellung_id, bestelldatum, artikel_id, umsatz, kunden_id)
38
39         db_query(bestellung_query)
40     end
41 end
42 end
43
44 function event()
45     insert_data()
46 end

```

Die letzte Anweisung, die wir noch brauchen, ist die Select - Abfrage. Hierbei muss man sich Gedanken machen, welche Abfrage benötigt wird, damit die untersuchten Effekte auch tatsächlich auftreten. In dem Beispiel brauchen wir deswegen einen Join zwischen den beiden Tabellen über den Fremdschlüssel.

### Codeblock 2.10: Lua - Script für das Abfragen von Daten

```

1 function select_query()
2     local join_query = [[
3         SELECT k.STADT, SUM(b.UMSATZ) AS Total_Umsatz
4         FROM KUNDENMITVARCHAR k
5         JOIN BESTELLUNGMITVARCHAR b ON k.NAME = b.FK_KUNDEN
6         GROUP BY k.STADT;
7     ]]
8     db_query(join_query)
9 end
10
11 function event()
12     select_query()
13 end

```

Damit haben wir für unseren Vergleich alle vier Operationen genauer definiert und müssen diese 4 Funktionen nur noch leicht anpassen für die Implementierung mit dem Namen als Fremdschlüssel und ohne die Kundennummer in der Kundentabelle. Daraufhin benötigen wir noch ein Skript, dass die Operationen in der korrekten Reihenfolge ausführt und die Grafiken generiert. Wichtig dafür ist die folgende Dateienstruktur, die anhand der Int - Verbunds dargestellt wird.

Damit wir die unterschiedlichen Operationen voneinander trennen können, gibt es folgende Dateienstruktur: Es gibt einen Ordner mit einem beliebigen Namen, z.B. `int_queries`, in diesem Ordner befinden sich folgende Dateien:

- `int_queries.lua` ⇒ enthält die `prepare()`- und `cleanup()`-Funktionen
- `int_queries_insert.lua` ⇒ enthält die `insert()`-Funktion
- `int_queries_select.lua` ⇒ enthält die `select()`-Funktion

Analog muss auch ein Ordner für die Varchar - Vergleich erstellt werden. Als Letztes brauchen wir nur einen Orchestrator, der das korrekte Lua - Skript ausführt, die Ergebnisse in die richtige Log - Datei schreibt und anschließend die CSV - Dateien und die Grafiken erstellt. Dieser Orchestrator ist das Shell - Skript: `sysbench_script.sh`.

Zudem möchten wir unser Beispiel erweitern, da es auch möglich sein soll, unterschiedliche Längen von Varchar hinzuzufügen. Dadurch könnten wir nicht nur den Performanceunterschied zwischen Int und Varchar feststellen, sondern auch noch den Einfluss der Länge des Verbundoperators für Varchar. Dazu benötigen wir eine Hilfsfunktion in `varchar_queries_insert.lua`, die einen zufälligen Namen mit der Länge von einer vorgegebenen Zahl erstellt. Dieser Name ist damit kein natürlicher Name, sondern einfach eine Kombination von zufälligen Buchstaben, aber für unseren Testfall gehen wir diesen Kompromiss ein. Wenn wir jetzt zwei unterschiedlichen Längen für Varchar testen wollen, dann müssten wir den Varchar - Ordner mit den oben beschriebenen drei Dateien kopieren und nur die Zeile ändern, die die Länge des zufälligen Namens bestimmt. Dies würde zu extremer Redundanz führen, weshalb man beim Aufruf des Orchestrator - Scripts, Variablen definieren kann, die im Skript selbst exportiert und in der `varchar_queries_insert.lua` - Datei importiert werden können.

Dies ist die Zeile mit der festgelegten Länge:

```
1 local length = 10
```

Die Zeile mit der importierten Länge:

```
1 local length = tonumber(os.getenv("LENGTH"))
```

Den Orchestrator - Script ruft man wie folgt auf:

### Codeblock 2.11: Befehl zum Ausführen des Orchestrator Skripts

```
1 ./sysbench_script.sh \  
2 -out "YOUR_PATH_TO_DIRECTORY/Output" \  
3 -var '{"length": [1,64]}' \  
4 -scripts:"YOUR_PATH_TO_DIRECTORY/int_queries" \  
5 "YOUR_PATH_TO_DIRECTORY/varchar_queries:length"
```

Die Parameter haben folgende Funktion:

- -out: Gibt den Pfad an, an welchen der Output-Ordner gespeichert werden soll
- -var: Angabe der Variablen und deren Werte, die exportiert werden sollen im JSON-Format
- -scripts: Angabe der Pfade zu den Ordnern, die die Lua-Skripte enthalten. Nach dem Doppelpunkt wird angegeben, welche Variable das Skript benötigt. Int\_queries benötigt keine Variablen, deshalb gibt es auch keinen Doppelpunkt.

Die letzte Besonderheit ist es, dass man mehrere Select – Abfragen ohne unterschiedliche Insert - Befehle definieren kann. Zu einem späteren Punkt in der Bachelorarbeit werden wir zu unterschiedliche Indextypen kommen. Um zu untersuchen, ob ein bestimmter Indextyp bei Abfragen verwendet wird, müssen wir nur unterschiedliche Selects abfragen. Die eigentlichen Tabellen und deren Datensätze müssen dabei nicht immer wieder gleich befüllt werden. Wenn wir auch unsere Ordnerstruktur mit dem Int - Query Beispiel zurückkommen, dann könnte man anstelle von int\_queries\_select.lua auch einen Ordner erstellen mit den Namen int\_queries\_select. In diesem Ordner können beliebig viele unterschiedliche Lua - Skripts sein, die select – Funktionen haben. Dadurch werden alle Select - Befehle auf der gleichen Datenbasis verglichen und so können wir im Kapitel (TODO (Daniel): Index) erkennen, wann der Index verwendet wird und wann nicht.

Bevor wir uns das Ergebnis des Befehls anschauen, kommen wir zu der Funktionsweise des Orchestrator - Skript sysbench\_script.sh. Im Grundlegenden arbeitet dieses Skript ähnlich wie schon das Skript im Demo - Beispiel, aber durch die zusätzlichen Anwendungsfälle kommt es zu mehr Komplexität.

Zu Beginn des Skripts werden die Umgebungsvariablen aus der Datei db.env geladen. Die Variablen helfen zum einen wie bei dem Demo - Beispiel bei die Datenbank - Verbindung und zum anderen können sich auch die Parameter der Benchmarks verändern. Danach werden die Parameter, die an das Skript übergeben wurden, überprüft. Beispielsweise wird sichergestellt, dass die für die Skripts verwendeten Parameter, bei unserem Beispiel length für varchar, tatsächlich auch definiert worden sind mit -var.

Wenn wir den Befehl ausführen, wird der Output-Ordner an der gewünschten Stelle erstellt. In diesem Ordner werden verschiedene Grafiken generiert, die die Ergebnisse visualisieren. Dabei gibt es zwei unterschiedliche Arten von Grafiken. Die erste Art von Grafik ist ein Zeitreihendiagramm, welches auf der x-Achse den zeitlichen Verlauf zeigt. Auf der y-Achse werden in einigen Diagrammen die unterschiedlichen Metriken für jedes einzelne Skript dargestellt, während andere Diagramme die Werte einer bestimmten Metrik auf der y-Achse zeigen und dabei die Ergebnisse verschiedener Skripte vergleichen. Dadurch können beispielsweise die Metriken Reads und Writes analysiert werden, um herauszufinden, welches Skript in diesen Bereichen besser abschneidet. Danach wird der Output - Ordner erstellt und die Spaltenüberschriften in die CSV – Dateien geschrieben. Danach beginnt erst das eigentliche Durchgehen der unterschiedlichen Skripte, die unter dem Argument -script angegeben wurden. Zunächst schreibt man die einzelnen Dateien nach dem obigen Schema (2.3) auf, denn als Argument wurde nur der oberste Ordner angegeben. Als nächstes kommt eine Fallunterscheidung, die überprüft, ob dieses Skript exportierte Variable nutzt oder nicht. Für den Fall, dass keine Variablen exportiert werden (z.B. int\_queries) wird einfach die Prepare – Funktion aufgerufen, dann process\_script\_benchmark und anschließend die Cleanup - Funktion. Wenn aber Variablen exportiert werden, dann müssen weitere Zwischenschritte umgesetzt werden. Und zwar müssen alle Kombinationen zwischen den verschiedenen exportierten Variablen generiert werden. Wenn es drei Variablen gibt, von denen 2 jeweils 2 Werte und eine letzte nur einen Wert hat, dann gibt es  $2 * 2 * 1 = 4$  unterschiedliche Kombinationen. Als nächstes muss man für alle diese Kombinationen die Schritte ausführen, die man schon bei der Variante ohne exportierte Variable ausgeführt hat und dabei darf man nicht vergessen die Variablen an sich zu exportieren.

## Codeblock 2.12: Ausschnitt aus Orchestrator Script

```

1 # Main benchmark loop
2 for INFO in "${QUERY_INFO[@]"; do
3     IFS=: read -r QUERY_PATH MULTIPLE_KEYS <<< "$INFO"
4
5     MAIN_SCRIPT="${QUERY_PATH}/${(basename "$QUERY_PATH").lua"
6     INSERT_SCRIPT="${QUERY_PATH}/${(basename "$QUERY_PATH")}_insert.lua"
7     SELECT_SCRIPT="${QUERY_PATH}/${(basename "$QUERY_PATH")}_select"
8     LOG_DIR="$OUTPUT_DIR/logs/${(basename "$QUERY_PATH")}"
9
10    if [[ -n "$MULTIPLE_KEYS" ]]; then
11        IFS=', ' read -r -a KEYS <<< "$MULTIPLE_KEYS"
12
13        combinations=$(generate_combinations "" "${KEYS[@]}")
14
15        # Process each combination
16        while IFS=', ' read -r combination; do
17            # Export key-value pairs for the current combination
18            IFS=', ' read -ra key_value_pairs <<< "$combination"
19            for pair in "${key_value_pairs[@]"; do
20                key="{pair%#}"
21                value="{pair##}"
22                export "$(echo "$key" | tr '[:lower:]' '[:upper:]')=$value"

```

```

23     done
24
25     COMBINATION_NAME=$(echo "$combination" | tr ',' '_' | tr '=' '_')
26     LOG_DIR_KEY_VALUE="$LOG_DIR/$COMBINATION_NAME"
27     mkdir -p "$LOG_DIR_KEY_VALUE"
28
29     RAW_RESULTS_FILE="${LOG_DIR_KEY_VALUE}/${(basename "$QUERY_PATH")}_${COMBINATION_NAME}_prepare.log"
30     run_benchmark "$MAIN_SCRIPT" "prepare" "$RAW_RESULTS_FILE" "" "$COMBINATION_NAME"
31
32     process_script_benchmark "$QUERY_PATH" "$LOG_DIR_KEY_VALUE" "$INSERT_SCRIPT" "$SELECT_SCRIPT" "
33     $COMBINATION_NAME"
34
35     RAW_RESULTS_FILE="${LOG_DIR_KEY_VALUE}/${(basename "$QUERY_PATH")}_${COMBINATION_NAME}_cleanup.log"
36     run_benchmark "$MAIN_SCRIPT" "cleanup" "$RAW_RESULTS_FILE"
37 done <<< "$combinations"
38 else
39     # Process normally when no keys specified
40     mkdir -p "$LOG_DIR"
41     RAW_RESULTS_FILE="$LOG_DIR/${(basename "$QUERY_PATH")}_prepare.log"
42     run_benchmark "$MAIN_SCRIPT" "prepare" "$RAW_RESULTS_FILE"
43
44     process_script_benchmark "$QUERY_PATH" "$LOG_DIR" "$INSERT_SCRIPT" "$SELECT_SCRIPT"
45
46     RAW_RESULTS_FILE="$LOG_DIR/${(basename "$QUERY_PATH")}_cleanup.log"
47     run_benchmark "$MAIN_SCRIPT" "cleanup" "$RAW_RESULTS_FILE"
48 fi
done

```

Die Funktion `process_script_benchmark` überprüft noch es sich bei dem Select – Directory um einen Ordner handelt oder nicht. Wenn es ein Ordner ist, dann werden alle Dateien in diesem Ordner mit Sysbench durchgeführt, wenn nicht, dann wird an den Ordner nur die Endung `.lua` hinzugefügt.

### Codeblock 2.13: Methode Process Script Benchmark

```

1 process_script_benchmark() {
2     local SCRIPTS=()
3     local IS_FROM_SELECT_DIR=false
4
5     if [ -f "$SELECT_SCRIPT.lua" ]; then
6         # SELECT_SCRIPT is a Lua file
7         SCRIPTS=("$INSERT_SCRIPT" "$SELECT_SCRIPT.lua")
8     else
9         # SELECT_SCRIPT is a directory
10        SCRIPTS=("$INSERT_SCRIPT" "$SELECT_SCRIPT"/*)
11        IS_FROM_SELECT_DIR=true
12    fi
13
14    for SCRIPT in "${SCRIPTS[@]}; do
15        if [ -f "$SCRIPT" ]; then
16            local SCRIPT_NAME
17            # Removed script name definition for less complexity => multiple if-cases processed
18            local RAW_RESULTS_FILE="$LOG_DIR/${SCRIPT_NAME}.log"
19            run_benchmark "$SCRIPT" "run" "$RAW_RESULTS_FILE" "$SCRIPT_NAME" "$COMBINATION"
20        fi
21    done
22 }

```

Die Methode `run_benchmark` führt den `Sysbench` – Befehl aus und wenn es sich um die Methode `Run` handelt, dann müssen die Daten während der Ausführung und die Statistiken am Ende in je eine CSV - Datei speichern. Aus diesen beiden CSV - Dateien müssen die Insert - und Select - Queries der zugehörigen Skripte wieder vereint werden und die Attribute werden miteinander addiert. Als letzter Schritt erfolgt noch der bekannte Schritt der Grapherstellung.

#### Codeblock 2.14: Methode Run Benchmark

```
1 run_benchmark() {
2   # For OTHER_ENVIRONMENT_VARIABLES => see demo
3   sysbench --db-driver=mysql OTHER_ENVIRONMENT_VARIABLES "$SCRIPT_PATH" "$MODE" >> "$OUTPUT_FILE" 2>&1
4   if [ $? -ne 0 ]; then
5     echo "Benchmark failed for script $SCRIPT_PATH. Exiting."
6     exit 1
7   fi
8
9   # Only extract data if the mode is "run"
10  if [ "$MODE" == "run" ]; then
11    extract_run_data "$RAW_RESULTS_FILE" "$SCRIPT_NAME"
12    extract_statistics "$RAW_RESULTS_FILE" "$SCRIPT_NAME"
13  fi
14 }
```

INFO: Die Shell - Ausschnitte sind zum Teil verkürzt und würden auf diese Weise nicht funktionieren.

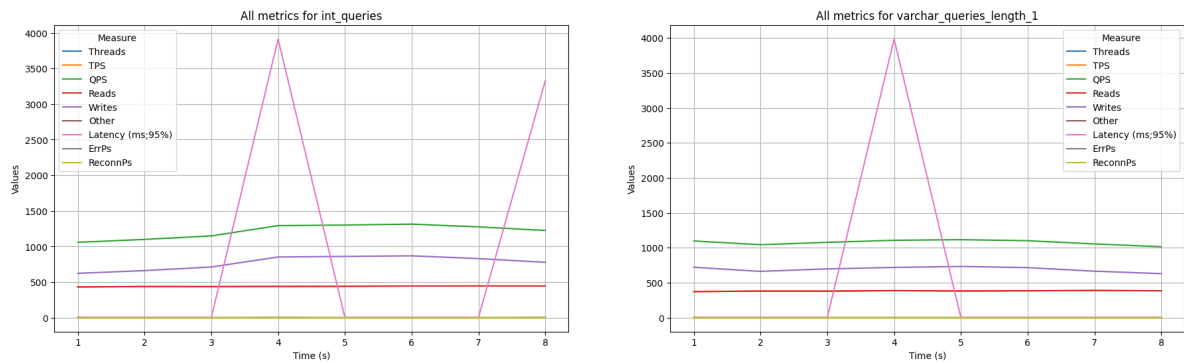
Wenn wir den Befehl ausführen, wird der Output-Ordner an der gewünschten Stelle erstellt. In diesem Ordner werden verschiedene Grafiken generiert, die die Ergebnisse visualisieren. Die erste Art stellen Zeitreihendiagramme dar, die auf der X-Achse den zeitlichen Verlauf zeigen. Auf der Y-Achse werden hingegen in einigen Diagrammen die unterschiedlichen Metriken eines einzelnen Skripts dargestellt, während andere Diagramme die Werte einer bestimmten Metrik auf der Y-Achse zeigen und dabei die Ergebnisse verschiedener Skripte vergleichen. Dadurch können beispielsweise die Metriken „Reads“ und „Writes“ analysiert werden, um herauszufinden, welches Skript in diesen Bereichen besser abschneidet.

Die zweite Art von Grafik, die erstellt wird, ist ein Hexagon - Diagramm. Dieses verzichtet auf eine Zeitachse und fasst die Performance über den gesamten Zeitraum hinweg zusammen. Im Vergleich zur Laufzeitanalyse liefert es zusätzliche Informationen, wie etwa die Latenz oder die Gesamtanzahl der Queries. Dadurch ist es auch möglich, dass mehrere Skripte und mehrere Kennzahlen in einer Grafik dargestellt werden können.

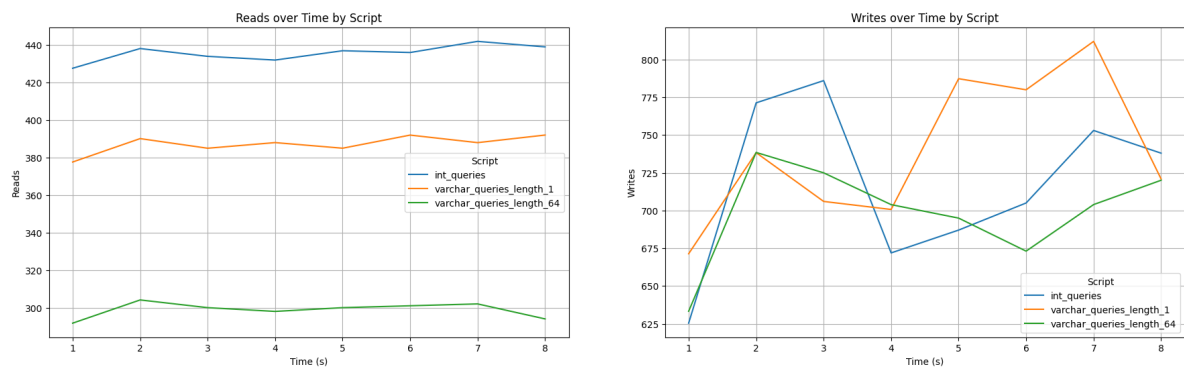
#### PNGs

Aus den Grafiken, die für ein Skript alle Metriken veranschaulichen, kann man möglicherweise Datenfehler erkennen. So springt bei Abbildung (`int_queries.png`) die Latenz bei einigen Messpunkten von 0 ms auf einen deutlich erhöhten Wert und danach wieder auf 0 ms zurück. Ansonsten aber sind die anderen Metriken auf einem konstanten Level, und es gibt wenige



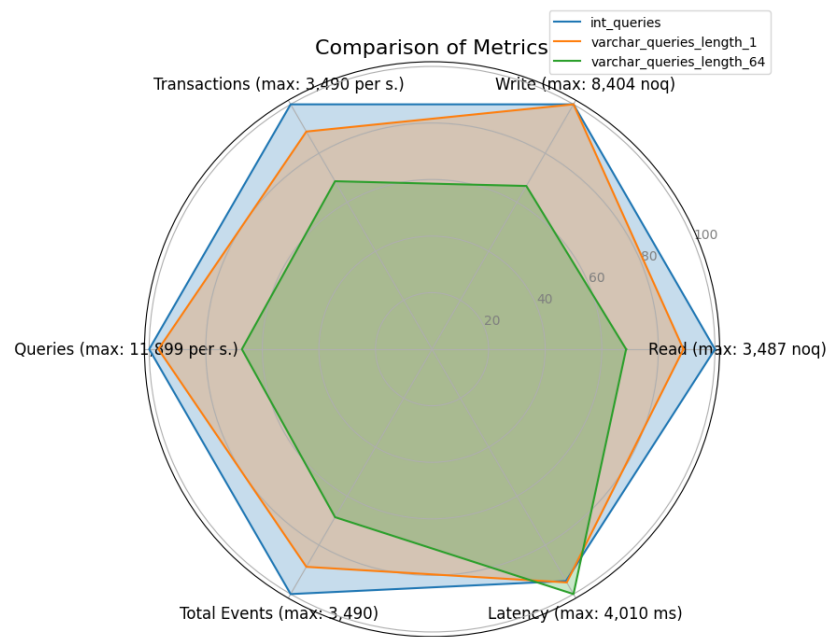


**Abbildung 2.2:** Die Grafik zeigt alle Metriken für die Skripte `int_queries` (links) und `varchar_queries_length_1` (rechts)



**Abbildung 2.3:** Die Grafik zeigt den Vergleich zwischen allen Skripten für die Metriken Reads (links) und Writes (rechts)

Schwankungen. Bei der Abbildung (`varchar_queries_length_1.png`) sieht dies sehr ähnlich aus, und auch dort schwankt die Latenz etwas mehr. Wenn wir jetzt die drei Skripte miteinander vergleichen wollen, können wir die Abbildungen `Reads.png` und `Writes.png` heranziehen. Was die Lesegeschwindigkeit angeht, kann man erkennen, dass `int_queries` am meisten Reads hat, als Nächstes kommt `varchar_queries_length_1` und dann `varchar_queries_length_64`. Damit sind die Abfragen, wie wir erwartet haben, bei `int_queries` am schnellsten, und je länger der String wird, desto langsamer werden die Abfragen. Bei den Schreibgeschwindigkeiten sieht das schon etwas anders aus, wobei es hier zunächst bei allen eine langsamere Startphase gibt. Anschließend an diesen Cold Start liegt das Niveau von `int_queries` am höchsten, also auch am schnellsten. Die beiden `varchar_queries` sind hier aber überraschenderweise auf einem ähnlichen Niveau. Bei der Abbildung (`statistics.png`) kann man die Effekte der Lese- und Schreibgeschwindigkeiten auch erkennen. Es fällt auch auf, dass, anders als bei Reads, Writes, Queries, die Latenz bei schnelleren Queries geringer ist und nicht der höchste Wert



**Abbildung 2.4:** Darstellung der Skripte und 6 Metriken in einem Hexagon  
der Beste ist.

## 3 Projektdurchführung

### 3.1 GitHub Action

Im Laufe der Bachelorarbeit sind immer mehr unterschiedliche Projekte dazugekommen, die alle das Orchestrator-Skript benutzen. Dadurch sind immer mehr Fallunterscheidungen in diesem Skript erforderlich geworden, und man hat schnell den Überblick verloren, wenn man Änderungen vorgenommen hat. Um zu überprüfen, ob diese Änderungen negative Nebeneffekte haben, mussten jedes Mal alle Skripte nacheinander ausgeführt werden, was nicht nur zeitintensiv war, sondern auch hohe Lasten für den lokalen Rechner bedeutete. Eine Möglichkeit wäre es gewesen, die Skripte parallel durchlaufen zu lassen, um Zeit zu sparen, aber damit wäre das Lastenproblem nicht gelöst worden. Eine deutlich bessere Variante ist die Auslagerung in eine Pipeline. In meinem Fall habe ich mich für GitHub Actions entschieden. Vereinfacht gesagt sollen in der GitHub Action alle Skripte parallel ausgeführt und am Ende alle Output-Dateien in einen Ordner zusammen als GitHub Artifact hochgeladen werden. Anschließend kann man die Zip-Datei herunterladen und überprüfen, ob alle Ergebnisse noch mit der Erwartung übereinstimmen. Wenn fehlerhafte Änderungen hochgeladen wurden, scheitert der Workflow-Run direkt, und man hat einen guten Überblick über alle Projekte.

Damit dies funktioniert, muss eine YAML-Datei im Ordner `.github/workflows/` erstellt werden. Anschließend bekommt der Workflow einen Namen und man kann definieren, wann er getriggert werden soll. In meinem Fall, wenn sich etwas im `./github`-Ordner verändert, im `Projects/`-Ordner, in dem sich alle Lua-Skripte befinden, oder wenn sich das Orchestrator-Skript und die in diesem Skript benutzten Python-Dateien ändern. Als Nächstes muss man eine JSON-Datei mit den Informationen zu den exportierten Variablen und den verwendeten Skripten befüllen und den einzelnen Projekten einen Namen geben. Diesen Namen muss man auch in der Matrix definieren, damit die Informationen parallel aus der JSON-Datei entnommen und die Befehle ausgeführt werden können.

#### Codeblock 3.1: JSON mit Konfiguration der Script

```
1 {  
2   "join-type": {  
3     "dirs": [  
4       "lua",  
5       "python",  
6       "orchestrator",  
7       "scripts",  
8       "tests",  
9       "utils",  
10      ]  
11    }  
12  }
```

```

4     "./Projects/Join_Typ/Scripts/int_queries",
5     "./Projects/Join_Typ/Scripts/varchar_queries:length"
6 ],
7 "var": {"length": [1, 64]}
8 }
9 }

```

Anschließend muss man wie schon bei den Skripten davor die Umgebungsvariablen in der YAML – Datei definieren, wenn es sich um vertrauliche Informationen handeln sollte, bietet es sich an GitHub Secrets zu benutzen. Danach beginnen erst die eigentlichen Schritte, die in dem Workflow ausgeführt werden. Zunächst muss man das Repository auschecken und die passende Konfiguration aus der JSON – Datei laden, die dem Testtypen entspricht aus der Matrix. Anschließend muss man nur noch die Abhängigkeiten installieren, die von den Skripten benötigt werden. Dazu gehören unter anderem Sysbench, Pandas und Matplotlib. Danach muss man den MySQL - Container mit den korrekten Umgebungsvariablen starten und als Nächstes wird das Orchestrator - Skript ausgeführt. Dieses Skript erstellt, wie schon erklärt, die Graphen, CSV - und Logdateien in einem Output Ordner. Um das Ganze aufzuräumen, kann man dem MySQL Container wieder stoppen und als letzten Schritt muss man nur noch den Output Ordner als Artifact hochladen.

### Codeblock 3.2: Ausschnitt aus der Workflow - Datei

```

1 name: Benchmark Workflow
2 on:
3   push:
4     paths:
5       - 'Projects/**'
6       - ...
7 jobs:
8   benchmark:
9     runs-on: ubuntu-latest
10    strategy:
11      matrix:
12        test-type: [high-count, low-count, b-tree-query-differences, selectivity-change, hash-query-differences,
13                    join-type, null-check, int-char, data-size]
14    env:
15      OTHER_ENVIRONMENT_VARIABLES
16    steps:
17      - name: Checkout repository
18        uses: actions/checkout@v3
19      - name: Load configuration from JSON
20        run: # removed the definition and export of variables for simplicity reasons
21      - name: Install dependencies
22        run: |
23          sudo apt-get update
24          sudo apt-get install -y sysbench
25          python -m pip install --upgrade pip
26          pip install pandas matplotlib
27      - name: Start MySQL container
28        run: |

```

```

28     docker run --name mysql-${{ matrix.test-type }} -d \
29         -e MYSQL_ROOT_PASSWORD=${DB_PASS} \
30         -e MYSQL_DATABASE=${DB_NAME} \
31         -p ${DB_PORT}:3306 mysql:8.0
32     until docker exec mysql-${{ matrix.test-type }} mysqladmin --user=root --password=${DB_PASS} --host
=127.0.0.1 --port=${DB_PORT} ping --silent; do sleep 1; done
33     echo "MySQL is ready!"
34     - name: Run sysbench script
35     run: |
36         chmod +x Tools/sysbench_script.sh
37         Tools/sysbench_script.sh \
38         -out "${{ env.output_dir }}" \
39         -var "${{ env.var }}" \
40         -scripts:${(echo "${{ env.dirs_string }}" | sed 's/ / /g')}
41     - name: Stop MySQL container
42     run: |
43         docker stop mysql-${{ matrix.test-type }}
44         docker rm mysql-${{ matrix.test-type }}
45     - name: Upload all
46     uses: actions/upload-artifact@v3
47     with:
48         name: combined-output
49         path: Output

```

Die eben beschriebene YAML - Datei reicht aus, damit alle angegebenen Skripten in dem JSON ausgeführt und die Output Dateien alle korrekt in einem Ordner als ZIP-Datei hochgeladen werden. Es bieten sich aber auch Alternativen an, die zu einer Optimierung des Workflows führen.

Zum einen kann man die zu installierenden Abhängigkeiten mithilfe des GitHub Caches ([12]) speichern. Dies bietet sich besonders an, da sich die Abhängigkeiten über die Workflows hinweg nur selten ändern. Falls sich doch etwas ändert, kann man beispielsweise die `requirements.txt`-Datei anpassen. Dadurch werden einmalig alle Abhängigkeiten neu installiert und anschließend im Cache abgelegt. Falls sich bis zum nächsten Workflow keine Änderungen an den Abhängigkeiten ergeben, wird der Cache automatisch genutzt. Der Zeitgewinn in unserem Beispiel ist jedoch nur gering und beträgt nur wenige Sekunden pro Workflow.

### Codeblock 3.3: Speichern der Abhängigkeiten im Cache

```

1 - name: Cache pip dependencies
2   if: env.should_run == 'true'
3   uses: actions/cache@v3
4   with:
5     path: ~/.cache/pip
6     key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}

```

Deutlich mehr Zeit und Ressourcen kann man aber sparen, wenn man zwischen zwei unterschiedlichen Arten von Dateien unterscheidet. Denn zum einen gibt es Dateien, die die Ergebnisse von allen Skripten beeinflussen. Dazu gehören das Workflow - Skript und die JSON

- Datei, aber auch das Orchestrator - Skript und die darin verwendeten Python - Skripte. Die Ordner an sich, die in der JSON angegeben werden, die beeinflussen nur sich selbst und nicht die anderen Skripte. Beispielsweise, wenn ich in Projekt A die Anzahl an Zeilen ändere, die ausgeführt werden, dann ändert dies nichts an dem Ergebnis von Projekt B oder C. In diesem Beispiel würde es sich anbieten, dass für Projekt A die Benchmarks neu durchgeführt werden, für Projekt B und C könnte hingegen jeweils der letzte erfolgreiche Output Ordner benutzt werden. Als Endresultat könnten damit die neue Durchführung von Projekt A zusammen mit der alten Ausführung der Projekte B und C in einer ZIP - Datei hochgeladen werden. Dadurch wird nur ein Drittel der eigentlichen Ressourcen verbraucht, wenn man davon ausgehen würde, dass alle 3 Projekte gleich viel Zeit benötigen würden.

Für die Implementierung dieser Optimierung muss zunächst die allgemeinen Skripte hashen und zusätzlich noch die Ordner mit den Lua - Skripten, die für das jeweilige Skript aus der JSON benötigt werden. Diese beiden Hashes kann zusammen mit den Testtypen kombinieren, damit bekommt die folgende Struktur für den Namen:

```
1 NAME="${{ matrix.test-type }}-${{ env.hash }}-${{ env.general_hash }}"
```

Nachdem wir unsere JSON geladen haben, machen wir nun nicht mehr direkt mit der Installation der Abhängigkeiten weiter, sondern davor hashen wir die unterschiedlichen Pfade und erstellen unseren Namen. Wenn es keinen Ordner mit dem gleichen Namen gibt, dann machen wir weiter wie bisher. Das einzige, was sich ändert, ist der Schritt vor dem Hochladen des gesamten Output-Ordners. Der vom Testtypen erzeugte Ordner muss zusammen mit seinem Namen hochgeladen werden, damit er im nächsten Workflow heruntergeladen werden kann, sofern die Hashes unverändert bleiben. Ist dies beim nächsten Workflow der Fall, dann muss der Ordner nur heruntergeladen werden und an die korrekte Adresse im Output Ordner verschoben werden. Die Installation der Abhängigkeiten, das Starten des MySQL - Containers und das Ausführen des Orchestrator - Skripts hat man sich damit erspart.

Die letzte Frage lautet, wo die Ordner mit den berechneten Namen gespeichert und beim nächsten Run wieder heruntergeladen werden sollen. Zum einen kann man Lösungen in GitHub selbst verwenden. Zum einen würde sich eine GitHub Cache - Lösung wieder anbieten, aber tatsächlich sind GitHub Artifacts für das Sichern von Dateien besser geeignet ([12]). Eine andere interessante Lösung kann auch das Nutzen von expliziten Branches nur für die Sicherung der Dateien sein. Das Problem ist hier, dass es manchmal durch bestimmtes Timing zu Problemen beim Pushen kommen kann, da zufällig ein anderer paralleler Workflow in der Zeit zwischen Rebase, Commit und Push den Code verändert hat, wodurch nach verhiertem Push erneut ein Rebase durchgeführt werden muss. Außerdem muss man dafür der GitHub Action Schreibberechtigungen geben. Des Weiteren eignen sich auch Cloud - Speicherlösungen sehr gut, um die Ordner zu speichern und wieder herunterzuladen. Dazu

gehören von Google Cloud Storage (GCS), AWS S3 oder MS Azure Storage, die sich zusammen mit GitHub Artifacts am besten eignen.

## 4 Optimierungen von Datentypen

Das erste Thema, das wir in Bezug auf die Performance - Optimierung von Datenbanken betrachten, sind die unterschiedlichen Datentypen und ihre Effizienzsteigerungen. Bei der Auswahl des korrekten Datentyps gibt es unterschiedliche Faktoren, die vom jeweiligen Typen abhängen. Es gibt aber auch allgemeinere Prinzipien, die auf fast alle Datentypen angewendet werden können.

### 4.1 Allgemeine Faktoren

Bei der Erstellung von Tabellen sollte man folgende Schritte für die Auswahl von Datentypen befolgen. Zunächst muss die allgemeine Klasse der Typen, wie beispielsweise numerisch, Zeichenketten oder zeitbezogen, festgelegt werden. Anschließend sollte der spezifische Typ ausgewählt werden. Für numerische Daten kommen beispielsweise Ganzzahlen wie INT oder Fließkommazahlen wie FLOAT und DOUBLE infrage. Die spezifischen Typen können dieselbe Art von Daten speichern, unterscheiden sich jedoch im Bereich der Werte, die sie speichern können. Auch sind sie unterschiedlich in der Precision (Genauigkeit), die sie erlauben und dem physischen Speicherplatz, den sie entweder auf der Festplatte oder im Arbeitsspeicher benötigen. Einige Datentypen haben auch spezielle Verhaltensweisen und Eigenschaften.

Allgemein gilt für Datentypen, dass kleiner besser ist, weshalb man den kleinstmöglichen Datentypen wählen sollte, den man speichern kann und der die vorhandenen Daten entsprechend repräsentieren kann. Dadurch wird zum einen weniger Speicherplatz (In-Memory und CPU-Cache) in Anspruch genommen, was meistens zu schnelleren Abfragen führt. Zum anderen spricht für die Benutzung von kleinstmöglichen Typen die einfache Typveränderung. Wenn die vorhandenen Daten falsch eingeschätzt wurden und nachträglich ein größerer Datentyp benötigt wird, kann der Typ ohne größere Probleme vergrößert werden. Eine weitere allgemeine Richtlinie ist die Einfachheit von Datentypen. Damit ist beispielsweise gemeint, dass Integer einfach zu verarbeiten ist als Character, weshalb man immer einen Integer wählen sollte, wenn man durch ihn die Daten korrekt abbilden kann. Dies liegt daran, dass weniger CPU-Zyklen benötigt werden, um Operationen auf einfacheren Datentypen zu verarbeiten. Bei dem Beispiel mit Integer und Character liegt dies an den Character Sets und Sortierregeln, die den Character-Vergleich erschweren.



Die letzte allgemeine Regel, die Performancegewinne bringt, ist die Vermeidung von NULL. Viele Tabellen enthalten NULLABLE Spalten, selbst wenn die Anwendung kein NULL (Fehlen eines Wertes) speichern muss, da dies die Standardeinstellung ist. Daher ist es am besten solche Spalten bei der Tabellenerstellung mit dem Identifier NOT NULL zu definieren. Wenn allerdings NULL-Werte gespeichert werden sollen, dann sollte der Identifier nicht genutzt werden. Für MySQL ist es dann schwieriger Abfragen zu optimieren, da dadurch Indizes, Indexstatistiken und Wertevergleiche mehr Speicherplatz benötigen und komplizierter werden. Dies liegt daran, dass indizierte nullable Spalten ein zusätzliches Byte pro Eintrag gebrauchen, was dazu führen kann, dass ein Index mit fester Größe in einen variablen Index umgewandelt wird. Allerdings fällt die Leistungssteigerung, die durch die Änderung von NULL-Spalten in NOT NULL erzielt wird, in der Regel gering aus. Besonders bei Verwendung von Indizes sollte aber darauf geachtet werden.

MySQL unterstützt auch viele Aliase, z.B. INTEGER, BOOL, NUMERIC. Diese Aliase können verwirrend sein, aber sie beeinflussen nicht die Performance. Wenn eine Tabelle mit einem aliasierten Datentyp erstellt wird und die Tabelle mit SHOW CREATE TABLE untersucht wird, fällt auf, dass statt des aliasierten Datentyps der Basistyp angezeigt wird, da der aliasierte Datentyp intern in den Basistyp umgewandelt wurde.

## 4.2 Einzelne Datentypen und weitere Faktoren

Bevor wir untersuchen, ob die eben beschriebene Prinzipien tatsächlich einen Einfluss auf die Performance haben, müssen die speziellen Verhaltensweisen der bekanntesten Datentypen betrachtet werden.

Für numerische Datentypen gibt es die Wahl zwischen Ganzzahlen und Fließkommazahlen. Die spezifischen Typen unterscheiden sich nur in der Anzahl der Bits, die sie speichern können. SMALLINT kann 16 Bits speichern, während INT 32 und BIGINT 64 Bits speichern kann. Dementsprechend verändert sich auch der mögliche Wertebereich der Zahlen, die durch den Speicherplatz abgedeckt sind. Mit den optionalen UNSIGNED - Attribute können keine negativen Werte gespeichert werden können, dafür verdoppelt sich aber die obere Grenze der Positiven. Zeitgleich bleiben der Speicherplatz und die Leistung gleich. Die Berechnung der Wertebereiche für den default, bzw. mit den Signed - Attribut, erfolgt in 4.1 und mit dem Unsigned - Attribut in 4.2.

$$\text{Signed: } -2^{(N-1)} \text{ bis } 2^{(N-1)} - 1 \quad (4.1)$$

$$\text{Unsigned: } 0 \text{ bis } 2^N - 1 \quad (4.2)$$

**Hinweis:**  $N$  entspricht der Anzahl der Bits.

Beispiel für 8 Bits:

- SIGNED: -128 bis 127
- UNSIGNED: 0 bis 255

Eine Breitenangabe wie INT(11) beeinflusst nur die Anzeige und nicht den Wertebereich oder die Speicheranforderungen. Um dies zu beweisen, können wir die folgende Table erstellen.

**Codeblock 4.1:** SQL-Befehl zur Erstellung der Testtabelle

```
1 CREATE TABLE test_int (  
2     int_5 INT(5),  
3     int_11 INT(11)  
4 );
```

Da wir den für die beiden Variablen den Typen INT gewählt haben und wir überprüfen wollen, ob die Breitenangabe einen Einfluss auf die Speicheranforderungen hat, können wir die Grenzen des Wertebereichs für INT einfügen: -2147483648 und 2147483647.

**Codeblock 4.2:** Inserts und Selects für Testtabelle aus 4.1

```
1 INSERT INTO test_int (int_5, int_11) VALUES (2147483647, 2147483647);  
2 INSERT INTO test_int (int_5, int_11) VALUES (-2147483648, -2147483648);  
3  
4 SELECT * FROM test_int;
```

**Tabelle 4.1:** Ergebnis der SQL-Abfrage aus 4.2

int_5	int_11
2147483647	2147483647
-2147483648	-2147483648

Für den maximalen Wert von INT werden 32 Bits benötigt:  $2^{(32-1)} - 1 = 2147483647$ . INT(5) und INT(11) können beide die Grenzwerte von INT speichern, weshalb wir bestätigt haben, dass die Breitenangabe keinen Einfluss auf die Speicheranforderungen hat, ansonsten hätten wir einen Fehler bei der Einfügung der Werte bekommen.

Ein spezifischer Typ für eine Festkommazahl ist DECIMAL, die auch für die Speicherung von Ganzzahlen geeignet ist. Außerdem kann man bei einer Festkommazahl auch die Genauigkeit angeben, da die maximale Anzahl der Ziffern vor und nach dem Dezimalpunkt definiert werden. DECIMAL(18, 9) beispielsweise speichert neun Ziffern vor und nach dem Dezimalpunkt und benötigt dafür 9 Bytes Speicherplatz. DECIMAL speichert Zahlen in einer binären

Zeichenkette (binary string) mit neun Ziffern pro vier Bytes und unterstützt bis zu 65 Ziffern insgesamt.

Zu den Fließkommazahlen gehören die FLOAT- und DOUBLE-Typen, die die standardmäßige Gleitkomma-Arithmetik verwenden und für ungefähre Berechnungen optimiert sind. FLOAT benötigt 4 Bytes, während DOUBLE 8 Bytes Speicherplatz beansprucht und eine höhere Präzision sowie einen größeren Wertebereich bietet. Die Gleitkomma-Arithmetik ist aufgrund der nativen Verarbeitung durch die CPU deutlich schneller als die präzise Berechnung mit DECIMAL, bringt jedoch einen gewissen Präzisionsverlust mit sich. Alternativ kann auch BIGINT genutzt werden, um sowohl die Ungenauigkeit von Gleitkomma-Speicherungen als auch die höheren Kosten der DECIMAL-Arithmetik zu vermeiden.

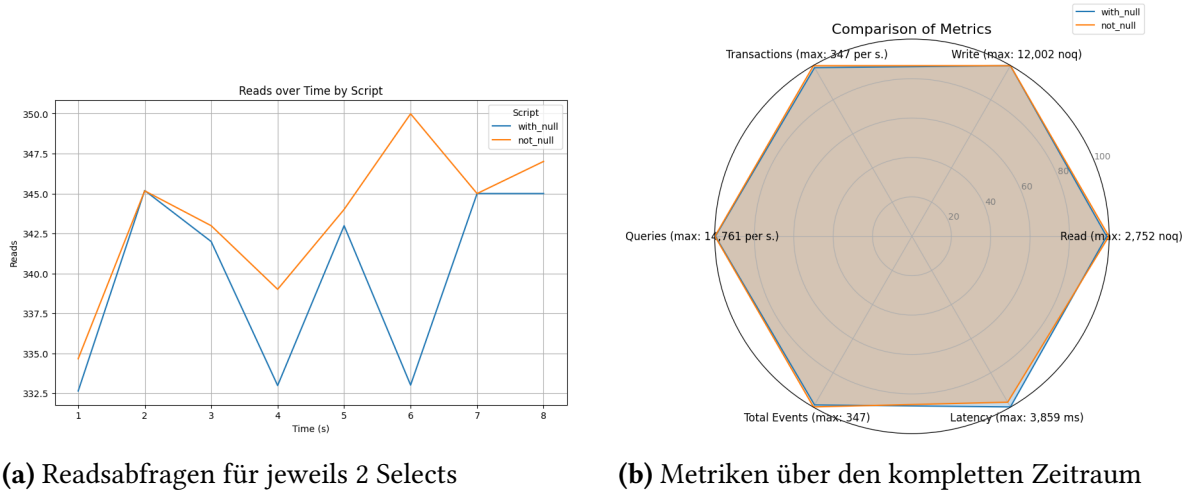
Die beiden Haupttypen für Zeichenketten sind VARCHAR und CHAR. VARCHAR speichert die Zeichenfolgen mit variabler Länge und benötigt daher weniger Speicherplatz als Typen mit fester Länge, da nur so viel Platz verwendet wird, wie tatsächlich benötigt wird. Zusätzlich werden ein oder zwei Bytes für die Speicherung der Länge der Zeichenfolge verwendet (1 Byte für < 255 Bytes Zeichenfolge). Durch diese effiziente Speichernutzung ist VARCHAR der am häufigsten verwendete Datentyp für Zeichenketten, aber es hat auch Nachteile, da Aktualisierungen an den Werten zu wachsenden Zeilen und damit auch zusätzliche Verarbeitung der Speicher - Engine erfordern kann. Und obwohl die Speicherung von hello in VARCHAR(5) oder VARCHAR(200) gleich viel Speicherplatz benötigt, kann es trotzdem ineffizienter für Sortierungen oder Operationen auf temporären Tabellen sein. Deshalb sollte trotzdem immer so viel Platz reserviert werden, wie tatsächlich benötigt wird.

CHAR hingegen hat eine feste Länge und MySQL reserviert immer auch den nicht gebrauchten Platz für die angegebene Anzahl an Zeichen. Daher ist CHAR ideal für sehr kurze Strings oder Werte, die alle nahezu gleich lang sind, da VARCHAR(1) zwei Bytes aufgrund des Längen - Bytes benötigt, CHAR(1) hingegen auch. Außerdem ändert sich bei CHAR die Speicherstruktur bei Aktualisierungen nicht, weshalb dieser Datentyp besser geeignet ist, wenn die Daten häufig verändert werden. Hingegen VARCHAR eignet sich besonders, wenn die maximale Länge einer Spalte deutlich größer ist als die durchschnittliche Länge der gespeicherten Werte.

DATETIME und TIMESTAMP können dieselbe Art von Daten speichern und beide haben dabei eine Genauigkeit von einer Sekunde. TIMESTAMP benötigt aber nur halb so viel Speicherplatz, ist zeitzonenbewusst und verfügt über spezielle Auto-Update-Funktionen. Allerdings hat TIMESTAMP einen viel kleineren Bereich an erlaubten Werten und manchmal können seine speziellen Fähigkeiten ein Nachteil sein.

## 4.3 Analyse der Benchmarks

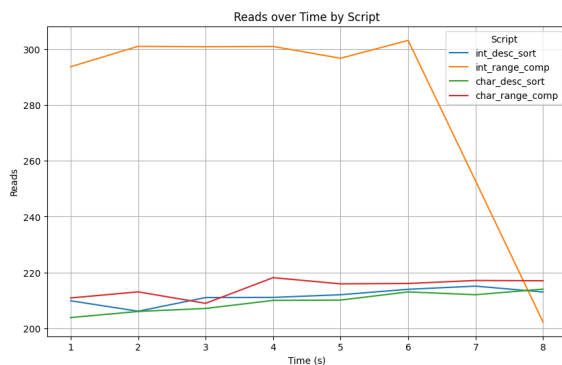
Der erste Leitsatz, den wir untersuchen, besagt, dass Spalten nach Möglichkeit als NOT NULL deklariert werden sollten. Für den Nachweis benutzen wir erneut die Kundentabelle (2.5). Diesmal erstellen wir eine Tabelle, bei der das Attribut NOT NULL für alle Spalten deklariert wird, sowie eine Tabelle ohne dieses Attribut. Wenn das Attribut nicht deklariert wird, dann können u.a. auch NULL-Werte in die Tabelle eingefügt werden.



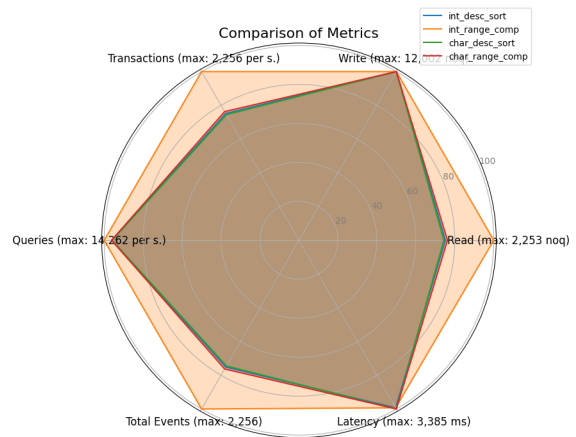
**Abbildung 4.1:** Vergleich von NULL und NOT NULL

In der erstellten Grafik (siehe Abbildung 4.1a) sind die beiden Resultate der Select-Befehle zu sehen, die sowohl einfache WHERE-Klauseln als auch Count- und Group-By-Befehle enthalten. Anhand der Grafik lässt sich erkennen, dass die Werte für NOT NULL höher liegen als für NULL, bzw WITH NULL. Höhere Werte bedeuten, dass mehr Abfragen pro Sekunde durchgeführt werden können, was auf eine bessere Performance hindeutet. Deshalb lässt es sich sagen, dass NOT NULL besser performt als NULL, aber wenn man auf die Y-Achse schaut, fällt auf, dass die Werte nicht so weit auseinanderliegen, sondern der Unterschied nur marginal ist. Daher sollten beim Datenbankentwurf Entscheidungen nicht aus Performancegründen getroffen werden, sondern aus Gründen der Datenintegrität und -konsistenz.

Um zu zeigen, dass man bei der Wahl zwischen unterschiedlichen Datentypen, den simpleren wählen sollte, haben wir erneut die Kundentabelle (2.5) benutzt. Für diesen Vergleich haben wir den Datentyp des Schlüsselattributs der Tabelle geändert, zunächst INT und anschließend CHAR. An den Ergebnissen fällt auf, dass die Schreibbefehle bei beiden Skripten etwa gleich schnell sind, aber bei den Abfragen gibt es deutliche Unterschiede. Hier zeigt sich, dass bei Wertevergleichen INT deutlich schneller ist als CHAR (etwa 50%). Bei der Sortierung ist die Reihenfolge gleich, jedoch fallen die Abstände deutlich geringer aus (4.2a).



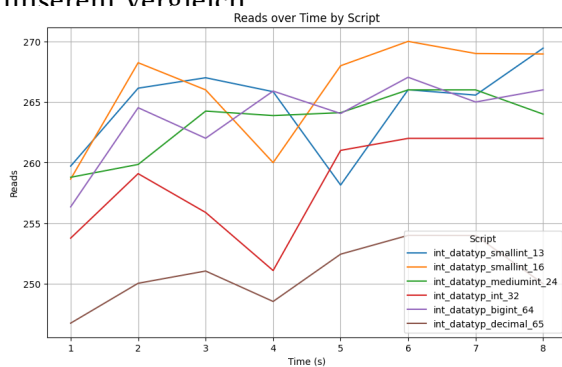
(a) Vergleich für jeweils 2 unters. Abfragen



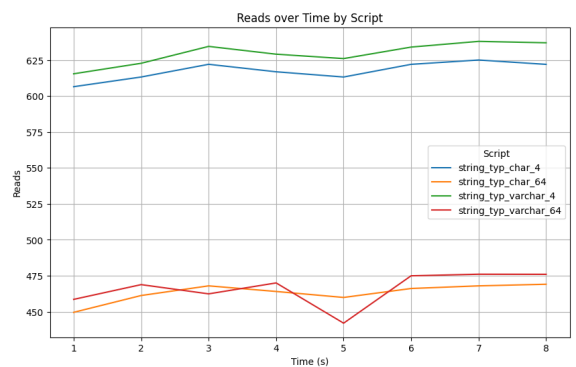
(b) Kennzahlen über den gesamten Zeitraum

### Abbildung 4.2: Vergleich von INT und CHAR

Als Letztes wollten wir unterschiedliche Datentypen vergleichen. Dazu haben wir die gleiche Tabelle wie beim Vergleich von INT und CHAR verwendet, jedoch diesmal unterschiedliche numerische oder Zeichenketten-Typen für den Primärschlüssel benutzt. Beim Vergleich der numerischen Typen zeigt sich, dass DECIMAL mit deutlichem Abstand am langsamsten ist (Abbildung 4.3a). Danach folgt, wie vermutet, der nächstgrößere Datentyp BIGINT. Auffällig ist, dass der Unterschied zwischen INT, MEDIUMINT und SMALLINT kleiner ist als erwartet. Dies wird vermutlich aber daran liegen, dass wir die Abfragen nur auf eine Tabelle mit wenigen tausend Datensätzen ausgeführt haben. In der Produktion mit Hunderttausenden oder Millionen von Datensätzen ist anzunehmen, dass die Unterschiede zwischen den Typen größer wären als in unserem Vergleich.



(a) Unterschiedliche numerische Typen

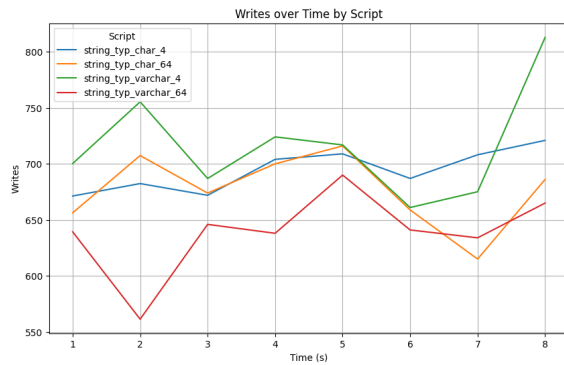


(b) Unterschiedliche Zeichenketten-Typen

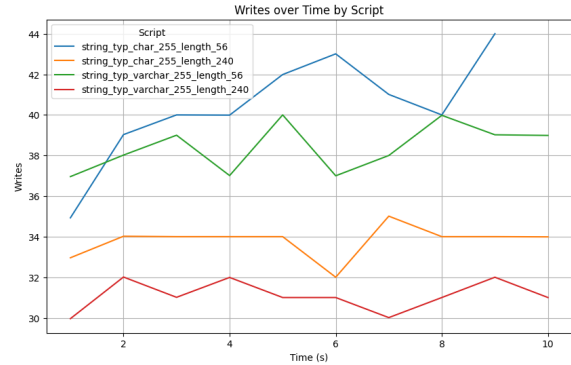
### Abbildung 4.3: Vergleich von unterschiedlichen Select-Abfragen

Beim Vergleich zwischen den beiden Zeichenketten-Typen CHAR und VARCHAR ist unabhängig von der Länge zu erkennen, dass VARCHAR effizienter ist als CHAR (4.3b). Im ersten Vergleich wurde jeweils eine Länge von 4 Stellen verwendet und beim zweiten Vergleich eine Länge von 64 Stellen. Bei beiden untersuchten Längen ist VARCHAR schneller als CHAR. Wenn man sich jedoch die Performance beim Einfügen von Werten anschaut, fällt auf, dass die Unterschiede

eher gering sind und es gibt auch stärkere Schwankungen bei den Werten (siehe Abbildung 4.4a).



(a) Bei gleicher Länge der Zeichen



(b) Bei unterschiedlichem Befüllungsgrad

**Abbildung 4.4:** Vergleiche von unterschiedlichen Write-Abfragen

Als letzten Vergleich haben beide Zeichenketten-Typen mit der Länge von 255 Stellen definiert, aber dafür mit unterschiedlich vielen Stellen befüllt. Anschließend haben wir bei beiden Tabellen die Werte aktualisiert und wenige Stellen bei der Namen-Spalte zufällig hinzugefügt. Wenn man dies tut, dann fällt auf, dass CHAR schneller ist als VARCHAR (4.4b). Zusätzlich wird der Unterschied zwischen den beiden Typen größer, je mehr Stellen befüllt werden. Damit haben wir gezeigt, dass die Vorteile von CHAR insbesondere bei der Aktualisierung von Werten liegen, während VARCHAR bei der Selektion von Werten besser abschneidet. Der Grund hierfür ist, dass CHAR die verbleibenden Stellen mit Leerzeichen auffüllt, was zu einem höheren Speicherbedarf führt.

# 5 Indexierung und Einfluss auf die Performance

## 5.1 Grundlagen der Indexierung

Das folgende Thema befasst sich mit der Indexierung und den damit verbundenen Performance-Optimierungen, die näher erläutert werden. Zunächst betrachten wir die Grundlagen der Indexierung, anschließend die verschiedenen Arten von Indizes und schließlich deren Auswirkungen auf die Performance.

Indizes (oder auch Indexes) sind Datenstrukturen, die von Speicher-Engines (engl. storage engines) verwendet werden, um unter anderem Zeilen schneller zu finden. Sie haben einen großen Einfluss auf die Performance der Datenbank und werden umso wichtiger, je größer die Datenbank wird. Weniger ausgelastete Datenbanken können ohne ordnungsgemäße Indizes gut funktionieren, aber die Leistung kann rapide sinken, wenn die Datenmenge wächst. Wenn ein solches Problem auftritt, ist die Index-Optimierung oft der effektivste Weg, die Abfrageleistung zu verbessern. Um wirklich optimale Indizes zu erstellen, ist es häufig notwendig, Abfragen umzuschreiben. Wie genau Indizes erstellt werden müssen, wird im weiteren Verlauf der Arbeit betrachtet.

Um die Funktionsweise eines Indexes zu verdeutlichen, betrachten wir ein Beispiel aus einem wissenschaftlichen Fachbuch. Am Ende solcher Bücher gibt es meist ein Stichwortverzeichnis oder Register. Dieses Register besteht aus einer alphabetisch geordneten Liste von Begriffen, Themen und Stichworten. Möchte man einen Begriff nachschlagen, sucht man ihn in der Liste und erhält die Seitenzahlen, auf denen er vorkommt. In MySQL verwendet die Storage-Engine Indizes auf ähnliche Weise. Sie durchsucht die Datenstruktur des Indexes nach einem Wert. Wird ein Treffer gefunden, kann die Engine die Zeile ermitteln, die den Treffer enthält. Betrachten wir dazu folgendes Beispiel:

### Codeblock 5.1: Variationen

```
1 SELECT name FROM customer WHERE cust_id = 7;
```

Es gibt einen Index auf der Spalte `cust_id`, sodass MySQL diesen Index nutzt, um Zeilen zu finden, deren `cust_id` gleich 7 ist. Mit anderen Worten wird eine Suche innerhalb der Indexwerte durchgeführt, und alle entsprechenden Zeilen werden zurückgegeben.

Ein Index kann Werte aus einer oder mehreren Spalten einer Tabelle enthalten. Bei mehreren Spalten ist die Reihenfolge der Spalten im Index entscheidend, da MySQL nur effizient auf ein linkes Präfix des Indexes zugreifen kann. Ein Index über zwei Spalten ist nicht gleichbedeutend mit zwei separaten einspaltigen Indizes. Es gibt verschiedene Typen von Indizes, die jeweils für unterschiedliche Zwecke optimiert sind und die im nächsten Abschnitt behandelt werden.

## 5.2 B-Baum-Index

Indizes werden auf der Ebene der Storage-Engine und nicht auf der Serverebene implementiert. Daher sind sie nicht standardisiert und unterscheiden sich je nach Engine. Zudem unterstützen nicht alle Engines alle Index-Typen. Eine Storage-Engine ist eine Kernkomponente eines Datenbankmanagementsystems (DBMS), die für die Speicherung und Verwaltung der Daten zuständig ist. Sie entscheidet, wie Daten physisch organisiert, gespeichert und abgerufen werden. Verschiedene Storage-Engines unterscheiden sich in ihrer Indexfunktionalität sowie in der Unterstützung von Transaktionen und Sperrmechanismen.

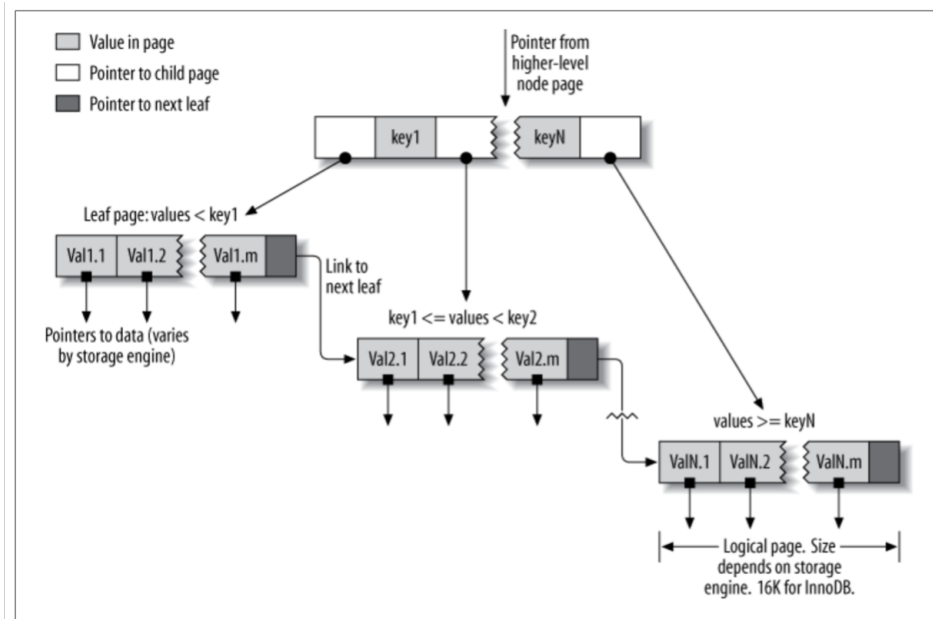
Der erste zu betrachtende Indextyp ist der B-Baum-Index (engl. B-Tree Index), der auf einer speziellen Baum-Datenstruktur basiert. Diese Struktur wird von den meisten MySQL-Storage-Engines unterstützt. Die Implementierung und Nutzung des B-Baum-Indexes kann jedoch je nach verwendeter Storage-Engine variieren.

Das Grundprinzip eines B-Baums ist, dass alle Werte in einer bestimmten Reihenfolge gespeichert werden und jede Blattseite den gleichen Abstand zum Wurzelknoten hat. Ein B-Baum-Index beschleunigt den Datenzugriff, da die Storage-Engine nicht die gesamte Tabelle durchsuchen muss, um die gewünschten Daten zu finden. Stattdessen beginnt die Suche beim Wurzelknoten.

Die Slots im Wurzelknoten enthalten Zeiger auf Kindknoten, und die Storage-Engine folgt diesen Zeigern. Der richtige Zeiger wird durch Vergleich der Werte in den Knoten-Seiten (engl. node pages) ermittelt, die die oberen und unteren Grenzen der Werte in den Kindknoten definieren. Letztlich stellt die Storage-Engine fest, ob der gewünschte Wert existiert, oder sie erreicht erfolgreich eine Blattseite (engl. leaf page).

Blattseiten sind besonders, da sie Zeiger auf die indexierten Daten enthalten, anstatt auf andere Seiten zu verweisen. Zwischen dem Wurzelknoten und den Blattseiten können viele Ebenen von Knoten-Seiten existieren. Die Tiefe des Baumes hängt von der Größe der Tabelle ab.





**Abbildung 5.1:** Darstellung des binären Baums mit Knoten und Blättern

Außerdem speichern B-Bäume die indexierten Spalten in einer festgelegten Reihenfolge, was sie besonders nützlich für die Suche nach Datenbereichen macht. Beispielsweise kann ein Index auf einem Textfeld (z.B. vom Typ VARCHAR) effizient alle Namen finden, die mit „K“ beginnen, da die Werte in alphabetischer Reihenfolge gespeichert sind.

Der Index sortiert die Werte entsprechend der Reihenfolge der in der CREATE TABLE-Anweisung angegebenen Spalten, beispielsweise des Primärschlüssels (`last_name`, `first_name`, `b_day`). B-Baum-Indizes eignen sich gut für Suchen mit dem vollständigen Schlüsselwert (engl. full key value), einem Schlüsselbereich (engl. key range) oder einem Schlüsselpräfix (engl. full key prefix). Beim Schlüsselpräfix ist dies jedoch nur der Fall, wenn die Suche das linkeste Präfix des Indexes verwendet.

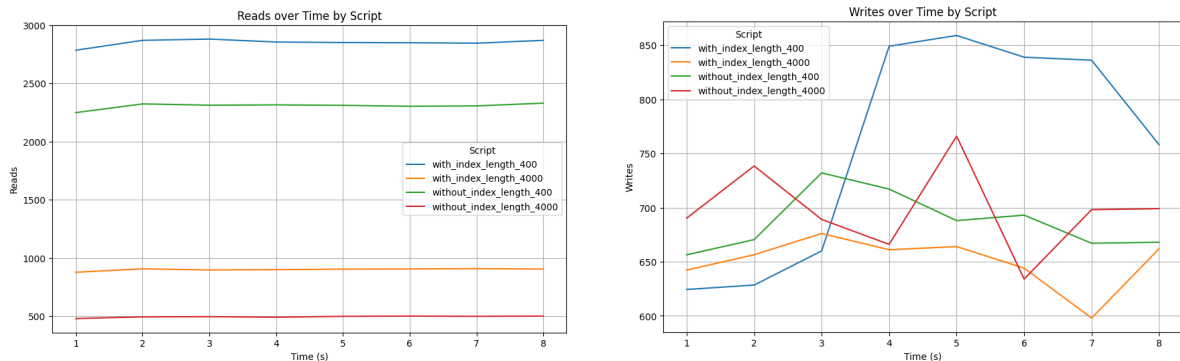
Als nächstes betrachten wir die möglichen Abfragen, bei denen B-Baum-Indizes besonders hilfreich sind, um ein besseres Verständnis für ihre optimale Nutzung zu erlangen. Eine Übereinstimmung mit dem vollständigen Schlüsselwert liefert Werte für alle Spalten im Index. Eine beispielhafte Abfrage wäre die Suche nach allen Einträgen für Max Mustermann, geboren am 2000-01-01, wenn der Schlüssel aus Nachname, Vorname und Geburtsdatum besteht. Für diesen Index sind auch Abfragen nützlich, die nur mit dem linken Präfix übereinstimmen, beispielsweise die Suche nach „Mustermann“. Eine weitere Möglichkeit ist die Übereinstimmung mit einem Spaltenpräfix, also dem ersten Teil eines Spaltenwerts, etwa alle Nachnamen, die mit „M“ beginnen. Ebenso effizient ist der Index bei der Übereinstimmung mit einem Wertebereich, z.B. Nachnamen zwischen „Mustermann“ und „Müller“.

Ein B-Baum-Index kann auch genutzt werden, um Abfragen effizient zu unterstützen, bei denen eine Spalte exakt und eine andere innerhalb eines Wertebereichs abgefragt wird. Beispielsweise könnte dies eine exakte Übereinstimmung mit dem Nachnamen „Mustermann“ und eine Bereichsabfrage für Vornamen, die mit „Ma“ beginnen, umfassen. Der letzte Anwendungsfall sind Abfragen, die nur den Index verwenden und nicht die gespeicherten Zeilen, etwa wenn alle benötigten Daten im Index enthalten sind.

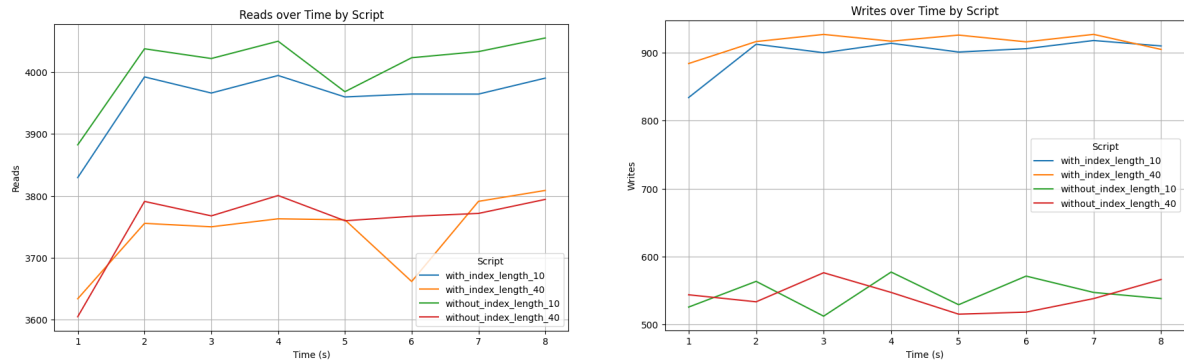
Ein weiterer Vorteil von B-Baum-Indizes ist, dass sie aufgrund der sortierten Baumstruktur nicht nur Abfragen, sondern auch ORDER BY-Bedingungen effizient unterstützen können. Wenn ein B-Baum für die Suche genutzt werden kann, kann er auch für die Sortierung der Ergebnisse verwendet werden.

Es gibt jedoch Einschränkungen von B-Baum-Indizes, die dazu führen, dass andere Indextypen für bestimmte Szenarien besser geeignet sind. Eine Einschränkung ist, dass die Suche nicht am linken Ende des Indexes beginnen kann. Beispielsweise ist ein Index, der aus Nachname, Vorname und Geburtsdatum besteht, nicht geeignet, um alle Personen zu finden, die vor dem Jahr 2000 geboren wurden, ohne dass der Nachname und Vorname ebenfalls spezifiziert werden.

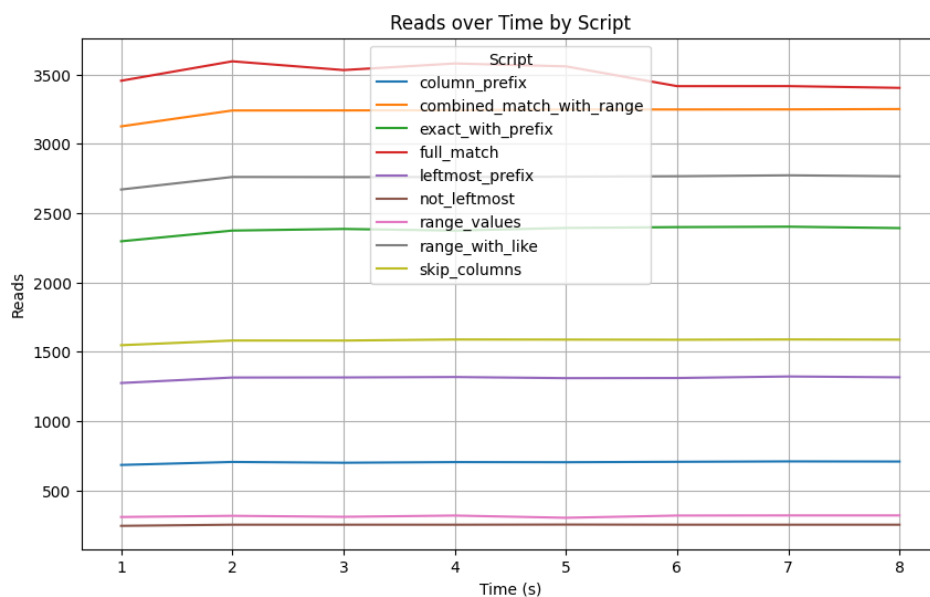
Für optimale Leistung sollten Indizes mit den gleichen Spalten, jedoch in unterschiedlicher Reihenfolge erstellt werden, um die häufigsten Abfragen zu optimieren. Eine Analyse der am häufigsten verwendeten Abfragen kann dabei helfen zu entscheiden, ob zusätzliche Indizes erforderlich sind.



**Abbildung 5.2:** Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts) mit 500 bzw. 5000 Zeilen



**Abbildung 5.3:** Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts) mit 10 bzw. 100 Zeilen dar



**Abbildung 5.4:** Grafik visualisiert die Unterschiede von verschiedenen Select - Queries auf dieselben Daten. Je nach Geschwindigkeit greift der Index besser oder nicht

## 5.3 Hash - Index

Ein weiterer Indextyp, den wir betrachten, ist der Hash-Index. Dieser basiert auf einer Hash-Tabelle und ist daher nur für exakte Suchanfragen geeignet, die alle Spalten im Index verwenden. Die Funktionsweise der Storage-Engine lässt sich wie folgt beschreiben: Für jede Zeile wird mithilfe einer Hash-Funktion ein Hash-Wert der indexierten Spalte berechnet. Der Hash-Wert (engl. *hash code*) ist eine kleine Zahl, die sich in der Regel von den Hash-Werten anderer Zeilen mit unterschiedlichen Schlüsselwerten unterscheidet.

In MySQL unterstützt nur die Memory-Storage-Engine explizite Hash-Indizes. Der bereits besprochene Standard-Indextyp für Memory-Tabellen, der B-Baum-Index, ist jedoch ebenfalls möglich. Außerdem unterstützt die Memory-Engine keine eindeutigen Hash-Indizes. Das bedeutet, wenn mehrere Werte denselben Hash-Wert besitzen, speichert der Index die Zeiger auf die Zeilen (engl. *row pointers*) in demselben Hash-Tabelleneintrag, typischerweise mithilfe einer verketteten Liste (z.B. einer *Linked List*). Im Gegensatz dazu stellen eindeutige Hash-Indizes sicher, dass für jeden Hash-Wert nur ein einziger Eintrag existiert. Bei Konflikten wird ein Mechanismus wie die *Open Addressing*-Strategie (z.B. *Linear Probing* oder *Quadratic Probing*) eingesetzt, um Konflikte zu lösen und den Speicherplatz effizient zu verwalten. Hierbei wird versucht, Konflikte direkt innerhalb der Hash-Tabelle zu bewältigen, anstatt auf zusätzliche Datenstrukturen wie verkettete Listen zurückzugreifen.

Um die Berechnung der Hash-Funktion genauer zu erläutern, folgt ein Beispiel:

### Codeblock 5.2: Variationen

```
1 SELECT lname FROM testhash WHERE fname = 'Peter';
```

Zunächst berechnet MySQL den Hash-Wert für 'Peter' und verwendet diesen, um den entsprechenden Zeiger im Index zu finden. Angenommen, die Hash-Funktion liefert für 'Peter' den Wert **7654**. MySQL sucht nun im Index an der Position 7654 und findet einen Zeiger auf Zeile 3. Im letzten Schritt wird der Wert in Zeile 3 mit 'Peter' verglichen, um sicherzustellen, dass es sich um die richtige Zeile handelt. Da die Indizes nur kompakte Hash-Werte speichern, sind Hash-Indizes äußerst platzsparend, und Suchvorgänge erfolgen in hoher Geschwindigkeit.

Ähnlich wie der B-Baum-Index hat auch der Hash-Index einige Einschränkungen, auf die wir nun eingehen:

Da der Index nur Hash-Werte und Zeiger auf Zeilen (engl. *row pointers*) enthält, jedoch nicht die Werte selbst, kann MySQL den Index nicht verwenden, um das Einlesen der Zeilen zu vermeiden. Da der Zugriff auf die in den Speicher geladenen Zeilen jedoch sehr schnell ist, wird die Leistung dadurch nicht wesentlich beeinträchtigt.

Ein wesentlicher Nachteil von Hash-Indizes ist, dass sie nicht für Sortierungen verwendet werden können, da die Werte nicht in einer geordneten Reihenfolge gespeichert sind. Im Gegensatz dazu können B-Baum-Indizes Sortierungen unterstützen, wenn sie entsprechend erstellt und genutzt werden.

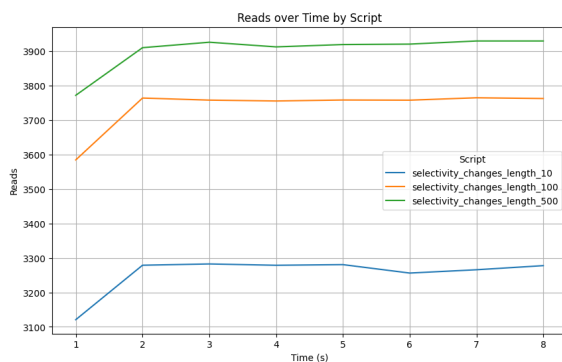
Darüber hinaus ermöglichen Hash-Indizes keine partiellen Schlüsselübereinstimmungen (engl. *partial key matching*). Da der Hash-Wert aus dem gesamten indexierten Wert berechnet wird, hilft ein Hash-Index beispielsweise nicht, wenn ein Index aus den Spalten (A, B) besteht und die WHERE-Klausel nur auf A verweist.

Ein weiterer Nachteil besteht darin, dass Hash-Indizes keine Bereichsabfragen (engl. *range queries*) unterstützen. Sie eignen sich lediglich für Gleichheitsvergleiche, wie die Operatoren = (gleich), <=> (null-sicher gleich) und IN().

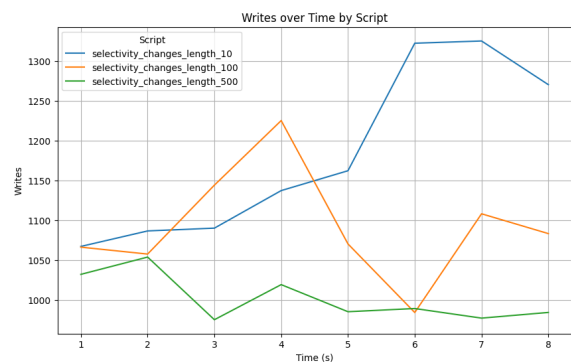
Obwohl Hash-Indizes sehr performant sind, können Hash-Kollisionen ihre Leistung beeinträchtigen. Wenn viele Werte denselben Hash-Wert aufweisen, muss die Storage-Engine jeden Zeiger in der verketteten Liste durchlaufen und die entsprechenden Werte mit dem Suchwert vergleichen, um die richtige(n) Zeile(n) zu finden.

Auch Index-Wartungsoperationen können bei vielen Kollisionen langsamer werden. Wenn beispielsweise ein Index auf einer Spalte mit sehr geringer Selektivität erstellt wird und eine Zeile gelöscht werden soll, kann das Finden des entsprechenden Zeigers im Index sehr aufwendig sein, was auch das Löschen der Zeile verzögert.

Einige Speicher-Engines, wie beispielsweise InnoDB, können zudem erkennen, wenn bestimmte Index-Werte besonders häufig verwendet werden, und automatisch einen Hash-Index für diese Werte im Speicher (engl. *memory*) erstellen, der zusätzlich zu den vorhandenen B-Baum-Indizes genutzt wird.

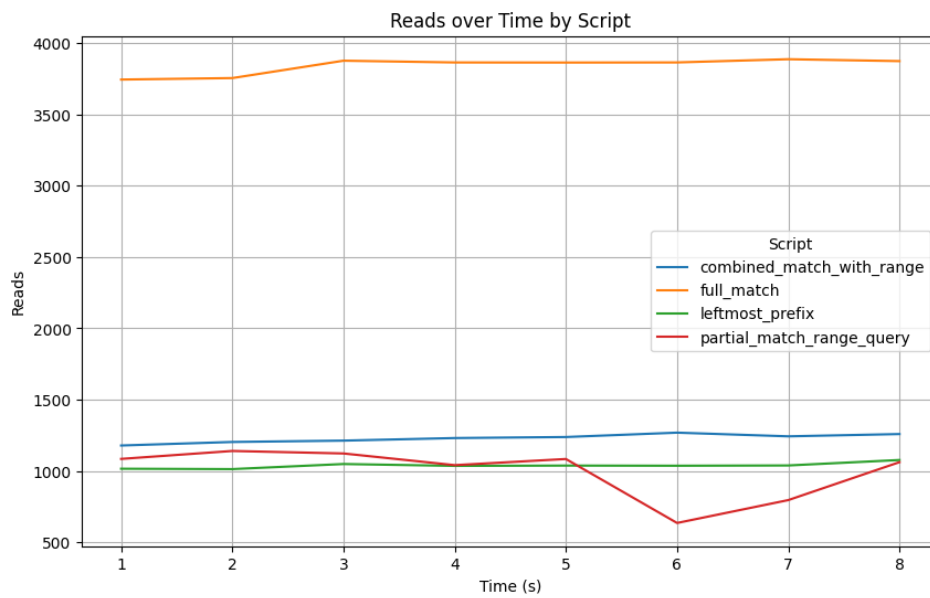


(a) Unterschiede von Readsabfragen



(b) Unterschiede von Schreibbefehlen

**Abbildung 5.5:** Vergleich der Auswirkungen von Hashkollisionen auf Lese- und Schreibvorgänge in Abhängigkeit von der Anzahl an Hashkollisionen.



**Abbildung 5.6:** Grafik visualisiert die Unterschiede von verschiedenen Select - Queries auf dieselben Daten. Je nach Geschwindigkeit greift der Index besser oder nicht

# Literatur

- [1] M. Vogel. „EDV-Lexikon, Bottleneck.“ (2009), Adresse: <https://martinvogel.de/lexikon/bottleneck.html> (besucht am 28. 10. 2024).
- [2] R. Germany. „Was ist ein Snapshot Backup?“ (2024), Adresse: <https://www.rubrik.com/de/insights/what-is-a-snapshot-backup> (besucht am 28. 10. 2024).
- [3] N. Reimers. „Virtuelle, dezidierte und Cloud-Server, MySQL-Benchmark mittels sysbench.“ (2017), Adresse: <https://www.webhosterwissen.de/know-how/server/mysql-benchmark-mittels-sysbench/> (besucht am 28. 10. 2024).
- [4] A. Kopytov. „Sysbench Github Repository.“ (2024), Adresse: <https://github.com/akopytov/sysbench> (besucht am 28. 10. 2024).
- [5] D. E. Difallah, A. Pavlo, C. Curino und P. Cudré-Mauroux, „OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases,“ *PVLDB*, Jg. 7, Nr. 4, S. 277–288, 2013. Adresse: <http://www.vldb.org/pvldb/vol7/p277-difallah.pdf>.
- [6] Shopify. „Mybench Github Repository.“ (2024), Adresse: <https://github.com/Shopify/mybench> (besucht am 28. 10. 2024).
- [7] Shopify. „What is mybench?“ (2022), Adresse: <https://shopify.github.io/mybench/introduction.html> (besucht am 28. 10. 2024).
- [8] Shopify. „Detailed design documentation.“ (2022), Adresse: <https://shopify.github.io/mybench/detailed-design-doc.html#live-monitoring-user-interface> (besucht am 28. 10. 2024).
- [9] T. Williams, C. Kelley und many others, *Gnuplot 4.4: an interactive plotting program*, <http://gnuplot.sourceforge.net/>, März 2010.
- [10] T. pandas development team, *pandas-dev/pandas: Pandas*, Version latest, Feb. 2020. DOI: [10.5281/zenodo.3509134](https://doi.org/10.5281/zenodo.3509134). Adresse: <https://doi.org/10.5281/zenodo.3509134>.
- [11] J. D. Hunter, „Matplotlib: A 2D graphics environment,“ *Computing in Science & Engineering*, Jg. 9, Nr. 3, S. 90–95, 2007. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- [12] GitHub. „Caching dependencies to speed up workflows.“ (2025), Adresse: <https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows#comparing-artifacts-and-dependency-caching> (besucht am 07. 01. 2025).

# Anhang

Hier beginnt der Anhang. Siehe die Anmerkungen zur Sinnhaftigkeit eines Anhangs in Abschnitt

Der Anhang kann wie das eigentliche Dokument in Kapitel und Abschnitte unterteilt werden. Der Befehl `\appendix` sorgt im Wesentlichen nur für eine andere Nummerierung.



# **Eigenständigkeitserklärung**

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

## **Performance - Optimierung von Datenbanken**

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

Hamburg, 21. Dezember 1940