HAW HAMBURG

**BACHELORARBEIT**

# Entwurf 4: Performance – Optimierung von Datenbanken

vorgelegt am 04. Februar 2025

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

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

## 1.1 Einführung in Benchmarks

TODO

## 1.2 Measures

TODO

# 2 Grundlagen

In dem diesem Kapitel betrachten wir die Grundlagen der Bachelorarbeit, die in den späteren Kapiteln für die Durchführung der Benchmarktests und Analysen erforderlich sind. Zunächst werden die Auswahl der einzelnen Tools, ihre Funktionsweise und die jeweiligen Vor- und Nachteile evaluiert. Anschließend werden die einzelnen Schritte dargestellt, damit man die Tools korrekt verwenden kann. Besonders beim Benchmark-Tool untersuchen wir die verschiedenen Argumente, die übergeben werden können, und zeigen anhand eines kurzen Beispiels, wie die Resultate dieses Tools aussehen könnten. Danach betrachten wir ein komplexeres Beispiel, welches wir zu großen Teilen bei späteren Tests wiederverwenden können. Zu guter Letzt zeigen wir, wie GitHub Actions funktionieren und uns bei den Benchmarktests Aufwand ersparen können und wie wir die Workflows zeitlich und resourcentechnisch optimieren können.

## 2.1 Auswahl der Tools

Die Grundlage für diese Bachelorarbeit ist das Verhalten der MySQL – Datenbank [1] 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 veranschaulich 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** [2] 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** [3] oder **mybench** [4]. 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 hervornehmen. In dem Kapitel (2.3) 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 [5]. 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 Ergebnisraten 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 [6]. 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 [7] 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 [8] und matplotlib.pyplot [9] 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, 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 angegebend 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_only). 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:

```
sysbench oltp_read_write \
  --db-driver=mysql \
  --mysql-user=YOUR_USER \
  --mysql-password=YOUR_PASSWORD \
  --mysql-db="sbtest" \
  --time=10 \
  --report-interval=1 \
  run
```

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

```
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

```
# Define necessary variables: DB_HOST, DB_USER, DB_PASS, DB_NAME, TABLES, TABLE_SIZE, DURATION,
# Example: RAW_RESULTS_FILE="output/sysbench.log"

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"

# Prepare the database
echo "Preparing the database...";
sysbench oltp_read_write $SYSBENCH_OPTS prepare >> "$RAW_RESULTS_FILE" 2>&1
echo "Database prepared."

# Run the benchmark
echo "Running benchmark...";
sysbench oltp_read_write $SYSBENCH_OPTS --time=$DURATION --threads=1 --report-interval=1 run >> "$RAW_RESULTS_FILE"
     2>&1
echo "Benchmark complete."

# Cleanup the database
echo "Cleaning up...";
sysbench oltp_read_write $SYSBENCH_OPTS cleanup >> "$RAW_RESULTS_FILE" 2>&1
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 entsprechende 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

```
# Define necessary variables
RAW_RESULTS_FILE="output/sysbench.log"
OUTPUT_FILE="output/sysbench_output.csv"

echo "Script,Time (s),Threads,TPS,QPS,Reads,Writes,Other,Latency (ms;95%),ErrPs,ReconnPs" > "$OUTPUT_FILE"
grep '^\[ ' $RAW_RESULTS_FILE | while read -r line; do
    time=$(echo $line | awk '{print $2}' | sed 's/s//')
    threads=$(echo $line | awk -F 'thds: ' '{print $2}' | awk '{print $1}')
    tps=$(echo $line | awk -F 'tps: ' '{print $2}' | awk '{print $1}')
    qps=$(echo $line | awk -F 'qps: ' '{print $2}' | awk '{print $1}')
    read_write_other=$(echo $line | sed -E 's/.*\(r\/w\/o: ([0-9.]+)\/([0-9.]+)\/([0-9.]+)\).*/\1,\2,\3/')
    reads=$(echo $read_write_other | cut -d',' -f1)
    writes=$(echo $read_write_other | cut -d',' -f2)
    other=$(echo $read_write_other | cut -d',' -f3)
    latency=$(echo $line | awk -F 'lat \\(ms,95%\\): ' '{print $2}' | awk '{print $1}')
    err_per_sec=$(echo $line | awk -F 'err/s: ' '{print $2}' | awk '{print $1}')
    reconn_per_sec=$(echo $line | awk -F 'reconn/s: ' '{print $2}' | awk '{print $1}')

    echo "demo,$time,$threads,$tps,$qps,$reads,$writes,$other,$latency,$err_per_sec,$reconn_per_sec" >> "
     $OUTPUT_FILE"
done

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-Script, 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

```
OUTPUT_FILE="$OUTPUT_DIR/sysbench_output.csv"

# Gnuplot
GNUPLOT_SCRIPT="YOUR_PATH_TO_PROJECT/plot_sysbench.gp"
gnuplot $GNUPLOT_SCRIPT
echo "Plots generated with gnuplot"

# Python with Library Pandas
PYTHON_SCRIPT="YOUR_PATH_TO_PROJECT/generatePlot.py"
python3 "$PYTHON_SCRIPT" "$OUTPUT_FILE"
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.

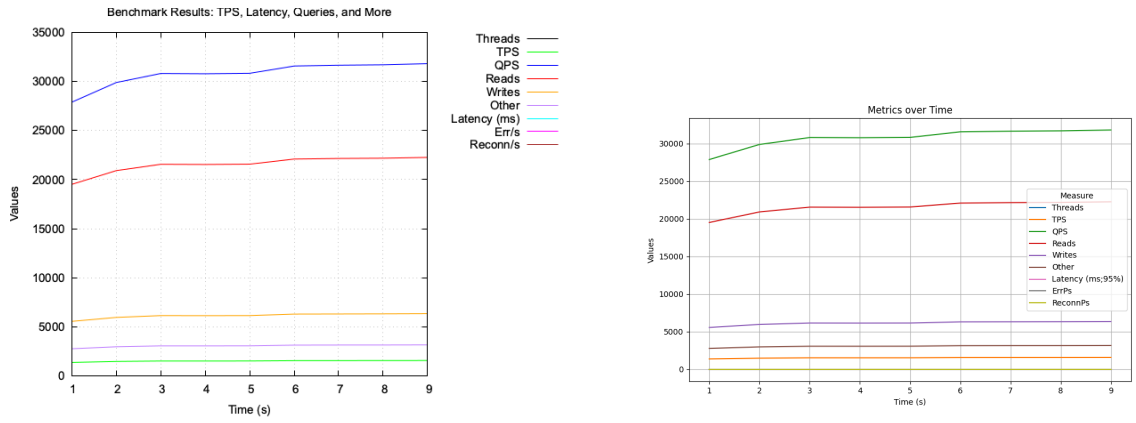



**Abbildung 2.1:** Die Grafik zeigt die erstellten Graphen mit Gnuplot (links) und Pandas (rechts)

- **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 Projektaufbau mit Beispiel

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

```
CREATE TABLE KUNDEN (
    KUNDEN_ID     INT PRIMARY KEY,
    NAME          VARCHAR(255),
    GEBURTSTAG    DATE,
    ADRESSE       VARCHAR(255),
    STADT         VARCHAR(100),
    POSTLEITZAHL  VARCHAR(10),
    LAND          VARCHAR(100),
    EMAIL         VARCHAR(255) UNIQUE,
    TELEFONNUMMER VARCHAR(20)
);
```

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

```
CREATE TABLE BESTELLUNG (
    BESTELLUNG_ID INT PRIMARY KEY,
    BESTELLDATUM DATE,
    ARTIKEL_ID   INT,
    UMSATZ       INT,
    FK_KUNDEN    INT NOT NULL,
    FOREIGN KEY (FK_KUNDEN) REFERENCES KUNDEN (KUNDEN_ID)
);
```

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

```
function prepare()
    local create_kunden_query = [[
        CREATE TABLE KUNDENMITID (
            ....
        );
    ]]
    local create_bestellung_query = [[
        CREATE TABLE BESTELLUNGMITID (
            ...
        );
    ]]

    db_query(create_kunden_query)
    db_query(create_bestellung_query)
    print("Tables KUNDENMITID and BESTELLUNGMITID have been successfully created.")
end
```

Wenn die Datenbank beispielsweise in einer Produktivumgebung läuft, dann wollen wir, dass die Benchmarks möglichst wenig Einfluss aus 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 EXITS“ 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

```
function cleanup()
    local drop_kunden_query = "DROP TABLE IF EXISTS KUNDENMITID;"
    local drop_bestellung_query = "DROP TABLE IF EXISTS BESTELLUNGMITID;"

```

```
    db_query(drop_bestellung_query)
    db_query(drop_kunden_query)
    print("Cleanup successfully done")
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:

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

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

```
local num_rows = 1000
local bestellungProKunde = 4

function delete_data()
    local delete_bestellung_query = "DELETE FROM BESTELLUNGMITID;"
    local delete_kunden_query = "DELETE FROM KUNDENMITID;"
    db_query("START TRANSACTION")
    db_query(delete_bestellung_query)
    db_query(delete_kunden_query)
    db_query("COMMIT")
end

function insert_data()
    delete_data()
    for i = 1, num_rows do
        local kunden_id = i -- define name, geburtstag, adresse, stadt, postleitzahl, land, email, telefonnummer
        local kunden_query = string.format([[
            INSERT IGNORE INTO KUNDENMITID
            (KUNDEN_ID, NAME, GEBURTSTAG, ADRESSE, STADT, POSTLEITZAHL, LAND, EMAIL, TELEFONNUMMER)
            VALUES (%d, '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s');
        ]], kunden_id, name, geburtstag, adresse, stadt, postleitzahl, land, email, telefonnummer)
        db_query(kunden_query)

        for j = 1, bestellungProKunde do
```

```
            local bestellung_id = (i-1) * bestellungProKunde + j -- define bestelldatum, artikel_id, umsatz
            local bestellung_query = string.format([[
                INSERT IGNORE INTO BESTELLUNGMITID
                (BESTELLUNG_ID, BESTELLDATUM, ARTIKEL_ID, UMSATZ, FK_KUNDEN)
                VALUES (%d,'%s', %d, %d, %d);
            ]], bestellung_id, bestelldatum, artikel_id, umsatz, kunden_id)
            db_query(bestellung_query)
        end
    end
end

function event()
    insert_data()
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

```
function select_query()
    local join_query = [[
        SELECT k.STADT, SUM(b.UMSATZ) AS Total_Umsatz
        FROM KUNDENMITVARCHAR k
        JOIN BESTELLUNGMITVARCHAR b ON k.NAME = b.FK_KUNDEN
        GROUP BY k.STADT;
    ]]
    db_query(join_query)
end

function event()
    select_query()
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:

```lua
local length = 10
```

Die Zeile mit der importierten Länge:

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

Den Orchestrator - Script ruft man wie folgt auf:

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

```
./sysbench_script.sh \
  -out "YOUR_PATH_TO_DIRECTORY/Output" \
  -var '{"length":[1,64]}' \
  -scripts:"YOUR_PATH_TO_DIRECTORY/int_queries" \
  "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 4.1 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 sichergegangen, 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

```
# Main benchmark loop
for INFO in "${QUERY_INFO[@]}"; do
  # Definition of the variables comes here

  if [[ -n "$EXPORTED_VARS" ]]; then
    IFS=';' read -r -a KEYS <<< "$EXPORTED_VARS"
    combinations=$(generate_combinations "" "${KEYS[@]}")
    # Process each combination
    while IFS=',' read -r combination; do
        IFS=',' read -ra key_value_pairs <<< "$combination"
        for pair in "${key_value_pairs[@]}"; do
          export "$(echo "${pair%%=*}" | tr '[:lower:]' '[:upper:]')=${pair#*=}"
        done
        COMBINATION_NAME=$(echo "$combination" | sed -E 's/(^|,)num_rows=[^,]*//g;s/^,//;s/,$//' | tr ',' '_' | tr '=' '_')
        LOG_DIR_KEY_VALUE="$LOG_DIR/$COMBINATION_NAME"
        mkdir -p "$LOG_DIR_KEY_VALUE"

        process_script_benchmark "$QUERY_PATH" "$LOG_DIR_KEY_VALUE" "$INSERT_SCRIPT" "$SELECT_SCRIPT" "$COMBINATION_NAME"
    done <<< "$combinations"
  else
    # Process normally when no keys specified
    mkdir -p "$LOG_DIR"
    process_script_benchmark "$QUERY_PATH" "$LOG_DIR" "$INSERT_SCRIPT" "$SELECT_SCRIPT"
  fi
done
```

Die Funktion `process_script_benchmark` führt wie beim Demo - Beispiel schon erwähnt, die Methoden prepare, insert, selct und cleanup durch 2.2. Außerdem überprüft sie auch, ob 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

```
process_script_benchmark() {
  local QUERY_PATH="$1" LOG_DIR="$2" INSERT_SCRIPT="$3" SELECT_SCRIPT="$4" COMBINATION="${5:-}"
  local SCRIPTS=()
```

```
  local IS_FROM_SELECT_DIR=false

  if [ -f "$SELECT_SCRIPT.lua" ]; then
    # SELECT_SCRIPT is a Lua file
    SCRIPTS=("$INSERT_SCRIPT" "$SELECT_SCRIPT.lua")
  else
    # SELECT_SCRIPT is a directory
    SCRIPTS=("$INSERT_SCRIPT" "$SELECT_SCRIPT"/*)
    IS_FROM_SELECT_DIR=true
  fi

  # Prepare benchmark
  PREPARE_LOG_FILE="$LOG_DIR/$(basename "$QUERY_PATH")${COMBINATION:+_${COMBINATION}}_prepare.log"
  run_benchmark "$MAIN_SCRIPT" "prepare" "$PREPARE_LOG_FILE" "" "${COMBINATION_NAME:-}"

  # Select and Insert benchmark
  for SCRIPT in "${SCRIPTS[@]}"; do
    if [ -f "$SCRIPT" ]; then
      local SCRIPT_NAME
      # Removed script name definition for less complexity => multiple if-cases processed
      local RAW_RESULTS_FILE="$LOG_DIR/${SCRIPT_NAME}.log"
      run_benchmark "$SCRIPT" "run" "$RAW_RESULTS_FILE" "$SCRIPT_NAME" "$COMBINATION"
    fi
  done

  #Cleanup benchmark
  run_benchmark "$MAIN_SCRIPT" "cleanup" "$LOG_DIR/$(basename "$QUERY_PATH")${COMBINATION:+_${COMBINATION}}_cleanup
     .log"
}
```

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

Die Methode `run_benchmark` führt den Sysbench – Befehl (siehe 2.2) aus und nur wenn es sich um die Methode `RUN` handelt, werden die Daten während der Ausführung und die Endstatistiken in je eine CSV - Datei gespeichert. Aus diesen beiden CSV - Dateien müssen die Insert - und Select - Queries der zugehörigen Skripte wieder vereint werden und die Attribute werden miteinander addiert. Der letzte Schritt, der noch fehlt, ist die Erstellung der Graphen.

Wenn wir das komplette Skript für unser Join-Typ-Beispiel ausführen, dann ist das Endresultat ein Output-Ordner, der an angegebener Stelle erstellt wird. 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.

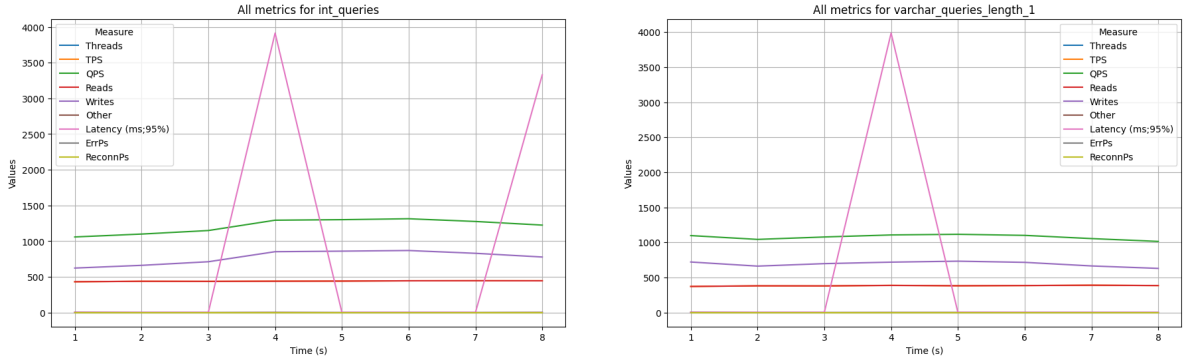



**Abbildung 2.2:** Die Grafik zeigt alle Metriken für die Skripte `int_queries` (links) und `varchar_queries_length_1` (rechts)

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 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.

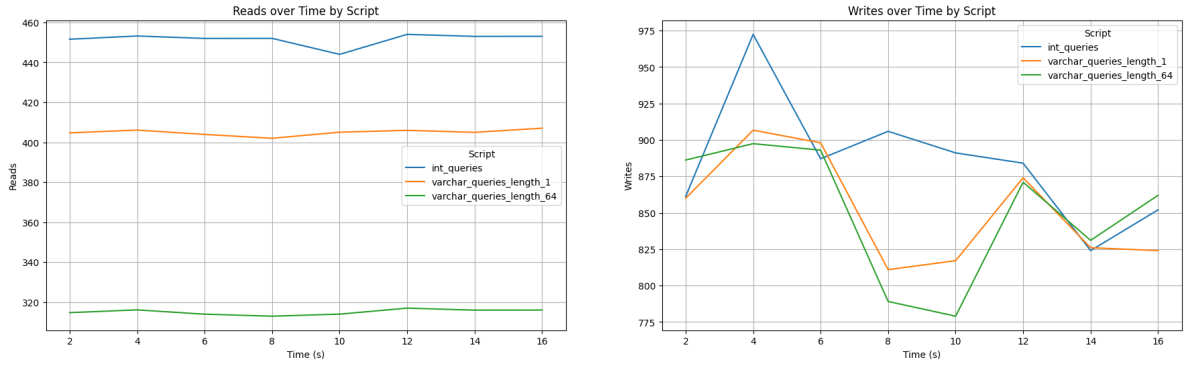



**Abbildung 2.3:** Die Grafik zeigt den Vergleich zwischen allen Skripten für die Metriken Reads (links) und Writes (rechts)

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 der Beste ist.

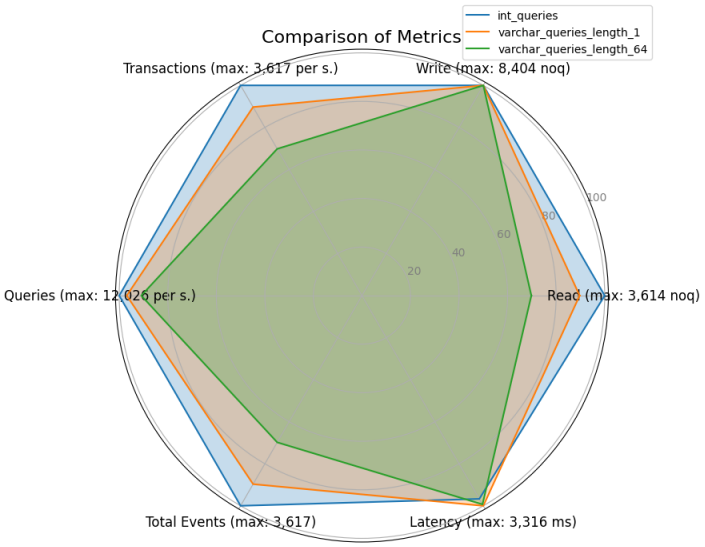



**Abbildung 2.4:** Darstellung der Skripte und 6 Metriken in einem Hexagon

## 2.4 GitHub Action

Im Verlauf der Bachelorarbeit sind immer mehr unterschiedliche Projekte dazugekommen, wie in den späteren Kapiteln zu sehen sein wird, die alle das Orchestrator-Skript verwenden. Dadurch sind immer mehr Fallunterscheidungen im Hauptskript erforderlich geworden und man hat schnell den Überblick verloren, wenn man Änderungen vorgenommen hat. Um diese

Änderungen zu überprüfen, 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, damit wäre aber nicht das Lastenproblem und der manuelle Aufwand gelöst worden. Eine deutlich bessere Variante ist das Automatisieren dieser Befehle unabhängig von dem lokalen Rechner auf virtuellen Maschinen in der Cloud. Als Plattform für diese Continuous Integration und Continuous Delivery (CI/CD) habe ich mich für GitHub Actions entschieden ([10]). Mit GitHub Actions kann man Workflows erstellen, die bei einem bestimmten Event getriggert werden und anschließend eine Anzahl von Aufträgen nacheinander oder gleichzeitig ausführen können. Jeder Auftrag (engl. Job) wird innerhalb eines eigenen Runners der virtuellen Maschine in einem Container ausgeführt und mann über einen oder mehrere Schritte (eng. Step) verfügen. Die Schritte können wiederum beliebige Shell-Befehle, Skripte oder Aktionen ausführen.

Wie in den Kapiteln 2.2 und 2.3 erklärt, müssen wir das Hauptskript ausführen, in dem die Benchmarktests durchführen werden. Außerdem benötigen wir auch die Pfade zu den Lua-Skripten, die getestet werden sollen und in einigen Fällen müssen wir auch Variablen definieren, die später exportiert und in den Lua-Skripts verwendet werden. Diese Pfade und Variablen müssen wir pro Projekt definieren und zusammen mit je einem Namen in einer JSON-Datei speichern.

**Codeblock 2.14:** JSON mit Konfiguration der Script

```
{
  "example-name": {
    "dirs": [
      "./example-project/Scripts/int_queries",
      "./example-project/Scripts/varchar_queries:length"
    ],
    "var": {"length": [1, 64]}
  }
}
```

Damit wir das Hauptskript ausführen können, müssen wir im ersten Auftrag (engl. Job) die Daten dieser JSON-Datei verarbeiten und bestimmte Variablen, wie beispielweise den Outputordner definieren. Zudem müssen wir alle Namen der verschiedenen Projekte in einer Liste zusammenfügen und als Output für den nächsten Job definieren. Dieser Job wird erst gestartet, wenn der Erste beendet ist und ist verantwortlich für das eigentliche Durchführen der Benchmarks. Da wir die Vorteile des gleichzeitigen Ausführens der Aufträge nutzen wollen, müssen wir die Matrixstrategie verwenden, damit die Benchmarks parallel ausgeführt werden. Bei der Matrixstrategie kann man Variablen definieren, um automatisch mehrere

Auftragsausführungen parallel zu erstellen. In unserem Fall verwenden wir dafür die Liste mit den unterschiedlichen Projektnamen.

Damit nun die Benchmarks ausgeführt werden können, müssen wir innerhalb dieser Matrixausführung einige Vorbereitungen treffen. Zum einen müssen wir die Dependencies für Sysbench und die Python-Libraries installieren und zum anderen müssen wir einen MySQL-Container mit passenden Konfigurationen starten und vorbereiten. Nach diesen Schritten können wir das Hauptskript ausführen und die Outputdateien werden an dem angegebenen Pfad erstellt.

Um Zugriff auf diese Dateien zu erhalten, müssen wir sie als GitHub Artifact hochladen. Die GitHub Artifacts können wir anschließend entweder über die GitHub REST Api oder die Übersicht des Workflows in GitHub als Zip-Datei herunterladen. Als letzten Auftrag, nach Beendigung beider vorangegangenen Jobs, können wir alle GitHub Artifacts zu diesem Workflow herunterladen und gemeinsam als Artifact wieder hochladen, damit wir z.B. bei 10 Projekten, nicht 10 Zip-Dateien herunterladen und einzeln entpacken müssen, um die Änderungen der Dateien zu überprüfen. Wenn fehlerhafte Änderungen den Workflow triggern, kann es dazu kommen, dass je nach Fehler unterschiedliche Jobs oder Steps nicht erfolgreich ausgeführt werden und damit der komplette Workflow scheitert.

Der Workflow wird in einer YAML-Datei im Ordner `.github/workflows/` definiert. Zunächst muss man den Names des Workflows definieren und anschließend, wann er getriggert werden soll. Dies kann beispielsweise manuell auf GitHub mit dem Tag `workflow_dispatch` oder bei jedem Push mit `push`, u.a. kann das auch auf bestimmte Dateien oder Ordner einschränken. Unter dem Tag `env` muss man die Umgebungsvariablen definieren, dazu gehören zum Beispiel der Datenbankname oder die Länge der Durchführung des Benchmarks. Wenn es sich um vertrauliche Informationen handelt, sollte man GitHub Secrets dafür verwenden. Ein Beispiel dafür wäre das Downloaden der Artefakte im letzten Job, um einen gemeinsamen Output-Ordner zu erstellen. Dafür wird die GitHub REST API benötigt, die ein vertrauliches `Personal Access Token` erfordert, welches Repository- sowie Lese- und Schreibrechte für GitHub Registries besitzt.

**Codeblock 2.15:** Ausschnitt aus der Workflow - Datei

```yaml
name: Benchmark Workflow
on:
  push:
    paths:
      - 'Projects/**'
      - ...
jobs:
  prepare-benchmark:
    outputs:
      matrix: ${{ steps.set-matrix.outputs.matrix }}
      configurations: ${{ steps.prepare-config.outputs.configurations }}
    steps:
```

```yaml
    - name: Checkout repository
      uses: actions/checkout@v3
    - name: Read and generate list of matrix name
      run: # ... echo "matrix=$matrix" >> $GITHUB_OUTPUT
    - name: Prepare configurations for all test types
      run: # ... export variables like  test_type, dirs, var, output_dir, artifact_name, should_run as "configurations"
run-tests:
  needs: prepare-benchmark
  strategy:
    matrix:
      test-type: ${{ fromJson(needs.prepare-benchmark.outputs.matrix) }}
  env:
    OTHER_ENVIRONMENT_VARIABLES
  steps:
    - name: Checkout repository
      uses: actions/checkout@v3
    - name: Extract and save values to GitHub environment
      run: # ...
    - name: Install dependencies
      run: # install sysbench,pandas and matplotlib
    - name: Start MySQL container
      run: |
        docker run --name mysql-${{ env.test_type }} -d -e MYSQL_ROOT_PASSWORD=$DB_PASS -e MYSQL_DATABASE=$DB_NAME -p $DB_PORT:3306 mysql:8.0
        until docker exec mysql-${{ env.test_type }} mysqladmin --user=root --password=$DB_PASS --host=127.0.0.1 --port=$DB_PORT ping --silent; do sleep 1; done
        echo "MySQL is ready!"
    - name: Run sysbench script
      run: |
        chmod +x Tools/sysbench_script.sh
        Tools/sysbench_script.sh -out "${{ env.output_dir }}" -var '${{ env.var }}' -scripts:'${{ env.dirs }}'
    - name: Upload individual outputs
      uses: actions/upload-artifact@v4
      with:
        name: ${{ env.artifact_name }}
        path: ${{ env.output_dir }}

upload-combined-output:
  needs: [prepare-benchmark, run-tests]
  steps:
    - name: Loop through configurations, download artifacts with artifact_name and unzip it
      run: # ... ALL_ARTIFACTS=$(curl -s -H "Authorization: Bearer ${{ secrets.GITHUB_TOKEN }}" "https://api.github.com/repos/${{ github.repository }}/actions/artifacts") ...
    - name: Upload "Output" - folder as artifact
      run: # ...
```

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.

## 2.5 Optimierungen des Workflows

Zum einen kann man die zu installierenden Abhängigkeiten mithilfe des GitHub Caches ([11]) 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 2.16:** Speichern der Abhängigkeiten im Cache

```
- name: Cache pip dependencies
  if: env.should_run == 'true'
  uses: actions/cache@v3
  with:
    path: ~/.cache/pip
    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. Beispielweise, 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:

```
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 and 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 ([11]). Eine andere interessante Lösung kann auch das Nutzen von expliziten Branches nur für die Sicherung der Dateien seien. 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 verhindertem Push erneut ein Rebase durchgeführt werden muss. Außerdem muss man dafür der GitHub Action Schreibberichtigungen 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.

# 3 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.

## 3.1 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.

## 3.2 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 nummerische 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 3.1 und mit dem Unsigned - Attribut in 3.2.

$$\text{Signed: } -2^{(N-1)} \text{ bis } 2^{(N-1)} - 1 \tag{3.1}$$

$$\text{Unsigned: } 0 \text{ bis } 2^{N} - 1 \tag{3.2}$$

**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 3.1:** SQL-Befehl zur Erstellung der Testtabelle

```
CREATE TABLE test_int (
    int_5 INT(5),
    int_11 INT(11)
);
```

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 3.2:** Inserts und Selects für Testtabelle aus 3.1

```
INSERT INTO test_int (int_5, int_11) VALUES (2147483647, 2147483647);
INSERT INTO test_int (int_5, int_11) VALUES (-2147483648, -2147483648);

SELECT * FROM test_int;
```

**Tabelle 3.1:** Ergebnis der SQL-Abfrage aus 3.2

| 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.

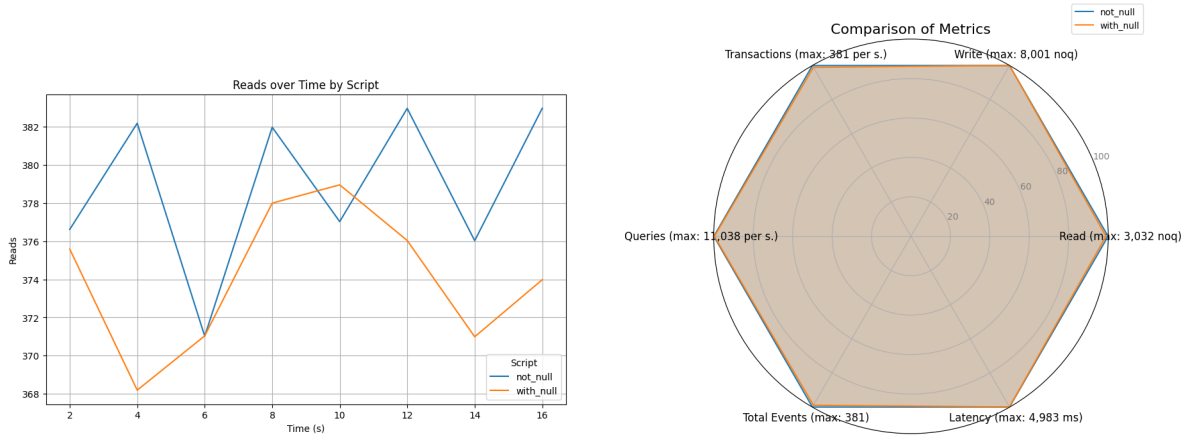
Die beiden Haupttypen für Zeichenketten sind `VARCHAR` und `CHAR`. `VARCHAR` speichert die Zeichenfolgen mit variabler Länge und benötigt daher weniger Speicherplatz als Typen mit fester Länge, da nur so viel Platz verwendet wird, wie tatsächlich benötigt wird. Zusätzlich werden ein oder zwei Bytes für 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.

## 3.3 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.



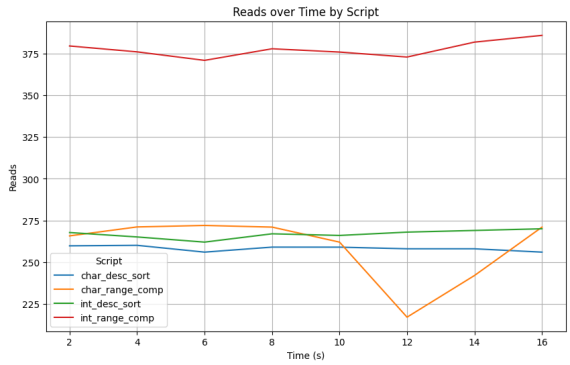

**(a)** Readsabfragen für jeweils 2 Selects

**(b)** Metriken über den kompletten Zeitraum

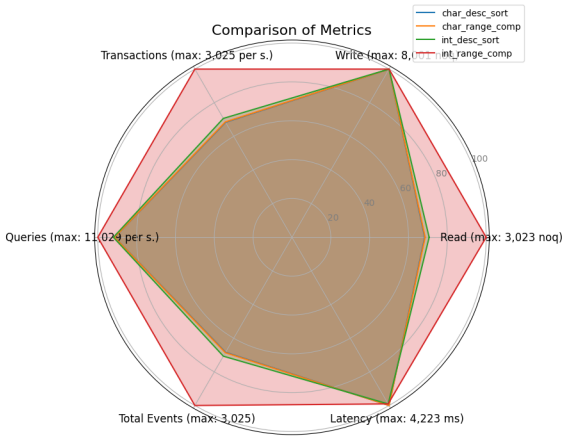
**Abbildung 3.1:** Vergleich von NULL und NOT NULL

In der erstellten Grafik (siehe Abbildung 3.1a) 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 (3.2a).



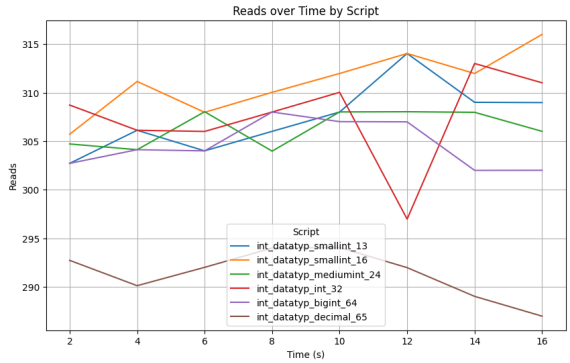

**(a)** Vergleich für jeweils 2 unters. Abfragen



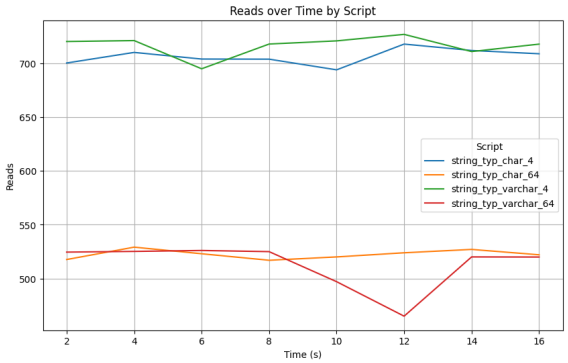

**(b)** Kennzahlen über den gesamten Zeitraum

**Abbildung 3.2:** 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 3.3a). 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

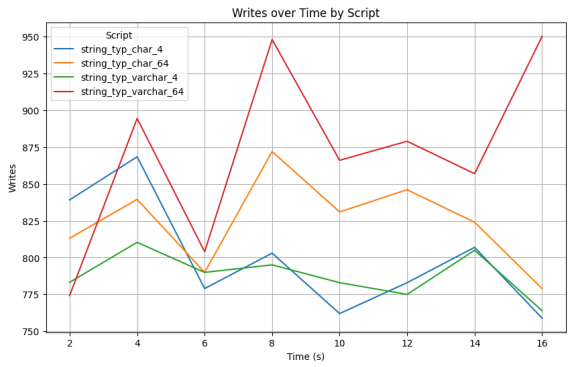



**(b)** Unterschiedliche Zeichenketten-Typen

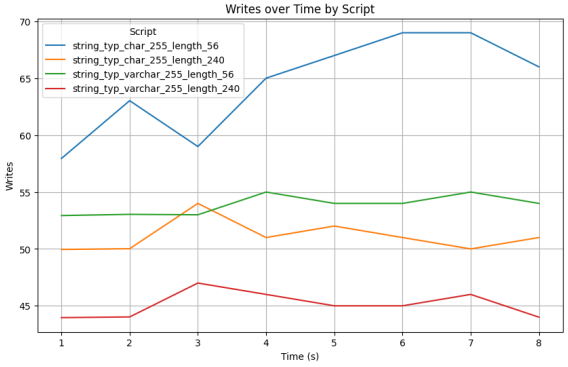
**Abbildung 3.3:** Vergleich von unterschiedlichen Select-Abfragen

Beim Vergleich zwischen den beiden Zeichenketten-Typen `CHAR` und `VARCHAR` ist unabhängig von der Länge zu erkennen, dass `VARCHAR` effizienter ist als `CHAR` (3.3b). 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 3.4a).




**(a)** Bei gleicher Länge der Zeichen




**(b)** Bei unterschiedlichem Befüllungsgrad

**Abbildung 3.4:** 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` (3.4b). 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.

# 4 Indexierung und Einfluss auf die Performance

## 4.1 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.

## 4.2 B-Baum-Index

TODO

## 4.3 Hash - Index

TODO

# 5 Views

In dem folgenden Kapitel betrachten wir die Performance Vorteile von Sichten, bzw. Views in SQL. Zunächst betrachten wir dafür virtuelle Sichten und ihre Vor- und Nachteile und wie sie mit Inserts umgehen. Anschließend betrachten wir materialisierte Sichten, die tatsächlich physisch in der Datenbank gespeichert werden und wie man sie implementiert. Im Zuge dessen müssen wir Trigger erklären und verwenden, da die native Umsetzung von materialisierten Sichten in MySQL nicht unterstützt wird. Zum Schluss betrachten wir die Durchführung der Benchmarks näher und interpretieren die entstandenen Ergebnisse.

## 5.1 Virtuelle Views

Grundlegend existieren Relationen, bzw. Tabellen, die durch das `CREATE TABLE` – Statement definiert werden, physisch in der Datenbank. Damit sind sie persistent, was bedeutet, dass sie dauerhaft existieren und sich nicht ändern, es sei denn, sie werden explizit durch eine SQL-Änderungsanweisung dazu aufgefordert. Dies entspricht der Dauerhaftigkeit des ACID-Prinzips [12], die sicherstellt, dass bestätigte Transaktionen dauerhaft gespeichert bleiben und auch bei Systemausfällen nicht verloren gehen. Es gibt jedoch eine weitere Klasse von SQL-Relationen, die nicht wie Tabellen tatsächlich physisch gespeichert werden. Diese werden als virtuelle Sichten bezeichnet. Virtuelle Sichten werden durch einen Ausdruck definiert, der einer Abfrage ähnelt und können aber auch abgefragt werden, als ob sie tatsächlich physische existieren. In einigen Fällen kann man sogar die Datensätze über die Sicht geändert werden.

Die grundsätzliche Definition einer virtuellen View sieht so aus:

**Codeblock 5.1:** Allgemeine View-Deklaration

```
1 CREATE VIEW ‹name› AS ‹definition›
```

Als Nächstes müssen wir die `view-definition` mit einer SQL-Abfrage ersetzen, die den Inhalt der virtuellen Sicht abbilden soll. Um dieses Vorgehen mit einem Beispiel zu erklären, nutzen wir erneut die Tabellen Kunde (2.5) und Bestellung (2.6). Nun wollen wir, dass in der Sicht die beiden Tabellen zusammengefügt werden über die `KUNDEN_ID`, da diese zum einen der Primärschlüssel der Kundentabelle ist und zum anderen der Fremdschlüssel in Bestellung.

Damit wir nicht nur diese Join-Operation haben, wollen wir zusätzlich pro Jahr und pro Stadt den Umsatz aggregieren. Diese Aggregation könnte beispielsweise vom Marketingteam genutzt werden, um schwache Regionen zu identifizieren und gezielt in diesen nachzusteuern. Die View mit dem Namen `KUNDEN_OVERVIEW` hat folgende Struktur:

**Codeblock 5.2:** View Deklarierung

```
CREATE VIEW KUNDEN_OVERVIEW AS
SELECT
    EXTRACT(YEAR FROM B.BESTELLDATUM) AS Jahr,
    K.LAND AS Land,
    SUM(B.UMSATZ) AS Gesamtumsatz
FROM KUNDEN K
     JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
GROUP BY EXTRACT(YEAR FROM B.BESTELLDATUM), K.LAND;
```

Wenn wir jetzt die Daten dieser virtuellen Sicht abfragen wollen, dann adressieren wir den Namen in der FROM-Klausel und verlassen uns darauf, dass das Datenbankmanagementsystem die benötigten Tupel erzielt. Dabei operiert das DBMS direkt auf den Relationen, die die virtuelle Sicht definieren. In unserem Fall sind das die Kunden- und Bestelltabelle.

**Codeblock 5.3:** SQL-Befehl mit Sicht

```
SELECT * FROM KUNDEN_OVERVIEW
ORDER BY Jahr ASC, Gesamtumsatz DESC;
```

Eine weitere Möglichkeit, um sich die Funktionsweise der Sicht besser vorzustellen, ist es, wenn man jede Sicht in einer FROM-Klausel durch eine Unterfrage ersetzen, die mit der Sichtdefinition identisch ist. Diese Unterabfrage müssen wir noch mit einer Tupelvariablen ergänzen, damit wir auch Bezug auf die Tupel nehmen können. Man kann den Attributen einer Sicht auch eigene Namen vergeben, indem wir sie in Klammern hinter dem Namen der Sicht aus der `CREATE VIEW`-Anweisung auflisten.

**Codeblock 5.4:** Select-Befehl ohne Sicht

```
SELECT YEAR (B.BESTELLDATUM) AS Jahr, K.STADT, SUM (B.UMSATZ) AS Gesamtumsatz
FROM KUNDEN K
    JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
GROUP BY YEAR (B.BESTELLDATUM), K.STADT
ORDER BY Jahr DESC, Gesamtumsatz DESC;
```

Die Definition einer Sicht kann mit `DROP VIEW ‹view-name›` gelöscht werden, wodurch keine Abfragen oder Änderungsbefehle mehr auf dieser Sicht ausgeführt werden können. Das

Löschen der Sicht hat jedoch keine Auswirkungen auf die Tupel der zugrundeliegenden Tabelle(n). Im Gegensatz dazu würde `DROP TABLE ‹table-name›` die Tabelle löschen und damit auch die darauf basierenden Sichten unbrauchbar machen, da ihre Definitionen auf der gelöschten Tabelle beruhen. Abgesehen vom Löschen der Tabellen kann man auch Einfügungen an der View durchführen. Dies ist aber nicht uneingeschränkt möglich und nur unter bestimmten Bedingungen erlaubt. Zum einen muss die Sicht durch eine einfache Abfrage aus nur einer einzigen Relation definiert sein. Zum anderen muss die `SELECT`-Klausel ausreichend Attribute umfassen, sodass fehlende Werte bei Einfügungen mit `NULL` oder Standardwerten ergänzt werden können. Die Änderungen werden dann direkt auf die Basistabelle angewendet, wobei nur die in der Sicht definierten Attribute berücksichtigt werden. Wenn die eben beschriebenen Bedingungen erfüllt sind, werden auch bei Löschungen und Aktualisierungen wird die Änderung auf die zugrundeliegende Relation R übertragen. Dabei word die `WHERE`-Bedingung der View zu den Bedingungen der Änderung im `WHERE`-Block hinzugefügt. Wenn die Bedingungen nicht erfüllt sind, wie bei unserem Beispiel (5.2), dann müssen Änderungen über die zugrunde liegenden Tabellen erfolgen und die View kann nur für Select-Abfragen benutzt werden.

Das Einfügen über die Sicht ist jedoch nicht die intuitivste Möglichkeit, um Änderungen an die unterliegenden Tabellen durchzuführen. Das liegt vor allem an dem Umgang mit den nicht definierten Werten, weshalb sich das Konzept von Triggern anbietet. Trigger in SQL sind Datenbankobjekte, die mit einer Tabelle verknüpft sind und sobald bestimmte Ereignisse eintreten, führen sie eine Reihe von Anweisungen aus ([13]). Die Auslösung eines Triggers kann entweder vor (`BEFORE`) oder nach (`AFTER`) einem bestimmten Ereignis erfolgen, wie `INSERT`, `UPDATE` oder `DELETE`. Bei Triggern auf Sichten können auch `INSTEAD-OF`-Trigger verwendet werden, die Änderungsversuche an der Sicht abfangen und stattdessen eine frei definierbare Aktion ausführen.

**Codeblock 5.5:** Allgemeine Trigger Deklaration

```
1 CREATE TRIGGER trigger_name
2 {BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
3 ON {table_name | view_name}
4 FOR EACH ROW
5 trigger_body;
```

Das Problem in MySQL mit Triggern ist aber, dass sie nur auf Tabellen angewendet werden können. Später werden wir im Kapitel 5.3 ein genaueres Beispiel dazu betrachten. Wir können uns jedoch an noch einem anderen Konzept, den Stored Procedures bedienen. Stored Procedures sind Funktionen, die direkt im DB-Server hinterlegt werden und wie andere integrierte Funktionen, wie z.B. round(), aufgerufen werden können.

**Codeblock 5.6:** Allgemeine Prozedur Deklaration

```
CREATE PROCEDURE stored_procedure_name(IN param1 INT, IN param2 VARCHAR(255))
BEGIN
    -- smth
END
```

Da wir in unserem Beispiel eine View (5.2) haben, die 2 Tabellen verbindet, ist es nicht möglich, die Daten direkt einzufügen. Deshalb verwenden wir stattdessen eine Prozedur. Die Prozedur bekommt die gleichen Parameter, wie die Spalten der View und verarbeitet diese, um die entsprechenden Daten in die zugrunde liegenden Tabellen einzufügen. Wenn die Prozedur korrekt ist, dann werden die Änderungen bei der nächsten `SELECT`-Abfrage der View sichtbar.

**Codeblock 5.7:** Deklaration der Prozedur

```
CREATE PROCEDURE insert_view(IN Jahr INT, IN Stadt VARCHAR (255), IN Umsatz INT)
BEGIN
INSERT INTO BESTELLUNG (BESTELLDATUM, FK_KUNDEN, UMSATZ)
VALUES (STR_TO_DATE(CONCAT(Jahr, '-01-01'), '%Y-%m-%d'),
        (SELECT K.KUNDEN_ID FROM KUNDEN K WHERE K.STADT = Stadt LIMIT 1), Umsatz);
END;
```

Jetzt können wir die Methode `insert_view` einfach mit dem `CALL`-Befehl aufrufen und die Werte für die drei Parameter in Klammern übergeben. Dadurch werden die Werte in die Bestelltabelle eingefügt. Als Bestelldatum wird stets der erste Tag des Jahres verwendet und als Kunde wird einer gewählt, der in der jeweiligen Stadt lebt. Im Vergleich zum direkten Einfügen in die Bestelltabelle verlieren wir jedoch an Datenpräzision. Einerseits haben wir nicht das genaue Datum und andererseits fehlt die Information zur `ARTIKEL_ID`. Zusammengefasst lässt sich sagen, dass je nach Definition der Sicht Daten entweder direkt eingefügt oder mithilfe von Stored Procedures befüllt werden können. Es ist dabei jedoch nicht ausgeschlossen, dass in den zugrunde liegenden Tabellen zu geringerer Datenqualität kommen kann, da zum Beispiel NULL-Werte oder andere Standardwerte verwendet werden.

## 5.2 Materialisierte Views

Allgemein sind Sichten so definiert, dass sie eine neue Relation aus Basistabellen durch Ausführen einer Abfrage auf diesen Tabellen erstellen. Bisher haben wir Sichten ausschließlich als logische Beschreibungen von Relationen betrachtet. In bestimmten Fällen kann es jedoch aus Performancegründen sinnvoll sein, sie zu materialisieren, also die Ergebnisse physisch zu

speichern. Durch die physische Speicherung verringert sich der Rechenaufwand für Abfragen, da beispielsweise in unserem Fall (siehe View (5.2) oder materialisierte View (5.8)) der Join nicht erneut ausgeführt werden muss. Die bereits gespeicherten Ergebnisse sind direkt abrufbar. Dies führt zu einer schnelleren Antwortzeit der Query, bringt jedoch einen höheren Pflegeaufwand mit sich. Passend zu unserer virtuellen Sicht (5.2) sieht die Materialisierte wie folgt aus:

**Codeblock 5.8:** Materialized View

```
1 CREATE MATERIALIZED VIEW UmsatzProJahrStadt AS
2 SELECT
3     EXTRACT(YEAR FROM B.BESTELLDATUM) AS Jahr,
4     K.STADT AS Stadt,
5     SUM(B.UMSATZ) AS Gesamtumsatz
6 FROM KUNDEN K
7          JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
8 GROUP BY EXTRACT(YEAR FROM B.BESTELLDATUM), K.STADT;
```

Wie zu sehen ist, unterscheidet sich die materialisierte Sicht nur in der ersten Zeile von der Virtuellen. Einen Nachteil der materialisierten Sicht gegenüber der Virtuellen ist der zusätzliche Aufwand, wie bei Indizes, bei Aktualisierungen Das liegt daran, dass Teile der Sicht bei jeder Änderung der zugrunde liegenden Basistabelle neu berechnet werden müssen. Da die Anzahl an Neuberechnungen einen großen Einfluss auf die Performance hat, muss man sich ein Konzept überlegen, mit dem die Anzahl auf ein Minimum begrenzt wird. Ansonsten kann es durch Sperren auf die zugrundeliegenden Tabellen dazu kommen, dass die Produktivumgebung eingeschränkt ist. Allerdings muss das DBMS Änderungen, die nicht in der Abfrage enthalten sind, sowie Relationen, die nicht betroffen sind, nicht berücksichtigen.

Es gibt aber weitere Optimierungen, um nicht jedes Mal die gesamte Sicht vollständig neu erstellen zu müssen. Man muss sich vor Augen führen, dass alle Änderungen an der zugrunde liegende Tabelle inkrementell sind. Damit können Einfügungen, Löschungen und Aktualisierungen an einer Basistabelle durch eine kleine Anzahl von Abfragen auf die Basistabellen und anschließende Änderungsanweisungen an der materialisierten Sicht umgesetzt werden.

Die inkrementelle Aktualisierung der materialisierten Sicht ist deutlich effizienter als die ständige Neuberechnung der Sicht. Nicht jedes Datenbankmanagementsystem bietet die inkrementelle Auffrischung an. Oracle unterstützt die inkrementelle Auffrischung nativ mit Materialized View Logs. In PostgreSQL ist eine manuelle Planung erforderlich, da eine automatische inkrementelle Aktualisierung nicht unterstützt wird ([14]). Wenn man die Option `CONCURRENTLY` beim Aktualisieren einer materialisierten Sicht verwendet, können andere Prozesse weiterhin darauf zugreifen, da die bestehende Sicht erst ersetzt wird, wenn die neue Version fertiggestellt ist (siehe 5.11). MySQL bietet gar nicht erst eine Möglichkeit an,

um materialisierte Sichten nativ zu erstellen. In MySQl kann man jedoch die Funktionsweise mithilfe einer physischen Tabelle als Sicht und mit einem Trigger auf die zugrundeliegenden Tabellen nachstellen, die bei Aktualisierung Änderungen an der Sichttabelle durchführt.

Ein Anwendungsfall für die Nutzung von aggregierten Daten in einer materialisierten Sicht ist die Analyse von Daten, um Vorhersagen zu treffen. Wenn beispielsweise Analysten eines Motorradbetriebes für die Zukunft den Einkauf planen wollen, dann müssen die vergangenen Daten häufig in einer aggregierten Form betracht werden. Diese materialisierte Sicht wird damit eher selten abgefragt, wohin hingegen Änderungen an unterliegenden Tabellen, wie z.B. den Bestand von Motorrädern oder Motorradteilen in der Lagertabelle sehr häufig passieren. Wenn man die Sicht bei jeder Änderung im Lager aktualisieren würde, wäre dies sehr aufwendig und hätte wahrscheinlich kaum einen Einfluss auf die Entscheidungen, die mithilfe der Sicht getrioffen werden. Daher ist es sinnvoll, dass die Daten nur einmal täglich aktualisiert werden, beispielweise als Cron-Job in der Nacht, wenn die Systemlast gering ist. In diesem Fall haben die Analysten zwar nicht die tagesaktuellen Daten, sondern nur den Stand des Vortages. Und da die Analysten in der Regel mit historischen Daten arbeiten, ist dieses Risiko eingehbar, wenn jedoch eine schnelle Lieferung an den Kunden versprochen wird, ist es für den Verkauf entscheidend, über aktuelle Daten zu verfügen. Andernfalls riskieren wir, dem Kunden falsche Versprechungen zu machen, etwa indem wir Produkte als verfügbar anzeigen, die in Wirklichkeit nicht mehr auf Lager sind.

Eine materialisierte Sicht kann wie eine normale Sicht in der FROM-Klausel einer Abfrage verwendet werden, genauso wie eine virtuelle Sicht. In Oracle gibt es zusätzlich noch eine spezielle Funktionalität, die es ermöglicht, Abfragen automatisch umzuschreiben. Damit wird eine materialisierte Sicht verwendet, auch wenn diese nicht explizit in der Abfrage referenziert wird. Bei dieser Funktionalität muss die materialisierte Sicht für Query Rewrite aktiviert sein mit `ENABLE QUERY REWRITE`. Die Abfrage wird aber nur dann umformuliert, wenn alle Relationen in der Sicht enthalten sind und die Bedingungen entsprechend angepasst werden. Folgendes Beispiel:

```
1 SELECT Stadt, Jahr, Gesamtumsatz
2 FROM KUNDEN K JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
3 WHERE Stadt = 'Berlin' AND Jahr = 2024;
```

In Oracle könnte diese Abfrage intern so umgeschrieben werden, dass stattdessen die Materialized View, die aggregierte Umsätze enthält, genutzt wird. Anders sieht dies in PostgreSQL aus, da dort diese automatische Abfrageumschreibung nicht existiert.

Bei der zweiten Abfrage ist dies aber nicht möglich, da die materialisierte Sicht nicht die Spalten Land und Monat enthalten. Deshalb werden in diesem Fall die zugrunde liegenden Tabellen direkt abgefragt, da die materialisierte Sicht nicht die benötigten Spalten enthält, und die Abfrage muss explizit auf diese Tabellen zugreifen, um die gewünschten Daten zu erhalten.

```sql
SELECT Land, Monat, Gesamtumsatz
FROM KUNDEN K JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
WHERE LAND = 'Deutschland' AND EXTRACT(MONTH FROM K.GEBURTSTAG) = 8;
```

Zusammengefasst lässt es sich sagen, dass die Auswahl von materialisierten Sichten deutlich komplexer ist als die von Indizes, da potenziell jede Abfrage eine Sicht definieren könnte. Damit gibt es potenziell deutlich mehr mögliche Sichten als Indexes. Es sollten aber nur Sichten erstellt werden, die mindestens eine Abfrage der erwarteten Workload verbessern, wobei Kriterien wie Relationen, Bedingungen und Attribute berücksichtigt werden. Zudem muss der Nutzen einer Sicht nicht nur anhand der Laufzeitverbesserung, sondern auch im Verhältnis zu ihrem Speicherbedarf bewertet werden, da materialisierte Sichten oft nicht nur erheblich mehr Speicherplatz beanspruchen können, sondern untereinander sich von der Größe deutlich unterscheiden.

## 5.3 Durchführung der Benchmarks

Das Ziel für die Durchführung ist es den Performanceunterschied zwischen einer virtuellen und einer materialisierten Sicht darzustellen. Da MySQL keine nativen materialisierten Sichten unterstützt, verfolgen wir einen alternativen Ansatz und vergleichen diesen später mit der nativen Implementierung in PostgreSQL, einem anderen Datenbankmanagementsystem.

Zuallererst beginnen wir mit der Umsetzung der virtuellen Sicht, für die wir, wie bei den anderen Sichten auch, zunächst die Basistabelle erstellen müssen. Als Basistabelle verwenden wir die Tabellen Kunden (2.5) und Bestellung (2.6) und erstellen die View (5.2), die wir in Kapitel (5.1) schon beschrieben haben. Es werden weiterhin Daten in die Kundentabelle eingefügt und pro Kunde wird eine festgelegte Anzahl an Bestellungen hinzugefügt. Wir fügen keine Datensätze über die virtuelle Sicht ein und damit die Sicht bei Select-Befehlen verwendet wird, muss sie explizit angesprochen werden. Da wir die Unterschiede in der Lesegeschwindigkeit möglichst repräsentativ feststellen wollen, betrachten wir mehrere Select-Befehle auf die unterschiedlichen Spalten der virtuellen Sicht. Ein beispielhafter Befehl sieht so aus:

**Codeblock 5.9:** Einfache Select-Query

```sql
SELECT Jahr, SUM(Gesamtumsatz) AS UmsatzProJahr FROM KUNDEN_OVERVIEW GROUP BY Jahr;
SELECT * FROM KUNDEN_OVERVIEW WHERE Jahr = 2020;
SELECT * FROM KUNDEN_OVERVIEW WHERE Land = 'Germany';
SELECT * FROM KUNDEN_OVERVIEW WHERE Gesamtumsatz > 2500;
```

Wie schon im Kapitel (5.1) erklärt, lassen sich diese Abfragen auf die virtuelle Sicht in direkte Abfragen auf die Kundentabelle umwandeln. Um den Einfluss der virtuellen Sicht auf die Performance zu sehen, führen wir deshalb einen Benchmark mithilfe der Sicht durch und bei dem anderen deklarieren wir gar keine Sicht und wandeln alle Befehle auf die Sicht direkt in SQL-Befehle auf die Kundentabelle um.

Als Nächstes befassen wir uns mit der materialisierten Sicht. Da MySQL, wie bereits erwähnt, keine materialisierten Views direkt unterstützt, erstellen wir zusätzlich zur Kundentabelle eine weitere physische Tabelle namens KUNDEN_MAT_OVERVIEW. Die Tabelle KUNDEN_MAT_OVERVIEW besteht, wie die virtuelle Sicht auch, aus den Spalten JAHR, LAND und GESAMTUMSATZ, wobei die Kombination aus Jahr und Land der Schlüssel der Tabelle ist. Als Nächstes erstellen wir die Tabellen KUNDEN, BESTELLUNG und KUNDEN_MAT_OVERVIEW und befüllen weiterhin die ersten beiden Tabellen mit Testdaten. Nach dem Einfügen der Testdaten wollen wir wieder die Select-Performance überprüfen und ersetzen bei unseren Select-Befehlen (5.9) die virtuellen Sicht KUNDEN_OVERVIEW mit der Tabelle KUNDEN_MAT_OVERVIEW. Wenn wir die Ergebnisse der Select-Befehle betrachten, dann fällt auf, dass die Tabelle KUNDEN_MAT_OVERVIEW keine Einträge hat. Das Problem beim bisherigen Verfahren liegt daran, dass die beiden Tabellen und KUNDEN_MAT_OVERVIEW separat voneinander existierende Tabellen sind und die Einfügungen wirken sich demnach nicht auf die KUNDEN_MAT_OVERVIEW-Tabelle aus. Lösen kann man das mithilfe der Definition von Triggern, die ausgelöst werden, wenn sich Werte in der Kundentabelle verändern. In MySQL kann ein Trigger nur für einen Datenbankmanipulationsoperator gleichzeitig verwendet werden ([15]), weshalb wir für INSERT und DELETE jeweils einen Trigger definieren müssen. Da wir keine Datensätze aktualisieren, vernachlässigen wir aus Simplizitätsgründen den Trigger für UPDATE. Für unser Beispiel sieht der INSERT-Trigger wie folgt aus:

**Codeblock 5.10:** Insert Trigger für Kundentabelle

```
1  CREATE TRIGGER UPDATE_BESTELLUNG_MAT_OVERVIEW_AFTER_INSERT
2  AFTER INSERT ON BESTELLUNG
3  FOR EACH ROW
4  BEGIN
5      DECLARE v_land VARCHAR(255);
6      DECLARE v_jahr INT;
7      SELECT LAND INTO v_land FROM KUNDEN WHERE KUNDEN_ID = NEW.FK_KUNDEN;
8      SELECT EXTRACT(YEAR FROM NEW.BESTELLDATUM) INTO v_jahr;
9
10     IF EXISTS (
11         SELECT 1
12         FROM KUNDEN_MAT_OVERVIEW
13         WHERE LAND = v_land AND JAHR = v_jahr
14     ) THEN
```

```
15          UPDATE KUNDEN_MAT_OVERVIEW
16          SET GESAMTUMSATZ = GESAMTUMSATZ + NEW.UMSATZ
17          WHERE LAND = v_land AND JAHR = v_jahr;
18      ELSE
19          INSERT INTO KUNDEN_MAT_OVERVIEW (JAHR, LAND, GESAMTUMSATZ)
20          VALUES (v_jahr, v_land, NEW.UMSATZ);
21      END IF;
22  END;
```

Ein Nachteil für den Ansatz mit einer normalen Tabelle im Zusammenspiel mit den Triggern wird damit auch deutlich, da es einen erhöhten Aufwand für jede materialisierte Sicht gibt, der zusätzlich noch stark abhängig vom jeweiligen Anwendungsbeispiel abhängt.

Damit wir die Performanceunterschiede zwischen Postgres und MySQL ermitteln können, implementieren wir zusätzlich den Ansatz mit den Triggern auch in Postgres. Die Implementierungen für die Insert- und Select-Befehle sind in MySQL und Postgres identisch, bei der Erstellung der Tabellen und Trigger gibt es aber Unterschiede. Zum einen unterscheiden sich die Mechanismen zur automatischen Generierung von Primärschlüsseln, da PostgreSQL `SERIAL` und MySQL `AUTO_INCREMENT` verwendet. Zum anderen kann in MySQL die Logik eines Triggers direkt in der `CREATE TRIGGER`-Anweisung definiert werden, während in PostgreSQL ein Trigger eine separate Funktion aufrufen muss, die die Logik enthält und mit `RETURNS TRIGGER` definiert ist. Auch die Deklarierung der Variablen unterscheidet sich, da in PostgreSQL mehrere Variablen in einem `DECLARE`-Block und in MySQL jede Variable einzeln im `BEGIN...END`-Block deklariert werden muss.

Als letzten Schritt betrachten wir die native Implementierung in PostgreSQL. In Postgres kann man direkt die materialisierte Sicht, siehe Beispiel (5.8), direkt erstellen und muss nicht eine zusätzliche Tabelle und die passenden Trigger dazu definieren. Wie man sieht, ist der Befehl nahezu identisch mit dem Befehl für das Erstellen der virtuellen Sicht (5.1). Die Select-Befehle sind komplett gleich im Vergleich zur Umsetzung mit dem Trigger und auch die Einfügungen erfolgen erneut auf den darunterliegenden Tabellen. Wie bereits beim MySQL-Ansatz aufgefallen ist, dass ohne Trigger keine Daten in der Tabelle vorhanden sind, gibt es hier auch keine, es sei denn, man aktualisiert die materialisierte Sicht explizit mit diesem Befehl:

**Codeblock 5.11:** Aktualisierung der materialisierten Sicht

```
1 REFRESH MATERIALIZED VIEW KUNDEN_MAT_OVERVIEW;
```

Da die Umsetzung von inkrementeller Auffrischung, siehe Kapitel 5.2, in Postgres nicht unterstützt wird, muss die materialisierte Sicht immer auf diese Art und Weise aktualisiert werden. Den Befehl führen wir nach den `INSERT` und `DELETE`-Befehlen auf der Kundentabelle

aus. Da wir den Einfluss auf die Performance des Befehls untersuchen möchten, führen wir ihn einmal nach jeder Einfügung in die Kundentabelle aus und einmal, nachdem alle Datensätze eingefügt wurden.

Zusammengefasst haben wir damit sechs unterschiedliche Benchmarks, die in zwei unterschiedlichen Graphen vergleichen. Zum einen haben wir den Vergleich zwischen keiner View, der virtuellen View und dem Ansatz mit Triggern in MySQL. Zum anderen haben wir auch eine Analyse des MySQL-Ansatzes im Vergleich zum gleichen Ansatz in PostgreSQL sowie der beiden nativen Implementierungen mit unterschiedlichen Aktualisierungshäufigkeiten durchgeführt.

## 5.4 Analyse der Ergebnisse

Nach der Durchführung der Benchmarks betrachten wir vor allem die Ergebnisse der `READS` und `WRITES` - Kennzahlen. Zunächst fällt, wie zu erwarten war, auf, dass die Unterschiede zwischen der virtuellen Sicht und den direkten SQL-Befehlen (`without_view`) nur minimal sind. Dies gilt sowohl für die Lese- als auch für die Schreibwerte.

Man kann auch einen klaren Performancevorteil beim Ansatz mit den Triggern erkennen, da die Lesewerte dort deutlich höher sind. Anders hingegen sieht es bei der Schreibperformance aus, da dort die Trigger greifen und zusätzliche Aktualisierungen durchführen, weshalb sich dort etwas geringe Werte ergeben. Der Unterschied bei Writes ist aber nicht wirklich signifikant in unserem Beispiel, kommt aber daher, da gar keine Daten physisch gespeichert werden. Dies liegt aber auch daran, dass wir relativ wenige Datensätze in den Testtabellen haben.

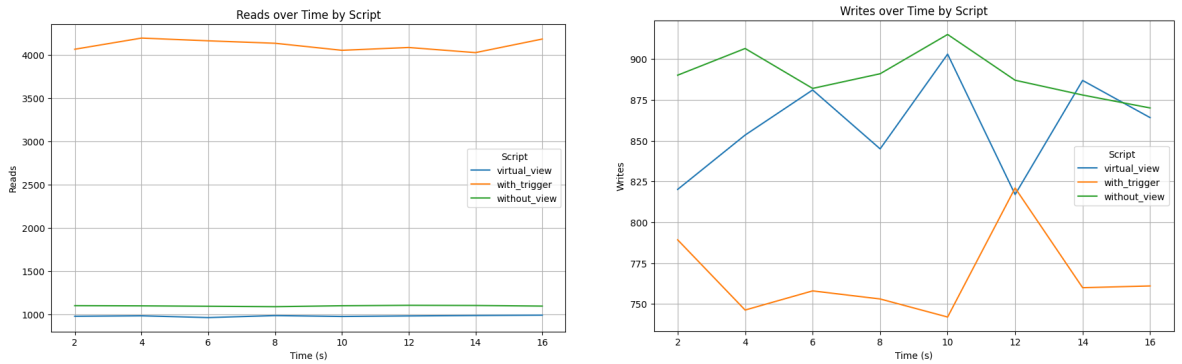



**Abbildung 5.1:** Vergleich zwischen keiner View, der virtuellen View und dem Ansatz mit Triggern in MySQL

Bei dem zweiten Vergleich sehen wir zuallererst einen sehr deutlichen Performanceunterschied beim Ansatz mit den Triggern zwischen Postgres und MySQL. Begründet werden kann dieser Unterschied mit den verschiedenen Vorteilen des Datenbankmanagementsystems und

dessen Umgebung. Wir haben beide Systeme mit dem gleichen Ansatz gebenchmarkt, damit wir die Implementierung der nativen materialisierten Sicht, die nur in Postgres möglich ist, besser vergleichen zu können. Denn die Ergebnisse der nativen Implementierung sind in Bezug auf die Abfragegeschwindigkeit tatsächlich am performantesten und die Anzahl an Aktualisierungen (5.11) hat dabei keinen Einfluss. Anders hingegen sieht es bei der Einfüge-Geschwindigkeit aus, da dort die Implementierung, die nach jedem Insert-Befehl aktualisiert nicht am schnellsten, sondern am langsamsten ist. Der Vergleich zwischen den DBMS fällt wieder schwer, da die Unterschiede zwischen `with_trigger` und `with_trigger_postgres` sehr groß sind (etwa um Faktor 45). Damit wird noch einmal deutlich wie stark die Einfügedauer bei den materialisierten Sichten von der Anzahl an Refreshs abhängig ist, da `mat_view_refresh_every` unterhalb der Performance von `with_trigger` liegt. In unserem Beispiel ist die einmalige Aktualisierung der Sicht besser als das Verwenden der Trigger in Postgres.

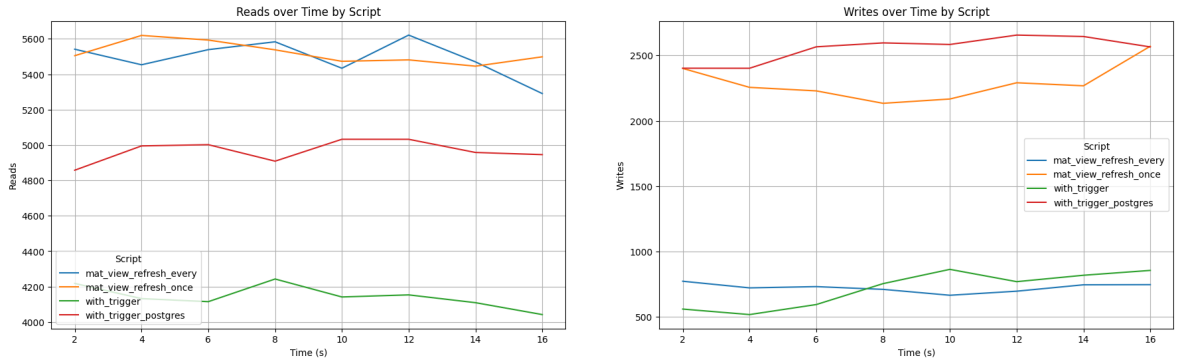



**Abbildung 5.2:** Vergleich zwischen Triggeransatz in MySQL und Postgres, sowie zwei nativen Implementierungen in Postgres unterschiedlicher Aktualisierungshäufigkeit

Es lässt sich also zusammenfassen, dass virtuelle Sichten keine Auswirkungen auf die Performance haben. Dies ist im eigentlichen Sinne aber auch nicht der Absicht der virtuellen Sicht, denn sie ist besser geeignet, um beispielsweise die Organisation der Rechte für unterschiedliche Nutzer der Datenbank zu gewährleisten. Wenn man hingegen beispielweise in OLTP-Systemen die Notwendigkeit hat, dass man aggregierte Daten häufig zu der Analyse von bestimmten Daten benötigt, dann sind materialisierte Sichten nützlich. Man sollte allerdings vor allem die Performanceauswirkungen von diesen Sichten nicht unterschätzen und sich gut überlegen, wie häufig und zu welcher Zeit die Daten aktualisiert werden müssen.

# 6 Evaluation

TODO

# 7 Fazit

TODO

# Literatur


[1] 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).

[2] A. Kopytov. „Sysbench Github Repository.“ (2024), Adresse: https://github.com/akopytov/sysbench (besucht am 28. 10. 2024).

[3] 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.

[4] Shopify. „Mybench Github Repository.“ (2024), Adresse: https://github.com/Shopify/mybench (besucht am 28. 10. 2024).

[5] Shopify. „What is mybench?“ (2022), Adresse: https://shopify.github.io/mybench/introduction.html (besucht am 28. 10. 2024).

[6] Shopify. „Detailed design documentation.“ (2022), Adresse: https://shopify.github.io/mybench/detailed-design-doc.html#live-monitoring-user-interface (besucht am 28. 10. 2024).

[7] T. Williams, C. Kelley und many others, *Gnuplot 4.4: an interactive plotting program*, http://gnuplot.sourceforge.net/, März 2010.

[8] T. pandas development team, *pandas-dev/pandas: Pandas*, Version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. Adresse: https://doi.org/10.5281/zenodo.3509134.

[9] 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.

[10] GitHub. „Understanding GitHub Actions.“ (2025), Adresse: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions (besucht am 20. 01. 2025).

[11] 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).

[12] S. Luber. „Was ist ACID?“ (2018), Adresse: https://www.bigdata-insider.de/was-ist-acid-a-776182/ (besucht am 26. 01. 2025).

[13] DataScientest. „SQL TRIGGER zur Automatisierung deines DBMS.“ (2023), Adresse: https://datascientest.com/de/sql-trigger-zur-automatisierung-deines-dbms (besucht am 31. 01. 2025).

[14] A. Ouko. „SQL Materialized View: Verbesserung der Abfrageleistung.“ (2025), Adresse: https://www.datacamp.com/de/tutorial/sql-materialized-view (besucht am 26. 01. 2025).

[15] Oracle. „27.3.1 Trigger Syntax and Examples.“ (2018), Adresse: https://dev.mysql.com/doc/refman/8.4/en/trigger-syntax.html (besucht am 27. 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