HAW HAMBURG

BACHELORARBEIT

Entwurf 1: Performance – Optimierung von Datenbanken

vorgelegt am 07. Januar 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 Überblick

1.1 Einführung in Benchmarks

TODO

1.2 Measures

TODO

2 Tools

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 (TODO (Daniel): Kapitel mit Join Typ) betrachten wir ein beispielhaftes Projekt mit Sysbench, bei dem der Einfluss von unterschiedlichen Datentypen als Join-Operator zwischen zwei Tabellen verglichen wird. Wenn wir später die Performance von unterschiedliche Index-Typen betrachten, dann müssen wir nur an wenigen Stellen Veränderungen durchführen, die in dem Kapitel genauer besprochen werden.

Ein weiterer Vorteil von Sysbench ist, dass es als de facto Standard im Bereich der Datenbankbenchmarks angesehen wird [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, zwischen speichert, da diese gebraucht werden, um den Benchmarktest mit Sysbench zu starten. Nachdem das RDBMS gestartet worden ist, muss zudem eine Datenbank erstellt werden. Dies könnte unter anderem so aussehen:

Codeblock 2.1: Create Database

```
1 CREATE DATABASE sbtest;
```

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

- `--db-driver`: Gibt den Treiber für die Datenbank an, die Sysbench verwenden soll. In diesem Fall `mysql`, um die MySQL-Datenbank zu testen.
- `--mysql-host`: Der Hostname oder die IP-Adresse des MySQL-Servers. Standardmäßig wird `localhost` verwendet, wenn nichts angegeben wird.
- `--mysql-user`: Der Benutzername, mit dem Sysbench auf die MySQL-Datenbank zugreift.
- `--mysql-password`: Das Passwort für den MySQL-Benutzer. Falls der Benutzer kein Passwort hat oder der Zugriff über eine andere Authentifizierungsmethode erfolgt, kann dieses Argument weggelassen werden.
- `--mysql-db`: Der Name der MySQL-Datenbank, auf die zugegriffen wird. In diesem Beispiel `sbtest`.
- `--time`: Gibt die Laufzeit des Benchmarks in Sekunden an und muss immer mit angegeben werden.
- `--report-interval`: Gibt das Intervall in Sekunden an, in dem Zwischenergebnisse während des Tests ausgegeben werden. Sofern `--report-interval` nicht gesetzt wird, werden die Ergebnisse nur am Ende des Tests angezeigt.
- `--tables`: Die Anzahl der Tabellen, die für den Test erstellt werden sollen. Standardmäßig wird nur eine Tabelle erstellt.
- `--table-size`: Die Anzahl der Datensätze (Zeilen) pro Tabelle. Muss auch nicht zwingend 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="127.0.0.1"
DB_USER="root"
DB_PASS="password"
DB_NAME="sbtest"
TABLES=10
TABLE_SIZE=10000
DURATION=10
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"

# Create output directory
rm -rf "output"
mkdir -p "output"

# 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"
rm -rf "$OUTPUT_DIR/gnuplot"
mkdir -p "$OUTPUT_DIR/gnuplot"
echo "Generating plot with gnuplot..."
gnuplot $GNUPLOT_SCRIPT
echo "Plots generated with gnuplot"

# Python with Library Pandas
PYTHON_SCRIPT="YOUR_PATH_TO_PROJECT/generatePlot.py"
echo "Generating plots with pandas..."
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.

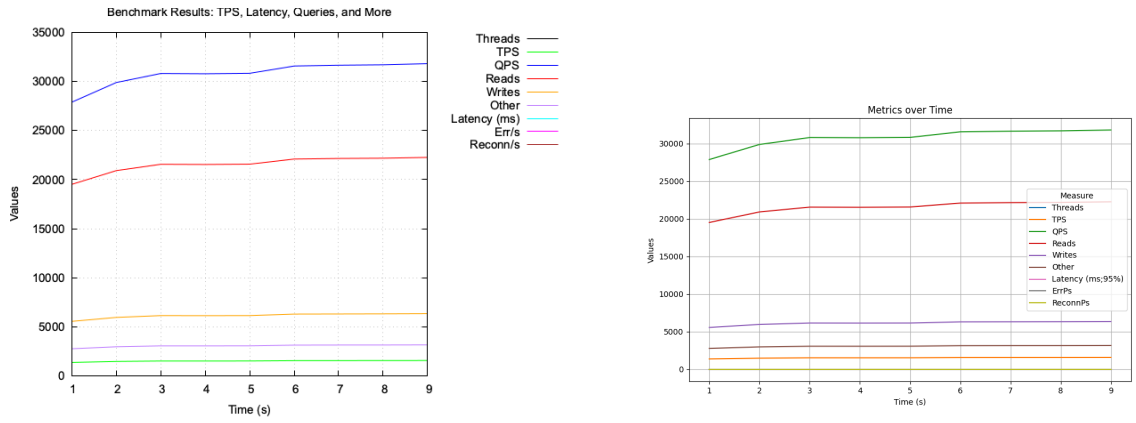


Abbildung 2.1: Die Grafik zeigt die erstellten Graphen mit Gnuplot (links) und Pandas (rechts)

- **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 Genauere Darstellung anhand eines Beispiels

In dem vorausgegangenen Kapitel Einführung in die Tools wurde das Tool Sysbench und seine Funktionsweise anhand eines Demo - Projekts näher erläutert. Damit die Reihenfolge und die Bedeutungen der unterschiedlichen Methoden (prepare → run → cleanup) sowie die

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

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

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

Codeblock 2.5: Create Table - Befehl für Tabelle Kunden

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

```
    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

```lua
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

```lua
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
        local name = string.format("Kunde_%d", i)
```

```
        local geburtstag = string.format("19%02d-%02d-%02d", math.random(50, 99), math.random(1, 12), math.random(1, 28))
        -- define 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:

```
1 local length = 10
```

Die Zeile mit der importierten Länge:

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

Den Orchestrator - Script ruft man wie folgt auf:

Codeblock 2.11: Befehl zum Ausführen des Orchestrator Skripts

```
./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 (TODO (Daniel): Index) erkennen, wann der Index verwendet wird und wann nicht.

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

Zu Beginn des Skripts werden die Umgebungsvariablen aus der Datei `db.env` geladen. Die Variablen helfen zum einen wie bei dem Demo - Beispiel bei die Datenbank - Verbindung und zum anderen können sich auch die Parameter der Benchmarks verändern. Danach werden die Parameter, die an das Skript übergeben wurden, überprüft. Beispielsweise wird 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 "${QUERY_INFO[@]}"; do
  IFS=: read -r QUERY_PATH MULTIPLE_KEYS <<< "$INFO"

  MAIN_SCRIPT="${QUERY_PATH}/$(basename "$QUERY_PATH").lua"
  INSERT_SCRIPT="${QUERY_PATH}/$(basename "$QUERY_PATH")_insert.lua"
  SELECT_SCRIPT="${QUERY_PATH}/$(basename "$QUERY_PATH")_select"
  LOG_DIR="$OUTPUT_DIR/logs/$(basename "$QUERY_PATH")"

  if [[ -n "$MULTIPLE_KEYS" ]]; then
    IFS=',' read -r -a KEYS <<< "$MULTIPLE_KEYS"

    combinations=$(generate_combinations "" "${KEYS[@]}")

    # Process each combination
    while IFS=',' read -r combination; do
        # Export key-value pairs for the current combination
        IFS=',' read -ra key_value_pairs <<< "$combination"
        for pair in "${key_value_pairs[@]}"; do
            key="${pair%%=*}"
            value="${pair#*=}"
            export "$(echo "$key" | tr '[:lower:]' '[:upper:]')=$value"
```

```
        done

        COMBINATION_NAME=$(echo "$combination" | tr ',' '_' | tr '=' '_')
        LOG_DIR_KEY_VALUE="$LOG_DIR/$COMBINATION_NAME"
        mkdir -p "$LOG_DIR_KEY_VALUE"

        RAW_RESULTS_FILE="${LOG_DIR_KEY_VALUE}/$(basename "$QUERY_PATH")_${COMBINATION_NAME}_prepare.log"
        run_benchmark "$MAIN_SCRIPT" "prepare" "$RAW_RESULTS_FILE" "" "$COMBINATION_NAME"

        process_script_benchmark "$QUERY_PATH" "$LOG_DIR_KEY_VALUE" "$INSERT_SCRIPT" "$SELECT_SCRIPT" "
    $COMBINATION_NAME"

        RAW_RESULTS_FILE="${LOG_DIR_KEY_VALUE}/$(basename "$QUERY_PATH")_${COMBINATION_NAME}_cleanup.log"
        run_benchmark "$MAIN_SCRIPT" "cleanup" "$RAW_RESULTS_FILE"
    done <<< "$combinations"
  else
    # Process normally when no keys specified
    mkdir -p "$LOG_DIR"
    RAW_RESULTS_FILE="$LOG_DIR/$(basename "$QUERY_PATH")_prepare.log"
    run_benchmark "$MAIN_SCRIPT" "prepare" "$RAW_RESULTS_FILE"

    process_script_benchmark "$QUERY_PATH" "$LOG_DIR" "$INSERT_SCRIPT" "$SELECT_SCRIPT"

    RAW_RESULTS_FILE="$LOG_DIR/$(basename "$QUERY_PATH")_cleanup.log"
    run_benchmark "$MAIN_SCRIPT" "cleanup" "$RAW_RESULTS_FILE"
  fi
done
```

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

Codeblock 2.13: Methode Process Script Benchmark

```
process_script_benchmark() {
  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

  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
}
```

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

Codeblock 2.14: Methode Run Benchmark

```
run_benchmark() {
  # For OTHER_ENVIRONMENT_VARIABLES => see demo
  sysbench --db-driver=mysql OTHER_ENVIRONMENT_VARIABLES "$SCRIPT_PATH" "$MODE" >> "$OUTPUT_FILE" 2>&1
  if [ $? -ne 0 ]; then
    echo "Benchmark failed for script $SCRIPT_PATH. Exiting."
    exit 1
  fi

  # Only extract data if the mode is "run"
  if [ "$MODE" == "run" ]; then
    extract_run_data "$RAW_RESULTS_FILE" "$SCRIPT_NAME"
    extract_statistics "$RAW_RESULTS_FILE" "$SCRIPT_NAME"
  fi
}
```

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

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

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

PNGs

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

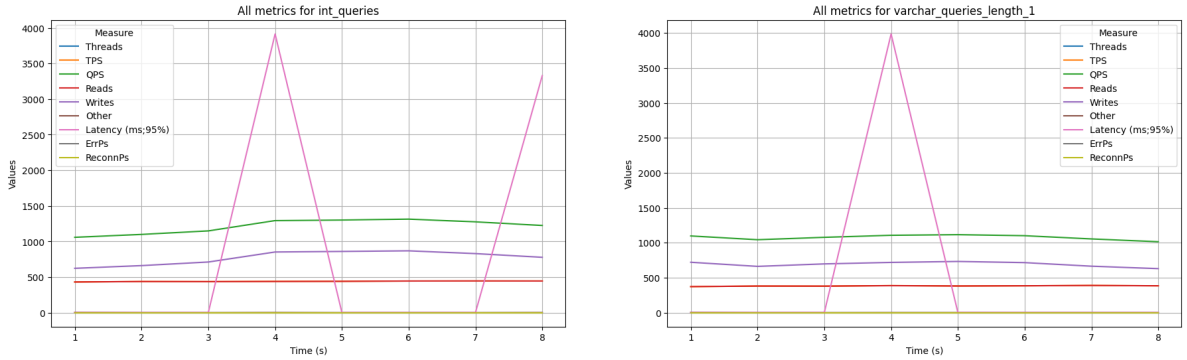


Abbildung 2.2: Die Grafik zeigt alle Metriken für die Skripte `int_queries` (links) und `varchar_queries_length_1` (rechts)

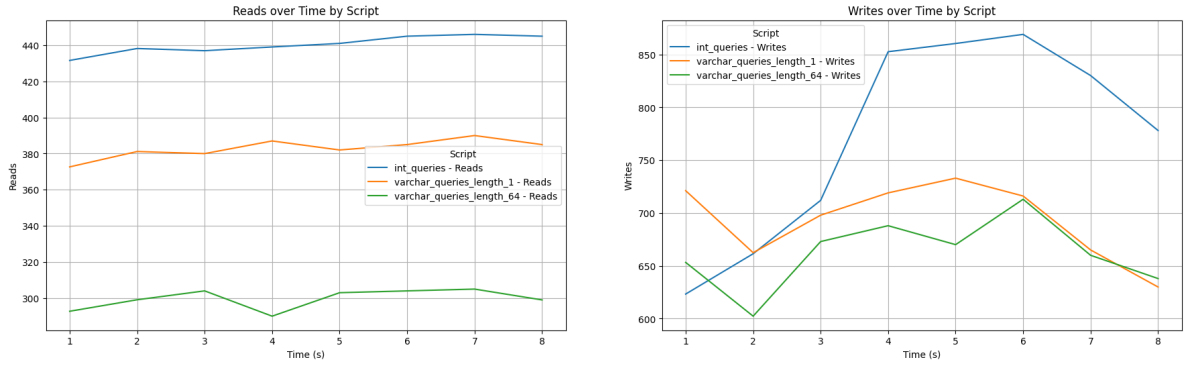


Abbildung 2.3: Die Grafik zeigt den Vergleich zwischen allen Skripten für die Metriken Reads (links) und Writes (rechts)

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

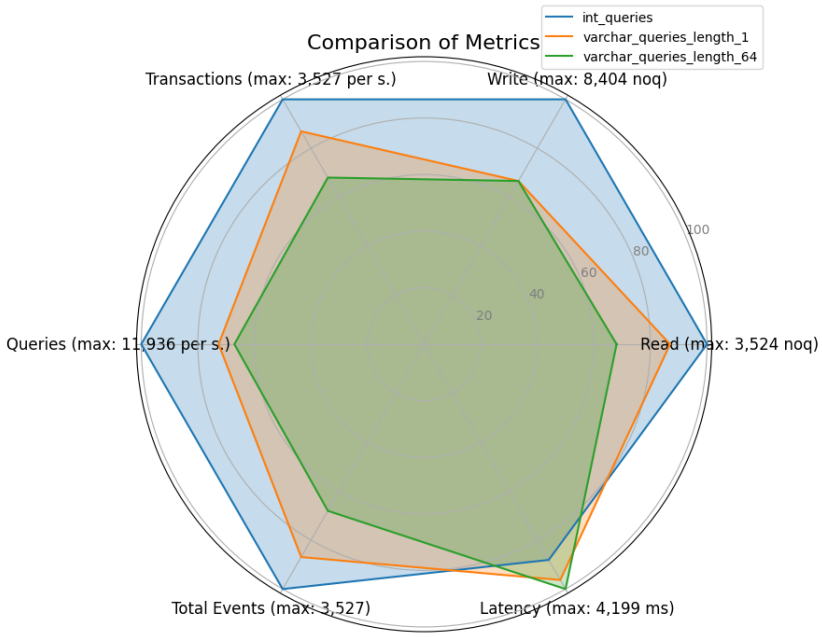


Abbildung 2.4: Darstellung der Skripte und 6 Metriken in einem Hexagon

der Beste ist.

3 Projektdurchführung

3.1 GitHub Action

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

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

Codeblock 3.1: JSON mit Konfiguration der Script

```
1 {
2   "join-type": {
3     "dirs": [
```

```
      "./Projects/Join_Typ/Scripts/int_queries",
      "./Projects/Join_Typ/Scripts/varchar_queries:length"
    ],
    "var": {"length": [1, 64]}
  }
}
```

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

Codeblock 3.2: Ausschnitt aus der Workflow - Datei

```yaml
name: Benchmark Workflow
on:
  push:
    paths:
      - 'Projects/**'
      - ...
jobs:
  benchmark:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        test-type: [high-count, low-count, b-tree-query-differences, selectivity-change, hash-query-differences, join-type, null-check, int-char, data-size]
    env:
      OTHER_ENVIRONMENT_VARIABLES
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
      - name: Load configuration from JSON
        run: # removed the definition and export of variables for simplicity reasons
      - name: Install dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y sysbench
          python -m pip install --upgrade pip
          pip install pandas matplotlib
      - name: Start MySQL container
        run: |
```

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

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

Zum einen kann man die zu installierenden Abhängigkeiten mithilfe des GitHub Caches ([10]) speichern. Dies bietet sich besonders an, da sich die Abhängigkeiten über die Workflows hinweg nur selten ändern. Falls sich doch etwas ändert, kann man beispielsweise die `requirements.txt`-Datei anpassen. Dadurch werden einmalig alle Abhängigkeiten neu installiert und anschließend im Cache abgelegt. Falls sich bis zum nächsten Workflow keine Änderungen an den Abhängigkeiten ergeben, wird der Cache automatisch genutzt. Der Zeitgewinn in unserem Beispiel ist jedoch nur gering und beträgt nur wenige Sekunden pro Workflow.

Codeblock 3.3: Speichern der Abhängigkeiten im Cache

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

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

die JSON - Datei, aber auch das Orchestrator - Skript und die darin verwendeten Python - Skripte. Die Ordner an sich, die in der JSON angegeben werden, die beeinflussen nur sich selbst und nicht die anderen Skripte. 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:

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

Nachdem wir unsere JSON geladen haben, machen wir nun nicht mehr direkt mit der Installation der Abhängigkeiten weiter, sondern davor hashen wir die unterschiedlichen Pfade und erstellen unseren Namen. Wenn es keinen Ordner mit dem gleichen Namen gibt, dann machen wir weiter wie bisher. Das einzige, was sich ändert, ist der Schritt vor dem Hochladen des gesamten Output-Ordners. Der vom Testtypen erzeugte Ordner muss zusammen mit seinem Namen hochgeladen werden, damit er im nächsten Workflow heruntergeladen werden kann, sofern die Hashes unverändert bleiben. Ist dies beim nächsten Workflow der Fall, dann muss der Ordner nur heruntergeladen werden und 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 ([10]). 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.

4 Optimierungen von Datentypen

4.1 Allgemeine Faktoren

TODO

4.2 Einzelne Datentypen und weitere Faktoren

TODO

Grafiken für Null

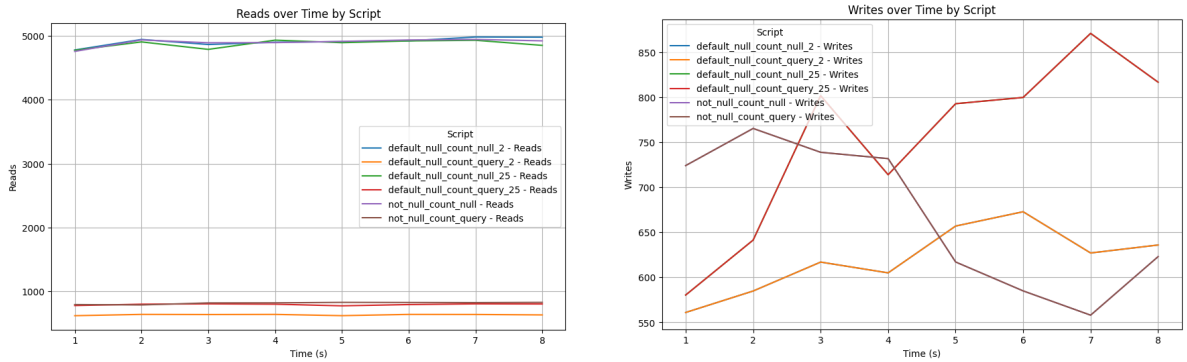


Abbildung 4.1: Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts)

Grafiken für Simple

Grafiken für Smaller

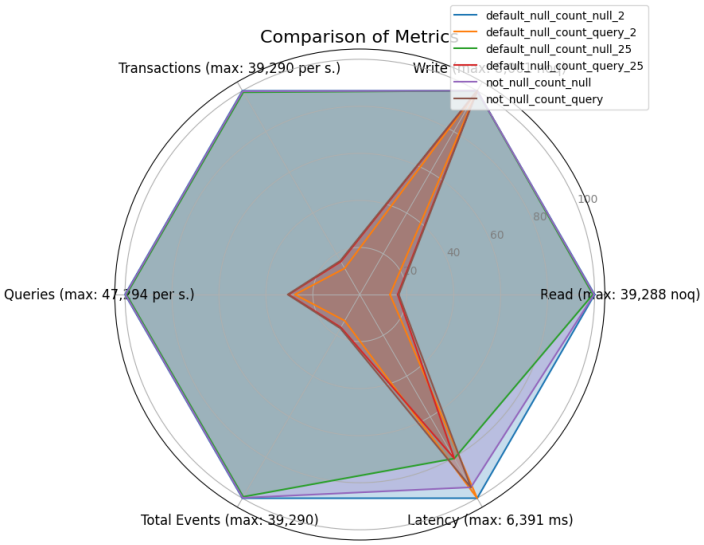


Abbildung 4.2: Grafik visualisiert die unterschiedlichen Skripte und 6 Metriken in einem Hexagon

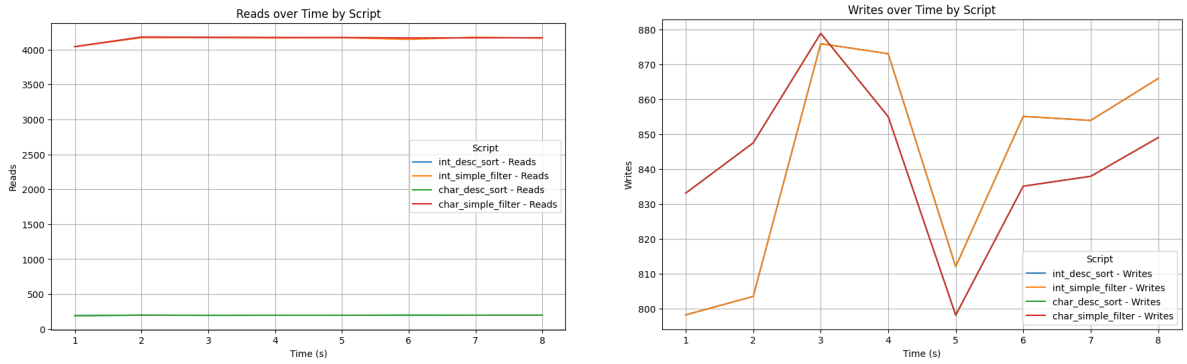


Abbildung 4.3: Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts)

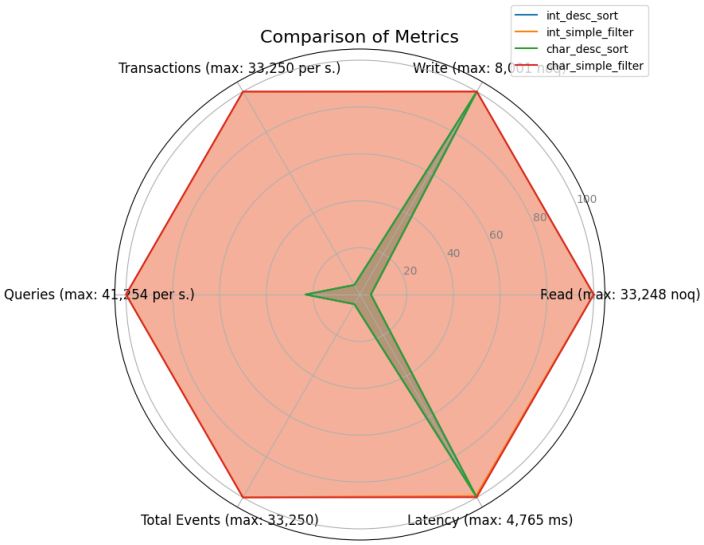


Abbildung 4.4: Grafik visualisiert die unterschiedlichen Skripte und 6 Metriken in einem Hexagon

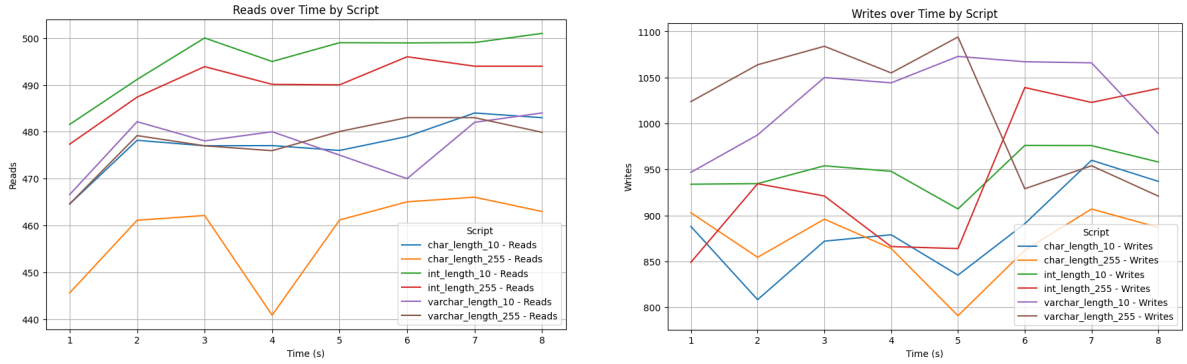


Abbildung 4.5: Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts)

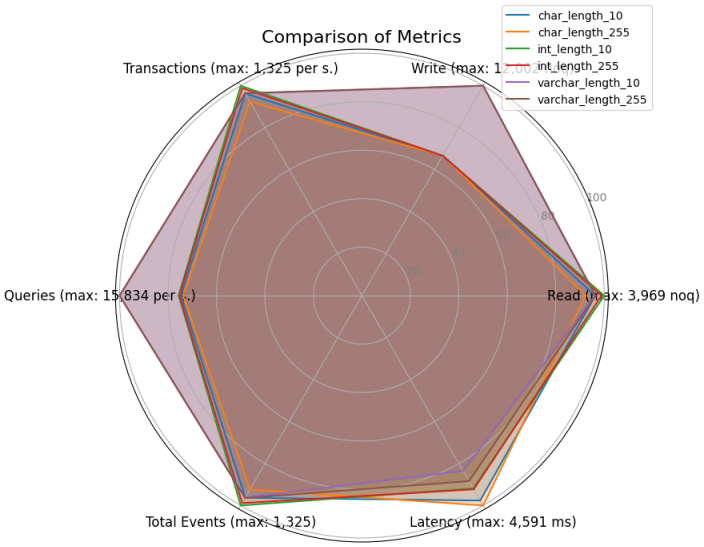


Abbildung 4.6: Grafik visualisiert die unterschiedlichen Skripte und 6 Metriken in einem Hexagon

5 Indexierung und Einfluss auf die Performance

5.1 Grundlagen der Indexierung

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

Codeblock 5.1: Variationen

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

TODO

5.2 B-Baum-Index

TODO

TODO

5.3 Hash - Index

TODO

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

Codeblock 5.2: Variationen

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

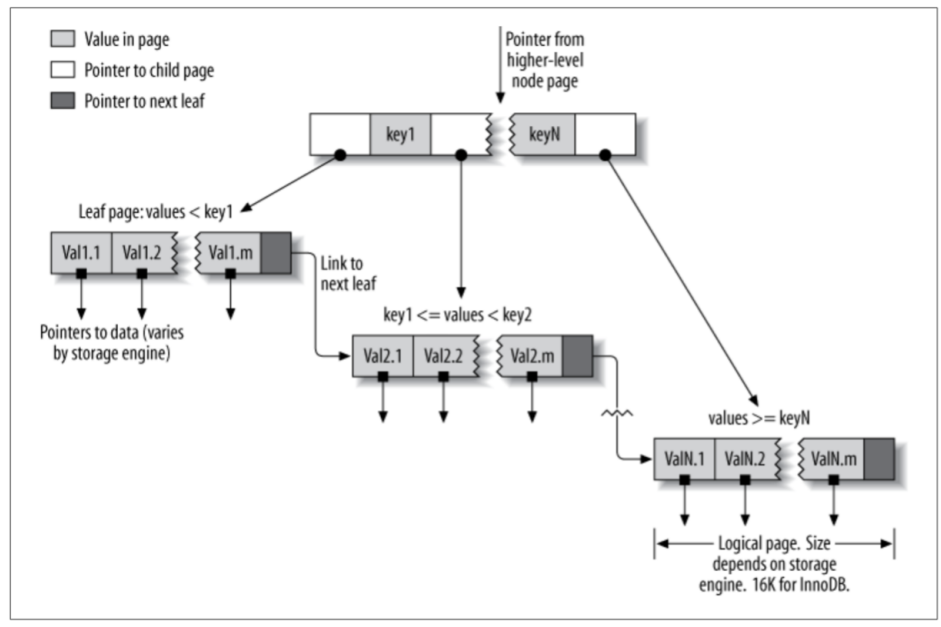


Abbildung 5.1: Darstellung des binären Baums mit Knoten und Blättern

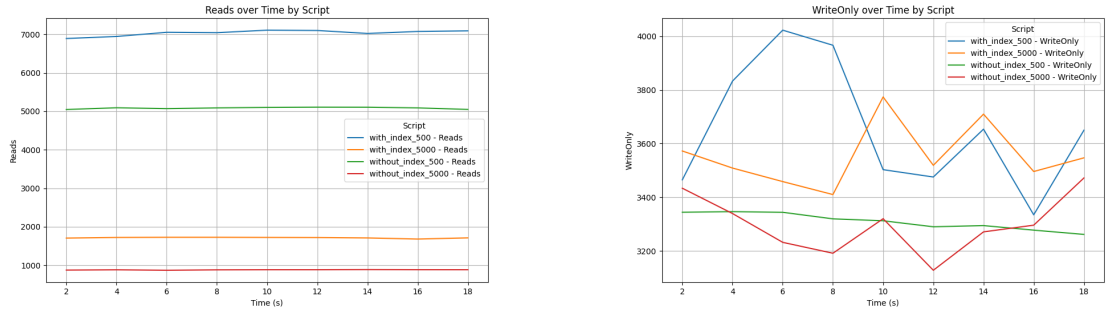


Abbildung 5.2: Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts) mit 500 bzw. 5000 Zeilen

TODO

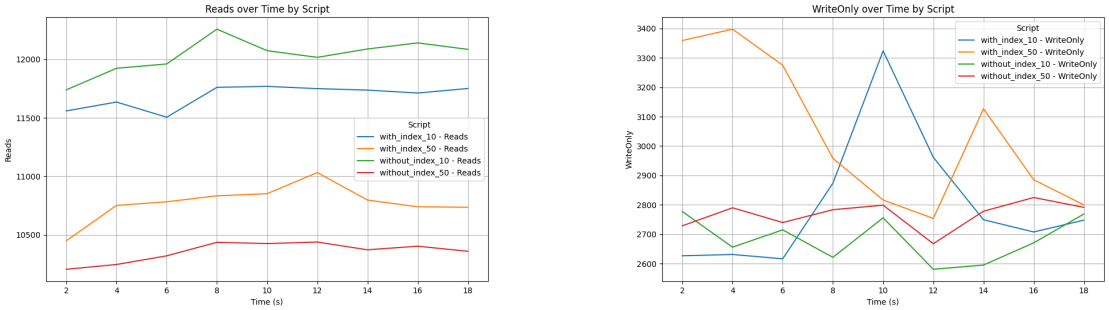


Abbildung 5.3: Grafik zeigt die verschiedenen Zeiten (in ms) für Readsabfragen (links) und Schreibbefehle (rechts) mit 10 bzw. 100 Zeilen dar

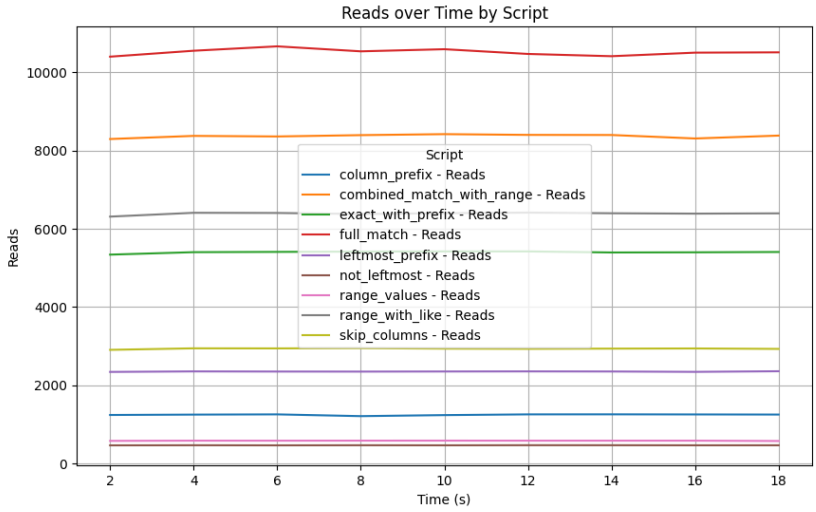
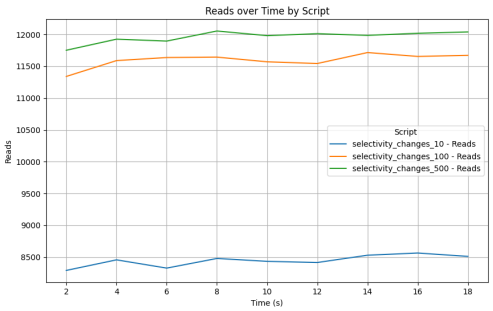
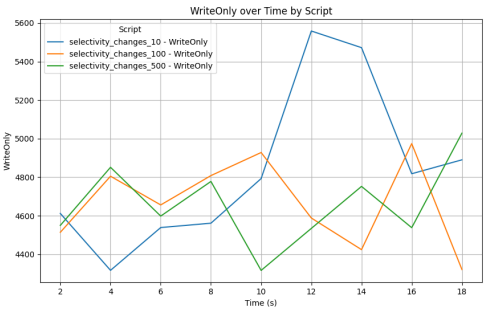


Abbildung 5.4: Grafik visualisiert die Unterschiede von verschieden Select - Queries auf dieselben Daten. Je nach Gewschiwndikeit greift der Index besser oder nicht



(a) Unterschiede von Readsabfragen



(b) Unterschiede von Schreibbefehlen

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

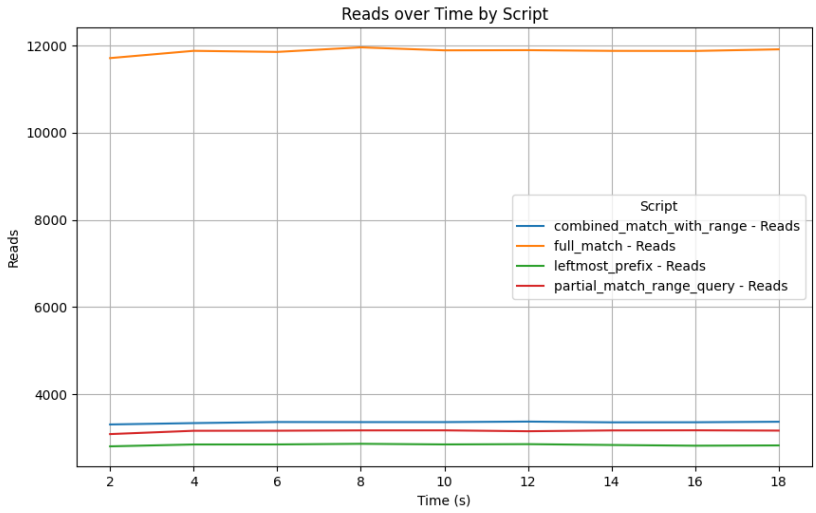


Abbildung 5.6: Grafik visualisiert die Unterschiede von verschieden Select - Queries auf dieselben Daten. Je nach Gewschiwndikeit greift der Index besser oder nicht

Literatur

[1] N. Reimers. „Virtuelle, dezidierte und Cloud-Server, MySQL-Benchmark mittels sysbench," besucht am 28. Okt. 2024. Adresse: https://www.webhosterwissen.de/know-how/server/mysql-benchmark-mittels-sysbench/.

[2] A. Kopytov. „Sysbench Github Repository," besucht am 28. Okt. 2024. Adresse: https://github.com/akopytov/sysbench.

[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," besucht am 28. Okt. 2024. Adresse: https://github.com/Shopify/mybench.

[5] Shopify. „What is mybench?" Besucht am 28. Okt. 2024. Adresse: https://shopify.github.io/mybench/introduction.html.

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

[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. „Caching dependencies to speed up workflows," besucht am 7. Jan. 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.

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