HAW HAMBURG

BACHELORARBEIT

Performance – Optimierung von Datenbanken

vorgelegt am 13. März 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

Relationale Datenbanken sind ein essenzieller Bestandteil moderner IT-Systeme und bilden die Grundlage für zahlreiche Anwendungen, die täglich von Millionen von Nutzern verwendet werden. Mit wachsender Datenmenge steigen jedoch die Antwortzeiten von Abfragen, was die Systemnutzung erschwert. Die Herausforderung besteht darin, geeignete Optimierungsstrategien zu finden, die sowohl Lese- als auch Schreiboperationen effizient gestalten und eine hohe Skalierbarkeit gewährleisten. Diese Arbeit untersucht verschiedene Ansätze zur Optimierung der Performance, darunter Datentypen, Indexierung, Views, Partitionierung und Replikation. Zur Analyse der Auswirkungen dieser Methoden wird das Tool Sysbench für Benchmarks eingesetzt. Die Ergebnisse zeigen, dass die Wahl des kleinstmöglichen Datentyps und die Verwendung von `NOT NULL`-Spalten die Leistung optimieren, indem sie Speicherplatz sparen. Hash-Indizes sind besonders bei exakten Schlüsselvergleichen effektiv, während B-Baum-Indizes vielseitigere Einsatzmöglichkeiten bieten. Materialisierte Sichten bieten Performancevorteile durch gespeicherte Abfrageergebnisse, während virtuelle Sichten Echtzeitdaten liefern, aber bei jedem Zugriff die Abfrage neu ausführen und daher langsamer sind. Bei großen Datenmengen kann Partitionierung eine effektive Lösung darstellen, während Replikation die Lastverteilung insbesondere bei hoher CPU-Last verbessert. Es gibt keine universelle Lösung, aber je nach Anwendungsfall können geeignete Konzepte ausgewählt, optimiert und auch miteinander kombiniert werden.

Abstract

Relational databases are an essential component of modern IT systems and form the foundation for numerous applications used daily by millions of users. However, as data volumes grow, query response times increase, making system usage more challenging. The challenge lies in identifying suitable optimization strategies that make both read and write operations efficient while ensuring high scalability. This paper examines various approaches to performance optimization, including data types, indexing, views, partitioning, and replication. The impact of these methods is analyzed through benchmarking with the Sysbench tool. The results show that choosing the smallest possible data type and using `NOT NULL` columns optimize performance by saving storage space. Hash indexes are particularly effective for exact key comparisons, while B-tree indexes offer more versatile applications. Materialized views provide performance benefits by storing query results, whereas virtual views deliver real-time data but execute the query anew with each access, making them slower. For large datasets, partitioning can be an effective solution, while replication improves load distribution, especially under high CPU load. There is no universal solution, but depending on the use case, suitable concepts can be selected, optimized, and even combined.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Diese Bachelorarbeit untersucht verschiedene Datenbankobjekte in relationalen Datenbanken. Dabei werden zunächst die einzelnen Konzepte detailliert erläutert, bevor ihre praktische Umsetzung in einem Datenbankmanagementsystem erfolgt. Um die Effizienz der verschiedenen Implementierungen bewerten zu können, wird eine geeignete Messmethode benötigt. Dafür sind Benchmarks das optimale Werkzeug. Anhand der Benchmark-Ergebnisse lassen sich Rückschlüsse auf die Leistungsfähigkeit der untersuchten Ansätze ziehen. Dieses Kapitel behandelt die Grundlagen und Typen von Benchmarks, die relevanten Kennzahlen sowie die Auswahl geeigneter Tools für eine korrekte Durchführung.

1.1 Einführung in Benchmarks

Benchmarks dienen dazu, das Verhalten eines Systems unter Last praktisch und effektiv zu untersuchen. Die wichtigste Erkenntnis, die aus Benchmarks gewonnen werden kann, ist die Identifikation von Problemen und Fehlern, die systematisch dokumentiert und nach Priorität bearbeitet werden sollten. Das Ziel von Benchmarks ist die Reduzierung und Bewertung von unerwünschtem Verhalten sowie die Analyse, wie sich das System derzeit und unter simulierten, zukünftigen und anspruchsvolleren Bedingungen verhalten könnte.

Es gibt zwei verschiedene Techniken für Benchmarks (**schwartz2012high**). Die erste zielt darauf ab, die Applikation als Ganzes zu testen (engl. full-stack). Dabei wird nicht nur die Datenbank getestet, sondern die gesamte Applikation, einschließlich des Webservers, des Netzwerks und des Applikationscodes. Der Grundgedanke dahinter ist, dass der Nutzer genauso lange auf eine Antwort warten muss, wie das gesamte System für die Verarbeitung der Anfrage benötigt. Um den Kunden kürzere Wartezeiten zu ermöglichen, sollte das Ziel sein, diese Zeit möglichst zu verkürzen. Es kann dabei auch vorkommen, dass das Datenbanksystem nicht das Bottleneck ist.[1] Full-Stack-Benchmarks haben jedoch auch Nachteile, denn sie sind schwieriger zu erstellen und insbesondere schwieriger korrekt einzurichten.

[1] Ein Bottleneck ist ein Engpass beim Transport von Daten oder Waren, der einen maßgeblichen Einfluss auf die Abarbeitungsgeschwindigkeit hat. Wenn der Bottleneck weiterhin bestehen bleibt, führen Optimierungsversuche an anderen Stellen nur zu marginalen oder gar keinen messbaren Verbesserungen der Gesamtleistung (**bottleneck**).

Die zweite Art von Benchmarks sind sogenannte Single-Component-Benchmarks, die verwendet werden, wenn man lediglich verschiedene Schemas und Abfragen im DBMS auf ihre Performance testen möchte. Sie analysieren ein spezifisches Problem in der Applikation und sind deutlich einfacher zu erstellen. Ein weiterer Vorteil besteht darin, dass nur ein Teil des gesamten Systems getestet wird, wodurch die Antwortzeiten kürzer sind und man schneller Ergebnisse erhält. Da sich diese Bachelorarbeit ausschließlich mit den verschiedenen Objekten in Datenbanken beschäftigt, habe ich mich für einen Single-Component-Benchmark entschieden.

Das Gefährliche bei der Verwendung von Benchmarks ist, dass schlechte Designentscheidungen zu falschen Interpretationen des Systems führen. Ein möglicher Grund dafür kann sein, dass die Ergebnisse nicht die Realität wiederspiegeln. Deshalb ist es wichtig, dass die Größe des Datensatzes und des Workloads realistisch sind. Idealerweise verwendet man einen Snapshot des tatsächlichen produktiven Datensatzes.[2] Weil in dieser Arbeit keine echten Produktionsdaten zur Verfügung stehen, müssen die Daten und der Workload nicht durch Snapshots, sondern zufällig generiert werden. Ein Problem bei der Verwendung zufällig erzeugter Werte ist die unrealistisch gleichmäßige Verteilung der Datensätze. Im Gegensatz dazu können in der Realität Hotspots auftreten, die die Verteilung erheblich beeinflussen. In den folgenden Kapiteln werden die Ansätze möglichst allgemein erläutert, weshalb dieser Kompromiss in Kauf genommen werden kann. Für die Analyse eines speziellen Systems sollten jedoch Snapshots aus der Produktivumgebung verwendet werden, um fehlerhafte Schlüsse zu vermeiden.

Ein weiterer Fehler, der noch bei Benchmarks auftreten kann, ist das falsche Nachstellen des tatsächlichen Benutzerverhaltens. Zudem sollte darauf geachtet werden, dass Caching-Effekte nicht zu falschen Annahmen über die Performance führen. Teilweise wird auch die Aufwärmphase des Systems vollständig ignoriert und kurze Benchmarks können die Performance ebenfalls verfälschen.

Um zuverlässige Ergebnisse zu erzielen, sollten Benchmarks über einen ausreichend langen Zeitraum durchgeführt werden, um den stabilen Zustand des Systems zu erfassen. Dies gilt besonders für Server mit großen Datenmengen und viel Speicher. Zudem muss gewährleistet sein, dass der Benchmark reproduzierbar ist, da unzureichende oder fehlerhafte Tests keine aussagekräftigen Ergebnisse liefern. Zu guter Letzt empfiehlt es die Ergebnisse eines Benchmarks in einem Diagramm darzustellen, da bestimmte Phänomene oft nur so erkennbar werden und nicht in tabellarischer Form sichtbar sind.

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

1.2 Kennzahlen

Bevor ein geeignetes Benchmark-Tool ausgewählt wird, sollte zunächst geklärt werden, welche Kennzahlen im Datenbankkontext relevant sind und welche davon für die jeweiligen Zwecke von besonderem Interesse sind. Das Benchmark-Tool, für das man sich entscheidet, muss in der Lage sein, diese Kennzahlen zu erfassen und zugänglich zu machen.

Die erste Kennzahl, die betrachtet wird, ist der Durchsatz (engl. throughput). Der Durchsatz gibt an, wie viele Transaktionen pro Zeiteinheit durchgeführt werden, wobei ein höherer Wert eine bessere Performance zur Folge hat. Üblicherweise wird als Einheit Transaktionen pro Sekunde verwendet, gelegentlich auch Transaktionen pro Minute. Man kann Transaktionen auch in verschiedene Kategorien unterteilen, wie beispielsweise Lese- und Schreibtransaktionen. Diese Unterscheidung ist wichtig, da bestimmte Implementierungen schnellere Lese-, aber langsamere Schreibtransaktionen zur Folge haben können.

Die nächste Metrik ist die Antwortzeit (engl. latency), die die gesamte Zeit misst, die für eine Abfrage benötigt wird. Abhängig von der Applikation kann sie in Mikrosekunden (µs), Millisekunden (ms), Sekunden oder sogar Minuten angegeben werden. Oft wird die Latenz in einer aggregierten Form angegeben, wie beispielsweise dem Durchschnitt, Maximum, Minimum oder Perzentilen. Bei der Betrachtung von Latenzzeiten macht es aber wenig Sinn, Maximal- oder Minimalwerte zu betrachten, da diese oft Ausreißer sind und die allgemeine Performance nicht repräsentieren. Daher nutzt man eher Perzentile bei den Antwortzeiten. Perzentile bezeichnen den Wert, unter dem ein bestimmter Prozentsatz der gemessenen Latenzzeiten liegt. Wenn beispielsweise das 95. Perzentil der Antwortzeit bei 5 ms liegt, bedeutet dies, dass 95% der Abfragen in weniger als 5 ms abgeschlossen sind (**perzentil_erklaerung**).

Eine weitere Kennzahl ist die Gleichzeitigkeit (engl. concurrency), die angibt, wie viele Anfragen gleichzeitig bearbeitet werden können. Ein genauerer Ansatz zur Messung der Gleichzeitigkeit auf dem Webserver besteht darin, die Anzahl der gleichzeitig ausgeführten Anfragen zu einem bestimmten Zeitpunkt zu bestimmen. Es kann auch geprüft werden, ob der Durchsatz sinkt oder die Antwortzeiten steigen, wenn die Gleichzeitigkeit zunimmt. Beispielsweise könnte eine Website mit 50.000 gleichzeitigen Benutzern nur 10 oder 15 gleichzeitige Abfragen erfordern. Ein weiterer wichtiger Messwert, der die Leistung bei mehreren Nutzern beschreibt, ist die Skalierbarkeit (engl. scalability). Sie gibt an, wie sich das Verhalten des Systems verändert, wenn die Anzahl der Benutzer oder die Größe der Datenbank steigt. In einem idealen System würden doppelt so viele Abfragen bearbeitet werden, wenn doppelt so viele „Arbeiter“ versuchen, die Aufgaben zu erledigen.

Es gibt noch zahlreiche weitere Messgrößen, wie beispielsweise die Verfügbarkeit oder die CPU-Auslastung. Auf Letztere wird im Kapitel 7 näher eingegangen. Für das Benchmark-Tool sind die Metriken Durchsatz und Antwortzeit unverzichtbar und sollten daher unbedingt

berücksichtigt werden. Das Tool sollte auch dazu in der Lage sein, zwischen Lese- und Schreibtransaktionen zu unterscheiden. Die anderen Metriken sind vor allem im Zusammenhang mit Mehrbenutzer-Systemen wichtig und nehmen daher in den meisten Kapiteln dieser Arbeit eine untergeordnete Rolle ein.

1.3 Auswahl der Tools

Zu Beginn muss ein geeignetes relationales Datenbankmanagementsystem ausgewählt werden. In dieser Bachelorarbeit wird MySQL in der Version 8.0 verwendet, das erstmals am 19. April 2018 veröffentlicht wurde (**mysql_release**). Die aktuellste eingesetzte Version ist 8.0.41. Im Kapitel 5 wird zudem das DBMS PostgreSQL verwendet, um ein spezifisches Konzept zu untersuchen. Dieses Konzept wird mit MySQL verglichen, da MySQL keine native Implementierung dafür bereitstellt. Das ist aber die einzige Ausnahme, denn der Schwerpunkt im weiteren Verlauf der Arbeit wird überwiegend auf MySQL liegen.

Die Grundlage dieser Bachelorarbeit bildet die Untersuchung des Verhaltens der MySQL-Datenbank (**sysbench_mysql**) im Hinblick auf verschiedene Konzepte mithilfe eines zentralen Benchmark-Tools. Nach eingehender Überlegung habe ich mich für Sysbench (**sysbench_repo**) entschieden. Sysbench ist ein Open-Source-Tool, das ein skriptfähiges und multi-threaded Benchmark-Tool ist und auf LuaJIT basiert (**schwartz2012high**). Es wird hauptsächlich für Datenbankbenchmarks verwendet, kann jedoch auch dazu eingesetzt werden, um beliebig komplexe Arbeitslasten zu erstellen, die keinen Datenbankserver erfordern. Das Tool erfasst verschiedene Metriken, die im vorherigen Kapitel vorgestellt wurden, wie etwa Transaktionen pro Sekunde und Latenz. Außerdem kann genauer spezifiziert werden, wie oft diese Metriken geloggt werden sollen. Ein weiterer Vorteil von Sysbench ist, dass es nicht auf ein einzelnes Datenbanksystem beschränkt ist, sondern die Auswahl aus mehreren DBMS ermöglicht, darunter auch PostgreSQL.

Bei der Auswahl des Benchmark-Tools habe ich auch andere Optionen wie Benchbase (**DifallahPCC13**) und Mybench (**mybench_repo**) in Erwägung gezogen. Im Vergleich zu diesen Tools bietet Sysbench jedoch eine deutlich höhere Skriptfähigkeit und Flexibilität. Das bedeutet, dass Sysbench in großem Umfang über Skripte gesteuert werden kann, was eine benutzerdefinierte Konfiguration der Tests ermöglicht. Allerdings ist die Verwendung von Sysbench im ersten Projekt aufwendiger, da die Skripte von Grund auf neu erstellt werden müssen. Sobald jedoch ein Projekt einmal eingerichtet ist, können viele Aspekte übernommen und präzise sowie schnelle Änderungen vorgenommen werden. Dieser Vorteil wird im Kapitel 2.2 näher erklärt.

Sysbench zeichnet sich zudem dadurch aus, dass es als de facto Standard im Bereich der Datenbankbenchmarks gilt (**mybench_comparison**). Durch diese Positionierung im Markt

gibt es viele aktive Nutzer und dadurch bedingt viele verfügbaren Ressourcen. Ein Vorteil der anderen Tools besteht jedoch in der präziseren Steuerung der Ergebnisraten und Transaktionen im Vergleich zu Sysbench. Zudem beschränkt sich Sysbench hinsichtlich des Outputs auf das Wesentliche, da es lediglich eine Reihe von Log-Dateien erzeugt. Die Visualisierung der Ergebnisse muss vom Benutzer selbst mithilfe anderer Tools umgesetzt werden. Anders sieht das bei dem Tool Mybench aus, da dort die Möglichkeit besteht, in Echtzeit umfassende Abbildungen anzuzeigen (**mybench_user_interface**). Trotz dieses Features habe ich mich aufgrund der hohen Anpassbarkeit sowie der Stellung als de facto Standard für Sysbench entschieden.

Auf die Erstellung von Grafiken sollte aber auch mit Sysbench nicht verzichtet werden. Durch Abbildungen lassen sich Entwicklungen im Zeitverlauf wesentlich besser erkennen als in einer Log-Datei. Anhand der reinen Zahlen in einem Log lassen sich möglicherweise einige Trends erkennen, doch vor allem zyklische Schwankungen sind ohne Grafiken schwer zu identifizieren. Mit Graphen, die eine Zeitachse enthalten, werden Zyklen sofort erkennbar und der Vergleich unterschiedlicher Messungen wird wesentlich erleichtert. Um die Kennzahlen, die mithilfe von Sysbench ermittelt worden sind, in eine grafische Darstellung umzuwandeln, gibt es unterschiedliche Tools. Eine erste Möglichkeit bietet Gnuplot (**gnuplot**), das sich gut für die Darstellung von CSV-Dateien eignet. Wenn jedoch nur bestimmte Spalten der Tabelle angezeigt werden sollen, stößt man schnell an seine Grenzen. Aus diesem Grund habe ich mich mit einem Python-Script für eine flexiblere Alternative entschieden. Für die grafische Darstellung kommen dabei die Bibliotheken pandas (**reback2020pandas**) und matplotlib.pyplot (**hunter_2007**) zum Einsatz. Die genaue Verwendung von Sysbench wird im nächsten Kapitel erklärt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der Bachelorarbeit betrachtet, die in den späteren Kapiteln für die Durchführung der Benchmark-Tests und Analysen erforderlich sind. Zunächst werden die einzelnen Schritte dargelegt, um die im vorherigen Kapitel 1 ausgewählten Tools korrekt zu verwenden. Besonders beim Benchmark-Tool werden die verschiedenen Argumente untersucht, die übergeben werden können und es wird anhand eines kurzen Beispiels gezeigt, wie die Resultate dieses Tools aussehen könnten. Danach wird eine komplexere Demonstration betrachtet, die bei späteren Tests wiederverwendet werden kann. Zu guter Letzt wird gezeigt, wie GitHub Actions funktionieren, uns bei den Benchmark-Tests Aufwand ersparen und wie die Workflows sowohl zeitlich als auch ressourcenschonend optimiert werden können.

2.1 Einführung in die Tools

Zuallererst 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, dass man sich die Zugangsdaten, bestehend aus Benutzer- und Passwortdaten, speichert, da diese gebraucht werden, um den Benchmark-Test mit Sysbench zu starten. Nachdem das RDBMS gestartet worden ist, muss zunächst eine Datenbank erstellt werden. Das könnte beispielsweise folgendermaßen aussehen:

```
1 CREATE DATABASE sbtest;
```

Zusätzlich zu erfolgreichen Erstellung der Datenbank muss das Tool Sysbench installiert werden. Auf einem Linux-basierten System kann Sysbench wie folgt installiert werden. Wenn man das Betriebssystem macOS verwendet, muss `sudo apt` durch `brew` ersetzt werden.

```
1 sudo apt install sysbench
```

Damit auch die Grafiken erstellt werden können, müssen die Tools `Gnuplot` und `Pandas` in Kombination mit `matplotlib` installiert werden. Auch hier kann `sudo apt` durch `brew` ersetzt werden, wenn macOS verwendet wird. Es kann sein, dass `pip3` anstelle von `pip` verwendet werden muss.

```
pip install pandas matplotlib
sudo apt install gnuplot
```

Im nächsten Schritt wird sicht mit dem Tool Sysbench näher vertraut gemacht. Dazu werden die verschiedenen Argumente, die beim Aufruf mitgegeben werden können oder müssen, durchgegangen und kurz erklärt:

- **`db-driver`**: Treiber der Datenbank, in diesem Fall **`mysql`**
- **`mysql-host`**: Hostname oder IP-Adresse des Servers (Standard: `localhost`)
- **`mysql-user`**: Benutzername der Datenbank
- **`mysql-password`**: Passwort des DB-Benutzers (kann weggelassen werden, wenn keine Authentifizierung erforderlich ist)
- **`mysql-db`**: Name der zu verwendenden Datenbank, bei uns: **`sbtest`**
- **`time`**: Laufzeit des Benchmarks in Sekunden und ist verpflichtend
- **`report-interval`**: Intervall in Sekunden, in dem Zwischenergebnisse angezeigt werden (Standard: nur Gesamtstatistiken am Ende)
- **`tables`**: Anzahl der zu erstellenden Tabellen (Standard: 1)
- **`table-size`**: Anzahl der Datensätze pro Tabelle (optional)

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

1. Wie im Abschnitt 1.3 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.

Um die korrekte Verwendung des Tools zu überprüfen, wird ein kurzes Demo-Beispiel betrachtet, bei dem vordefinierte Testtypen von Sysbench genutzt werden. Auf diese Weise kann man schnell kontrollieren, ob die Einrichtung des Tools korrekt ist, ohne dafür SQL-Befehle oder Lua-Skripte für die eigenen Bedürfnisse zu schreiben. Es stehen verschiedene Testtypen zur Auswahl, wie das Einfügen von Daten (**oltp_insert**), das Abfragen von Daten (**oltp_read_only**) oder beides (**oltp_read_write**). Als letztes müssen die unterschiedlichen Methoden aufgelistet werden, um sie mit den Testtypen kombinieren zu können:

- **prepare**: Bereitet die Datenbank für den Test vor, u.a. das Erstellen der Tabellen.
- **run**: Ist die Ausführungsphase des Tests. Je nach Testtyp führt diese Methode die spezifizierten Operationen aus, wie etwa **oltp_read_write**. Dabei wird die Performance der Datenbank unter der angegebenen Arbeitslast gemessen.

- **cleanup**: Diese Methode stellt die Datenbank in ihren ursprünglichen Zustand zurück und stellt sicher, dass keine Testdaten zurückbleiben.

Für das Demo-Beispiel wird der Testtyp **oltp_read_write** ausgewählt und mit allen Methoden kombiniert. Für die Methode `RUN` würde die Query so aussehen, wobei `YOUR_USER` und `YOUR_PASSWORD` durch die tatsächlichen Benutzerdaten der verwendeten Datenbank 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. Daher muss vor der Ausführung der `RUN`-Methode zunächst die `PREPARE`-Methode durchgeführt werden. Um die Datenbank wieder in den Ausgangszustand zu versetzen, muss nach dem `oltp_read_write`-Testtyp auch die `CLEANUP`-Methode aufgerufen werden. Um sich die manuelle Ausführung dieser drei Befehle in der korrekten Reihenfolge zu sparen, bietet es sich an, ein Shell-Script zu schreiben, indem die Methoden nacheinander aufgerufen werden.

Codeblock 2.1: Ausführung der Sysbench-Methoden in korrekten Reihenfolge

```
# Define necessary variables: DB_HOST, DB_USER, DB_PASS, DB_NAME, TABLES, TABLE_SIZE, DURATION,
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
sysbench oltp_read_write $SYSBENCH_OPTS prepare >> "$RAW_RESULTS_FILE" 2>&1

# Run the benchmark
sysbench oltp_read_write $SYSBENCH_OPTS --time=$DURATION --threads=1 --report-interval=1 run >> "$RAW_RESULTS_FILE"
     2>&1

# Cleanup the database
sysbench oltp_read_write $SYSBENCH_OPTS cleanup >> "$RAW_RESULTS_FILE" 2>&1
```

Die Ergebnisse werden nun der Log-Datei (unter output/sysbench.log) gespeichert, aber uns fehlt noch die Erstellung der Graphen. Um die Erstellung zu vereinfachen, bietet es

sich an, die Kennzahlen aus der Log-Datei zu extrahieren und die Werte mit korrekten Spaltenüberschriften in einer CSV-Datei zu speichern. Dies geht mit dem Shell-Kommando `grep`:

Codeblock 2.2: Extraktion der Ergebnisse aus der Log-Datei in eine Tabelle

```
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 $1}')
    # Extract other measures
    latency=$(echo $line | awk -F 'lat \\(ms,95%\\): ' '{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."
```

Der letzte Schritt ist die Erstellung der Graphen mithilfe 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 für die Visualisierung ist, 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. In der Abbildung 2.1 sind die Ergebnisse der Grapherstellung zu sehen.

Codeblock 2.3: Erstellung der Graphen aus der CSV-Datei

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

# Gnuplot
GNUPLOT_SCRIPT="/Users/danielmendes/Desktop/Bachelorarbeit/Repo/plot_sysbench.gp"
gnuplot $GNUPLOT_SCRIPT
echo "Plots generated with gnuplot"

# Python with Library Pandas
PYTHON_SCRIPT="/Users/danielmendes/Desktop/Bachelorarbeit/Repo/generatePlot.py"
python3 "$PYTHON_SCRIPT" "$OUTPUT_FILE"
echo "Plots generated with pandas"
```

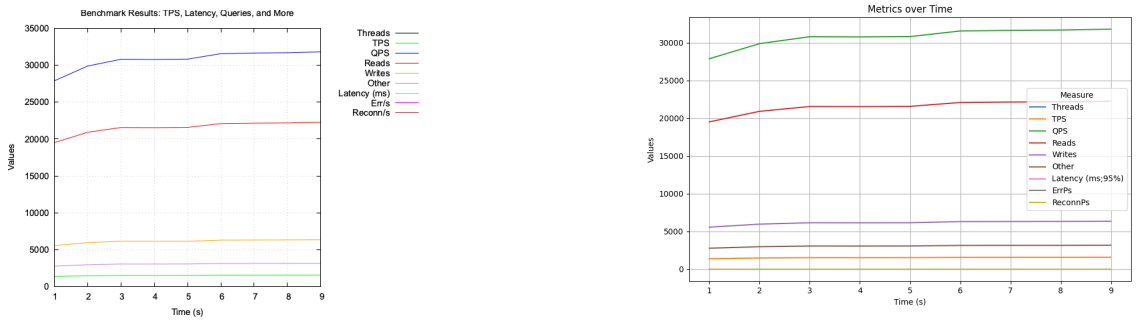


Abbildung 2.1: Grafik zeigt Erstellung mit Gnuplot (links) und Pandas (rechts)

Die Metriken in der Abbildung sind: Transaktionen, Abfragen, Fehler und Wiederverbindungen pro Sekunde (engl. TPS, QPS, ErrPs, ReconnPs), Anzahl der Operationen (engl. Reads, Writes, Other), die Latenz im 95. Perzentil und die Anzahl der Threads.

2.2 Projektaufbau mit Beispiel

In dem vorausgegangenen Abschnitt 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 der Grafiken deutlich geworden. Das bisherige Problem besteht jedoch darin, dass bei dem dargelegten Beispiel keine Kontrolle über die getesteten Daten besteht. Wenn man sich die Logs genauer anschaut, dann zeigt sich, dass zwar über die Parameter des Sysbench-Befehls die Anzahl der erstellten Tabellen und eingefügten Datensätze von außen gesteuert werden kann, aber die genaue Implementierung auf diese Weise nicht verändert werden kann. 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 zu erklären, wird ein Beispiel angeschaut, bei dem zwei Tabellen erstellt und mit zufälligen Testdaten befüllt werden. Die Abfrage, die auf Performance getestet werden soll, ist das Verbinden (Joinen) dieser beiden Tabellen. In diesem Fall wird eine Kundentabelle mit Name, Geburtstag und Adresse sowie eine Bestelltabelle mit Artikeldetails, Bestelldatum und einem Bezug zu dem Kunden, der die Bestellung aufgibt, erstellt. Damit nicht nur ein Beispiel dargestellt wird, wird ein Vergleich zwischen zwei verschiedenen Implementierungen benötigt. Der Unterschied zwischen den beiden besteht darin, dass die Tabelle in der einen Version eine Kundennummer vom Typ `INT` enthält, während sie in der anderen vom Typ `VARCHAR` ist. Da Verbundoperationen aufwendig sind, wird angenommen, dass der speichereffizientere Typ `INT` Performancevorteile bietet. Dies gilt es nun mit Benchmark-Tests genauer zu untersuchen.

Für die Durchführung der Benchmarks wird zunächst unabhängig von Sysbench und den Lua-Skripten mit der Spezifizierung der Tabellen begonnen, die erstellt werden sollen. Dies muss einmal mit der Kundennummer und einmal mit dem Namen als Fremdschlüssel in der Bestelltabelle durchgeführt werden. Damit müssen insgesamt vier unterschiedliche `CREATE TABLE`-Befehle umgesetzt werden. So sehen die `CREATE TABLE`-Ausdrücke für den Fall mit `INT` aus:

Codeblock 2.4: Create Table-Befehl für Tabelle Kunden

```
1 CREATE TABLE KUNDEN (
2     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 2.5: 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 diese Befehle in der `prepare()`-Funktion verwendet werden. Dafür müssen einfach die `CREATE TABLE`-Befehle an die Datenbank gesendet werden. Wenn bestimmte Indexe oder andere Datenbankstrukturen erstellt werden sollen, müsste dies ebenfalls in dieser Funktion erfolgen. Dies ist ein Auszug aus der `Prepare`-Funktion:

Codeblock 2.6: Lua-Script für die Erstellung der Tabellen

```
local con = sysbench.sql.driver():connect()
function prepare()
    local create_kunden_query = [[
        CREATE TABLE KUNDEN (...);
    ]]
    local create_bestellung_query = [[
        CREATE TABLE BESTELLUNG (...);
    ]]

    con:query(create_kunden_query)
    con:query(create_bestellung_query)
    print("Tables KUNDEN und BESTELLUNG have been successfully created")
end
```

Wenn die Datenbank beispielsweise in einer Produktivumgebung läuft, dann wollen wir, dass Benchmarks möglichst wenig Einfluss auf sie haben. Damit ist es das Ziel, dass die Datenbank möglichst nach dem Durchlauf wieder in ihrem Anfangszustand ist. Außerdem sollte der Benchmark idempotent sein, also beliebig oft nacheinander ausgeführt werden können, ohne zu Problemen zu führen. Wenn eine Tabelle erstellt wird, ohne sie vorher zu löschen, schlägt der `CREATE TABLE`-Befehl im nächsten Durchlauf fehl. Dies lässt sich durch die Klausel `IF NOT EXISTS` vermeiden oder noch besser, indem die Tabelle am Ende des Benchmarks gelöscht wird. Dafür ist die `cleanup()`-Funktion vorgesehen:

Codeblock 2.7: Lua-Script für das Aufräumen

```lua
local con = sysbench.sql.driver():connect()
function cleanup()
    con:query("DROP TABLE IF EXISTS BESTELLUNG;")
    con:query("DROP TABLE IF EXISTS KUNDEN;")
    print("Cleanup successfully done")
end
```

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

Jetzt wurde das Gerüst für die eigentlichen Insert- und Select-Befehle geschaffen. Bei den Insert-Befehlen kann entweder eine Zufallszahl generiert oder aus vordefinierten Listen zufällig gewählt werden. Allerdings muss bei den zufällig generierten Daten darauf geachtet werden, dass die Primärschlüsselbedingung nicht verletzt wird. Deshalb bietet es sich an, mit inkrementellen Werten zu arbeiten. In diesem Beispiel wird die `KUNDEN_ID` fortlaufend mit dem Schleifendurchgang vergeben und die `BESTELLUNG_ID` wird aus einer Kombination der Kundennummer und der Bestellnummer berechnet. Es muss festgelegt werden, wie viele Kunden und Bestellungen pro Kunde erstellt werden. Um sicherzustellen, dass keine Werte in den Tabellen enthalten sind, können alle Datensätze aus den Tabellen entfernt werden, bevor neue hinzugefügt werden. Damit die Performance der Insert-Query auch gemessen wird, ist es wichtig, dass die `insert()`-Funktion in der `event()`-Funktion aufgerufen wird.

Codeblock 2.8: Lua-Script für das Einfügen von Daten

```lua
local con = sysbench.sql.driver():connect()
local num_rows = 1000
local bestellungProKunde = 4

function delete_data()
    con:query("DELETE FROM BESTELLUNG;")
    con:query("DELETE FROM KUNDEN;")
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 KUNDEN
            (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)
        con: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 BESTELLUNG
                (BESTELLUNG_ID, BESTELLDATUM, ARTIKEL_ID, UMSATZ, FK_KUNDEN)
                VALUES (%d,'%s', %d, %d, %d);
```

```
            ]], bestellung_id, bestelldatum, artikel_id, umsatz, kunden_id)
            con:query(bestellung_query)
        end
    end
end

function event()
    insert_data()
end
```

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

Codeblock 2.9: Lua-Script für das Abfragen von Daten

```
local con = sysbench.sql.driver():connect()
function select_query()
    local join_query = [[
        SELECT k.STADT, SUM(b.UMSATZ) AS Total_Umsatz
        FROM KUNDEN k
        JOIN BESTELLUNG b ON k.NAME = b.FK_KUNDEN
        GROUP BY k.STADT;
    ]]
    con:query(join_query)
end

function event()
    select_query()
end
```

Damit sind für den Vergleich alle vier Operationen genauer definiert und es muss lediglich die Implementierung mit `VARCHAR` als Primärschlüssel der Kundentabelle angepasst werden. Dazu muss beim `CREATE TABLE`-Befehl der Typ für die Spalten `KUNDEN_ID` und `FK_KUNDEN` angepasst werden und beim Einfügen muss die Variable `i` zu einem String umgewandelt werden.

Neben dem Vergleich zwischen `INT` und `VARCHAR` soll auch das Verhalten mit unterschiedlichen Längen analysiert werden. Dadurch kann der Performanceunterschied zwischen beiden Datentypen sowie der Einfluss der Länge des Verbundoperators festgestellt werden. Dazu wird für beide Typen eine Hilfsfunktion benötigt, die eine Zeichenkette bzw. eine Zahl mit einer bestimmten Länge erstellt. Das Ergebnis der Funktion wird in der `INSERT`-Methode verwendet und zur Sicherstellung der Eindeutigkeit der `KUNDEN_ID` mit der Schleifenvariable `i` konkateniert. Ein Problem besteht jedoch noch darin, dass bisher nur eine Länge pro `INSERT`-Methode festgelegt werden kann. Wie könnten jetzt die beiden Ordner mit den Skripten duplizieren und die Längen in den neuen Dateien anpassen. Dies würde zu extremer Redundanz führen, weshalb es eine intuitivere Lösung gibt. Und zwar könnte man beim Aufruf des Shell-Scripts Variablen definieren, die im Skript exportiert werden und in den Lua-Dateien importiert werden können. Die Zeile mit der festgelegten Länge könnte so aussehen:

```lua
local length = 10
```

Um die im Skript exportierte Variable, beispielsweise `LENGTH`, zu verwenden, muss man Folgendes tun:

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

Jetzt muss noch ermittelt werden, welche Längen überhaupt zulässig sind. Bei `VARCHAR` gestaltet sich das einfach, da dort alle Längen bis 255 bei `VARCHAR(255)` möglich sind. `INT` kann Werte bis $2^{32} - 1$ (4.294.967.295) speichern, also bis zu 10 Stellen, während `BIGINT` Werte bis $2^{64} - 1$ (18.446.744.073.709.551.615) kann und damit 20 Stellen umfasst. Um größere Längen zu testen, wird der Typ der `Kundentabelle` von `INT` auf `BIGINT` geändert und es werden 4 sowie 16 Stellen als getestete Längen gewählt.

Es wurde also gezeigt, dass sich mit Lua-Skripten Tabellen gezielt erstellen, eingefügte Daten verwalten und Abfragen steuern lassen. Um die Operationen in der korrekten Reihenfolge auszuführen und die Grafiken zu generieren, wird ein Shell-Skript benötigt. Dieses Skript soll möglichst wenige Parameter erhalten, weshalb eine festgelegte Dateistruktur erforderlich ist. Es wird ein Ordner mit einem beliebigen Namen, z.B. `int_queries`, benötigt, in dem sich folgende Dateien befinden:

- `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 den Varchar-Fall erstellt werden. Wichtig ist dabei, dass die Namen der Dateien mit dem Namen des Ordners übereinstimmen. Das Shell-Script bedient sich dieser Struktur, führt korrekte Lua-Skript aus und geht die einzelnen Schritte bis zur Erstellung der Grafiken durch. Wenn Variablen definiert werden, werden diese exportiert, um sie in den Lua-Dateien importieren zu können. Der Dateiname dieses Orchestrators ist `sysbench_script.sh` und man kann ihn wie folgt aufrufen:

Codeblock 2.10: Befehl zum Ausführen des Orchestrator Skripts

```sh
./sysbench_script.sh \
  -out "YOUR_PATH_TO_DIRECTORY/Output" \
  -var '{"length":[4, 16]}' \
  -scripts '{
    "YOUR_PATH_TO_DIRECTORY/Scripts/varchar_queries": {
      "vars": "length"
    },
    "YOUR_PATH_TO_DIRECTORY/Scripts/int_queries": {
      "vars": "length"
```

```
10      }
11    }'
```

Wenn man will, kann man mehrere Select-Abfragen ohne unterschiedliche Insert-Befehle definieren. Dies wird später in der Bachelorarbeit nützlich sein, wenn verschiedene Indextypen untersucht und mithilfe unterschiedlicher SELECT-Abfragen überprüft werden, ob ein bestimmter Indextyp bei Abfragen verwendet wird. Die eigentlichen Tabellen und deren Datensätze müssen dabei nicht immer wieder neu befüllt werden. Wenn auf die Ordnerstruktur mit dem Int-Query-Beispiel zurückgekommen wird, könnte anstelle von `int_queries_select.lua` auch ein Ordner mit dem Namen `int_queries_select` erstellt werden. In diesem Ordner können beliebig viele unterschiedliche Lua-Skripts sein, die Select-Befehle durchführen. Dadurch werden alle Select-Befehle auf der gleichen Datenbasis verglichen und es kann im Kapitel 4.1 erkannt werden, wann der Index verwendet wird und wann nicht.

Auflistung aller möglichen Parameter:

- `–out`: Gibt den Pfad des Speicherorts für den Output-Ordner an
- `–var`: Gibt die Variablen und deren Werte im `JSON`-Format an
- `–scripts`: Gibt die Pfade der Ordner mit den jeweiligen Lua-Skripten im `JSON`-Format an. Der Schlüssel für jedes Skript ist der Pfad zur Datei, während die zu exportierenden Variablen unter dem Schlüssel `vars` angegeben werden.
  Innerhalb von `–scripts` kann man folgendes angeben:
  - `–vars`: Wählt aus, welche unter der `–var` angegebenen Variablen für das jeweilige Skript verwendet werden sollen
  - `–selects`: Legt fest, welche Select-Abfragen verwendet werden sollen, wenn man mehrere in einem Ordner definiert
  - `–db`: Gibt den Namen aller verwendeten Datenbankverbindungen aus der `db.env`-Datei in einer Liste an. Standardmäßig wird `MySQL` verwendet.

Damit wird kurz die Funktionsweise des Orchestrator-Skripts `sysbench_script.sh` erläutert. Im Grundlegenden arbeitet das 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 Argumente des Skripts, die bereits in 2.10 gesehen wurden, definiert und überprüft. Beispielsweise wird sichergegangen, dass die für die Skripts verwendeten Parameter, in diesem Beispiel `length`, tatsächlich definiert werden mit `–var`. Danach wird der Output-Ordner erstellt und die Spaltenüberschriften in die CSV-Dateien geschrieben. Anschließend beginnt erst das eigentliche Durchgehen der unterschiedlichen Skripte, die unter dem Argument `–script` angegeben wurden. Zu Beginn der Schleife entnimmt man die Werte das Skript die verwendeten Datenbanken (unter dem Argument `–db`) und die Select-Abfragen (unter dem Argument `–selects`). Daraufhin geht man in weitere Schleife, um die unterschiedlichen Datenbankverbindungen durchzugehen. Innerhalb dieser Schleife

wird eine Methode aufgerufen, die alle Variablen vorbereitet. Zum Beispiel werden für die jeweilige Datenbank die richtigen Umgebungsvariablen aus der Datei `envs.json` geladen. Diese Variablen sind unverzichtbar, da sonst keine Verbindung zur Datenbank aufgebaut werden kann.

Als Nächstes kommt eine Fallunterscheidung, die überprüft, ob das Skript im aktuellen Durchlauf Variablen exportiert. Für den Fall, dass keine Variablen exportiert werden, wird direkt die Methode `process_script_benchmark` aufgerufen. Wenn aber Variablen exportiert werden, dann müssen weitere Zwischenschritte umgesetzt werden. Zunächst 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 $2 \times 2 \times 1 = 4$ unterschiedliche Kombinationen. Anschließend muss man für jede Kombination die entsprechenden Werte exportieren und dann die Methode `process_script_benchmark` aufrufen.

Die Funktion `process_script_benchmark` führt wie schon beim Demo-Beispiel (siehe 2.1) erwähnt, die Methoden `PREPARE`, `INSERT`, `SELECT` und `CLEANUP` durch. Außerdem überprüft sie, ob es sich bei dem Select-Directory um einen Ordner handelt oder nicht. Wenn es ein Ordner ist, dann werden alle `SELECT`-Funktionen in diesem Ordner nacheinander ausgeführt, wenn nicht, dann wird nur eine Datei mit der Endung `_select.lua` betrachtet. Die Methode `run_benchmark` führt den Sysbench-Befehl (siehe 2.1) aus und wenn es sich um die Methode `RUN` handelt, werden die Daten während der Ausführung und die Endstatistiken in je eine CSV-Datei gespeichert.

Codeblock 2.11: Verkürzter Ausschnitt aus Orchestrator Script

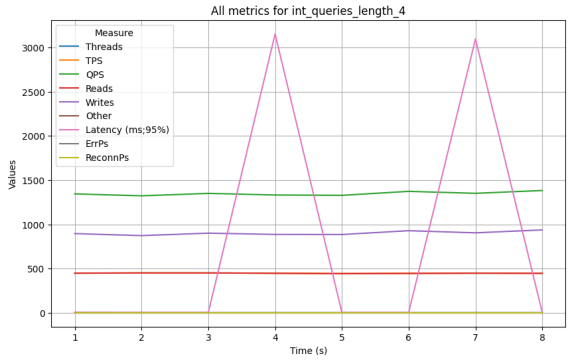
```
for SCRIPT_PATH in $SCRIPT_DIRS; do
  DBMS=$(echo "$SCRIPTS" | jq -r --arg key "$SCRIPT_PATH" '.[$key].db // ["mysql"]')
  SELECT_QUERIES=$(echo "$SCRIPTS" | jq -r --arg key "$SCRIPT_PATH" '.[$key].selects')
  for DB in $(echo "$DBMS" | jq -r '.[]'); do
    prepare_variables "$SCRIPT_PATH" "$DB"
    DB_INFO="$( [ "$DBMS_COUNT" -ne 1 ] && echo "${DB}" )"
    if [[ -n "$EXPORTED_VARS" ]]; then
        IFS=',' read -r -a KEYS <<< "$EXPORTED_VARS"

        COMBINATIONS=$(generate_combinations "" "${KEYS[@]}")
        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
              export "$(echo "${pair%%=*}" | tr '[:lower:]' '[:upper:]')=${pair#*=}"
            done
            COMBINATION_NAME=$(echo "$combination" | sed -E 's/(^|,)num_rows=[^,]*//g;s/^,//;s/,$//' | tr ',' '_' |
      tr '=' '_')
            LOG_DIR_COMBINATION="$LOG_DIR/$COMBINATION_NAME"
            process_script_benchmark "$DB_INFO" "$SCRIPT_PATH" "$LOG_DIR_COMBINATION" "$INSERT_SCRIPT" "
     $SELECT_SCRIPT" "$COMBINATION_NAME"
        done <<< "$COMBINATIONS"
    else
      process_script_benchmark "$DB_INFO" "$SCRIPT_PATH" "$LOG_DIR" "$INSERT_SCRIPT" "$SELECT_SCRIPT"
```

```
    fi
    eval $(jq -r --arg env "$DB" '.[$env] | to_entries | .[] | "unset " + .key' "$ABS_PATH/envs.json")
  done
done
# Combine csv files during runtime and end statistics and generate plots
python3 "$PYTHON_PATH/generateCombinedCSV.py" "$STATISTICS_FILE_TEMP" "$STATISTICS_FILE" --select_columns "
    $STATS_SELECT_COLUMNS" --insert_columns "$STATS_INSERT_COLUMNS" --prefixes "$PREFIXES"
python3 "$PYTHON_PATH/generateCombinedCSV.py" "$RUNTIME_FILE_TEMP" "$RUNTIME_FILE" --select_columns "
    $RUNTIME_SELECT_COLUMNS" --insert_columns "$RUNTIME_INSERT_COLUMNS" --prefixes "$PREFIXES"
python3 "$PYTHON_PATH/generatePlot.py" "$RUNTIME_FILE" "$STATISTICS_FILE"
```

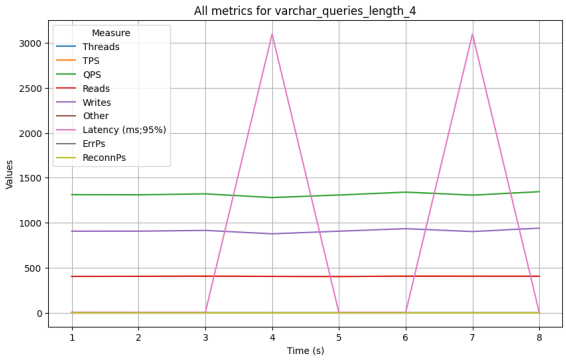
Nach dem Durchführen aller Schleifen sind alle Messwerte in CSV-Dateien gespeichert. Jetzt müssen mithilfe von Python-Skripten die Ergebnisse der Insert- und Select-Benchmarks aus den CSV-Dateien pro Skript wieder vereint werden, indem die Attribute miteinander addiert werden. Der letzte fehlende Schritt ist die Erstellung der Graphen mithilfe von Python und Pandas.

Wenn der Befehl aus 2.10 ausgeführt wird, wird ein Output-Ordner an der gewünschten Stelle erstellt. Dieser besteht es den Unterordner `pngs`, `logs` und den CSV-Dateien. In dem Unterordner `pngs` befinden sich verschiedene Grafiken, 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. 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.

Damit wird zum finalen Schritt übergegangen, der Analyse der Ergebnisse für die verschiedenen Datentypen und Längen des Verbundoperators. Die ersten beiden Abbildungen aus 2.2 sind Zeitreihendiagramme, die für die Skripte `int_queries_length_4` und `varchar_queries_length_4` alle Metriken darstellen. Aus den Grafiken, die für ein Skript alle Metriken veranschaulichen, kann man möglicherweise Datenfehler erkennen. Bei beiden springt die Latenz bei einigen Messpunkten von 0 ms auf einen höheren Wert und wieder zurück. Ansonsten aber sind die anderen Metriken auf einem konstanten Level und es gibt wenige Schwankungen.



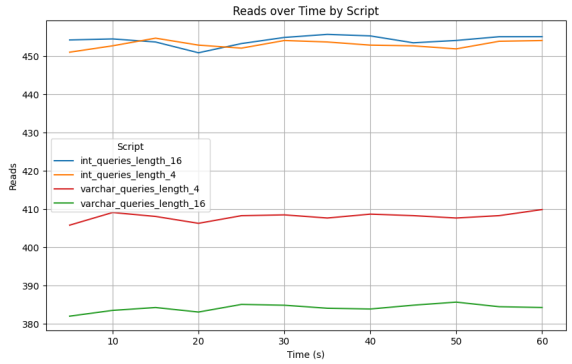
(a) `int_queries_length_4`



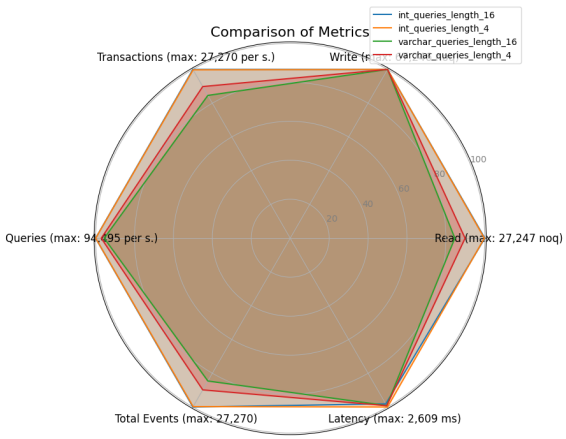
(b) `varchar_queries_length_4`

Abbildung 2.2: Die Grafik zeigt alle Metriken für die jeweiligen Skripte

Wenn alle vier Skripte miteinander verglichen werden sollen, können die Abbildungen aus 2.3 herangezogen werden. Was die Lesegeschwindigkeit angeht, kann man erkennen, dass `INT` eine etwa 30% bessere Lese-Performance hat als `VARCHAR`. Aus dem Vergleich von den unterschiedlichen Löngen mit `INT` kann man schließen, dass je länger die Zahl oder bei `VARCHAR` die Zeichenkette ist, desto langsamer wird die Abfrage. Es scheint auch so, dass die Länge bei `VARCHAR` sogar einen stärkeren Einfluss auf die Performance hat. Dennoch ist der Unterschied zwischen den Datentypen deutlich größer. Außerdem ist zu erkennen, dass die Werte bis auf wenige Ausnahmen konstant bleiben und es keine großen Schwankungen gibt. Aus der Gesamtstatistik in 2.3b kann ein ähnliches Verhalten abgeleitet werden. Bei der Schreibgeschwindigkeit kann man kaum Unterschiede erkennen und bei der Latenz haben interessanterweise die Varianten mit kleineren Längen höhere, also schlechtere Wert.



(a) Reads



(b) Gesamtstatistik

Abbildung 2.3: Die Grafik zeigt den Vergleich zwischen allen Skripten für die Metriken

2.3 GitHub Actions

Im Verlauf der Bachelorarbeit kommen immer mehr Projekte mit unterschiedlichen Lua-Dateien, die alle das Orchestrator-Skript verwenden, dazu. Manche dieser Projekte erfordern keine Anpassungen an dem Skript, während andere wiederum viele benötigen. Das Problem dabei ist, dass man durch die Komplexität des Skripts schnell den Überblick über die Auswirkungen der Änderungen auf andere Projekte verliert. Um sicherzugehen, müssen die Benchmarks für alle Projekte durchgeführt und anschließend die Output-Ergebnisse überprüft werden. Dazu muss jedes Skript nacheinander ausführt werden, was zum einen zeitintensiv ist und zum anderen hohe Lasten für den lokalen Rechner bedeutet. Das Vorgehen könnte man zeitlich optimieren, indem man die Skripte parallel ausführt, aber auch das würde nicht das Problem der hohen Lasten und des manuellen Aufwands lösen. 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 (**github_action_doku**). 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 kann über einen oder mehrere Schritte (engl. Step) verfügen. Die Schritte können wiederum beliebige Shell-Befehle, Skripte oder Aktionen ausführen.

Im Kapitel 2.2 wurde gezeigt, wie das Hauptskript für das Beispiel auf dem lokalen Rechner ausgeführt werden kann (2.10). Jetzt werden diese Informationen für alle Projekte gebraucht, die getestet werden sollen. Es werden immer die Pfade zu den Lua-Skripten benötigt, die getestet werden sollen, sowie in einigen Fällen die zusätzlich definierten Variablen. Diese Pfade und Variablen werden in einer JSON-Datei gesammelt und jedem Projekt wird ein Name zugewiesen.

Codeblock 2.12: JSON-Datei mit dem Join-Typ Beispiel

```
{
  "join-typ": {
    "scripts": {
      "./YOUR_PATH_TO_PROJECT/Scripts/varchar_queries": {
        "vars": "length"
      },
      "./YOUR_PATH_TO_PROJECT/Scripts/int_queries": {
        "vars": "length"
      }
    },
    "var": {"length": [4, 16]}
  }
}
```

Damit das Hauptskript ausgeführt werden kann, müssen im ersten Job die Daten dieser JSON-Datei verarbeitet und bestimmte Variablen, wie beispielsweise der Output-Ordner, definiert werden. Zudem müssen alle Namen der verschiedenen Projekte in einer Liste zusammengefügt und als Output für den nächsten Job bereitgestellt werden. Der nächste Auftrag ist verantwortlich für das eigentliche Durchführen der Benchmarks und wird erst gestartet, wenn der Vorherige beendet ist. Um die Vorteile des gleichzeitigen Ausführens der Aufträge zu nutzen, muss die Matrixstrategie verwendet werden. Bei der Matrixstrategie kann man eine Liste von Variablen angeben, um mehrere Auftragsausführungen parallel zu erstellen. In diesem Fall wird dafür die Liste mit den unterschiedlichen Projektnamen genutzt.

Damit die einzelnen Benchmarks ausgeführt werden können, müssen innerhalb der Matrixausführung einige Vorbereitungen getroffen werden. Zuallererst müssen, abhängig vom Projektnamen, die entsprechenden Variablen aus der JSON-Datei, die im ersten Job erstellt wurde, geladen und exportiert werden. Anschließend werden die Dependencies für Sysbench und die Python-Libraries installiert sowie die Datenbank-Container mit passenden Konfigurationen gestartet und vorbereitet. Nach diesen Schritten kann das Hauptskript ausgeführt werden und die Outputdateien werden am angegebenen Pfad erstellt.

Um Zugriff auf diese Dateien zu erhalten, müssen sie als GitHub Artifact hochgeladen werden. Die GitHub Artifacts können anschließend entweder über die GitHub REST API oder die Übersicht des Workflows auf der GitHub-Webseite als Zip-Datei heruntergeladen werden. Als letzten Schritt, nach Beendigung beider vorangegangenen Jobs, können alle GitHub Artifacts des aktuellen Workflows heruntergeladen und gemeinsam als ein neues Artifact wieder hochgeladen werden. Dadurch entfällt beispielsweise bei 10 Projekten die Notwendigkeit, 10 Zip-Dateien einzeln herunterzuladen und zu entpacken, um die Änderungen in den 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 Namen des Workflows festlegen und anschließend, wann er getriggert werden soll. Dies kann beispielsweise manuell auf GitHub mit dem Tag `workflow_dispatch` oder bei jedem Push mit `push` geschehen. Zudem kann der Trigger auch auf bestimmte Dateien oder Ordner beschränkt werden. Als Nächstes kann man unter dem Tag `jobs` die verschiedenen Aufträge definieren. Der Schlüssel `outputs` beschreibt die Ausgaben eines Jobs, die von anderen Jobs verwendet werden können, während `steps` die Aufgaben festlegt, die innerhalb eines Jobs ausgeführt werden. Unter dem Tag `env` muss man die Umgebungsvariablen definieren, dazu gehören zum Beispiel beim zweiten Job die Länge der Durchführung des Benchmarks. Wenn es sich um vertrauliche Informationen handelt, sollte man GitHub Secrets 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. Die Workflow-Datei für das Durchführen der Benchmarks sieht in verkürzter Form wie folgt aus:

Codeblock 2.13: Ausschnitt aus der Workflow-Datei

```yaml
name: Run All Benchmarks
on:
  push:
    paths: ['Projects/**', ...]
jobs:
  prepare-benchmark:
    outputs:
      matrix: ${{ steps.set-matrix.outputs.matrix }}
      configurations: ${{ steps.prepare-config.outputs.configurations }}
    steps:
      - { name: Checkout repository, uses: actions/checkout@v3 }
      - name: Read and generate list of matrix name  #  echo "matrix=$matrix" >> $GITHUB_OUTPUT
      - name: Prepare configurations for all test types
        run: # ... export variables like  test_type, dirs, var, output_dir, artifact_name as "configurations"
  run-tests:
    needs: prepare-benchmark
    strategy:
      matrix:
        test-type: ${{ fromJson(needs.prepare-benchmark.outputs.matrix) }}
    env: { TIME: 32, THREADS: 1, EVENTS: 0, REPORT_INTERVAL: 2 }
    steps:
      - { name: Checkout repository, uses: actions/checkout@v3 }
      - name: Extract and save values to GitHub environment
      - name: Install dependencies (sysbench, pandas, matplotlib)
      - name: Start MySQL container (and wait for it to be ready)
        run: |
          docker run --name mysql-${{ env.test_type }} -d -e MYSQL_ROOT_PASSWORD=$DB_PASS -e MYSQL_DATABASE=
    $DB_NAME -p $DB_PORT:3306 mysql:8.0
      - name: Run sysbench script
        run: |
          chmod +x Tools/Shell-Scripts/sysbench_script.sh
          Tools/Shell-Scripts/sysbench_script.sh -out "${{ env.output_dir }}" \
          -var '${{ env.var }}' -scripts:'${{ env.dirs }}'
      - name: Stop MySQL and PostgreSQL containers
      - name: Upload outputs # with actions/upload-artifact@v4
  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 with all downloaded benchmarks as one artifact named "combined-output"
```

Wenn Änderungen an den Skripten vorgenommen werden, wird der Workflow automatisch getriggert und die Benchmarks für alle Projekte werden durchgeführt. Nach Abschluss des Workflows können die kombinierten Ergebnisse aus dem `combined-output`-Artifact als ZIP-Datei heruntergeladen und damit analysiert werden. Es bieten sich aber auch weitere Verbesserungen an, die zu einer Optimierung des Workflows führen.

2.4 Optimierung des Workflows

Es gibt verschiedene Möglichkeiten, die Laufzeit und den Ressourcenverbrauch des Workflows zu optimieren. Zum einen kann man die zu installierenden Abhängigkeiten mithilfe des GitHub Caches (**github_cache_doku**) 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.

Codeblock 2.14: Speichern der Abhängigkeiten im Cache

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

Falls sich bis zum nächsten Workflow keine Änderungen an den Abhängigkeiten ergeben, wird der Cache automatisch heruntergeladen. Der Zeitgewinn in diesem Beispiel ist jedoch nur gering und beträgt nur wenige Sekunden pro Workflow.

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 in eine Tabelle eingefügt werden, dann ändert dies nichts an dem Ergebnis von Projekt B oder C. Daher würde es sich anbieten, dass für Projekt A die Benchmarks neu durchgeführt werden, für Projekte B und C könnte hingegen jeweils der letzte erfolgreiche Output-Ordner benutzen. 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 die JSON geladen wurde, wird nicht direkt mit der Installation der Abhängigkeiten fortgefahren. Stattdessen werden zunächst die unterschiedlichen Pfade gehasht und ein entsprechender Name erstellt. Falls kein Ordner mit diesem Namen existiert, wird wie bisher fortgefahren. Existiert jedoch bereits ein Ordner mit diesem Namen, werden alle weiteren Schritte nach dem Extrahieren der Werte aus der JSON im Job `run-tests` übersprungen. Dadurch erspart man sich die Installation der Abhängigkeiten, das Starten der Datenbank-Container sowie das Ausführen des Orchestrator-Skripts.

Als letztes stellt sich die Frage, 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 (**github_cache_doku**). Eine andere mögliche Lösung kann auch das Nutzen von expliziten Branches nur für die Sicherung der Dateien seien. Das Problem ist dabei, dass durch Timing-Probleme beim Pushen ein paralleler Workflow den Code zwischen Rebase, Commit und Push verändert haben könnte, wodurch nach einem verhinderten Push erneut ein Rebase nötig wird. Außerdem muss die GitHub Action über Schreibberechtigungen verfügen. 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. Wie man in der `workflow.yaml` erkennen kann, habe ich mich für die Lösung mit GitHub Artifacts entschieden. Wenn man eine andere Lösung umsetzen möchte, dann muss man aber nur wenige Zielen im Workflow anpassen.

3 Optimierungen von Datentypen

Das erste Thema in Bezug auf die Performance-Optimierung von Datenbanken sind die unterschiedlichen Datentypen und deren Auswirkungen auf die Performance. Bei der Auswahl des korrekten Datentyps gibt es unterschiedliche Faktoren, die vom jeweiligen Typen abhängen. Besonders werden die unterschiedlichen Implementierungen von numerischen und zeichenkettenbasierten Datentypen betrachtet. Es gibt aber auch allgemeinere Prinzipien, die auf fast alle angewendet werden können.

3.1 Allgemeine Faktoren

In diesem Abschnitt werden die geltenden Grundsätze behandelt, die allgemein bei der Wahl der Datentypen angewendet werden können. Bei der Erstellung von Tabellen sollte man folgende Schritte für die Auswahl von Datentypen befolgen (**schwartz2012high**). Zunächst muss die übergeordnete Kategorie des Datentyps, wie beispielsweise numerisch, textbasiert 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 Genauigkeit (engl. Precision), 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.

Der erste Grundsatz für Datentypen besagt, 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 weniger Speicherplatz im Arbeitsspeicher und CPU-Cache benötigt, was wiederum zu schnelleren Abfragen führt. Außerdem ist bei der Benutzung des kleinstmöglichen Typs eine einfache Typveränderung möglich. Wenn die vorhandenen Daten beispielsweise falsch eingeschätzt wurden, lässt sich der Typ nachträglich mit wenig Aufwand in einen größeren umwandeln.

Eine weitere allgemeine Richtlinie ist die Einfachheit von Datentypen. So sind Integer-Werte beispielsweise leichter zu verarbeiten als Character. Daher sollte man stets einen Integer

wählen, wenn er die Daten korrekt abbilden kann. Begründen kann es damit, dass für einfachere Datentypen weniger CPU-Zyklen benötigt werden, um Operationen auszuführen. Im Fall von Integer und Character liegt der Unterschied in den Character Sets und Sortierregeln, die den Vergleich von Character erschweren.

Die letzte Regel zur Performanceverbesserung ist die Vermeidung von `NULL`. Viele Tabellen enthalten `NULLABLE`-Spalten, obwohl keine `NULL`-Werte gespeichert werden müssen, da `NULL` die Standardeinstellung ist. Daher sollten solche Spalten bei der Tabellenerstellung mit dem Identifier `NOT NULL` definiert werden, es sei denn, `NULL`-Werte sind erforderlich.

> „A missing NOT NULL constraint can prevent index usage in an Oracle database-especially for count(*) queries.“ (**winand2011sql**)

Mit `NULL` wird es auch für MySQL schwieriger, Abfragen zu optimieren, da Indizes und Wertevergleiche mehr Speicherplatz benötigen. Dies liegt daran, dass indizierte nullable Spalten ein zusätzliches Byte pro Eintrag erfordern, wodurch ein Index mit fester Größe in einen variablen umgewandelt wird. Der Leistungsunterschied zwischen `NULL` und `NOT NULL` ist zwar gering, kann jedoch, besonders in Verbindung mit Indizes, spürbar sein.

Es muss erwähnt werden, dass MySQL viele Aliase für Datentypen unterstützt, wie zum Beispiel `INTEGER`, `BOOL` und `NUMERIC`. Diese Aliase können verwirrend sein, aber sie beeinflussen nicht die Performance. Im Wesentlichen funktioniert es so, dass ein aliasierter Datentyp beim Erstellen einer Tabelle intern in den Basistyp umgewandelt wird. Dies lässt sich mit dem Befehl `SHOW CREATE TABLE` bestätigen, da dort der Basistyp angezeigt wird.

3.2 Verhaltensweise einzelner Datentypen

Der erste Datentyp, bei dem das Verhalten genauer betrachtet wird, ist der numerische Datentyp. Bei diesem kann zwischen Ganzzahlen und Fließkommazahlen gewählt werden. 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 (**mysql_data_types_numeric**). Dementsprechend verändert sich auch der mögliche Wertebereich der Zahlen, die durch den Speicherplatz abgedeckt sind. Mit den optionalen `UNSIGNED`-Attributen können keine negativen Werte gespeichert werden können. Dafür verdoppelt sich die obere Grenze der positiven Werte, während der Speicherplatz und die Leistung unverändert bleiben. Die Berechnung des Wertebereichs für `SIGNED` und `UNSIGNED` erfolgt nach den folgenden Formeln:

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

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

Hinweis: *N* entspricht der Anzahl der Bits.

Wenn die Wertebereiche für den Datentyp `TINYINT` in MySQL berechnet werden sollen, muss für *N* der Wert 8 eingesetzt werden. Als Ergebnis ergeben sich für `SIGNED` die Werte von -128 bis 127 und für `UNSIGNED` die Werte von 0 bis 255. Bei einem Beispiel mit 150 Werten kann anstelle von `SMALLINT` also einfach `UNSIGNED` verwendet werden, um Speicherplatz zu sparen.

Eine Breitenangabe wie `INT(11)` beeinflusst nur die Anzeige und nicht den Wertebereich oder die Speicheranforderungen. Um dies zu beweisen, wird die folgende Tabelle erstellt:

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

Es wurde für beide Spalten der Datentyp `INT` gewählt und da überprüft werden soll, ob die Breitenangabe einen Einfluss auf die Speicheranforderungen hat, wird versucht, die Grenzen von `INT` einzufügen. Da `INT` 32 Bits benötigt, ergeben sich folgende Grenzen: $2^{(32-1)} - 1 = 2147483647$ und $-2^{(32-1)} = -2147483648$.

Codeblock 3.1: Inserts und Selects für Testtabelle

```
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 3.1

	int_5	**int_11**
obere Grenze	2147483647	2147483647
untere Grenze	-2147483648	-2147483648

Bei der Ausführung der Insert-Befehle wird keine Fehlermeldung angezeigt, weshalb `INT(5)` und `INT(11)` beide die Grenzwerte speichern können. Damit wurde gezeigt, dass die Breitenangabe keinen Einfluss auf die Speicheranforderungen hat, sondern lediglich die Anzeige beeinflusst.

Neben dem Typ für Ganzzahlen gibt es auch den Typ für Festkommazahlen, der in MySQL als `DECIMAL` bezeichnet wird. Eine Festkommazahl ist eine Zahl mit einem festen Dezimalpunkt, bei der sowohl die Anzahl der Dezimalstellen als auch die maximale Anzahl der Ziffern vor und nach dem Dezimalpunkt definiert sind. Damit ist er auch für die Speicherung von Ganzzahlen geeignet. `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 mit neun Ziffern pro vier Bytes und unterstützt bis zu 65 Ziffern insgesamt.

Ein weiterer numerischer Datentyp sind die Fließkommazahlen, zu denen `FLOAT` und `DOUBLE` gehören. Fließkommazahlen verwenden die Gleitkomma-Arithmetik und sind für ungefähre Berechnungen optimiert. `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 man auch `BIGINT` nutzen, um sowohl die Ungenauigkeit von Gleitkomma-Speicherungen als auch die höheren Kosten der `DECIMAL`-Arithmetik zu vermeiden.

Als Nächstes werden die zeichenkettenbasierten Datentypen betrachtet. Die beiden Haupttypen 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. Es gibt jedoch auch Nachteile, da Aktualisierungen der Werte zu wachsenden Zeilen führen und damit auch zu zusätzlicher Verarbeitung der Speicher-Engine. Interessant ist auch, dass die Speicherung von `hello` in `VARCHAR(5)` oder `VARCHAR(200)` zwar gleich viel Speicherplatz benötigt, jedoch für Sortierungen oder Operationen auf temporären Tabellen ineffizienter sein kann. Deshalb sollte immer so viel Platz reserviert werden, wie tatsächlich benötigt wird.

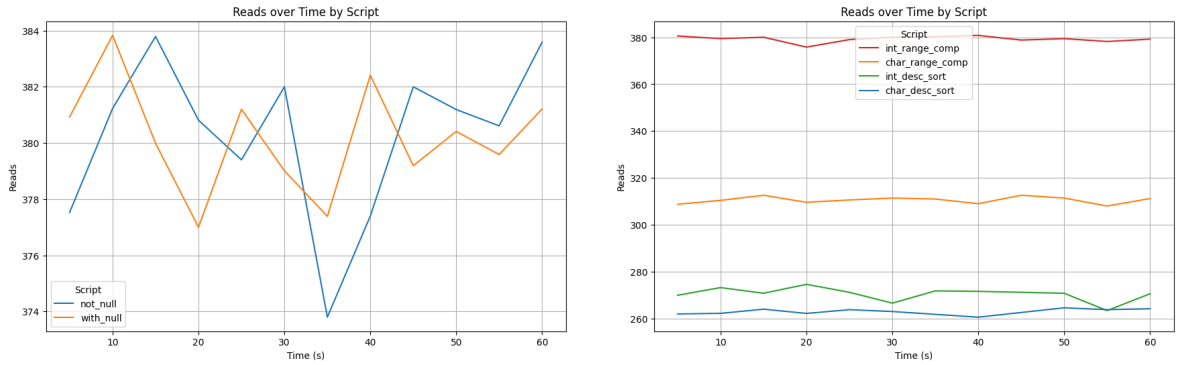
`CHAR` hingegen hat im Gegensatz zu `VARCHAR` eine feste Länge und MySQL reserviert auch den nicht gebrauchten Platz für die angegebene Anzahl an Zeichen. Daher ist `CHAR` ideal für sehr kurze Strings oder Werte, die alle nahezu gleich lang sind, da `VARCHAR(1)` aufgrund des Längen-Bytes 2 Bytes benötigt, `CHAR(1)` hingegen nur 1 Byte. Außerdem bleibt die Speicherstruktur bei Aktualisierungen von `CHAR` unverändert, weshalb er besser geeignet ist, wenn die Daten häufig geändert werden. Dafür ist `CHAR` nicht dafür geeignet, wenn die maximale Spaltenlänge deutlich größer ist als die durchschnittliche Wertelänge.

Als Letztes werden die zeitbezogenen Datentypen `DATE`, `TIME`, `TIMESTAMP` und `DATETIME` behandelt. Der Datentyp `DATE` speichert nur das Datum ohne Uhrzeit und ist besonders speichereffizient, während `TIME` ausschließlich eine Uhrzeit oder Zeitspanne, auch über 24 Stunden hinaus, erfasst. Die anderen beiden speichern das Datum mit Uhrzeit und haben eine Genauigkeit von einer Sekunde. `TIMESTAMP` benötigt nur halb so viel Speicherplatz wie `DATETIME` und ist zeitzonenbewusst, hat aber dafür einen deutlich kleineren Wertebereich. Abhängig von der Information, die gespeichert werden soll, wählt man den passenden zeitbezogenen Datentyp.

3.3 Analyse der Benchmarks

Der erste Leitsatz, der untersucht wird, besagt, dass Spalten nach Möglichkeit als `NOT NULL` deklariert werden sollten. Zum Nachweis wird die Kundentabelle aus 2.4 verwendet, bei der einmal alle Spalten als `NOT NULL` deklariert sind und einmal der Standardwert genutzt wird. Wenn das Attribut nicht deklariert wird, können `NULL`-Werte in die Tabelle eingefügt werden. Um bei Select-Abfragen mit `WHERE`-Klauseln sowie `COUNT`- und `GROUP BY`-Befehlen die gleiche Anzahl an Zeilen zu erhalten, werden `NULL`-Werte ausgeschlossen.

In der Grafik 3.1a sind die Ergebnisse der Select-Befehle zu sehen, wobei die Werte für `NOT NULL` im Durchschnitt höher sind als für `WITH NULL`. Höhere Werte bedeuten mehr Abfragen pro Sekunde und deuten auf bessere Performance hin, weshalb man sagen kann, dass `NOT NULL` besser performt als `WITH NULL`. Wenn man auf die y-Achse schaut, fällt aber auf, dass die Werte nicht so weit auseinanderliegen und damit sind die Unterschiede sehr gering. Daher sollte die Entscheidung, eine Spalte als `NOT NULL` zu deklarieren, vor allem aus Gründen der Datenintegrität und -konsistenz und nicht aus Performancegründen getroffen werden.



(a) Vergleich von NULL und NOT NULL **(b)** Vergleich von INT und CHAR

Abbildung 3.1: Vergleich von `NULL` und `NOT NULL`, sowie `INT` und `CHAR`

Um zu zeigen, dass bei der Wahl zwischen unterschiedlichen Datentypen der einfachere bevorzugt werden sollte, wird erneut die Kundentabelle verwendet. Für diesen Benchmark wird jeweils der Datentyp des Schlüsselattributs der Tabelle geändert. Zunächst wird eine Kundentabelle mit einem `INT`-Primärschlüssel erstellt, gefolgt von einer weiteren mit `CHAR`. Die Performance der Schreibbefehle ist in beiden Fällen etwa gleich. Bei den Lesebefehlen sieht das anders aus (siehe 3.1b). Wenn man einen Wertebereich abfragt, dann ist `INT` deutlich schneller (etwa 50%) als `CHAR`. Bei der Sortierung hat `INT` ebenfalls einen Vorteil, jedoch fallen die Abstände deutlich geringer aus.

Als letztes soll der Vergleich unterschiedlicher Datentypen erfolgen. Hierfür wird die gleiche Tabelle wie beim Vergleich von `INT` und `CHAR` verwendet, jedoch werden diesmal verschiedene numerische oder zeichenkettenbasierte Typen als Primärschlüssel eingesetzt. Beim Vergleich der numerischen Typen zeigt sich, dass `DECIMAL` mit deutlichem Abstand am langsamsten ist

(Abbildung 3.2). Danach folgt, wie vermutet, der nächstgrößere Datentyp `BIGINT`. Das lässt sich sowohl an der Grafik als auch an der Legende erkennen, die die Typen nach Performance absteigend sortiert. Die Legende hilft dabei, da auffällig ist, dass die unterschiedlichen Werte auch aufgrund der Skalierung der y-Achse sehr stark schwanken. Daraufhin kommen `INT`, `MEDIUMINT` und `SMALLINT`, wobei die Unterschiede kleiner sind als erwartet. Dies wird vermutlich darauf zurückzuführen sein, dass die Abfragen nur auf einer Tabelle mit wenigen tausend Datensätzen ausgeführt wurden. In der Praxis mit Millionen von Datensätzen dürften die Unterschiede zwischen den Typen größer sein als in diesem Vergleich.

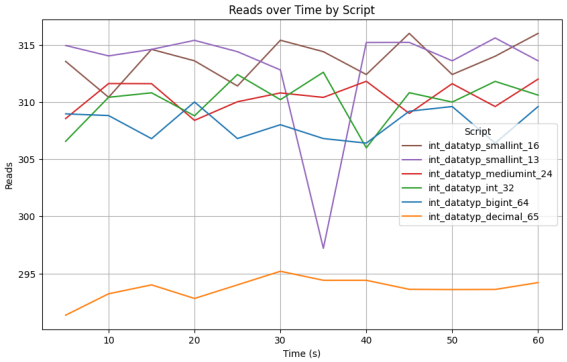
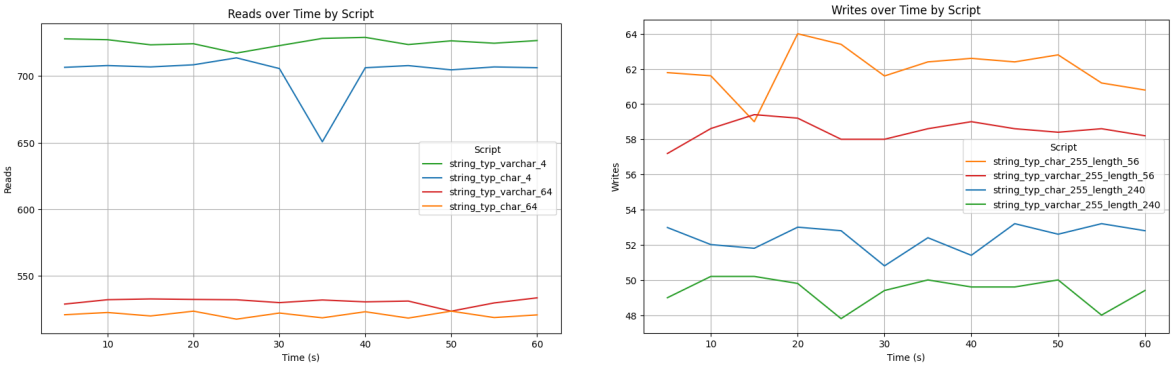


Abbildung 3.2: Vergleich von unterschiedlichen zeichenkettenbasierten Typen

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

Als letzten Vergleich wurden beide Zeichenketten-Typen mit der Länge von 255 Stellen definiert, aber mit unterschiedlich vielen Stellen befüllt. Anschließend wurden bei beiden Tabellen die Werte aktualisiert, wobei in der Namen-Spalte zufällig wenige Stellen hinzugefügt wurden. Dabei war `CHAR` schneller als `VARCHAR` (3.3b). Dies bestätigt die Vermutungen aus Abschnitt 3.2, da die Vorteile von `CHAR` insbesondere bei der Aktualisierung von Werten zum Tragen kommen, während `VARCHAR` bei der Selektion von Werten besser abschneidet.



(a) Unterschiedliche Zeichenketten-Typen **(b)** Bei unterschiedlichem Befüllungsgrad

Abbildung 3.3: Vergleich von unterschiedlichen zeichenkettenbasierten Typen

4 Indizes

Das folgende Kapitel befasst sich mit der Indexierung und den damit verbundenen Performance-Optimierungen, die näher erläutert werden. Zunächst werden einige Grundlagen der Indexierung betrachtet. Anschließend werden die verschiedenen Arten von Indizes näher erläutert und unterschiedliche Benchmarks mit ihnen durchgeführt. Im letzten Schritt werden die Ergebnisse analysiert, um festzustellen, welche Verwendung der Indizes am besten funktioniert.

4.1 Grundlagen

Indizes sind Datenstrukturen, die von Speicher-Engines (engl. storage engines) verwendet werden, um unter anderem Zeilen schneller zu finden (**schwartz2012high**). Die Storage-Engine ist eine Kernkomponente eines Datenbankmanagementsystems, die für die Speicherung und Verwaltung der Daten verantwortlich ist. Verschiedene Storage-Engines unterscheiden sich hinsichtlich ihrer Indexfunktionalität sowie der Unterstützung von Transaktionen und Sperrmechanismen. Im weiteren Verlauf werden verschiedene Indextypen vorgestellt, die nicht von allen Engines unterstützt werden.

Die Indizes haben einen großen Einfluss auf die Datenbank-Performance und werden mit zunehmender Größe der Datenbank immer wichtiger, da das Scannen aller Tupel zunehmend aufwendiger wird. Weniger ausgelastete Datenbanken können ohne ordnungsgemäße Indizes gut funktionieren, aber die Leistung kann rapide sinken, wenn die Datenmenge wächst. Wenn ein solches Problem auftritt, ist die Index-Optimierung oft der effektivste Weg, um die Abfrageleistung schnell zu verbessern. Um wirklich optimale Indizes zu erstellen, ist es häufig notwendig, Abfragen umzuschreiben. Besonders nützlich sind Indizes bei Abfragen, die Joins zwischen mehreren Tabellen enthalten, da sie ermöglichen, die Anzahl der zu prüfenden Tupel erheblich zu reduzieren, wenn eine einschränkende Bedingung vorliegt. Wie genau Indizes erstellt werden müssen, wird im Laufe dieses Kapitels geklärt.

Um die Funktionsweise eines Indexes anschaulicher zu erklären, wird als Beispiel ein wissenschaftliches Fachbuch betrachtet. Am Ende dieser Bücher gibt es meist ein Stichwortverzeichnis oder Register. Dieses Register besteht aus einer alphabetisch geordneten Liste von Begriffen, Themen und Stichworten. Möchte man einen Begriff nachschlagen, sucht

man ihn im Stichwortverzeichnis und erhält die Seitenzahlen, auf denen er vorkommt. In DBMS verwendet die Storage-Engine Indizes auf eine ähnliche Weise. Sie durchsucht die Datenstruktur des Indexes nach einem Wert. Und wenn ein Treffer gefunden wird, kann die Engine die Zeilen ermitteln, die den Treffer enthalten. Das folgendes Beispiel veranschaulicht dies:

```
1 SELECT NAME FROM KUNDEN WHERE KUNDEN_ID = 7;
```

Angenommen, es existiert ein Index auf der Spalte `KUNDEN_ID`, dann wird MySQL diesen verwenden, um Zeilen zu finden, bei denen die `KUNDEN_ID` den Wert 7 hat. Mit anderen Worten wird eine Suche innerhalb der Indexwerte durchgeführt und alle entsprechenden Zeilen werden zurückgegeben.

Ein Index kann Werte aus einer oder mehreren Spalten einer Tabelle enthalten. Bei mehreren Spalten ist die Reihenfolge der Spalten im Index entscheidend, da MySQL nur effizient auf ein linkes Präfix des Indexes zugreifen kann. Gibt man nur das zweite Attribut an, ohne das erste zu referenzieren, kann der Index nicht direkt verwendet werden. Außerdem darf man nicht verwechseln, dass ein Index über zwei Spalten nicht gleichbedeutend ist mit zwei separaten einspaltigen Indizes. Es gibt verschiedene Typen von Indizes, die jeweils für unterschiedliche Zwecke optimiert sind und in den nächsten Abschnitten behandelt werden.

Um zu verstehen, wie man Indizes für eine Datenbank auswählt, ist es wichtig zu wissen, welcher Teil der Abfrage am meisten Zeit in Anspruch nimmt (**garcia2008database**). Das Datenbanksystem ist so aufgebaut, dass die Tupel einer Relation üblicherweise auf viele Seiten einer Festplatte verteilt sind, die jeweils mehrere Tausend Bytes umfassen und viele Tupel speichern. Um die Werte eines Tupels zu prüfen, muss die gesamte Seite, auch Block genannt, in den Hauptspeicher geladen werden. Dabei kostet es kaum mehr Zeit, alle Tupel einer Seite anstatt nur ein einzelnes zu prüfen.

In der Regel stellt der Schlüssel den sinnvollsten Index für eine Tabelle dar, weshalb MySQL standardmäßig den B-Tree-Index für Primary Keys verwendet (**mysql_primary_key**). Die Entscheidung, ob für ein bestimmtes Attribut ein Index definiert werden soll, hängt von drei Faktoren ab: Erstens ist ein Index besonders nützlich, wenn Abfragen häufig auf ein bestimmtes Attribut zugreifen. Zweitens kann ein Index sinnvoll sein, wenn es nur wenige Tupel für einen bestimmten Wert des Attributs gibt, da dies den Festplattenzugriff bei einer Abfrage reduziert. Sobald nicht alle Blöcke geladen werden müssen, kann der Index Zeit sparen. Der letzte Fall betrifft Situation, in denen Tupel nach einem Attribut geclustert sind. Durch einen Index können müssen hier weniger Datenblöcke geladenen werden, da die Werte des Attributs aufeinanderfolgender gespeichert sind. Mit diesen Faktoren kann begründet werden, warum die Schlüssel einer Tabelle gut geeignet sind. Zum einen kommen sie oft in Abfragen vor (erster Punkt) und zum anderen enthalten sie keine doppelten Werte, da jedes Tupel einen eindeutigen Wert hat (zweiter Punkt).

Das Auswählen von Indizes erfordert von den Entwicklern eine Tradeoff abzuwägen. Es gibt dabei zwei Faktoren, die die Entscheidung beeinflussen. Zum einen kann ein Index auf einem Attribut Abfragen mit diesem Attribut erheblich beschleunigen. Zum anderen erschwert jeder Index Einfügungen, Löschungen und Aktualisierungen, da diese mehr Zeit und Aufwand erfordern. Aber selbst wenn Modifikationen die häufigste Form von Datenbankaktionen sind, kann ein Index auf ein häufig verwendetes Attribut die Leistung verbessern. Dies liegt daran, dass einige Modifikationsbefehle zuvor die Datenbank abfragen. Im Kapitel Partitionen wird uns dieses Thema wieder begegnen.

Um Zeitersparnis durch die Nutzung von Tupeln ohne vollständige Durchsuchung der Relation zu erreichen, müssen Indizes auf der Festplatte gespeichert werden. Dies führt jedoch zu zusätzlichen Festplattenzugriffen. Allgemein lässt sich sagen, dass Modifikationen in etwa doppelt so kostenintensiv sind wie der Zugriff auf den Index oder die Daten während einer Abfrage. Damit berechnet werden kann, ob sich ein Index für eine Spalte lohnt, muss bekannt sein, mit welcher Wahrscheinlichkeit Abfragen und Modifikationen durchgeführt werden.

Um die Vorgehensweise anhand einer beispielhaften Berechnung durchzuführen, wird die folgende Tabelle benutzt (abgeändertes Beispiel aus **garcia2008database**):

```
1 Fakten(Id, Bestelldatum, Artikel_Id, Kunden_Id, ...)
```

Der Schlüssel der Faktentabelle ist die Spalte `Id` und für die Attribute `Artikel_Id` sowie `Kunden_Id` wird jeweils ein Index erstellt. Damit stehen uns insgesamt drei unterschiedliche Indexe zur Verfügung, einschließlich des Primärschlüssels. Als Nächstes werden Befehle benötigt, bei denen die Indexe benutzt werden (siehe 4.1). In der ersten Zeile wird nur der Kundenindex verwendet, in der zweiten nur der Artikelindex und in der Letzten fügen wie eine Zeile ein.

Codeblock 4.1: Select-Queries für die Faktentabelle

```
1 SELECT Bestelldatum, Artikel_Id FROM Fakten WHERE Kunden_Id = k;
2 SELECT Bestelldatum, Kunden_Id FROM Fakten WHERE Artikel_Id = a;
3 INSERT INTO Fakten VALUES(i, b, a, k);
```

Damit berechnet werden kann, ob es sinnvoll ist, die Indizes zu erstellen, müssen bestimmte Voraussetzungen festgelegt werden. Zuallererst wird davon ausgegangen, dass die Faktentabelle 10 Datenblöcke belegt und im Durchschnitt jeder Kunde 3 Artikel kauft, während ein Artikel von 3 Kunden gekauft wird. Die Tupel für einen bestimmten Kunden oder Artikel sind gleichmäßig über die 10 Seiten verteilt. Trotzdem sind mit einem Index nur 3 Festplattenzugriffe erforderlich, um die durchschnittlich 3 Tupel für einen Kunden oder Artikel zu finden. Dazu ist ein Festplattenzugriff erforderlich, um die Seite des Indexes zu lesen und ein weiterer, um die modifizierte Seite zurückzuschreiben, falls eine Indexseite geändert werden muss. Ohne Index sind 10 Festplattenzugriffe zum Lesen und zwei Festplattenzugriffe zum Schreiben erforderlich. Unter diesen Bedingungen ergibt sich die folgende Kostentabelle:

Aktion	Kein Index	Kunden Index	Artikel Index	Beide Indizes
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
Durchschnitt	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

Tabelle 4.1: Kosten der unterschiedlichen Queries in Abhängigkeit der Indizes

Die letzte Zeile aus Tabelle 4.1 gibt die durchschnittlichen Kosten einer Aktion an. Unter der Annahme, dass der Anteil der Zeit, in der die erste Abfrage ausgeführt wird, p1 beträgt und der Anteil der Zeit für die zweite Abfrage p2 ist, ergibt sich der Anteil der Zeit, in der I ausgeführt wird, zu 1 − p1 − p2. Der Durchschnitt für den Kundenindex wird wie folgt berechnet:

$$4p_1 + 10p_2 + 4 \cdot (1 - p_1 - p_2) = 4p_1 + 10p_2 + 4 - 4p_1 - 4p_2 = 4 + 6p_2$$

Abhängig von den Werten für p1 und p2 kann jede der vier Optionen die geringsten Kosten für die drei Operationen verursachen. Zum Beispiel, wenn p1 = p2 = 0,1, dann ist der Ausdruck 2 + 8p1 + 8p2 am kleinsten, sodass keine Indizes bevorzugt werden würden. Wenn jedoch p1 = 0,5 und p2 = 0,1 gelten, ergibt ein Index für die Kunden den besten Durchschnittswert.

Damit wurde gezeigt, dass es sinnvoll ist, keinen Index zu verwenden, wenn überwiegend Einfügungen durchgeführt werden und nur sehr wenige Abfragen anfallen. Intuitiv gilt, dass bei vielen Abfragen und einer ungefähr gleichen Häufigkeit von Abfragen, die Artikel und Kunden angeben, beide Indizes vorteilhaft sind. Wird hingegen nur ein Typ von Abfrage häufig genutzt, sollte nur der Index definiert werden, der bei dieser Abfrageart hilft.

Es gibt zahlreiche Tools, die entwickelt wurden, um die Verantwortung der Wahl der Indizes vom Datenbankdesigner zu übernehmen. Dabei optimiert das System sich selbst oder dem Designer werden zumindest Empfehlungen für sinnvolle Entscheidungen gegeben. Ein bewährter Ansatz zur Auswahl von Indizes ist das sogenannte Greedy-Verfahren (**garcia2008database**), bei dem zunächst ohne ausgewählte Indizes der Nutzen jedes Kandidaten-Index bewertet wird. Wenn es einen Index mit positivem Nutzen gibt, wird dieser ausgewählt und anschließend wird eine Neubewertung ausgeführt, wobei davon ausgegangen wird, dass der zuvor ausgewählte Index bereits verfügbar ist. Dieser Prozess wird so lange wiederholt, bis es keinen Kandidaten-Index mit positivem Nutzen mehr gibt.

4.2 B-Baum-Index

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

Das Grundprinzip eines B-Baums ist, dass alle Werte in einer bestimmten Reihenfolge gespeichert werden und jede Blattseite den gleichen Abstand zum Wurzelknoten hat.

> „The height of a B+ tree depends on the number of data entries and the size of index entries." (**ramakrishnan2002database**)

Ein B-Baum-Index beschleunigt den Datenzugriff, da die Storage-Engine nicht die gesamte Tabelle durchsuchen muss, um die gewünschten Daten zu finden. Stattdessen beginnt die Suche beim Wurzelknoten. Die Slots im Wurzelknoten enthalten Zeiger auf Kindknoten und die Storage-Engine folgt diesen Zeigern. Der richtige Zeiger wird durch Vergleich der Werte in den Knoten-Seiten (engl. node pages) ermittelt, die die oberen und unteren Grenzen der Werte in den Kindknoten definieren. Letztlich stellt die Storage-Engine fest, ob der gewünschte Wert existiert oder ob sie erfolgreich eine Blatt (engl. leaf page) erreicht.

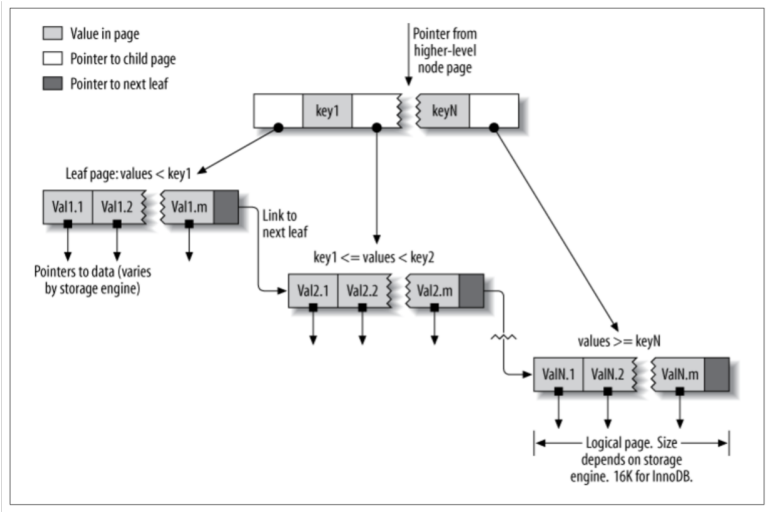


Abbildung 4.1: Binär-Baums-Darstellung (Abbildung 5–1 aus **schwartz2012high**)

Die Blätter sind besonders, da sie Zeiger auf die indexierten Daten enthalten, anstatt auf andere Seiten zu verweisen. Zwischen dem Wurzelknoten und den Blattseiten können viele Ebenen von Knoten-Seiten existieren. Die Tiefe des Baumes hängt von der Größe der Tabelle ab. Außerdem speichern B-Bäume die indexierten Spalten in einer festgelegten Reihenfolge, was sie besonders nützlich für die Suche nach Datenbereichen macht. Beispielsweise kann ein Index auf einem Textfeld (z.B. vom Typ `VARCHAR`) effizient alle Namen finden, die mit „K" beginnen, da die Werte in alphabetischer Reihenfolge gespeichert sind.

Der Index sortiert die Werte entsprechend der Reihenfolge der in der `CREATE INDEX`-Anweisung angegebenen Spalten, beispielsweise kann man wie folgt ein Index erstellen:

Codeblock 4.2: B-Baum-Index bestehend aus mehreren Attributen

```
1 CREATE INDEX combined_index ON KUNDEN(NAME, VORNAME, GEBURTSTAG);
```

Als Nächstes werden die möglichen Abfragen betrachtet, bei denen B-Baum-Indizes besonders hilfreich sind, um ein besseres Verständnis für ihre optimale Nutzung zu gewinnen. Eine Übereinstimmung mit dem vollständigen Schlüsselwert liefert Werte für alle Spalten im Index. Eine beispielhafte Abfrage zur Suche nach allen Einträgen mit dem Index aus 4.2 ist die Suche nach allen Kunden, die Max Mustermann heißen und am 01.01.2000 geboren wurden. Auch Abfragen, die nur mit dem linken Präfix übereinstimmen, können von diesem Index profitieren. So lässt sich etwa gezielt nach dem Nachnamen „Mustermann“ suchen. Ebenso ist es möglich, nur ein Spaltenpräfix zu verwenden, etwa um alle Nachnamen zu finden, die mit „M“ beginnen. Ein weiterer Vorteil ergibt sich bei Bereichsabfragen, denn der Index kann effizient genutzt werden, um Nachnamen zwischen „Mustermann“ und „Müller“ zu ermitteln. Darüber hinaus unterstützt ein B-Baum-Index auch Kombinationen aus exakten und Bereichsabfragen, beispielsweise wenn nach dem Nachnamen „Mustermann“ gesucht wird, während der Vorname innerhalb eines Bereichs liegt, etwa ab „Ma“. Ein weiterer Vorteil von B-Baum-Indizes ist, dass sie aufgrund der sortierten Baumstruktur nicht nur Abfragen, sondern auch `ORDER BY`-Bedingungen effizient unterstützen können.

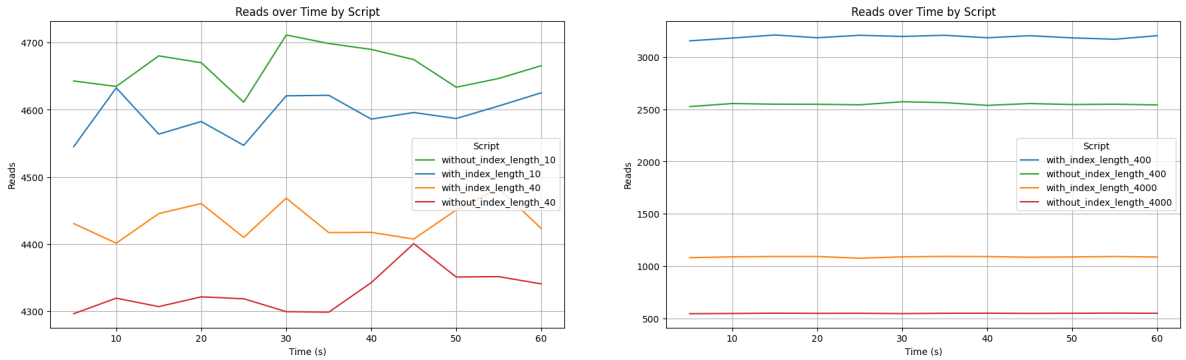
Es gibt jedoch Einschränkungen von B-Baum-Indizes, die dazu führen, dass andere Indextypen für bestimmte Szenarien besser geeignet sind. Eine Einschränkung ist, dass die Suche nicht am rechten Ende des Indexes beginnen kann. Beispielsweise ist der Beispiels-Index nicht dazu geeignet, alle Personen zu finden, die vor dem Jahr 2000 geboren wurden, ohne dass der Nachname und Vorname ebenfalls spezifiziert werden. Für optimale Leistung kann es auch erforderlich sein, dass Indizes mit den gleichen Spalten, jedoch in unterschiedlicher Reihenfolge erstellt werden. Auf diese Weise könnten mehr Kombinationen abgedeckt und zusätzlich einige Abfragen optimiert werden.

Im nächsten Abschnitt werden die Benchmarks durchgeführt, um das Verständnis für die Funktionsweise des B-Baum-Index zu bestätigen. Dazu wird zunächst wieder die Kundentabelle (2.4) erstellt und für den ersten Vergleich werden folgende Indizes definiert:

Codeblock 4.3: Definition mehrere Indizes

```
1 CREATE INDEX idx_stadt ON KUNDEN(STADT);
2 CREATE INDEX idx_postleitzahl ON KUNDEN(POSTLEITZAHL);
3 CREATE INDEX idx_geburtstag ON KUNDEN(GEBURTSTAG);
```

Um die Effizienz dieser Indizes einordnen zu können, wird diese Konfiguration mit einer verglichen, bei der nur die Kundentabelle ohne Indizes erstellt wird. In beiden Fälle werden eine bestimmte Anzahl an Datensätze eingefügt. Um die Performance der Select-Abfragen zu messen, werden verschiedene Queries an die Datenbank ausgeführt, bei denen die Attribute `GEBURTSTAG`, `STADT` und `POSTLEITZAHL` berücksichtigt werden. Dazu gehören `GROUP BY`- und `COUNT`-Abfragen, bei denen die Index-Attribute verwendet werden oder sie spielen in der `WHERE`-Bedingung eine Rolle. Damit es übersichtlich bleibt, werden einmal 10 Datensätze mit 40 und einmal 400 mit 4000 Zeilen verglichen.



(a) Mit 10 und 40 Datensätze

(b) Mit 400 und 4000 Zeilen

Abbildung 4.2: Grafik zeigt Performance mit und ohne Index für Readsabfragen

In der Abbildung 4.2a ist zu erkennen, dass bei 10 Datensätzen die Kundentabelle ohne Indizes schneller ist als die mit Indizes. Bei 40, 400 oder 4000 Zeilen (siehe 4.2b) wird schon die Wirkung der Indizes deutlich, da in diesen Fällen jeweils die Version mit Indizes effizienter ist. Der Unterschied bei 40 Datensätzen ist zwar etwas geringer, aber in den anderen Fällen sehen noch eindeutigere Unterschiede. Interessant ist, dass es nicht linear oder quadratisch mit der Anzahl an Datensätzen in der Tabelle steigt, sondern bei 400 und 4000 Zeilen beträgt der Unterschied zur Tabelle ohne Index jeweils etwa 500–700 Abfragen. Bei der Schreibgeschwindigkeit liegen beide auf einem sehr ähnlichen Niveau, wobei die Version ohne Index tendenziell einen leichten Vorteil hat.

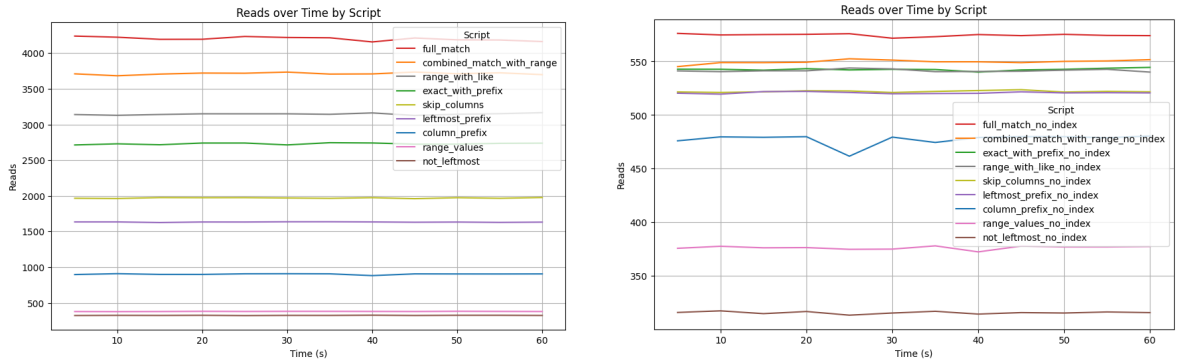
Mit dem vorherigen Benchmark können die Vorteile eines Indexes bereits deutlich erkannt werden. Nun soll jedoch auch die Funktionalität des B-Tree-Indexes in Bezug auf unterschiedliche Selects untersucht werden. Dazu wird erneut die Kundentabelle erstellt, aber diesmal wird nur ein Index definiert (siehe 4.2).

Anschließend wird die Tabelle mit einer festgelegten Anzahl an Datensätzen befüllt und es werden unterschiedliche Select-Befehle ausgeführt. Im Codeblock 4.4 sind aus Platzgründen nur die Where-Bedingungen zu sehen und am Ende jeder Zeile steht der Name der Query, damit später in der Analyse nachvollzogen werden kann, welche Query welche Performance liefert.

Codeblock 4.4: Unterschiedliche Where-Bedingungen für B-Tree-Index

```
WHERE NAME LIKE 'M%'; -- columm_prefix
WHERE NAME = 'Müller' AND VORNAME = 'Max' AND GEBURTSTAG < '1980-01-01'; -- combined_match_with_range
WHERE NAME = 'Müller' AND VORNAME LIKE 'M%' ORDER BY GEBURTSTAG; -- exact_with_prefix
WHERE NAME = 'Müller' AND VORNAME = 'Max' AND GEBURTSTAG = '1960-01-01'; -- full_match
WHERE NAME = 'Müller'; -- leftmost_prefix
WHERE GEBURTSTAG < '1980-01-01'; -- not_leftmost
WHERE NAME BETWEEN 'Müller' AND 'Schulz'; -- range_values
WHERE NAME = 'Müller' AND VORNAME LIKE 'M%' AND GEBURTSTAG < '1980-01-01'; -- range_with_like
WHERE NAME = 'Müller' AND GEBURTSTAG < '1980-01-01'; -- skip_columns
```

Anhand der Grafik in Abbildung 4.3 lässt sich erkennen, bei welchen Abfragen der Index am effizientesten ist. Auf der linken Seite können die Ergebnisse für die Read-Befehle mit Index betrachtet werden, während auf der rechten Seite die Werte ohne Index zu sehen sind.



(a) Mit Index **(b)** Ohne Index

Abbildung 4.3: Visualisierung von unterschiedlichen Select-Queries mit und ohne Index

Zunächst fällt auf, dass die Reihenfolge für die Werte mit und ohne Index komplett identisch ist. Dies ist direkt erkennbar, da die Legenden beider Grafiken nach dem durchschnittlichen Wert über die gesamte Zeit sortiert sind. Damit die richtigen Schlüsse aus der Grafik gezogen werden können, muss zunächst ermittelt werden, wie viele Zeilen die unterschiedlichen Queries zurückgeben. Dazu werden die Abfragen zusätzlich mit dem `COUNT(*)`-Operator durchgeführt und die Ergebnisse in die Log-Datei geschrieben. Anschließend werden die Werte entnommen und in einer Tabelle zusammengefasst.

Select-Query	Anzahl an Zeilen	Faktor	Index benutzt?
full_match	0	6.42	ja
combined_match_with_range	13	5.78	ja
range_with_like	31	5.22	ja
exact_with_prefix	51	5.14	ja
skip_columns	147	3.47	ja
leftmost_prefix	263	2.73	ja
column_prefix	551	1.69	ja
range_values	1343	0.96	nein
not_leftmost	2371	0.98	nein

Tabelle 4.2: Ergebnisse der COUNT(*)-Abfragen für B-Tree-Index

Anhand der Spalte `Anzahl an Zeilen` lässt sich erkennen, dass die Queries, die am wenigsten Zeilen zurückgeben, auch diejenigen sind, bei denen die höchste Performance erzielt wird. Deshalb ist auch die Reihenfolge mit und ohne Index gleich, weshalb man meinen könnte, dass der Index keinen Einfluss auf die Performance hat. Dies betrifft jedoch nur die Reihenfolge, nicht aber die Werte der Abfragen, da hier deutliche Unterschiede erkennbar sind. Anschaulich wird das mit der Betrachtung der Spalte `Faktor`. Um den Wert zu berechnen, werden die Werte aus der Gesamtstatistik entnommen und die Version mit Index durch die Version ohne Index geteilt. Dadurch lässt sich erkennen, dass `full_match` zwar bei beiden Versionen am schnellsten ist, jedoch mit Index etwa 6-mal schneller als ohne. Es lässt sich auch erkennen, dass je weniger Zeilen zurückgegeben werden, desto größer ist der Faktor. Bei den Queries `range_values` und `not_leftmost` liegt der Faktor sehr nah 1, was bedeutet, dass der Index keinen Einfluss auf die Performance hat. Deshalb stellt sich auch die Frage, ob der Index überhaupt verwendet wird. Um das zu überprüfen, wird der `EXPLAIN`-Operator verwendet, das Ergebnis erneut geloggt und der Tabelle hinzugefügt (siehe Spalte `Index benutzt?`). Und tatsächlich sehen wir, dass die vermuteten Queries die einzigen sind, bei denen der Index nicht verwendet wird.

4.3 Hash-Index

Ein weiterer Indextyp, der betrachtet wird, ist der Hash-Index. Dieser basiert auf einer Hash-Tabelle und ist daher nur für exakte Suchanfragen geeignet, die alle Spalten des Indexes verwenden. In MySQL unterstützt nur die Memory-Storage-Engine explizite Hash-Indizes. Einige Storage-Engines, wie zum Beispiel InnoDB, können erkennen, wenn bestimmte Index-Werte besonders häufig abgefragt werden. Sie erstellen dann automatisch einen Hash-Index für diese Werte im Speicher, der zusätzlich zu den bestehenden B-Baum-Indizes genutzt wird. Die Funktionsweise der Storage-Engine lässt sich wie folgt beschreiben.

Für jede Zeile wird mithilfe einer Hash-Funktion ein Hash-Wert der indexierten Spalte berechnet. Der Hash-Wert (engl. hash code) ist eine kleine Zahl, die sich in der Regel von den Hash-Werten anderer Zeilen mit unterschiedlichen Schlüsselwerten unterscheidet. Anschließend wird die Position im Index gesucht und man findet einen Zeiger auf die entsprechende Zeile. In letzten Schritt überprüft man die Werte der Zeile, um sicherzustellen, dass es sich um die richtige Zeile handelt.

Wenn mehrere Werte denselben Hash-Wert besitzen, speichert der Index die Zeiger auf die Zeilen (engl. row pointers) in demselben Hash-Tabelleneintrag, typischerweise mithilfe einer verketteten Liste (z.B. einer *Linked List*). Hash-Kollisionen können die Leistung eines Hash-Index beeinträchtigen, da jeder Zeiger in der verketteten Liste durchlaufen und die entsprechenden Werte mit dem Suchwert verglichen werden müssen, um die richtigen Zeilen

zu finden. Das ist auch Index-Wartungsoperationen mit viel Aufwand verbunden. Hingegen eindeutige Hash-Indizes stellen sicher, dass für jeden Hash-Wert nur ein einziger Eintrag existiert. Bei Konflikten wird ein Mechanismus wie die Open Addressing-Strategie (z.B. Linear Probing oder Quadratic Probing) eingesetzt, um Konflikte zu lösen und den Speicherplatz effizient zu verwalten. Hierbei wird versucht, Konflikte direkt innerhalb der Hash-Tabelle zu bewältigen, anstatt auf zusätzliche Datenstrukturen wie verkettete Listen zurückzugreifen. Die eindeutigen Hash-Indizes werden nicht von der Memory-Engine in MySQL unterstützt.

Um die Verwendung des Hash-Indexes zu veranschaulichen, benutzen folgt folgendes Beispiel:

```
1 SELECT * FROM KUNDEN WHERE NAME = 'Peter';
```

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

Ähnlich wie der B-Baum-Index hat auch der Hash-Index einige Einschränkungen. Zum einen enthält der Index nur Hash-Werte und Zeiger auf Zeilen (engl. row pointers), jedoch nicht die Werte selbst. Deshalb kann MySQL den Index nicht verwenden, um das Einlesen der Zeilen zu vermeiden. Allerdings erfolgt der Zugriff auf die in den Speicher geladenen Zeilen sehr schnell, wodurch die Leistung nicht wesentlich beeinträchtigt ist. Zum anderen können Hash-Indizes nicht für Sortierungen verwendet werden, da die Werte nicht in einer geordneten Reihenfolge gespeichert sind. Im Gegensatz dazu sind B-Baum-Indizes in der Lage. Darüber hinaus ermöglichen Hash-Indizes keine partiellen Schlüsselübereinstimmungen (engl. partial key matching). Da der Hash-Wert aus dem gesamten indexierten Wert berechnet wird, hilft ein Hash-Index beispielsweise nicht, wenn ein Index aus den Spalten (A, B) besteht und die `WHERE`-Klausel nur auf A verweist. Ein weiterer Nachteil besteht darin, dass Hash-Indizes keine Bereichsabfragen unterstützen. Sie eignen sich lediglich für Gleichheitsvergleiche, wie die Operatoren `=` (gleich), `<=>` (null-sicher gleich) und `IN()`.

Als Nächstes werden die Benchmarks mit Hash-Indizes betrachtet. Dazu wird erneut die Kundentabelle verwendet und diesmal nur ein Index für die Spalte `NAME` erstellt. Am Ende des `CREATE INDEX`-Befehls muss `USING HASH` hinzugefügt werden, damit statt des standardmäßigen B-Tree-Index der Hash-Index verwendet wird. Danach befüllen wieder die Tabelle mit Testdaten. Diesmal wird beim ersten Benchmark der Einfluss von Hash-Kollisionen auf die Performance untersucht. Um den Grad der Kollisionen zu verändern, wird eine Variable verwendet, die die obere Grenze für die zufällige Generierung einer Zahl darstellt. Anschließend werden alle Zeilen mit dem Wert `Kunde_1` für die Spalte `NAME` abgefragt und die Tests werden mit den Kollisionswahrscheinlichkeiten von 25%, 10%, 5% und 1% durchgeführt.

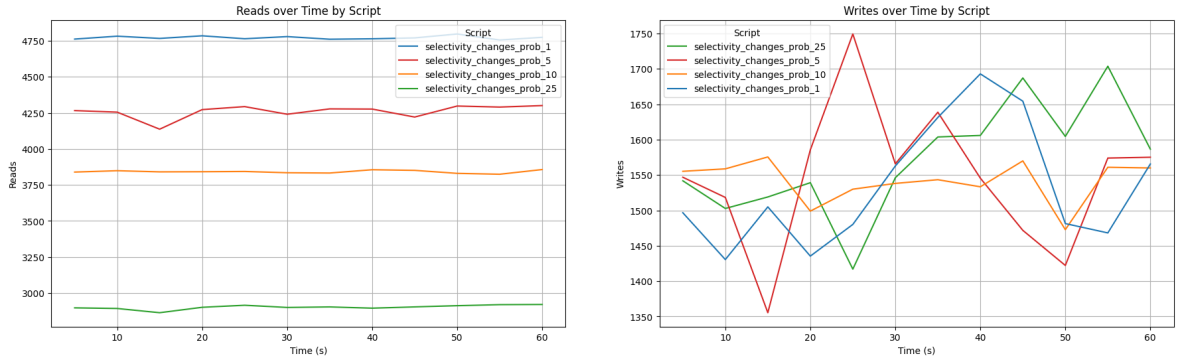


Abbildung 4.4: Vergleich der Auswirkungen von Hashkollisionen

An den Ergebnissen in Abbildung 4.4 ist zu erkennen, dass je geringer die Wahrscheinlichkeit für eine Kollision ist, desto schneller fällt die Select-Abfrage aus. Es fällt auch auf, dass die Unterschiede zwischen den verschiedenen Kollisionswahrscheinlichkeiten sehr groß sind. Hingegen die Einfüge-Performance ist bei allen 4 Varianten auf einem ähnlichen Niveau.

Als zweiten Test soll überprüft werden, ob der Index bei bestimmten Select-Queries benutzt wird oder nicht. Dazu wird erneut die Kundentabelle verwendet, der gleiche Index wie in Beispiel 4.2 erstellt, die Testdaten eingefügt und die Select-Befehle aus 4.4 genutzt. Dieses Mal werden aber nicht alle Select-Befehle verwendet, sondern nur die aus folgender Tabelle:

Select-Query	Anzahl an Zeilen	Faktor	Index benutzt?
full_match	0	2.67	ja
combined_match_with_range	6	0.97	nein
exact_with_prefix	45	1.01	nein
leftmost_prefix	206	1.00	nein

Tabelle 4.3: Ergebnisse der COUNT(*)-Abfragen für Hash-Index

Anhand der Spalten `Faktor` und `Index benutzt?` kann erkannt werden, dass der Index nur bei der `full_match`-Abfrage benutzt wird. Das stimmt auch mit den Ergebnissen aus der Abbildung 4.5 überein, da ohne Index alle Abfragen auch einem ähnlichen Niveau liegen, aber mit Index sticht eine deutlich hervor. Interessant ist, dass die Query mit 206 zurückgegebenen Zeilen nur unwesentlich langsamer ist als die anderen. Die Reihenfolge ist wieder bei beiden identisch und hängt von der Anzahl der zurückgegebenen Zeilen ab.

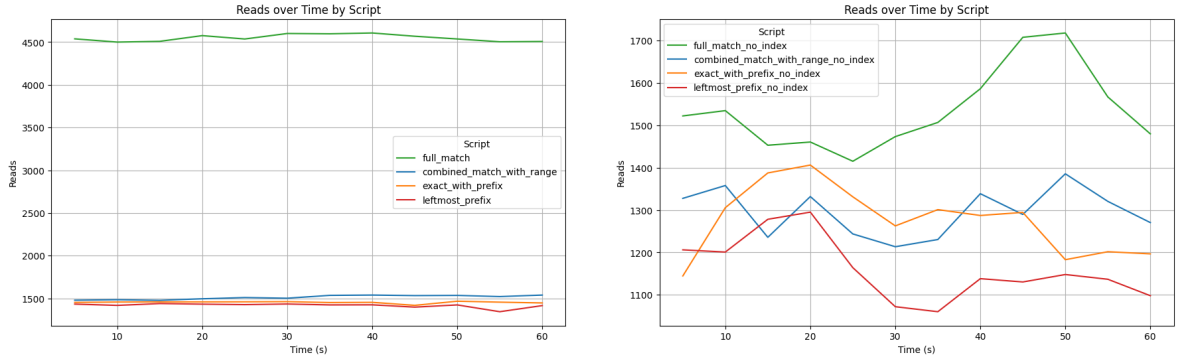
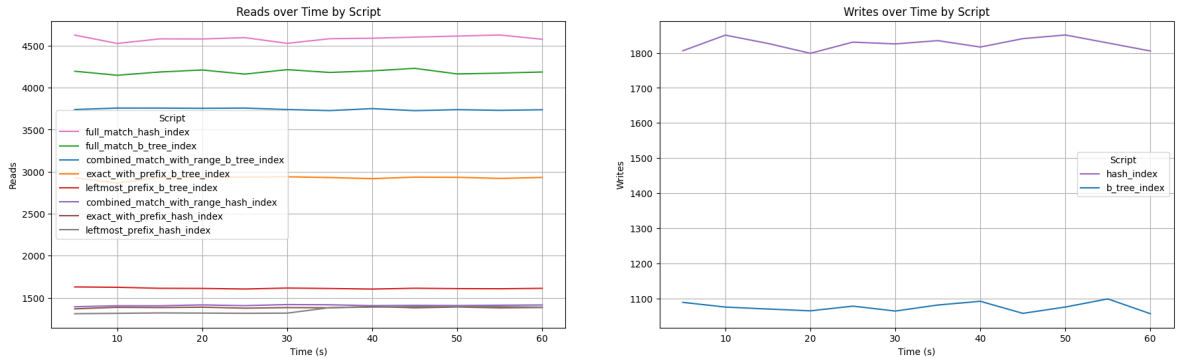


Abbildung 4.5: Grafik visualisiert Select-Queries mit (links) und ohne (rechts) Index

4.4 Vergleich von B-Tree- und Hash-Index

In den vorherigen Kapiteln wurden der B-Tree-Index und der Hash-Index jeweils getrennt voneinander betrachtet. Dabei wurde auch analysiert, bei welchen Select-Queries die Indizes Vorteile bieten und bei welchen nicht. Damit fehlt uns noch der Vergleich zwischen dem B-Tree-Index und dem Hash-Index.

Um die Unterschiede zwischen beiden Indexstrukturen genauer zu analysieren, wird ein neuer Benchmark durchgeführt, der die Skripte aus Kapitel 4.2 und 4.3 wiederverwendet. Da der Hash-Index aber nur 4 unterschiedliche Select-Queries aufruft, sollen auch nur diese mit dem B-Tree-Index ausgeführt werden. Dazu wird einfach der Parameter `selects` beim Aufruf des Orchestrator-Skripts hinzugefügt und das Skript anschließend ausgeführt.



(a) Anzahl der Lesezugriffe

(b) Anzahl der Schreibzugriffe

Abbildung 4.6: Vergleich der Select-Query-Performance von B-Tree- und Hash-Index

In der Abbildung 4.6a sehen die Performance für die unterschiedlichen Select-Befehle. Die höchste Transaktionsrate erzielt der Hash-Index, sofern der vollständige Schlüssel angegeben wird (`full_match`). Dicht darauf folgt der B-Tree-Index mit derselben Abfrage, allerdings mit etwa 10% weniger Zugriffen. Bei den übrigen Abfragen schneidet hingegen der B-Tree-Index deutlich besser ab, in einigen Fällen sogar bis zu dreimal schneller als der Hash-Index. Der Grund dafür ist bereits aus den anderen Kapiteln bekannt. Da der Hash-Index nur bei exaktem Schlüsselabgleich zum Einsatz kommt, wird er bei den anderen Abfragen nicht verwendet. Mithilfe des `EXPLAIN`-Operators wurde festgestellt, dass stattdessen der B-Tree-Index verwendet wird, was die starken Unterschiede erklärt.

Betrachtet man die Schreibperformance (Abbildung 4.6b), zeigt sich, dass der Hash-Index etwa 30–40% schneller ist als der B-Tree-Index. Wenn eine Anwendung also eine hohe Schreiblast hat, könnte der Hash-Index eine bessere Wahl sein, da er weniger Mehraufwand verursacht. Zusammenfassend lässt sich festhalten, dass der Hash-Index einen leichten Vorteil hat, wenn beide Indexe greifen. Andernfalls überwiegen die Stärken des B-Tree-Indexes.

5 Views

Im folgenden Kapitel werden die Performancevorteile von Sichten (engl. Views) in SQL betrachtet. Zunächst wird auf virtuelle Sichten, ihre Vor- und Nachteile, das Verhalten bei Inserts sowie auf mögliche Szenarien eingegangen, in denen sie besonders vorteilhaft sein können. Anschließend wird sich mit materialisierten Sichten beschäftigt, die physisch in der Datenbank gespeichert werden. Zunächst wird eine Version mit Triggern implementiert, da MySQL keine native Unterstützung für materialisierte Sichten bietet, bevor die native Implementierung in PostgreSQL genutzt wird. In den letzten beiden Kapiteln wird die Durchführung der Benchmarks näher betrachtet und die entstandenen Ergebnisse interpretiert.

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 (**silberschatz2011database**), 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 physisch gespeichert werden (**garcia2008database**). Sie werden als virtuelle Sichten bezeichnet.

> „A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition.“ (**ramakrishnan2002database**)

Virtuelle Sichten werden durch einen Ausdruck definiert, der einer Abfrage ähnelt. Sie können auch so abgefragt werden, als ob sie tatsächlich physisch existierten (vgl. **ramakrishnan2002database**). In einigen Fällen lassen sich sogar Datensätze über die Sicht ändern.

Codeblock 5.1: Allgemeine View-Deklaration

```
1 CREATE VIEW ‹name› AS ‹view-definition›;
```

In dem Codeblock 5.1 wird die Struktur der Definition einer View gezeigt. Als Nächstes muss die `view-definition` mit einer SQL-Abfrage ersetzt werden, die den Inhalt der virtuellen Sicht abbilden soll. Um dieses Vorgehen mit einem Beispiel näher zu veranschaulichen, werden die Tabellen Kunden (2.4) und Bestellung (2.5) genutzt. Nun wollen wir, dass die beiden Tabellen über die `KUNDEN_ID` in der Sicht zusammengefügt werden, da sie sowohl der Primärschlüssel in der Kundentabelle als auch der Fremdschlüssel in der Bestellung ist. Um in die SQL-Abfrage noch etwas mehr Komplexität zu bringen, soll neben der Join-Operation auch der Umsatz pro Jahr und pro Stadt aggregiert werden. Die View `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;
```

Diese Aggregation könnte beispielsweise von einem Marketingteam genutzt werden, um schwache Regionen pro Jahr zu identifizieren und gezielt in diesen nachzusteuern. Wenn die Daten dieser virtuellen Sicht abgefragt werden sollen, wird der Name in der FROM-Klausel adressiert und es wird darauf vertraut, dass das Datenbankmanagementsystem die benötigten Tupel erzielt (siehe 5.3). Dabei operiert das DBMS direkt auf den Relationen, die die virtuelle Sicht definieren. In diesem Fall handelt es sich um die Kunden- und Bestelltabelle.

Codeblock 5.3: SQL-Befehl mit verwendeter View

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

Eine weitere Möglichkeit, die Funktionsweise einer Sicht besser zu verstehen, besteht darin, sie in einer FROM-Klausel durch eine Unterabfrage zu ersetzen, die identisch mit der Sichtdefinition ist. Damit Bezug auf die Tupel genommen werden kann, muss die Unterabfrage noch mit einer Tupelvariablen ergänzt werden. Die SQL-Abfrage aus 5.4 liefert das gleiche Ergebnis wie die aus 5.3, wenn die View wie im Beispiel 5.2 definiert wird. Zu dem Einfluss auf die Performance wird im Unterkapitel 5.3 eingegangen.

Codeblock 5.4: Select-Befehl ohne Sicht

```
SELECT YEAR(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 YEAR(B.BESTELLDATUM), K.LAND
ORDER BY Jahr ASC, Gesamtumsatz DESC;
```

Man kann den Attributen einer Sicht auch eigene Namen vergeben, indem man sie in Klammern hinter dem Namen der Sicht aus der `CREATE VIEW`-Anweisung auflistet. Die Definition einer Sicht kann mit `DROP VIEW ‹view-name›` gelöscht werden, wodurch keine Abfragen mehr auf dieser Sicht ausgeführt werden können. Das Löschen der Sicht hat jedoch keine Auswirkungen auf die Tupel der zugrundeliegenden Tabellen. 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 anderen definierten 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 die Änderungen auf die zugrundeliegende Relation R übertragen. Dabei wird die `WHERE`-Bedingung der View zu den Bedingungen der Änderung im `WHERE`-Block hinzugefügt. Wenn die Bedingungen nicht erfüllt sind, wie im Beispiel (5.2), weil mehrere Relationen in der View verwendet werden, müssen Änderungen direkt an den zugrunde liegenden Tabellen vorgenommen werden. In diesem Fall kann die View nur für Select-Abfragen genutzt werden.

Das Einfügen über die Sicht ist jedoch nicht die intuitivste Möglichkeit, um Änderungen an den 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 (vgl. **silberschatz2011database**). 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

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
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 wird dazu im Kapitel 5.3 noch ein genaueres Beispiel betrachtet. Um Werte in eine virtuelle Sicht einzufügen, bietet sich jedoch das Konzept der Stored Procedures an. Stored Procedures sind Funktionen, die direkt im DB-Server hinterlegt werden und wie andere integrierte Funktionen, wie z.B. round(), aufgerufen werden können. Sie ermöglichen es, geschäftslogische Prozesse in der Datenbank zu speichern und direkt über SQL-Anweisungen auszuführen (vgl. **silberschatz2011database**).

Codeblock 5.6: Allgemeine Prozedur Deklaration

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

Damit für die Sicht aus dem Beispiel 5.2 Daten eingefügt werden können, muss die Prozedur die gleichen Parameter wie die Spalten der View erhalten. Die Parameter werden in der Funktion verarbeitet und die ermittelten Daten in die zugrunde liegenden Tabellen eingefügt. Wenn die Prozedur korrekt ist, dann werden die Änderungen bei der nächsten `SELECT`-Abfrage der View sichtbar.

Codeblock 5.7: Deklaration der Prozedur

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

Jetzt kann die Methode `insert_view` einfach mit dem `CALL`-Befehl aufgerufen werden und die Werte für die drei Parameter werden 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 dem jeweiligen Land lebt.

Im Vergleich zum direkten Einfügen in die Bestelltabelle geht jedoch Datenpräzision verloren. Einerseits fehlt das genaue Datum und andererseits sind die Informationen zur `KUNDEN_ID` und `ARTIKEL_ID` nicht vorhanden. 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 es in den zugrunde liegenden Tabellen zu einer geringeren Datenqualität kommen kann, da beispielsweise NULL-Werte oder andere Standardwerte verwendet werden. Deshalb sollten virtuelle Sichten grundsätzlich nur zur Abfrage von Daten benutzt werden und nicht für Änderungen. Stattdessen sollten die zugrunde liegenden Tabellen direkt angepasst werden.

5.2 Materialisierte Views

Allgemein werden Sichten so definiert, dass sie eine neue Relation aus den Basistabellen erzeugen, indem sie eine Abfrage auf diese Tabellen ausführen. Bisher wurden 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.

> „Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents." (**silberschatz2011database**)

Durch die physische Speicherung verringert sich der Rechenaufwand für Abfragen, da für das Beispiel (siehe 5.2) der Join nicht erneut ausgeführt werden muss. Die bereits gespeicherten Ergebnisse sind damit direkt abrufbar, was zu einer schnelleren Antwortzeit der Query führt. Passend zu der virtuellen Sicht (5.2) sieht die Materialisierte wie folgt aus:

Codeblock 5.8: Materialized View

```
1 CREATE MATERIALIZED VIEW UmsatzProJahrLand AS
2 SELECT
3     EXTRACT(YEAR FROM B.BESTELLDATUM) AS Jahr,
4     K.LAND AS Land,
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.LAND;
```

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, ähnlich wie bei Indizes. Wenn Änderungen an der zugrunde liegenden Basistabelle vorgenommen werden, ist die materialisierte Sicht nicht mehr aktuell. Die einfachste Lösung besteht darin, bei jeder Änderung eine vollständige Neuberechnung der Sicht durchzuführen (vgl. **silberschatz2011database**). Dies kann explizit mit dem folgenden Befehl durchgeführt werden:

Codeblock 5.9: Aktualisierung der materialisierten Sicht

```
REFRESH MATERIALIZED VIEW KUNDEN_MAT_OVERVIEW;
```

Die Anzahl an Neuberechnungen hat einen großen Einfluss auf die Performance, weshalb man sich ein Konzept überlegen, mit dem die Anzahl auf ein Minimum begrenzt wird. Ansonsten kann es durch Sperren auf die zugrunde liegenden Tabellen zu Einschränkungen in der Produktivumgebung kommen. In PostgreSQL erlaubt die Option `CONCURRENTLY` beim Aktualisieren einer materialisierten Sicht den gleichzeitigen Zugriff durch andere Prozesse, da die Sicht erst ersetzt wird, wenn die neue Version fertig ist (siehe 5.9).

Eine materialisierte Sicht kann wie eine virtuelle Sicht in der FROM-Klausel einer Abfrage verwendet werden. In Oracle gibt es zusätzlich noch eine Funktionalität, die es ermöglicht, Abfragen automatisch umzuschreiben. Damit kann die materialisierte Sicht auch verwendet werden, wenn sie nicht explizit in der Abfrage referenziert wird. Für diese Funktionalität muss die materialisierte Sicht mit der Funktion `ENABLE QUERY REWRITE` aktiviert werden. Die Abfrage wird aber nur dann umformuliert, wenn alle Relationen in der Sicht enthalten sind und die Bedingungen entsprechend angepasst werden.

Codeblock 5.10: Select mit View

```
SELECT Land, Jahr, Gesamtumsatz
FROM KUNDEN K JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
WHERE LAND = 'Deutschland' AND JAHR = 2024;
```

In Oracle könnte die Abfrage 5.10 intern so umgeschrieben werden, dass die Abfrage nicht auf diesen Tabellen erfolgt, sondern direkt auf die materialisierte Sicht `UmsatzProJahrLand`. Die materialisierte Sicht enthält bereits die aggregierten Umsätze und muss daher weniger Berechnungen durchführen. Bei der zweiten Abfrage 5.11 wird die materialisierte View nicht verwendet, da sie nicht die Spalten `STADT` und `MONAT` enthält. Wie in PostgreSQL bei beiden Befehlen erfolgt in diesem Fall auch bei Oracle keine automatische Abfrageumschreibung, weshalb die Abfrage explizit auf die Tabellen zugreifen muss.

Codeblock 5.11: Select nicht für View

```
SELECT Stadt, Monat, Gesamtumsatz
FROM KUNDEN K JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
WHERE STADT = 'Hamburg' AND EXTRACT(MONTH FROM K.GEBURTSTAG) = 8;
```

Neben der Verwendung der Option `CONCURRENTLY` gibt es noch weitere Optimierungen, um nicht jedes Mal die gesamte Sicht vollständig neu erstellen zu müssen. Dafür muss man sich vor Augen führen, dass alle Änderungen an der zugrunde liegenden Tabelle inkrementell sind. Auf diese Weise können Einfügungen, Löschungen und Aktualisierungen in einer

Basistabelle mit minimalem Abfrageaufwand durchgeführt und anschließend in der materialisierten Sicht aktualisiert werden. Diese inkrementelle Aktualisierung der materialisierten Sicht ist damit deutlich effizienter als die ständige Neuberechnung der Sicht. Aber nicht jedes Datenbankmanagementsystem unterstützt die inkrementelle Auffrischung. Oracle bietet diese Funktion nativ mithilfe von Materialized View Logs an, während in PostgreSQL eine manuelle Planung erforderlich ist, da keine automatische Auffrischung unterstützt wird (**mat_view_features_per_db**). MySQL bietet gar nicht erst eine Möglichkeit an, um materialisierte Sichten nativ zu erstellen. Allerdings kann man die Funktionsweise mithilfe einer physischen Tabelle und Triggern auf den zugrundeliegenden Tabellen nachstellen.

Zunächst eine physische Tabelle, z.B. mit dem Namen `KUNDEN_MAT_OVERVIEW`, erstellt. Diese Tabelle besteht, ähnlich wie die materialisierte Sicht 5.8, aus den Spalten `JAHR`, `LAND` und `GESAMTUMSATZ`, wobei die Kombination aus Jahr und Land der Schlüssel der Tabelle ist. Wenn nun Daten in die zugrundeliegenden Tabellen `KUNDEN` und `BESTELLUNG` eingefügt werden, bleibt die `KUNDEN_MAT_OVERVIEW`-Tabelle unverändert. Das leigt daran, weil bisher keine Verbindung zur neuen Tabelle hergestellt wurde. Dieses Problem kann gelöst werden, indem Trigger definiert werden, die bei Änderungen in der Bestell- oder Kundentabelle ausgelöst werden. Da die beiden Tabellen über die `KUNDEN_ID` verknüpft sind, kann der Fremdschlüssel mit der Option `ON DELETE CASCADE` versehen sein. Dadurch werden die Einträge, die in der Kundentabelle gelöscht werden, automatisch auch aus der Bestelltabelle entfernt und man muss nur die Änderungen in der Bestelltabelle als Auslöser für die Trigger beachten.

In MySQL kann ein Trigger nur für einen Datenbankmanipulationsoperator gleichzeitig verwendet werden (**mysql_trigger_syntax**), weshalb für `INSERT` und `DELETE` jeweils ein Trigger definiert werden muss. Da keine Datensätze aktualisiert werden, wird aus Simplizitätsgründen der Trigger für `UPDATE` vernachlässigt. Für das Beispiel sieht der `INSERT`-Trigger wie folgt aus:

Codeblock 5.12: Insert Trigger für die Tabelle Bestellung

```
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;
```

Nach dem Einfügen eines Datensatzes in die Bestelltabelle wird der Trigger aktiviert und überprüft, ob für das Land und Jahr bereits ein Eintrag in `KUNDEN_MAT_OVERVIEW` vorhanden ist. Ist dies der Fall, wird der Gesamtumsatz angepasst, andernfalls wird ein neuer Datensatz mit den entsprechenden Werten eingefügt. Ein Nachteil des Ansatzes mit einer normalen Tabelle in Kombination mit Triggern ist der erhöhte Aufwand für jede materialisierte Sicht. Zudem variiert dieser Aufwand stark, da er individuell vom jeweiligen Anwendungsfall abhängt.

Ein Anwendungsfall für die Nutzung von aggregierten Daten in einer materialisierten Sicht ist die Analyse von Daten, um Vorhersagen zu treffen. Wenn Analysten eines Motorradunternehmens beispielsweise den Einkauf für die Zukunft planen möchten, müssen sie oft auf aggregierte Daten aus der Vergangenheit zurückgreifen. Diese Entscheidungen werden jedoch nicht täglich abgefragt, sondern nur in regelmäßigen Abständen. Damit wird die materialisierte Sicht eher selten abgefragt, während Änderungen an den zugrundeliegenden Tabellen, wie z.B. der Bestand an Motorrädern oder die Anzahl der Motorradteile im Lager, sehr häufig vorkommen. Wenn man die Sicht bei jeder Änderung im Lager aktualisieren würde, würde dies zu einem enormen Aufwand führen. Daher kann es sinnvoll sein, die Daten nur einmal täglich zu aktualisieren, zum Beispiel durch einen Cron-Job in der Nacht. Zu dieser Zeit ist zusätzlich die Systemlast in der Regel gering. In diesem Fall haben die Analysten zwar nur den Stand des Vortages, aber da sie in der Regel mit vergangenen Daten arbeiten, ist dieses Risiko vertretbar. Anders ist es bei einer schnellen Lieferung an den Kunden, da es für den Verkauf entscheidend ist, über aktuelle Bestandsdaten zu verfügen.

Zusammengefasst lässt sich sagen, dass materialisierte Sichten und Indizes ähnliche Vorteile bei der Optimierung der Abfrageleistung bieten.

> „Indices are just like materialized views, in that they too are derived data, can speed up queries, and may slow down updates. Thus, the problem of index selection is closely related to that of materialized view selection, although it is simpler.“ (**silberschatz2011database**)

Allerdings ist 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 Indizes. 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 sich untereinander deutlich von der Größe unterscheiden.

5.3 Durchführung der Benchmarks

Das Ziel für die Durchführung ist es den Performanceunterschied zwischen einer virtuellen und einer materialisierten Sicht darzustellen. Dafür wird zuallererst mit der Umsetzung der virtuellen Sicht begonnen, für die, wie bei den anderen Sichten auch, zunächst die Basistabellen erstellt werden müssen. Als Basis werden die Tabellen Kunden (2.4) und Bestellung (2.5) verwendet und die View (5.2) erstellt, die bereits in Kapitel 5.1 beschrieben wurde. Anschließend werden die Testdaten direkt die beiden physischen Tabellen eingefügt und nicht über die virtuelle Sicht. Bei Select-Befehlen wird die Sicht explizit angesprochen und es werden mehrere Select-Befehle (siehe 5.13) auf verschiedenen Spalten untersucht, um die Unterschiede in der Lesegeschwindigkeit repräsentativ zu erfassen.

Codeblock 5.13: Select-Abfragen auf alle Spalten der View

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

Wie schon im Kapitel (5.1) erklärt, lassen sich die Abfragen auf die virtuelle Sicht in direkte Abfragen auf die Kundentabelle umwandeln. Um den Einfluss der virtuellen Sicht auf die Performance zu untersuchen, wird ein Benchmark mithilfe der Sicht durchgeführt, während bei der anderen Variante keine Sicht deklariert wird und alle Befehle auf die Sicht direkt in SQL-Befehle auf die zugrunde liegenden Tabellen umgewandelt werden.

Zusätzlich müssen die Ergebnisse mit und ohne virtualisierte Sicht auch mit dem im Kapitel 5.2 beschriebenen Ansatz unter Verwendung von Triggern in MySQL verglichen werden. Dafür müssen neben der Kunden- und Bestelltabelle auch die Tabelle `KUNDEN_MAT_OVERVIEW` sowie die Trigger für die `INSERT`- und `DELETE`-Operationen erstellt werden. Im nächsten Schritt werden die ersten beiden Tabellen mit Testdaten befüllt und bei den Select-Befehlen aus 5.13 der

Tabellenbezeichner angepasst. Damit sind alle Voraussetzungen für den Vergleich zwischen keiner View, der virtuellen View und dem Ansatz mit Triggern in MySQL erfüllt.

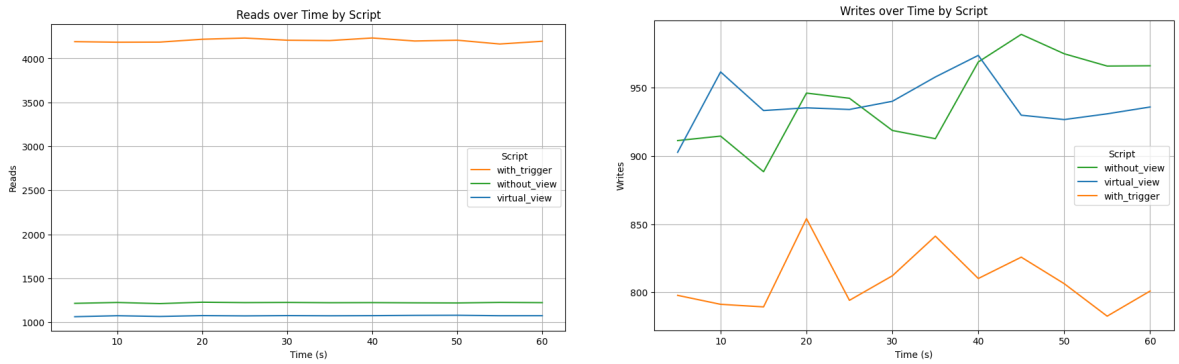


Abbildung 5.1: Vergleich zwischen keiner View, der virtuellen und Triggeransatz in MySQL

Bei den Ergebnissen fällt auf, dass die Unterschiede zwischen der virtuellen Sicht und den direkten SQL-Befehlen (`without_view`) nur minimal sind. Dennoch weist `with_view` eine leicht schlechtere Performance auf, sowohl bei Lese- als auch bei Schreiboperationen (5.1). Einen klaren Performancevorteil kann man beim Trigger-Ansatz erkennen, da die Lesewerte etwa um den Faktor 3 höher sind. Das liegt daran, dass direkt die Tabelle mit den aggregierten Werten abgefragt wird, wodurch weniger Rechenaufwand erforderlich ist. Anders hingegen sieht es bei der Schreibperformance aus, da die Trigger ausgelöst werden und zusätzliche Aktualisierungen an der Tabelle `KUNDEN_MAT_OVERVIEW` durchführt werden müssen. Dadurch sehen wie einen deutlichen Unterschied zu den anderen beiden Ansätzen, da die Werte bei den Schreibvorgängen etwa 15–20% langsamer sind.

Im letzten Benchmark sollen unterschiedliche Implementierungen von materialisierten Sichten getestet werden. Dazu wird der Ansatz mit Triggern in MySQL mit der nativen Implementierung in PostgreSQL verglichen. Die materialisierte Sicht in Postgres kann mithilfe des Befehls aus 5.8 direkt erstellt werden. Da Postgres die inkrementeller Auffrischung nicht unterstützt, muss die materialisierte Sicht nach den `INSERT` und `DELETE`-Befehlen auf der Kundentabelle immer vollständig aktualisiert werden. Da der Einfluss auf die Performance des Befehls 5.9 untersucht werden soll, wird dieser einmal nach der Einfügung jeder Zeile in die Kundentabelle und einmal, nachdem alle Datensätze eingefügt wurden, ausgeführt.

Um die Performanceunterschiede zwischen PostgreSQL und MySQL zu ermitteln, wird der Trigger-Ansatz auch in PostgreSQL implementiert. Die Implementierungen für die Insert- und Select-Befehle sind bei beiden DBMS 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.

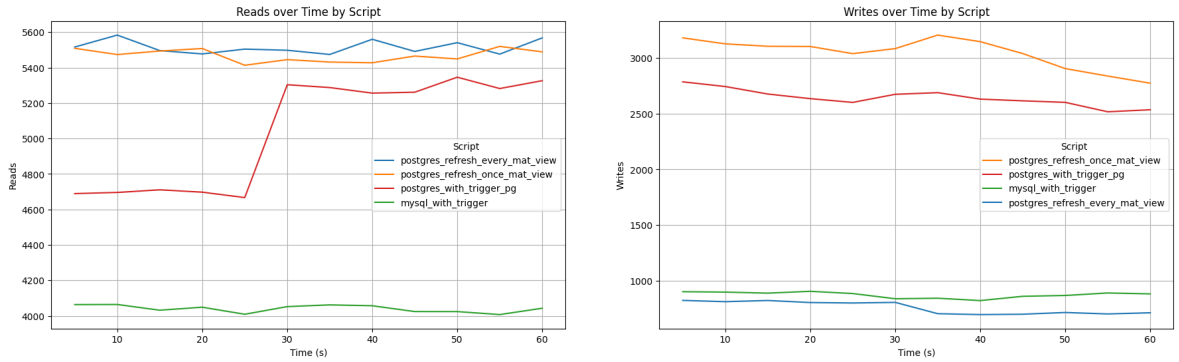


Abbildung 5.2: Vergleich zwischen Triggeransatz in MySQL und Postgres, sowie zwei nativen Implementierungen in Postgres

Bei der Grafik 5.2 wird zuallererst ein sehr deutlicher Performanceunterschied beim Ansatz mit den Triggern zwischen PostgreSQL und MySQL sichtbar. Begründet werden kann dieser Unterschied mit den verschiedenen Vorteilen des jeweiligen DBMS und dessen Umgebung. Beide Systeme wurden mit dem gleichen Ansatz gebenchmarkt, um die Implementierung der nativen materialisierten Sicht, die nur in PostgreSQL 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 (5.9) 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 Datenbankmanagementsystemen fällt wieder schwer, da die Unterschiede zwischen `with_trigger` und `with_trigger_postgres` sehr groß sind (etwa um Faktor 2–3). 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 vorliegenden Beispiel ist die einmalige Aktualisierung der Sicht besser als das Verwenden der Trigger in Postgres.

Es lässt sich also zusammenfassen, dass virtuelle Sichten wenig 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 Partitionen

In diesem Kapitel wird die Funktionsweise von Partitionen und das Verhalten des Abfrageoptimierers untersucht. Partitionierte Datenbanken existieren bereits seit den 1980er Jahren, haben jedoch in jüngerer Zeit durch NoSQL-Datenbanken und Hadoop-basierte Data Warehouses eine erneute Aufmerksamkeit erfahren (vgl. **kleppmann2017designing**). Auch in relationalen Datenbanken besteht die Möglichkeit, Daten auf verschiedene Partitionen zu verteilen. In MySQL stehen dabei verschiedene Partitionierungstypen zur Verfügung, darunter RANGE-, LIST-, HASH- und KEY-Partitionierung. Für jeden dieser Typen werden Anwendungsbeispiele präsentiert und mögliche Verwendungszwecke erläutert. Abschließend werden Benchmark-Tests durchgeführt, um die jeweiligen Vor- und Nachteile zu bewerten.

6.1 Grundlagen

Zunächst muss geklärt werden, was mit Partitionen gemeint ist.

> „Partitioning is a general term used to describe the act of breaking up data and distributing it across different hosts.“ (**da2015redis**)

In NoSQL-Datenbanken werden diese Partitionen häufig auf verschiedene Server verteilt, können jedoch auch innerhalb eines einzelnen Servers gespeichert werden. Im Gegensatz dazu bezieht sich die Partitionierung in MySQL ausschließlich auf die Verteilung von Tabellendaten innerhalb einer einzigen Datenbankinstanz. Dabei werden die Daten auf verschiedene physische oder logische Partitionen innerhalb desselben Servers verteilt. Zur Verteilung auf mehrere Server kann in relationalen Datenbanken Replikation eingesetzt werden, die im nächsten Kapitel 6 ausführlicher erläutert wird. Das System verwaltet sie intern so, dass der Benutzer nicht bemerkt, wie genau die Daten organisiert sind (**schwartz2012high**). Damit eine Tabelle die Partitionierung nutzt, muss bei ihrer Erstellung die `PARTITION BY`-Klausel angegeben werden, die festlegt, in welcher Partition jede Datenzeile gespeichert wird. Dies führt zu einer erhöhten Komplexität des `CREATE TABLE`-Befehls. In der Partitionsklausel selbst können nicht nur Ausdrücke und Berechnungen zur Bestimmung der Partitionierung eingesetzt werden, sondern auch Funktionen. Wenn Funktionen verwendet werden, müssen sie eine nicht-konstante und deterministische Ganzzahl zurückgeben, wie z.B. `YEAR()`.

Um Partitionen ordnungsgemäß definieren zu können, sind bestimmte Einschränkungen zu beachten. Zum einen müssen alle Spalten, nach denen die Partitionierung erfolgt, im Primärschlüssel oder Unique-Index enthalten sein. Andernfalls ist es nicht möglich, die Partitionen korrekt zu erstellen oder zu verwalten. Als logische Schlussfolgerung ergibt sich ein zusätzlicher Aufwand für die Pflege der neuen Indizes. Zudem können Fremdschlüssel-Bedingungen (engl. foreign key constraints) nicht verwendet werden. Ein weiteres Limit betrifft die maximale Anzahl an Partitionen pro Tabelle. Bei älteren MySQL-Versionen liegt dieses Limit bei 1024 und seit der MySQL-Version 8.0 bei 8192 Partitionen (**mysql_nof_partitions**). Wie später noch festgestellt wird, sollte aus verschiedenen Gründen die Anzahl der Partitionen so gering wie möglich gehalten werden. Aus diesem Grund ist diese Grenze nicht sehr relevant, sollte sie dennoch beachtet werden.

Nachdem die grundlegenden Bedingungen nun geklärt sind, wird im Folgenden die Funktionsweise erläutert. Wie bereits erwähnt, bestehen partitionierte Tabellen aus mehreren zugrunde liegenden Tabellen, die durch sogenannte Handler-Objekte verwaltet werden. Das Handler-Objekt fungiert als Schnittstelle, die es dem Datenbankmanagementsystem ermöglicht, mit den Partitionen zu interagieren. Dabei leitet es Anfragen an die Storage Engine weiter, die die Daten verwaltet. Aus Sicht der Storage Engine sind Partitionen normale Tabellen, unabhängig davon, ob sie eigenständig oder Teil einer partitionierten Tabelle sind. Abhängig vom jeweiligen Datenbankmanagementsystem können einzelne Partitionen entweder direkt oder, wie beispielsweise in MySQL, nicht direkt angesprochen werden. In Oracle ist dies wie folgt möglich:

```
1 SELECT * FROM your_table PARTITION (your_partition_name);
```

Die Verwendung von Indizes bei der Partitionierung wird von den verschiedenen DBMS unterschiedlich gehandhabt. In MySQL werden Indizes für jede Partition separat definiert, anstatt sie über die gesamte Tabelle hinweg zu erstellen. Dabei werden die Indizes identisch auf jede Partition angewendet. In Oracle hingegen gibt es neben den lokalen Indizes auch globale, die über die gesamte Tabelle hinweg erstellt werden, unabhängig von den Partitionen. Diese Methode ermöglicht eine effizientere Suche, erfordert jedoch eine komplexere Verwaltung.

Als Nächstes ist es wichtig zu verstehen, wie der Abfrageoptimierer (engl. Query Optimizer) arbeitet. Beim Ausführen von Abfragen versucht er, überflüssige Partitionen auszuschließen und sich gezielt auf diejenigen zu konzentrieren, die die relevanten Daten enthalten. Damit das sogenannte Pruning funktioniert, muss die `WHERE`-Klausel mit dem Partitionsausdruck übereinstimmen. Der Query Optimizer entscheidet bei SELECT-Abfragen, welche Partitionen ignoriert werden können und leitet die Anfrage gezielt weiter. Bei DELETE-Abfragen wird die betroffene Zeile lokalisiert und die Anfrage an die passende Partition übermittelt. Für INSERT-Abfragen wird zunächst die Zielpartition für die neue Zeile ermittelt. Erfolgt ein UPDATE innerhalb einer Partition, wird die Anfrage ebenfalls an die jeweilige Partition übermittelt. Wenn aber Teile der Partitionslogik verändert werden, dann stellt UPDATE eine

Kombination von INSERT und DELETE dar, da eine Einfügungsanfrage an die Zielpartition und eine Löschanfrage an die Quellpartition weitergeleitet wird. Die meisten dieser Operationen unterstützen Pruning, während, einige, wie z.B. INSERT-Abfragen von Natur aus ausschließend (engl. self-pruned) sind. Durch die Funktionsweise des Pruning lassen sich schon einige Vorteile der Partitionierung erkennen, die im Folgenden näher betrachtet werden.

Wie bei der Indexierung und der Datenclusterung einer Tabelle, trägt Partitionierung dazu bei, große Teile der Tabelle vom Zugriff auszuschließen und zusammengehörige Zeilen nahe beieinander zu speichern. Daher bietet es sich an, anstelle von indexierten Tabellen partitionierte Strukturen zu verwenden, um einen schnellen Zugriff auf die gewünschten Zeilen zu ermöglichen. Durch die korrekte Verteilung der Partitionen befindet man sich, wie bei Indizes, nahe der gewünschten Daten und kann von dort aus entweder das relevante Datengebiet sequentiell scannen oder es in den Speicher laden und indexieren. Im Gegensatz zu Indizes hat die Partitionierung aber zwei entscheidende Vorteile. Zum einen ist keine separate Datenstruktur erforderlich, auf die verwiesen werden muss und die ständig aktualisiert wird, wodurch nur geringer Mehraufwand verursacht wird. Stattdessen gibt es eine mathematische Formel, die bestimmt, welche Partitionen welche Kategorien von Zeilen enthalten können. Zum anderen lassen sich auch physisch verteilen, sodass der Server mehrere Festplatten effizienter nutzen kann. Besonders vorteilhaft ist dies, wenn die Tabellen sehr groß sind und nicht mehr vollständig in den Speicher passen.

> „The goal of partitioning is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load)."
> (**kleppmann2017designing**)

Außerdem steigt die Effizienz der Partitionierung, je mehr Partitionen durch die `WHERE`-Klausel in der Abfrage ausgeschlossen werden. Die Effizienz basiert aber auf zwei wesentlichen Annahmen. Erstens muss die Suche durch das Pruning von Partitionen bei der Abfrage eingegrenzt werden können. Zweitens darf die Partitionierung selbst keine hohen Kosten verursachen. Diese Annahmen sind jedoch nicht immer gültig, weshalb drei Anwendungsfälle mit möglichen Fehlern im Umgang mit Partitionen vorgestellt werden.

Zunächst sollte berücksichtigt werden, dass das Ergebnis einer Partitionierungsfunktion, wie z.B. `YEAR()`, den Wert `NULL` annehmen kann. Selbst wenn eine zeitbasierte Spalte als `NOT NULL` deklariert wird, können ungültige Datumswerte auftreten, die in MySQL in der ersten definierten Partition gespeichert werden. Eine Abfrage, die Jahre außer 2020 herausfiltert, muss daher zwei Partitionen durchsuchen, was insbesondere größeren Partitionen die Performance beeinträchtigt. Aus diesem Grund empfiehlt es sich, entweder eine dedizierte Partition für solche Sonderfälle einzuführen oder Erweiterung wie `RANGE COLUMNS` verwenden, die auf Funktionen in der Partitionsdefinition verzichten. Des Weiteren muss man bei der Definition von Indizes vorsichtig sein, wenn diese nicht mit der Partitionsklausel übereinstimmen, da sie

unerwartet zu umfassenderen Suche der Partitionen führen können. Daher sollte man es vermeiden, Indizes auf nicht partitionierten Spalten zu erstellen. Dies gilt nur dann nicht, wenn sichergestellt ist, dass die Abfragen Ausdrücke enthalten, die das Pruning der Partitionen unterstützen. Zuletzt sollte die Anzahl der definierten Partitionen begrenzt werden, da der Server die Partitionsdefinitionen linear durchsuchen muss, was mit steigender Partitionenzahl zunehmend ineffizient wird. Zusätzlich entsteht ein nicht vermeidbarer Mehraufwand durch das Öffnen und Sperren von Partitionen vor dem Pruning, was die Abfrageleistung weiter beeinträchtigen kann. Wie genau die Partitionen für die verschiedenen Typen definiert werden können, wird im Folgenden erläutert.

6.2 Arten der Partitionierung

In diesem Unterkapitel werden die verschiedenen Partitionierungsarten, die von MySQL unterstützt werden, jeweils mit einem Beispiel näher erläutert. Die Analyse der Ergebnisse erfolgt in Abschnitt 6.3.

Als Grundlage dienen die Kundentabelle (2.4) und die Bestelltabelle (2.5), die bereits in früheren Kapiteln verwendet wurden. Die Tabelle, die auf unterschiedliche Partitionen verteilt werden soll, ist die Kundentabelle. Allerdings müssen beide Tabellen noch angepasst werden, da es auch Einschränkungen für partitionierte Tabellen gibt. Zum einen muss bei der Bestelltabelle bei allen Typen die Fremdschlüssel-Bedingung entfernt werden und zum anderen muss der Primärschlüssel der Kundentabelle angepasst werden. Wie genau das passieren muss, wird an den Beispielen ersichtlich. Die Insert-Befehle sind bei allen Typen der Partitionierung gleich, bei den Select-Queries gibt es jedoch Unterschiede.

Der erste Typ, der betrachtet wird, ist die RANGE-Partitionierung. Bei dieser erfolgt die Zuordnung von Zeilen zu Partitionen basierend auf Spaltenwerten, die in einen definierten Wertebereich fallen. Für das Beispiel sollen unterschiedliche Partitionen je nach Alter des Kunden gebildet werden. Alle fünf Jahre wird eine neue Partition gebildet.

Damit es zu keinen Fehlern kommt, muss hier der Geburtstag auch Teil des Primärschlüssels der Kundentabelle sein. Außerdem muss die Spalte `GEBURTSTAG` bei der Bestelltabelle hinzugefügt werden, damit das Joinen der Tabellen über den Primärschlüssel effizienter ist. Seit MySQL 5.5 kann für Datumsspalten auch die Erweiterung `RANGE COLUMNS` verwendet werden, wodurch bei der Partitionsklausel die Funktion `YEAR()` nicht erforderlich ist. Um die Performance der beiden Ansätze zu vergleichen, werden beide Varianten bei der Tabellenerstellung verwendet. Die Tabelle mit der Funktion `YEAR` wird wie folgt erstellt:

Codeblock 6.1: Kundetabelle mit Range-Partitionierung

```sql
CREATE TABLE IF NOT EXISTS KUNDEN (
    KUNDEN_ID     INT NOT NULL,
    GEBURTSTAG    DATE NOT NULL,
    -- other attributes
    PRIMARY KEY (KUNDEN_ID, GEBURTSTAG)
) PARTITION BY RANGE (YEAR(GEBURTSTAG)) (
    PARTITION p1 VALUES LESS THAN (1955),
    PARTITION p2 VALUES LESS THAN (1960),
    -- other partitions
    PARTITION p15 VALUES LESS THAN (2025),
    PARTITION pmax VALUES LESS THAN MAXVALUE
);
```

Bei der Range-Partitionierung werden mehrere Select-Befehle getestet, da das Pruning je nach Art der Datumsabfrage besser oder schlechter funktioniert (siehe Abschnitt 6.1). Dazu wird zunächst die Kundentabelle mit der Bestelltabelle über die Attribute `KUNDEN_ID` und `GEBURTSTAG` gejoint. Die Testkunden werden so generiert, dass sie immer zufällig zwischen den Jahren 1950 und 2020 geboren sind. Die verschiedenen Select-Befehle unterscheiden sich in den folgenden `WHERE`-Bedingungen:

Codeblock 6.2: Unterschiedliche WHERE-Bedingungen

```sql
WHERE YEAR(k.GEBURTSTAG) = 1985;                              -- failing_pruning.sql
WHERE k.GEBURTSTAG BETWEEN '1985-01-01' AND '1985-12-31';       -- with_pruning.sql
WHERE k.GEBURTSTAG = '1985-01-01';                            -- with_primary_key.sql
```

Um zu überprüfen, ob der Optimierer die Partitionen pruned, wird der SQL-Befehl `EXPLAIN` vor dem `SELECT`-Befehl in 6.2 verwendet. Als Rückgabe des Befehls erhält man eine Übersicht, wie MySQL die Abfrage ausführt und welche Partitionen dabei verwendet wurden. Zunächst wird eine Select-Query analysiert, bei der keine `WHERE`-Klausel angegeben ist. Im Ergebnis von `EXPLAIN` sehen wir, dass der Abfragemechanismus alle Partitionen durchsuchen muss. Bei der `WHERE`-Klausel der Abfrage in Zeile 1 wird eigentlich erwartet, dass nur eine Partition abgefragt wird, jedoch wird das gleiche Resultat wie bei der vorherigen Abfrage zurückgegeben. Die Query aus der zweiten Zeile verweist direkt auf die Partitionsspalte und nicht auf einen Ausdruck. Und tatsächlich wird hier nur die Partition untersucht, die alle Kunden mit Geburtsdaten zwischen den Jahren 1985 und 1990 enthält. Diese Partition wird auch ausschließlich in der Abfrage der letzten Zeile benutzt. Daraus lässt sich schließen, dass MySQL nur dann Partitionen effizient prunen kann, wenn die Abfrage direkt auf die Partitionsspalte zugreift und keine Ausdrücke verwendet. Dieses Verhalten ähnelt dem von indexierten Spalten, die ebenfalls im Abfrageausdruck isoliert sein müssen, damit der Index zum Einsatz kommt.

Als Nächstes wird die LIST-Partitionierung betrachtet, bei der die Partitionen anhand von Spaltenwerten ausgewählt werden, die einem der vordefinierten diskreten Werte entsprechen. Zur Veranschaulichung soll pro Land eine eigene Partition erstellt werden. Dafür muss die Spalte `LAND` Teil des Primärschlüssels sein. Beim Befüllen der Tabellen wird für jeden Kunden ein zufälliges Land aus der Liste der 20 einwohnerreichsten Länder der Welt ausgewählt. Daher müssen bei der Erstellung der Tabelle auch 20 Partitionen sowie eine zusätzliche für sonstige Werte erstellt werden (6.3).

Codeblock 6.3: Kundetabelle mit List-Partitionierung

```
CREATE TABLE IF NOT EXISTS KUNDEN (
    KUNDEN_ID     INT NOT NULL,
    LAND          VARCHAR(100) NOT NULL,
    -- other attributes
    PRIMARY KEY (KUNDEN_ID, LAND)
) PARTITION BY LIST COLUMNS(LAND) (
    PARTITION p_china VALUES IN ('China'),
    PARTITION p_india VALUES IN ('India'),
    PARTITION p_united_states VALUES IN ('United States'),
    -- other partitions
    PARTITION p_thailand VALUES IN ('Thailand'),
    PARTITION p_other VALUES IN ('Other')
);
```

Im Rahmen der List-Partitionierung wird ebenfalls die Performance unterschiedlicher Select-Befehle überprüft. Zunächst wird, wie zuvor, die Kundentabelle mit der Bestelltabelle gejoint. Anschließend wird in der `WHERE`-Bedingung aber so gefiltert, dass nur Kunden aus Deutschland ausgewählt werden. Zusätzlich soll die Performance untersucht werden, wenn aus der Liste der Länder zufällig fünf ausgewählt werden und alle Kunden nur aus einem dieser fünf Länder stammen. Dazu werden drei verschiedene Ansätze getestet. Zum einen über den `OR`-Operator, zum anderen mithilfe des `IN`-Operators und zuletzt werden die 5 Länder, wie im ersten Beispiel, einzeln abgefragt und die Ergebnisse der 5 Abfragen mithilfe des `UNION`-Operators verbunden. Mithilfe von `EXPLAIN` wird sichtbar, dass bei allen Varianten nur die Partitionen der 5 Länder genutzt werden. Damit kann der Optimierer Bereiche in Listen diskreter Werte umwandeln, Elemente prunen und während der Abfrageverarbeitung Partitionen gezielt entfernen. Im Zusammenhang mit Joins ist der Effekt besonders stark, da MySQL bei einem partitionierten Schlüssel in der Join-Bedingung nur in den relevanten Partitionen nach übereinstimmenden Zeilen sucht. Welche der Varianten jedoch am effizientesten ist, wird erst bei der Analyse ersichtlich sein.

Zum Schluss wird die HASH-Partitionierung betrachtet, bei der die Partition anhand eines Hash-Werts zugewiesen wird, der aus den Spaltenwerten der Zeilen berechnet wird. Dadurch

kann eine gleichmäßige Verteilung der Daten auf die Partitionen werden.

> „Because of this risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key.“ (**kleppmann2017designing**)

Zur Umsetzung der Tabelle müssen ausschließlich die Zeilen aus dem Codeblock 6.4 am Ende des Create-Kunden-Befehls hinzugefügt werden. In diesem Fall wird auch nur eine einzige Select-Query getestet, die wieder beide Tabellen joined und in der `WHERE`-Bedingung überprüft, ob die `KUNDEN_ID` zwischen den Werten 1000 und 2000 liegt. Um mehr Werte miteinander zu vergleichen, werden verschiedene Varianten der Hash-Partitionierung getestet, indem die Anzahl der Partitionen variiert wird. Die Anzahl der Partitionen beträgt in den Benchmarks 5, 50 und 500. Einschließlich des Referenzfalls ergeben sich somit vier unterschiedliche Testfälle, die miteinander verglichen werden.

Codeblock 6.4: Hash-Partitonierung

```
1 PARTITION BY HASH(KUNDEN_ID)
2 PARTITIONS 5;
```

Die KEY-Partitionierung ähnelt der Hash-Partitionierung, verwendet jedoch die interne Hash-Funktion von MySQL und benötigt nur die Angabe einer oder mehrerer Spalten. Um die Performance zu überprüfen, wird genau das Gleiche wie bei der Hash-Partitionierung gemacht. Dafür muss im Codeblock 6.4 das Signalwort `HASH` mit `KEY` ersetzt werden und das Ergebnis wird mit dem von `HASH` verglichen. Die Erkenntnisse aus diesem Benchmark werden im nächsten Kapitel zusammengefasst.

6.3 Analyse

Im vorherigen Abschnitt wurden die verschiedenen Arten von Partitionen erläutert. Nun sollen für jedes dieser Beispiele Benchmarks durchgeführt und die Ergebnisse untersucht werden. Um den Einfluss der Partitionierung auf die Abfragen zu verdeutlichen, werden jeweils partitionierte mit nicht partitionierten Tabellen verglichen. Beide Varianten stellen die gleichen Insert-Befehle, während sich die Select-Queries je nach Partitionierungstyp leicht unterscheiden können. Abhängig vom Typ werden zusätzlich leicht unterschiedliche Abfragen gestellt. Die Ergebnisse des Referenzbenchmarks sollten weitgehend mit denen der partitionierten Varianten übereinstimmen. Es können jedoch kleinere Unterschiede auftreten, da jeweils zufällig generierte Daten eingefügt werden. Bei signifikanten Abweichungen sind die Performancemessungen jedoch schwerer miteinander vergleichbar.

Im ersten Benchmark mit der Range-Partitionierung fällt auf, dass die Benutzung von `RANGE` oder `RANGE COLUMNS` keinerlei Einfluss auf die Performance hat. Daher stellt sich bei der

Verwendung nicht die Frage nach der Performance, sondern nach der Präferenz des Nutzers. Bei der Analyse der Abbildung 6.1 wird deutlich, dass die `with_pruning`-Query mit erheblichem Abstand schneller ist als die anderen. Damit funktioniert das Pruning besonders gut, wenn die Geburtstage zwischen dem ersten und letzten Tage des Jahres abfragen. Als Nächstes kommen die Skripte ohne Partitionierung, die aber etwa 50% ineffizienter sind, als das Skript zuvor. Da beide auf einem sehr ähnlichen Niveau liegen, kann davon ausgegangen werden, dass die Verarbeitung durch `YEAR()` hier keine Rolle spielt. Etwas langsamer bei der Range-Partitionierung ist `with_primary_key`, bei der nur die Kunden abgerufen werden, die am 1. Januar 1985 geboren wurden. Wenn die Ergebnisse von `EXPLAIN` betrachtet werden, zeigt sich, dass bei der Abfrage nur eine Partition verwendet wurde. Als Letztes kommen die beiden Select-Queries, die alle Partitionen durchsuchen. Bei der einen Abfrage wurde keine `WHERE`-Bedingung angegeben und bei der anderen wurde die Funktion `YEAR()` verwendet. Es lässt sich also feststellen, dass Pruning mithilfe von `YEAR()` offensichtlich nicht funktioniert. Dies hat auch der Ausführungsplan für die Query bestätigt (siehe Kapitel 6.2). Bei den Insert-Befehlen ist der Fall ohne Partitionierung nur geringfügig schneller als mit Partitionierung.

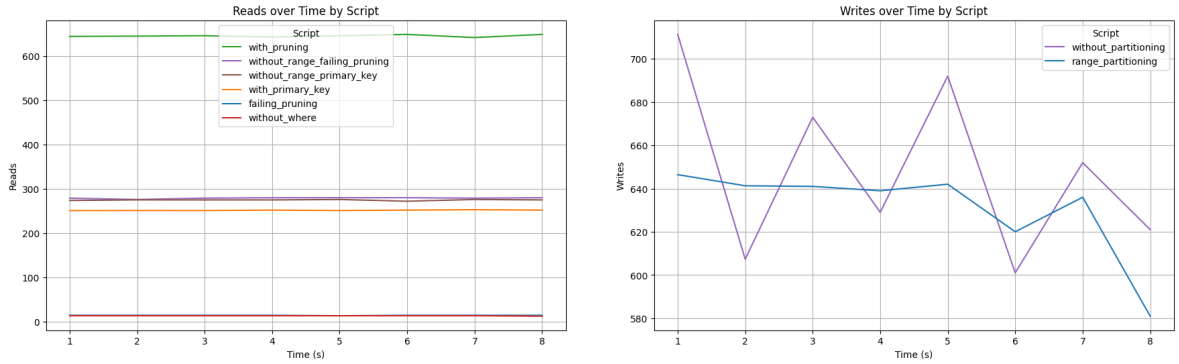


Abbildung 6.1: Vergleich zwischen der Range-Partitionierung und ohne Partition

Bei der List-Partitionierung 6.2 können ebenfalls einige interessante Beobachtungen gemacht werden. Beim ersten Fall ist nur ein Land in der WHERE-Bedingung vorhanden. Wenn dies so ist, dann hat die Partitionierung einen erheblichen Vorteil gegenüber der Version ohne Partitionierung (siehe rote Linie von `with_pruning_simple` und braune von `without_list_pruning_simple`). Wenn statt nur eines Landes mehrere abgefragt werden, zeigen sich für die verschiedenen Operatoren unterschiedliche Ergebnisse. Die beste Performance erzielt der IN-Operator. Dicht darauf folgt der OR-Operator, während der Fall ohne Partitionierung mit etwas größerem Abstand kommt. Deutlich abgeschlagen ist die Verbindung der Ergebnisse mit dem UNION-Operator. Die Performance beim Einfügen der Daten ist bei der Partitionierung und dem Referenzfall sehr ähnlich, wobei letzterer einen leichten Vorteil hat.

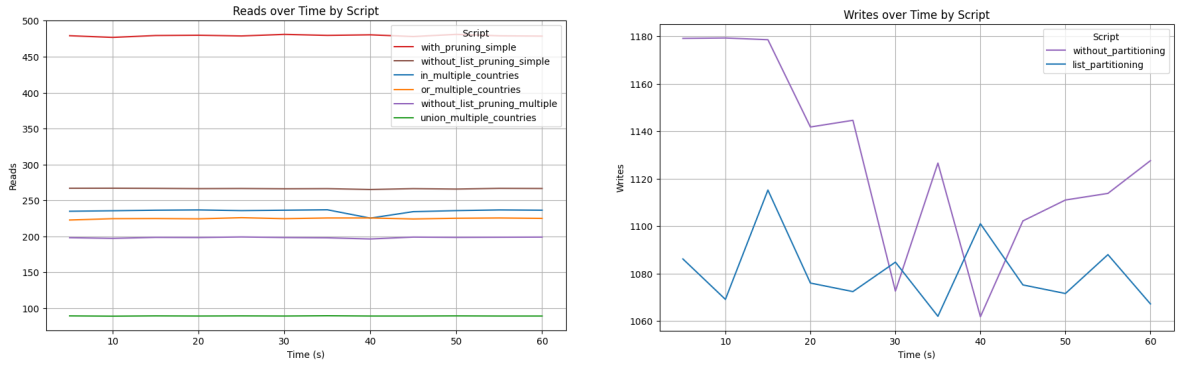


Abbildung 6.2: Vergleich zwischen der List-Partitionierung und ohne Partition

Bei der Key-Partitionierung fällt auf, dass es keinen signifikanten Performance-Unterschied zur Hash-Partitionierung gibt, sofern derselbe Datensatz und die gleiche Anzahl von Partitionen verwendet werden. Generell ist die Key-Partitionierung häufig stabiler und optimierter ist, insbesondere wenn es um Primärschlüssel geht. Bei der Hash-Partitionierung 6.3 fällt auf, dass die Werte der Abfragen sehr konstant sind. Nur bei der Variante mit 500 Partitionen gibt es deutlich sichtbare Schwankungen. Diese liegen aber nicht an dem Pruning, denn mit dem SQL-Befehl `EXPLAIN` sehen wir, dass alle 500 Partitionen benötigt werden und keine Partitionen zufällig geprunt werden. Die Hash-Partitionierung berechnet aus dem Wert einer bestimmten Spalte mithilfe einer Hash-Funktion einen Hash-Wert und anhand dessen wird die Zeile einer der Partitionen zugewiesen. Da der Hash-Wert bei 500 Partitionen mit modulo 500 gebildet wird, landet jeder 500-ste Wert in der gleichen Partition. Damit ist klargestellt, dass immer alle Partitionen benutzt werden, was die folgenden Ergebnisse zeigen. Was die Performance angeht, zeigt sich, dass die Abfrage ohne Partitionierung am schnellsten ist. Danach lässt sich die Regel ableiten, dass eine höhere Anzahl an Partitionen zu einer langsameren Abfrage führt. Dies liegt daran, dass mehr Partitionen die Suche innerhalb der Struktur komplexer machen und dadurch die Performance beeinträchtigen. Der Unterschied zwischen ohne Partitionen und 5 Partitionen ist noch überschaubar, aber bei 500 Partitionen sieht man einen sehr deutlichen Unterschied. Damit wurde die Regel aus dem Kapitel 6.1 bestätigt.

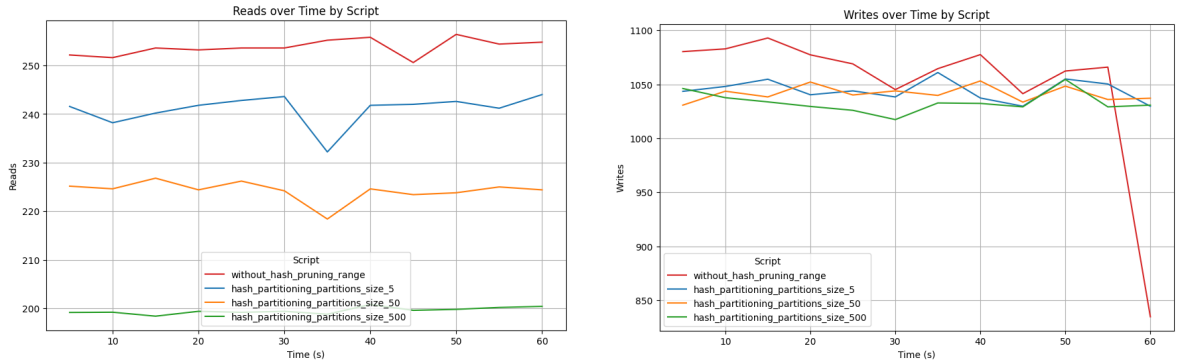


Abbildung 6.3: Vergleich zwischen der Hash-Partitionierung und ohne Partition

7 Replikation

In diesem Segment wird das Thema Replikation behandelt. Replikation kann die Grundlage für den Aufbau großer, leistungsstarker Anwendungen auf der Basis von MySQL sein. Es verfolgt dabei die sogenannte „Scale-Out"-Architektur, bei der mehrere Storage-Knoten parallelisiert arbeiten (vgl. **schwartz2012high**). Nach außen hin wirkt es trotzdem wie ein einziges Gesamtsystem. Bei dieser Architektur ist die Skalierbarkeit nahezu unbegrenzt, da man durch einfaches Hinzufügen weiterer Speicherknoten die Performance verbessern kann. Im Gegensatz dazu ist Scale-Up durch die Systemgrenzen eines einzelnen Geräts limitiert. Mit Scale-Out sind allerdings auch Nachteile verbunden, die später in diesem Abschnitt behandelt werden. Wie schon in den vorherigen Kapiteln werden zunächst die Grundlagen behandelt, gefolgt von der Betrachtung der Konfiguration, die die Basis für die Benchmarks bildet und abschließend erfolgt eine Analyse der Ergebnisse.

7.1 Grundlagen

Replikation ermöglicht die Konfiguration eines oder mehrere Server als Replikate eines anderen Servers, auch Master genannt. Sowohl die Begrifflichkeit Master-Replikat als auch die Varianten Primary-Secondary und Primary-Replica sind gebräuchlich.

> „Replication means keeping a copy of the same data on multiple machines that are connected via a network." (**kleppmann2017designing**)

Das grundlegende Problem, das die Replikation löst, besteht darin, die Daten eines Servers mit denen eines anderen synchron zu halten. Es können sich mehrere Replikate mit einem einzigen Master verbinden und dessen aktuellen Zustand widerspiegeln. Sie sind aber nicht dazu geeignet, richtige Backups zu ersetzten. Master und Replikate lassen sich auch in anderen Konfigurationen anordnen. Neben der klassischen Master-Replikat-Variante können Replikate selbst als Master für weitere Replikate dienen. Zudem ist eine Master-Master-Kombination denkbar. Die Datenreplikation kann aus verschiedenen Gründen vorteilhaft sein:

> „Replicas are very useful in a master failure scenario because they contain all of the most recent data and can be promoted to master." (**da2015redis**)

Zusätzlich sorgt sie dafür, dass die Daten näher an den Nutzern liegen, wodurch die Latenz verringert wird (vgl. **kleppmann2017designing**). Außerdem verbessert sie die Verfügbarkeit und ermöglicht eine bessere Skalierbarkeit. Effizienzvorteile gibt es insbesondere durch die Lastverteilung, bei der Leseanfragen auf mehrere Server verteilt werden. Daher ist Replikation besonders für leselastige Anwendungen vorteilhaft ist.

Im folgenden Abschnitt wird die Funktionsweise der Replikation erklärt, wobei der Fall mit einem Master und einem oder mehreren Replikaten betrachtet wird (**schwartz2012high**). Unmittelbar bevor eine Transaktion, die Daten aktualisiert, auf dem Master abgeschlossen wird, zeichnet der Master die Änderungen in seinem Binärlog (engl. binary log) auf. MySQL schreibt die Transaktionen seriell ins Binary-Log und informiert die Storage Engines nach dem Schreiben der Ereignisse darüber, die Transaktionen zu committen. Zu diesen Änderungen können beispielsweise neu deklarierte Tabellen oder Trigger sowie Einfügeoperationen in bestehende Tabellen gehören. Im nächsten Schritt muss das Replikat die Veränderungen auf dem Master mitbekommen. Dazu wird ein Worker-Thread gestartet, der als I/O-Replikations-Thread (engl. I/O-Slave-Thread) bezeichnet wird und eine Client-Verbindung zum Master öffnet (siehe Abbildung 7.1). Daraufhin wird ein spezieller Prozess gestartet, der die Ereignisse aus dem Binary-Log des Masters liest (engl. binlog dump process). Nach dem Verarbeiten schreibt der Thread die Werte auf seine eigene Festplatte in das sogenannte Relay-Log. Wenn er alle Ereignisse auf diesem Log verarbeitet hat, geht er in einen passiven Zustand und wartet auf Aktualisierungen. Den letzten Teil des Prozesses übernimmt der SQL-Slave-Thread. Dieser liest und spielt Ereignisse aus dem Relay-Log ab und aktualisiert die Daten der Replikate, sodass sie mit denen des Masters übereinstimmen. Wenn beide Threads eine etwa gleich schnelle Verarbeitung haben, dann bleibt das Relay-Log normalerweise im Cache des Betriebssystems und es gibt nur sehr geringe Mehrkosten (engl. Overhead). Die Ereignisse, die der SQL-Thread ausführt, können optional zusätzlich in das eigene Binary-Log der Replikate geschrieben werden.

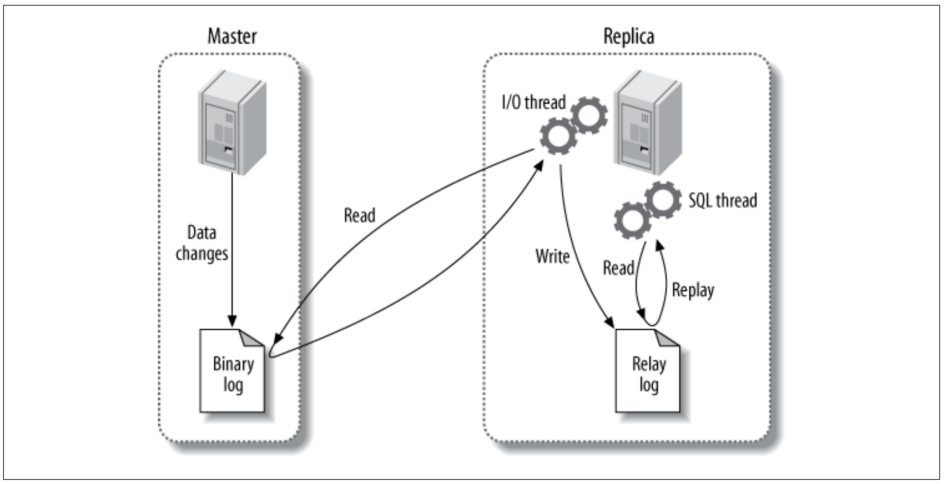


Abbildung 7.1: Darstellung der unterschiedlichen Threads

Die Abbildung 7.1 zeigt die beiden Replikation-Threads, die auf dem Replikat laufen. Zusätzlich gibt es jedoch einen weiteren Thread auf dem Master, der die vom Replikat zum Master geöffnete Verbindung auf dem Master startet.

Die Replikationsarchitektur entkoppelt die Prozesse des Abrufens und Schreiben von Ereignissen auf dem Replikat. Dadurch können die beiden Threads asynchron arbeiten, sodass der I/O-Thread unabhängig vom SQL-Thread agieren kann. Dies hat jedoch zur Folge, dass Änderungen, die auf dem Master parallel in verschiedenen Threads ausgeführt werden, auf dem Replikat nicht parallelisiert werden können. Das liegt daran, dass die Veränderungen auf dem Replikat in einem einzigen Thread abgearbeitet werden. Generell gibt es auch keine Garantie für die Latenz des Replikats und große Abfragen können dazu führen, dass das Replikat Sekunden, Minuten oder sogar Stunden hinter dem Master zurückbleibt. Der Flaschenhals (engl. bottleneck) des gesamten Systems stellt die Anzahl der Schreibvorgänge dar, die der langsamste Thread ausführen kann.

Wie aus der Funktionsweise der Replikation hervorgeht, verursacht der Master nur einen geringen Mehraufwand. Wenn das binäre Logging für Backups und Point-in-Time-Recovery genutzt wird, kann es jedoch deutlich mehr Ressourcen beanspruchen. Jedes angeschlossene Replikat verursacht nur eine geringe zusätzliche Last (hauptsächlich Netzwerk-I/O) auf dem Master. Trotzdem sollten die Auswirkungen vieler Replikate nicht unterschätzt werden, da sie im Wesentlichen zu unnötiger Daten-Duplikation führen.

Als Nächstes werden die zwei verschiedenen Arten der Replikation betrachtet, die von MySQL unterstützt werden: die anweisungsbasierte (engl. statement-based) und die zeilenbasierte (engl. row-based) Replikation. Die anweisungsbasierte Replikation wird seit MySQL 5.0 und älter unterstützt und funktioniert, indem die Abfrage, die die Daten auf dem Master geändert hat, protokolliert wird. Wenn ein Replikat das Ereignis aus dem Relay-Log liest und ausführt, wird die tatsächliche SQL-Abfrage erneut ausgeführt, die der Master ausgeführt hat. Der offensichtlichste Vorteil davon ist, dass sie relativ einfach zu implementieren ist und das Protokollieren sowie Wiederholen der Anweisungen das Replikat logischerweise mit dem Master synchron halten sollte. Außerdem sind die Binary-Log-Ereignisse in der Regel recht kompakt und verbrauchen nicht viel Bandbreite. In der Praxis gibt es jedoch Änderungen auf dem Master, die von Faktoren abhängen, die über den reinen Abfragetext hinausgehen. Beispielsweise werden Anweisungen zu leicht oder sogar deutlich unterschiedlichen Zeiten auf dem Master und dem Replikat ausgeführt. Deshalb muss das Binary Log nicht nur den Abfragetext, sondern auch Metadaten wie den aktuellen Zeitstempel enthalten. Einige Anweisungen kann MySQL nicht korrekt replizieren, wie zum Beispiel Abfragen, die die Funktion CURRENT_USER() verwenden. Auch gespeicherte Routinen und Trigger stellen bei dieser Art der Replikation ein Problem dar.

Die zeilenbasierte Replikation speichert die tatsächlichen Datenänderungen im Binary-Log. Ein großer Vorteil, der daraus folgt, ist, dass MySQL jede Anweisung korrekt replizieren kann.

Zudem können einige Änderungen mithilfe der zeilenbasierte Replikation effizienter sein, da das Replikat die Abfragen, die die Zeilen auf dem Master geändert haben, nicht erneut ausführen muss. Zum Beispiel, wenn eine Abfrage viele Zeilen in der Quelltabelle scannt, aber nur drei Zeilen in der Zieltabelle bearbeitet. Bei der anweisungsbasierten Replikation müsste ein Replikat die Anweisung erneut ausführen, nur um ein paar Zeilen zu erstellen, während dies bei der zeilenbasierten Replikation effizient und trivial ist. Andererseits ist das folgende Ereignis deutlich günstiger mit statement-basierter Replikation zu replizieren:

```
1 UPDATE master_table SET col1 = 0;
```

Die Verwendung der zeilenbasierten Replikation für diese Abfrage wäre sehr teuer, da jede Zeile geändert und somit ins Binary-Log geschrieben wird. Dadurch würde das Binary-Log-Ereignis extrem groß werden, was sowohl beim Protokollieren als auch bei der Replikation zu einer höheren Last auf dem Master führen würde. Damit werden nun weitere Vor- und Nachteile der unterschiedlichen Arten betrachtet.

Die anweisungsbasierte Replikation eignet sich besser, wenn das Schema auf Master und Replikat unterschiedlich ist und unterstützt Szenarien mit unterschiedlichen, aber kompatiblen Datentypen oder Spaltenreihenfolgen. Zudem erleichtert sie Schemaänderungen auf Replikaten, die später als Master dienen sollen, wodurch Ausfallzeiten reduziert werden können. Im Gegensatz dazu kann die zeilenbasierte Replikation bei Schemaänderungen auf einem Replikat bestimmte Operationen nicht ausführen, bietet jedoch eine zuverlässige Funktionalität mit allen SQL-Konstrukten. Sie stoppt auch bei Fehlern, z.B. wenn eine erwartete Zeile auf dem Replikat fehlt und weist damit auf Inkonsistenzen hin, während der andere Typ keine Hinweise auf fehlende Einträge gibt. Die Fehlersuche und das Verständnis von Problemen sind bei der anweisungsbasierten Replikation einfacher, da die Änderungen über verständliche SQL-Anweisungen erfolgen. Bei der zeilenbasierten Replikation ist die Nachvollziehbarkeit der Änderungen dagegen schwieriger, jedoch gibt es dafür weniger Locking-Probleme. Die zeilenbasierte Replikation erleichtert die Datenwiederherstellung durch das Speichern alter Daten, wobei eine Wiederherstellung zu einem bestimmten Zeitpunkt mit einem Binary-Log im zeilenbasierten Format zwar schwieriger, aber möglich ist. Außerdem benötigt sie häufig weniger CPU-Ressourcen, da keine komplexe SQL-Ausführungslogik erforderlich ist.

Da kein Format in jeder Situation perfekt ist, kann MySQL dynamisch zwischen statement-basierter und row-basierter Replikation wechseln. Standardmäßig wird die statement-basierte Replikation verwendet, aber wenn MySQL ein Ereignis erkennt, das nicht korrekt als Statement repliziert werden kann, wechselt es automatisch zur row-basierten Replikation. Alternativ kann das Format auch durch Setzen der Variable `binlog_format` manuell gesteuert werden.

7.2 Konfiguration des Master-Replikat-Ansatzes

Um Benchmarks ausführen zu können, muss der Master-Replika-Ansatz in MySQL konfiguriert werden. Zunächst gilt es, das Szenario der Replikation festzulegen, das umgesetzt werden soll. In diesem Kapitel wird das Modell mit einem Master und einer beliebigen Anzahl an Replikaten betrachtet. Die erforderlichen Schritte umfassen das Erstellen der Master- und Replikationsknoten und der anschließenden Anweisung an das Replikat, sich mit dem Master zu verbinden. Abschließend muss die Replikation gestartet werden.

Nach dem Starten der Knoten müssen einige spezielle MySQL-Privilegien berücksichtigt werden, die erforderlich sind, damit die Replikationsprozesse ordnungsgemäß ausgeführt werden können. Dazu muss ein Benutzer auf dem Master erstellt werden und diesem die richtigen Privilegien zugewiesen werden, damit der I/O-Thread sich als dieser Benutzer verbinden und das Binary-Log des Masters lesen kann. Außerdem darf nicht vergessen werden, die Datenbank zu erstellen, auf der die Benchmarks ausgeführt werden (wie in 2.1).

Codeblock 7.1: Datenbank- und Nutzererstellung sowie Rechtevergabe

```
1 CREATE DATABASE sbtest;
2 CREATE USER 'repl'@'%' IDENTIFIED WITH sha256_password BY 'repl_password';
3 GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
```

Diesen Nutzer muss nur auf dem Master erstellt werden, ist jedoch auch für die Verbindung der Replikate mit dem Master erforderlich. Im nächsten Schritt müssen einige Einstellungen auf dem Master und den Replikaten vorgenommen werden. Zum einen muss die Binärprotokollierung aktiviert und zum anderen eine einzigartige ID mit dem Parameter `server_id` angegeben werden. Wenn die Binärprotokollierung in der Konfigurations-Datei nicht bereits angegeben wurde, muss MySQL neu gestartet werden. Alternativ können die Einstellungen auch direkt beim Starten des Containers angegeben werden. Um zu überprüfen, ob die Binary-Logdatei auf dem Master erstellt wurde, kann man abhängig von der MySQL-Version folgende Befehle ausführen:

Codeblock 7.2: Anzeige der Konfiguration

```
1 SHOW BINARY LOG STATUS; --MySQL ≥8.0.23
2 SHOW MASTER STATUS; --MySQL <8.0.23
```

Die wichtigste Einstellung für das Binär-Logging auf dem Master ist `sync_binlog`, wobei der Wert auf `1` gesetzt werden muss. Diese Option sorgt dafür, dass MySQL den Inhalt des Binary-Logs, bei jedem Transaktions-Commit auf die Festplatte synchronisiert. Ist die Option deaktiviert, verringert sich der Arbeitsaufwand des Servers, jedoch könnten Binary-Log-Einträge bei einem Absturz verloren gehen. Auf einem Replikat, das nicht als Master

dient, erzeugt diese Option unnötigen Mehraufwand. Es wird außerdem empfohlen, einen Basisnamen für das Binary-Log explizit anzugeben, um einheitliche Namen auf allen Servern zu gewährleisten und Änderungen bei einem Hostnamenwechsel zu vermeiden. Dazu muss ein Argument für die `log_bin`-Option angegeben werden.

Es gibt noch weitere optionale Konfigurationsparameter, die hinzugefügt werden können. Einer davon ist der Parameter `relay_log`, der den Speicherort und den Namen des Relay-Logs festlegt. Ein weiterer wichtiger Parameter ist `log_slave_updates`, der es dem Replikat ermöglicht, replizierte Ereignisse in sein eigenes Binary-Log zu schreiben. Die Option `skip_slave_start` sorgt dafür, dass das Replikat nach einem Absturz automatisch startet, was die Möglichkeit offenlässt, den Server im Falle eines Problems zu reparieren. Zudem sorgt die Option `read_only` dafür, dass die meisten Benutzer keine nicht-temporären Tabellen ändern können. Die einzigen Ausnahmen bilden der Replikation-SQL-Thread und Threads mit dem `SUPER`-Privileg, weshalb dieses Privileg normalen Benutzern nicht zugewiesen werden sollte. In dem Anwendungsfall haben die Replikate die Option `read_only` aktiviert. Wenn das Replikat stark im Rückstand ist, kann der I/O-Thread den Festplattenspeicher füllen. Mit der Option `relay_log_purge` kann verhindert werden, dass der Replikation-SQL-Thread diese entfernt diese, sobald er mit deren Verarbeitung fertig ist.

Der nächste Schritt besteht darin, dem Replikat mitzuteilen, wie es sich mit dem Master verbinden und dessen Binary-Logs abspielen kann. Umgesetzt kann das mit der Ausführung des folgenden Befehls auf allen Replikaten:

Codeblock 7.3: Verbindung des Replikats zum Master

```
1 CHANGE MASTER TO
2   MASTER_HOST='YOUR_HOST_NAME',
3   MASTER_USER='YOUR_USER',
4   MASTER_PASSWORD='YOUR_PASSWORD',
5   MASTER_LOG_FILE='mysql-bin.000001',
6   MASTER_LOG_POS=0;
```

Die Spalten `MASTER_LOG_FILE` und `MASTER_LOG_POS` müssen mit dem Ergebnis von dem Befehl aus 7.2 übereinstimmen. Um sicherzustellen, dass die Datenbank `sbtest` und der Benutzer mit den entsprechenden Privilegien tatsächlich existieren, muss der Befehl aus 7.2 bereits vor der Ausführung des Befehls in 7.1 ausgeführt werden. Um die eigentliche Replikation zu starten, muss man den folgenden Befehl auf den Replikaten ausführen:

Codeblock 7.4: Starten der Replikation

```
1 START SLAVE;
```

Mit dem folgenden Befehl lässt sich überprüfen, ob die Durchführung erfolgreich war:

Codeblock 7.5: Status des Replikats

```
1 SHOW PROCESSLIST\G;
```

Die Spalten `Slave_IO_State`, `Slave_IO_Running` und `Slave_SQL_Running` zeigen an, ob die Replikationsprozesse laufen oder nicht. Wenn `Seconds_Behind_Master` nicht mehr NULL ist, bedeutet das, dass der I/O-Thread bereits alle Binary-Logs abgerufen hat und nun auf ein Ereignis vom Master wartet. Man sollte auch beobachten können, dass die verschiedenen Datei- und Positionswerte auf dem Replikat inkrementiert werden, wenn man Änderungen an dem Master vornimmt. Außerdem sollten zwei Threads auf dem Replikat aktiv sein, die unter dem Benutzer „system user“ laufen.

Bei den bisherigen Setup-Anweisungen wurde von einer frischen Installation ausgegangen. Es gibt aber auch andere Möglichkeiten, um ein Replikat von einem anderen Server zu initialisieren. Zum einen kann man bereits existierende Daten von einem Master kopieren, ein Replikat von einem anderen Replikat klonen oder ein Replikat aus einem aktuellen Backup starten. Um ein Replikat mit einem Master zu synchronisieren, sind drei Elemente erforderlich: eine Momentaufnahme der Master-Daten zu einem bestimmten Zeitpunkt, die Log-Datei des Masters mit dem entsprechenden Byte-Offset (ermittelbar durch den Befehl 7.2) sowie die Binary-Logs des Masters ab diesem Zeitpunkt. Eine kalte Kopie erfordert das Herunterfahren des Masters, um dessen Dateien zu kopieren, bevor er mit einem neuen Binary-Log neu gestartet wird, was jedoch zu Ausfallzeiten führt. Bei einer warmen Kopie können die Dateien übertragen werden, während der Server weiterhin läuft.

7.3 Analyse

Im vorherigen Abschnitt wurde erklärt, wie man den Master und vor allem die Replikate korrekt konfiguriert und den Prozess der Replikation startet. Für die Durchführung der Benchmarks werden erneut die Kundentabelle und die Bestelltabelle aus Kapitel 2.2 benötigt. Die einzigen erforderlichen Anpassungen betreffen das Festlegen des Binlog-Formats, das über die Variable `binlog_format` definiert wird und die Werte `STATEMENT`, `ROW` oder `MIXED` annehmen kann. Diese Einstellung kann entweder global für den gesamten Server oder lokal für die aktuelle Sitzung mithilfe des Befehls `SET SESSION` geändert werden. Damit lassen sich die Performanceunterschiede zwischen den einzelnen Arten, insbesondere bei den Einfügeoperationen, vergleichen. Der eigentliche Aufwand bei diesen Benchmarks besteht in der Einrichtung der Replikation auf dem lokalen Rechner und im Workflow, während die Veränderungen an den Lua-Skripten minimal sind.

Im ersten Vergleich sollen die Performanceunterschiede zwischen dem Master-Replikat-Ansatz und dem Ansatz mit einem einzelnen MySQL-Server festgestellt werden. Beim Master-

Replikat-Ansatz wird mit `ROW` immer der Standardwert des Binlog-Formats verwendet. Damit keine Fehler auftreten, müssen die beiden Ansätze miteinander kompatibel gemacht werden. Das Problem dabei ist, dass der Standardport von MySQL (3306) nicht gleichzeitig verwendet werden darf. Daher wird der Master auf Port 3307 gestartet und jedes Replikat erhält einen um 1 erhöhten Wert. Somit nutzt das dritte Replikat den Port 3310. Die Anzahl der Replikate lässt sich in der `envs.json`-Datei über die Variable `REPLICAS_COUNT` festlegen. Die Voraussetzung dafür ist, dass es lokal mindestens diese Anzahl an gestarteten und konfigurierten Replikate gibt. Innerhalb des Workflowjobs muss nichts anpasst werden, da dort `REPLICAS_COUNT` dazu genutzt wird, die exakte Anzahl an Replikate zu starten. Wichtig ist noch zu erwähnen, dass die Insert-Befehle nur auf dem Master, also Port 3307, ausgeführt werden. Die Select-Befehle werden sowohl auf dem Master als auch auf die Replikate ausgeführt.

Beim Betrachten der Ergebnisse aus Abbildung 7.2 fällt auf, dass die Version ohne Replikation am schnellsten ist. Danach folgen die Readabfragen auf dem Master auf Port 3307. Der Unterschied der beiden liegt bei etwa 15%. Nah an dem Ergebnis vom Master ist das erste Replikat (3308). Überraschenderweise folgen mit etwas mehr Abstand erst die anderen beiden Replikate auf den Ports 3309 und 3310. Bei der Schreibgeschwindigkeit gibt es einen deutlichen Unterschied zwischen den beiden Varianten, da durch den Prozess der Replikation die Schreibgeschwindigkeit ungefähr 60% langsamer ist.

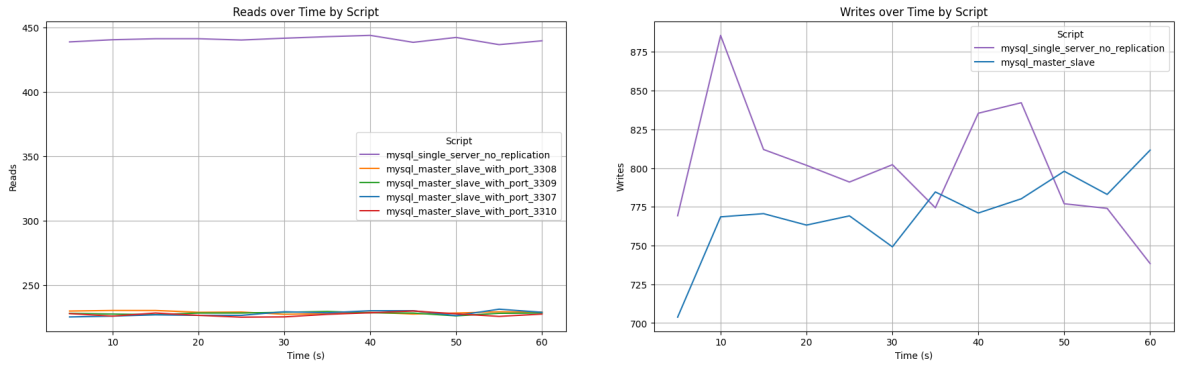
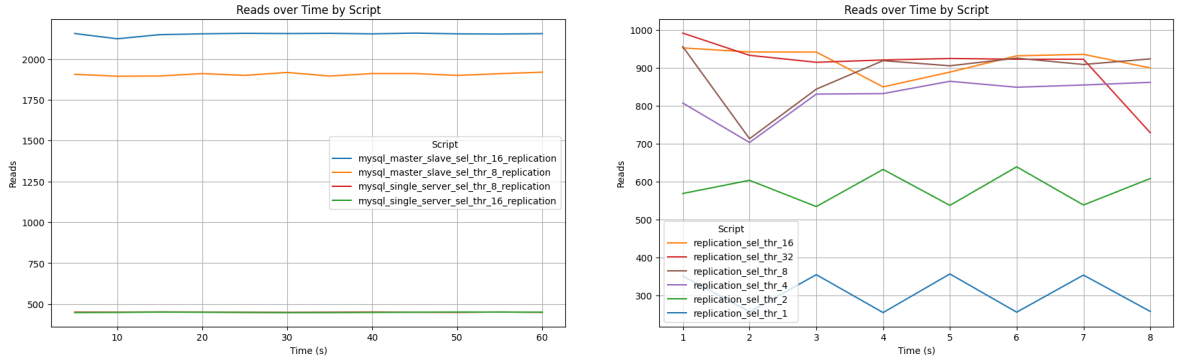


Abbildung 7.2: Vergleich zwischen Master mit 3 Replikaten und Single-Server-Ansatz

Bei dem ersten Vergleich betrug der Threads-Wert bei Einfügungen und Abfragen `1`, sodass die Leistung eines einzelnen Threads ersichtlich wurde. Aus den Ergebnissen lässt sich schlussfolgern, dass der Single-Server am effizientesten ist, solange er nicht stark ausgelastet wird. In diesem Fall bringt der Master-Replikat-Ansatz demnach keine Vorteile. Anders verhält es sich, wenn die Last auf den Server erhöht wird, indem die Anzahl der Threads gesteigert wird. Die Threadanzahl kann beim Aufführen des Sysbench-Befehls mit dem Parameter `--threads` festgelegt werden. Die Prozesse sollen die Last durch Nutzer in der Datenbank simulieren. Um den Vergleich zwischen dem Single-Server und dem Master-Replikat-Ansatz zu verdeutlichen, wird der Benchmark mit 8 und 16 Threads durchgeführt. Beim Single-Server werden alle Threads auf einem Server ausgeführt, während sie beim Master-Replikat-Ansatz auf den Master und die drei Replikate gleichmäßig aufgeteilt werden. Bei 8 Threads wird die

Last so aufgeteilt, dass der Master und alle Replikat je 2 Threads verarbeiten ($8/4 = 2$). Nach diesem Prinzip wird der Benchmark auch mit 16 Threads durchgeführt (siehe Abbildung 7.3a).



(a) Mit Replikation **(b)** Ohne Replikation

Abbildung 7.3: Vergleich von 8 Threads an Single-Server und jeweils 2 an die unters. Ports

Zunächst ist in der linken Grafik zu erkennen, dass beide Kurven des Single-Server-Ansatzes sehr ähnlich verlaufen. Auf den ersten Blick könnte das überraschen, da man erwarten würde, dass eine Verdopplung der Threadanzahl auch zu einer Verdopplung der Leseabfragen führt. Deshalb wurde noch ein weiterer Vergleich durchgeführt, bei dem die Threadanzahlen der 2er-Potenzreihe bis einschließlich 2^5 nur für den Single-Server-Ansatz getestet wurden. Zusätzlich wurde bei diesem Vergleich die CPU-Auslastung mit dem folgenden Befehl gemessen:

Codeblock 7.6: Messen der CPU-Auslastung

```
top -l 1 | grep 'CPU usage' | awk '{print $3 + $5}' %Für Mac
top -bn1 | grep 'Cpu(s)' | sed 's/.*, *\\([0-9.]*\\)%* id.*/\\1/' | awk '{print 100 - $1}' %Für Linux
```

Die Ergebnisse der CPU-Auslastung und der Anzahl der Leseabfragen sind in Tabelle 7.1 dargestellt, sortiert nach aufsteigender Threadanzahl. Die Abbildung 7.3b zeigt, dass sich die Leseperformance beim Anstieg von 1 auf 2 Threads nahezu verdoppelt. Zwischen 2 und 4 Threads fällt der Unterschied schon geringer aus, liegt aber dennoch bei etwa 40%. Die restlichen Threadanzahlen weichen nur geringfügig voneinander ab. Begründen lässt sich das durch die CPU-Auslastung. Bei einer geringeren Anzahl an Threads ist die CPU-Auslastung sehr niedrig, was die starken Anstiege erklärt. Irgendwann ist die CPU jedoch voll ausgelastet, sodass die Performance nicht mehr weiter steigt. Dies ist bei 8 Threads der Fall, da dort die CPU-Auslastung bei 89.34% liegt.

Anzahl an Threads	Durchschnittliche CPU-Auslastung	Anzahl an Leseabfragen
1	1.36%	2437
2	26.44%	4662
4	67.12%	6609
8	89.34%	7106
16	85.74%	7362
32	99.14%	7295

Tabelle 7.1: Auslastung mit unterschiedlichen Threadanzahlen

Wenn nun wieder das Ergebnis aus 7.3a betrachtet wird, werden die Vorteile der Replikation deutlich sichtbar. In beiden Fällen liegen die Werte mit Replikation stets deutlich über denen des Single-Server-Ansatzes. Die Performance des Single-Servers bleibt unabhängig von der Threadanzahl nahezu konstant. Bei der Replikation zeigt sich, dass ein Anstieg an Threads auch zu einer Steigerung der Leseabfragen führt. So ist die Kombination von 16 Threads, verteilt auf den Master und 3 Replikate, am effizientesten. Danach folgt mit etwas Abstand die Version mit 8 Prozessen, die etwa 25% langsamer ist. Aber selbst diese ist deutlich schneller als alle Varianten des Single-Server-Ansatzes, sodass festgestellt werden kann, dass die Replikation bei höherer Last klare Vorteile bietet.

Im letzten Vergleich wird ausschließlich der Master-Replikat-Ansatz verwendet, um die unterschiedlichen Binlog-Formate zu vergleichen. Zur Begrenzung der Variationen wird nur ein Replikat pro Master betrachtet. Daraus ergeben sich sechs unterschiedliche Leseergebnisse, da für jedes der drei Formate sowohl der Master- als auch der Replikat-Port abgefragt wird. In der Grafik 7.4 ist zu erkennen, dass es bei den verschiedenen Binlog-Formaten und Ports kaum Unterschiede gibt. Und auch die Schreibgeschwindigkeiten verhalten sich bei beiden Varianten sehr ähnlich. Das lässt sich auch mit dem Hexagon-Chart bestätigen, da dort alle Werte sehr nahe beieinander liegen.

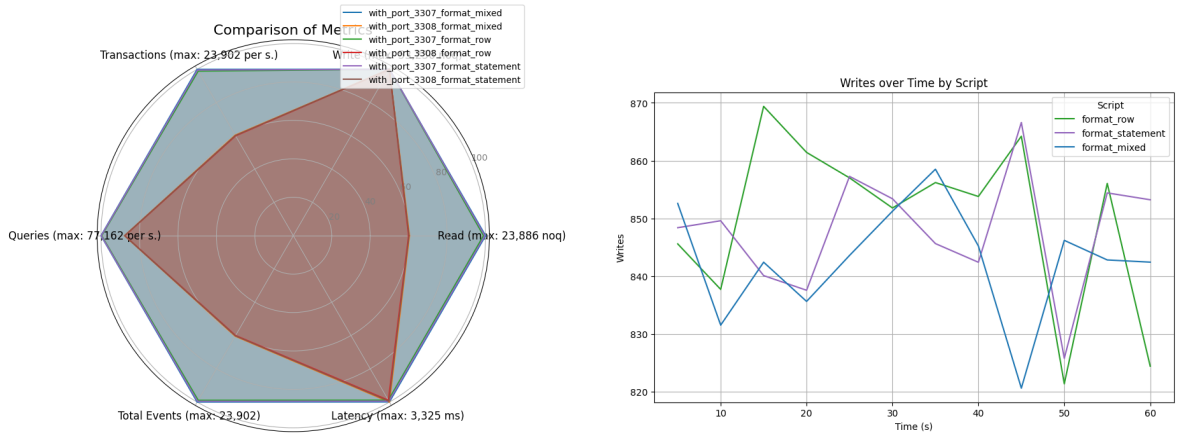


Abbildung 7.4: Vergleich zwischen den unterschiedlichen Binlog-Typen

Die Schlussfolgerungen aus den Messungen sehen dabei wie folgt aus. Es zeigt sich, dass durch Replikation, anders als beispielsweise mit Indexen oder Partitionen, mit einem einzelnen Thread keine deutlichen Performancevorteile gewonnen werden können. Wenn aber mehrere Nutzer auf der Datenbank interagieren und ausschließlich Lesezugriffe benötigen, dann können die Abfragen auf die unterschiedlichen Ports aufgeteilt werden. In diesen Szenarien mit höherer Parallelität und intensiveren Leseoperationen kann Replikation signifikante Vorteile bieten. Die Auswahl des Binlog-Formats hat beim Benchmark zu keinen Performanzgewinnen geführt. Möglicherweise könnte sich jedoch ein anderer Einfluss zeigen, wenn Schreiboperationen ins Spiel kommen oder die Konsistenzanforderungen geändert werden.

8 Fazit

Die vorliegende Bachelorarbeit ging der Frage nach, wie die Performance einer relationalen Datenbank verbessert werden kann. Ihr Aufbau orientiert sich am Prozess des Datenentwurfs. Beim Datenbankentwurf muss man die ermittelten Anforderungen aus den Interviews mit Stakeholdern verwenden, um einen konzeptionellen Entwurf, beispielsweise in Form eines ER-Modells, zu erstellen. Danach wird aus dem konzeptionellen Entwurf ein Logischer in Form eines Relationenschemas. Dieses Kapitel dient dazu, eine allgemeine Zusammenfassung der wesentlichen Erkenntnisse zu bieten.

Zuallererst wird der logische Entwurf betrachtet, bei dem neben der Normalisierung der Tabellen auch die Auswahl der korrekten Datentypen eine Rolle spielt. Das erste Kapitel behandelte dieses Thema im Detail. Mithilfe der Benchmarks wurde festgestellt, dass der kleinstmögliche Datentyp für eine Spalte deklariert werden sollte. Dazu muss zunächst festgelegt werden, welcher Bereich an Werten abgebildet werden soll, um darauf basierend den geeigneten Typ auszuwählen. Dabei ist durchaus von Vorteil, dass der Typ bei einer falschen Einschätzung des Wertebereichs ohne viel Aufwand verändert werden kann. Beim Betrachten der numerischen Datentypen fiel auf, dass je größer der Wertebereich und damit der Speicherbedarf ist, desto schlechter wird die Leistung. Deshalb zählen `DECIMAL` und `BIGINT` zu den ineffizientesten. Bei den zeichenkettenbasierten Typen ist die Wahl einfach zu treffen, da in den meisten Fällen der Typ `VARCHAR` am schnellsten ist. Nur wenn eine Spalte häufiger aktualisiert als abgefragt wird, kann es sinnvoll sein, den Typ `CHAR` in Erwägung zu ziehen. Einer der anderen Leitsätze bei der Wahl der Datenformate ist, dass man die simplere Datenstruktur wählen sollte. Im Vergleich zeigte sich, dass `INT` etwa 50% schneller ist als `CHAR`. Zu guter Letzt sollte berücksichtigt werden, dass die Spalten nicht nur aus Performancegründen, sondern auch zur Wahrung der Datenintegrität und -konsistenz an möglichst vielen Stellen als `NOT NULL` definiert werden sollten.

Nach dem logischen Entwurf einer Datenbank kommt als nächster Schritt die physische Implementierung. Bei diesem Schritt spielen auch die anderen Aspekte, die betrachtet wurden, wie Indexierung, Sichten, Partitionen oder Replikation, eine Rolle.

Bei der Indexierung wurde gezeigt, wie effektiv sie sein kann, indem der Aufbau und die Funktionsweise der B-Tree- und Hash-Indexe erläutert und getrennt voneinander untersucht wurden. Der Vergleich beider Varianten hat ergeben, dass der Hash-Index in bestimmten

Fällen effektiver ist als der B-Baum-Index. Auf der anderen Seite kann der B-Baum-Index bei deutlichen mehr Abfragen eingesetzt werden, insbesondere auch bei Bereichsabfragen oder Filtern von Teilen des Indexes. Im Gegensatz dazu funktioniert der Hash-Index nur bei einem exakten Schlüsselabgleich. Außerdem ist beim Hash-Index auch die Anzahl an Hashkollisionen relevant für die Performance. Der größte Nachteil der Verwendung von Indizes ist der höhere Pflegeaufwand, da bei jeder Datenänderung der Index ebenfalls angepasst werden muss. Wenn Performanceprobleme bei einer Datenbankumgebung auffallen, dann sollte man in den Logs nach Abfragen suchen, die zum einen besonders häufig vorkommen und zum anderen viel Zeit benötigen. Bei der Analyse kann man möglicherweise eine sinnvolle Nutzung von Indizes identifizieren und diese erstellen. Nach einigen Tagen oder Wochen bietet es sich an, eine Kontrolle durchzuführen und abhängig vom Ergebnis können einige Indizes entfernt und andere neue hinzugefügt werden. Ein ähnliches Vorgehen ist auch beim Einsatz von Views nützlich.

Wie die Benchmarks für die Sichten gezeigt haben, wirken sich virtuelle Views nicht auf die Performance aus. Dafür eignen sich virtuelle Sichten hervorragend für Gewährleistung von Rechtemanagement in einer Organisation, denn sie haben den Vorteil, dass die Daten nicht physisch gespeichert werden und somit keine Redundanzen entstehen. Materialisierte Sichten hingegen werden auf der Festplatte gesichert und bieten dafür ein erhebliches Performancepotenzial. Besonders geeignet sind sie in Szenarien, in denen häufig auf aggregierte oder komplexe Abfragen zugegriffen wird, wie zum Beispiel in OLTP-Systemen. Es ist durchaus sinnvoll, sich bereits beim Datenbankentwurf Gedanken über Sichten zu machen, doch es ist auch möglich, diese, wie bei Indizes, erst im Laufe der Zeit zu ergänzen. Wie genau die Implementierung von materialisierten Sichten umgesetzt werden kann, hängt vom jeweiligen Datenbankmanagementsystem ab. Einige DBMS unterstützen materialisierte Sichten, während andere sogar eine inkrementelle Auffrischung ermöglichen. In MySQL hingegen müssen materialisierte Sichten durch dedizierte Tabellen in Kombination mit Triggern nachgebildet werden. Bei den Tests ist jedoch deutlich geworden, dass die native Implementierung, z.B. in Postgres, einen klaren Performancevorteil gegenüber der Implementierung mit Triggern bietet. Daher ist es ratsam, diesen Aspekt bei der Auswahl des DBMS zu berücksichtigen. In Bezug auf die Schreibperformance muss ebenfalls erwähnt werden, dass die Pflege von materialisierten Sichten die Effizienz negativ beeinflusst.

Bei Partitionen fällt der Mehraufwand geringer aus als bei Indizes oder Sichten, da keine zusätzlichen Datenbankobjekte verwaltet werden müssen. Stattdessen werden die Datensätze auf mehrere Partitionen verteilt und nicht in einer einzelnen Tabelle gespeichert. Wenn eine Datenbankoperation ausgeführt wird, muss zunächst die Partition oder die entsprechenden Partitionen ermittelt werden, die die angeforderten Daten enthalten. Es ist auch möglich, dass eine Datenbankumgebung nicht für die Partitionierung geeignet ist. Normalerweise ist ein Merkmal, das für die Partitionierung spricht, ein natürliches Trennkriterium wie beispielsweise ein Zeitstempel oder geografische Regionen, da dadurch eine logische Auftei-

lung der Daten ermöglicht wird. Abhängig vom Trennkriterium muss man sich für einen der Partitionstypen entscheiden: Range, List, Hash oder Key. Der Performancevorteil der Partitionierung liegt darin, dass nur die relevanten Partitionen durchsucht werden müssen, anstatt die gesamte Tabelle zu scannen. Dieser Vorgang wird als Pruning bezeichnet und führt zu einer erheblichen Steigerung der Abfragegeschwindigkeit. Allerdings gibt es einige Einschränkungen beim Pruning, da es nicht mit allen Operatoren kompatibel ist. Bei der Range-Partitionierung mit einem Zeitstempel als Partitionsschlüssel können bei einigen Operatoren unerwartete Probleme auftreten. Obwohl eine Abfrage, die beispielsweise den `YEAR()`-Operator verwendet, dasselbe Ergebnis wie eine Bereichsabfrage nach dem ersten und letzten Tag eines Jahres liefert, kann die Query mit `YEAR()` nicht für das Partition-Pruning genutzt werden. In einem solchen Fall müssen alle Partitionen durchsucht werden, was die Abfrage sogar langsamer macht als ohne Partitionierung. Für die List-Partitionierung hat sich gezeigt, dass der Operator `IN` am effizientesten ist, gefolgt von `OR`, während `UNION` deutlich weniger effizient ist, weshalb von seiner Verwendung abgeraten werden sollte. Bei der Hash-Partitionierung wurde festgestellt, dass mit zunehmender Anzahl der Partitionen die Suche innerhalb der Partitionierungsstruktur aufwendiger wird und die Performance entsprechend schlechter ausfällt. Diese Erkenntnis gilt auch für die anderen Partitionierungstypen.

Zum Schluss wurde der Einfluss der Replikation im Rahmen des Master-Replikat-Ansatzes analysiert. Anders als bei der Partitionierung werden bei der Replikation vollständige Kopien der gesamten Datenbank auf mehreren Servern erstellt. Wenn Änderungen am Master vorgenommen werden, werden diese durch verschiedene Threads an die Replikate übertragen. Dadurch wird die Verfügbarkeit und Ausfallsicherheit erhöht, weshalb Replikation häufig in Verbindung mit Backups eingesetzt wird. Um die Performance zu testen, wurde die Leistung eines Single-Servers mit der eines Systems aus Master- und Replikaten verglichen. Dabei wurde festgestellt, dass der Single-Server bei Verwendung eines einzelnen Threads einen Leistungsvorteil hat. Sobald jedoch mehrere Threads die CPU-Auslastung auf dem Single-Server erhöhen und gleichzeitig die Last auf die Master- und Replikatknoten verteilt wird, zeigt sich der Vorteil der Replikationsverteilung. Allerdings treten auch Nachteile beim Einfügen von Daten mit Replikation auf, da das erneute Kopieren der Daten auf die Replikate die Performance negativ beeinflusst. Die Auswahl des Binlog-Formats hat bei keinem Benchmark zu Vor- oder Nachteilen geführt. Aus der Betrachtung der Ergebnisse ergibt sicht, dass Replikation kein geeigneter Lösungsansatz ist, wenn nur ein Nutzer auf die Datenbank zugreift. Sie wird jedoch besonders vorteilhaft, wenn die Last auf mehrere Server verteilt werden muss. In Bezug auf die Verteilung ähnelt die Replikation den zentralen Konzepten von NoSQL-Datenbanken, die horizontal skalieren, um Daten auf mehrere Knoten zu verteilen.

Zusammenfassend lässt sich festhalten, dass es keine allgemeingültige Lösung für optimale Performance gibt, sondern verschiedene Konzepte, deren Effizienz vom jeweiligen Anwendungsfall abhängt. Oft führt eine gezielte Kombination mehrerer Techniken zu den besten Ergebnissen. Ein bewährter Ansatz ist die Verbindung von Partitionierung und Replikation.

Hierbei wird jede Partition auf mehreren Knoten repliziert, wodurch die Datensätze weiterhin einer bestimmten Partition zugeordnet bleiben, gleichzeitig aber redundant gespeichert werden. Darüber hinaus wirkt sich die Verwendung kleinerer Datentypen positiv auf die Index-Performance aus. Indizes können effektiv mit Partitionierung oder materialisierten Views kombiniert werden und optimieren in replizierten Systemen die Lesezugriffe auf die Replikate. Letztlich zeigt sich, dass das Zusammenspiel der verschiedenen Strategien eine nachhaltige Antwort bietet.

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, 13. März 2025