

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 Leistungsmessungen eingesetzt. Die Ergebnisse zeigen, dass die Wahl des kleinstmöglichen Datentyps und die Verwendung von Not Null-Spalten die Effizienz verbessern, indem sie Speicherplatz sparen. Hash-Indizes sind besonders bei exakten Schlüsselvergleichen effektiv, wohingegen B-Baum-Indizes vielseitigere Einsatzmöglichkeiten bieten. Materialisierte Sichten bieten Performancevorteile durch gespeicherte Abfrageergebnisse, im Gegensatz dazu liefern virtuelle Sichten Echtzeitdaten, müssen jedoch bei jedem Zugriff die Abfrage neu ausführen und sind daher langsamer. 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 optimizes 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, with replication improving 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	III
1 Einleitung	1
1.1 Benchmarks	1
1.2 Kennzahlen	3
1.3 Auswahl der Tools	4
2 Grundlagen	6
2.1 Überblick über die Tools	6
2.2 Projektaufbau	10
2.3 GitHub Actions	18
2.4 Optimierung des Workflows	21
3 Optimierungen von Datentypen	23
3.1 Allgemeine Faktoren	23
3.2 Funktionsweise individueller Datentypen	24
3.3 Analyse der Benchmarks	27
4 Indizes	29
4.1 Grundlagen	29
4.2 B-Baum-Index	32
4.3 Hash-Index	36
4.4 Vergleich zwischen B-Tree- und Hash-Index	39
5 Views	40
5.1 Virtuelle Views	40
5.2 Materialisierte Views	44
5.3 Durchführung der Benchmarks	48
6 Partitionen	51
6.1 Grundlagen	51
6.2 Partitionierungstypen	54
6.3 Auswertung der Benchmarks	57

7	Replikation	60
7.1	Grundlagen	60
7.2	Konfiguration der Master-Replika-Architektur	64
7.3	Untersuchung der Replikation	66
8	Fazit	70
	Literatur	73
	Anhang	75

Abbildungsverzeichnis

2.1	Demo: Gnuplot vs. Pandas	9
2.2	Join-Typ: Skriptvergleich	17
2.3	Join-Typ: Metrikvergleich	18
3.1	Datentypen: Vergleich mit Not Null, sowie Int und Char	27
3.2	Datentypen: Numerische Datentypen	28
3.3	Datentypen: Zeichenkettenbasierte Typen	28
4.1	Binärbaum-Visualisierung	33
4.2	B-Tree-Indexing: Mit Index vs Ohne	34
4.3	B-Tree-Indexing: Unterschiedliche Selects mit Index und Ohne	35
4.4	Hash-Indexing: Auswirkungen von Hashkollisionen	38
4.5	Hash-Indexing: Unterschiedliche Abfragen mit Index und Ohne	38
4.6	Indexing: Vergleich von B-Tree- und Hash-Index	39
5.1	Views: Keine View, virtuelle View und Ansatz mit Triggern	48
5.2	Views: Beide Triggeransätze sowie materialisierte Sicht	49
6.1	Range-Partitionierung: Unterschiedliche Selects mit und ohne Partition	58
6.2	List-Partitionierung: Unterschiedliche Abfragen mit und ohne Partition	59
6.3	Hash-Partitionierung: Variationen der Partitionsanzahl	59
7.1	Master-Replik-Visualisierung	61
7.2	Replikation: Master-Replik-Ansatz vs Single-Server	67
7.3	Replikation: Threadanzahl aufgeteilt an Master-Replikate	68
7.4	Replikation: Unterschiedliche Binlog-Typen	69

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 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. Zudem kann analysiert werden, wie sich das System sowohl unter aktuellen als auch unter simulierten, zukünftigen und anspruchsvolleren Bedingungen verhalten könnte.

Es gibt zwei verschiedene Techniken für Benchmarks (Schwartz et al., 2012, S. 35–49). 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 darin bestehen, diese Zeit so weit wie möglich zu reduzieren. Es kann dabei auch vorkommen, dass das Datenbanksystem nicht das Bottleneck ist.¹ Full-Stack-Benchmarks haben jedoch

¹Ein 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 (Vogel, 2009).

auch Nachteile, denn sie sind schwieriger zu erstellen und insbesondere schwieriger korrekt einzurichten.

Die zweite Art von Benchmarks sind die sogenannten Single-Component-Benchmarks, die zum Einsatz kommen, wenn lediglich verschiedene Schemata und Abfragen im DBMS hinsichtlich ihrer Performance getestet werden sollen. 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, wurde sich 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 widerspiegeln. 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, werden die Daten und der Workload nicht durch Snapshots bereitgestellt, sondern stattdessen zufällig generiert. 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 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. Es empfiehlt sich auch die Ergebnisse eines Benchmarks in einem Diagramm darzustellen, da bestimmte Phänomene oft nur so erkennbar werden und nicht in tabellarischer Form sichtbar sind. Im nächsten Abschnitt wird ausgewählt, welche Messwerte betrachtet werden sollen.

²Snapshots 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 (Microsoft, 2024).

1.2 Kennzahlen

Zunächst sollte geklärt werden, welche Kennzahlen im Datenbankkontext relevant sind und für die jeweiligen Zwecke von besonderem Interesse. Darauf basierend kann ein Benchmark-Tool ausgewählt werden, das in der Lage ist, 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 (μ s), 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. 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 wurden (Reinboth, 2020).

Eine weitere Kennzahl ist die Gleichzeitigkeit (engl. concurrency), die angibt, wie viele Anfragen gleichzeitig bearbeitet werden können. Zur genaueren Messung der Gleichzeitigkeit auf dem Webserver wird die Anzahl der gleichzeitig ausgeführten Anfragen zu einem bestimmten Zeitpunkt ermittelt. Eine Website mit 50.000 gleichzeitigen Benutzern könnte unter Umständen nur 10 oder 15 gleichzeitige Abfragen erfordern. Es kann auch geprüft werden, ob der Durchsatz sinkt oder die Antwortzeiten steigen, wenn die Gleichzeitigkeit zunimmt. 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 (Oracle, 2025e). 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. Dies ist aber die einzige Ausnahme und der Schwerpunkt im weiteren Verlauf der Arbeit wird überwiegend auf MySQL liegen.

Die Grundlage dieser Bachelorarbeit bildet die Untersuchung des Verhaltens der MySQL-Datenbank (Reimers, 2017) im Hinblick auf verschiedene Konzepte mithilfe eines zentralen Benchmark-Tools. Nach eingehender Überlegung wurde sich für Sysbench (Kopytov, 2024) entschieden. Sysbench ist ein Open-Source-Tool, das auf LuaJIT basiert und Skriptfähigkeit sowie Multi-Threading unterstützt, um Benchmarks durchzuführen (Schwartz et al., 2012, S. 50–66). 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 wurden auch andere Optionen wie Benchbase (Difallah et al., 2013) und Mybench (Shopify, 2024) 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 (Shopify, 2022b). 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 (Shopify, 2022a). Trotz dieses Features wurde sich 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 Visualisierung schwer zu identifizieren. Mithilfe von Graphen mit einer Zeitachse werden Zyklen sofort sichtbar und der Vergleich unterschiedlicher Messungen wird erheblich vereinfacht.

Um die Kennzahlen, die mithilfe von Sysbench ermittelt worden sind, in eine grafische Darstellung umzuwandeln, gibt es unterschiedliche Tools. Eine erste Möglichkeit bietet Gnuplot (Williams et al., 2010), 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 fiel die Entscheidung auf ein Python-Skript als flexiblere Alternative. Für die grafische Darstellung kommen dabei die Bibliotheken pandas (pandas development team, 2020) 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 1.3 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, bei den Benchmark-Tests Aufwand ersparen und wie die Workflows sowohl zeitlich als auch ressourcenschonend optimiert werden können.

2.1 Überblick über die Tools

Um die Tools näher zu erläutern, ist die Grundvoraussetzung zuallererst ein laufendes Datenbankmanagementsystem, in diesem Fall MySQL. Dabei ist es egal, ob es lokal auf einem Rechner oder über Docker in einem GitHub CI/CD-Workflow gestartet wird. Wichtig ist aber, dass man sich die Zugangsdaten, bestehend aus Benutzer- und Passwortdaten, abspeichert, da diese gebraucht werden, um den Benchmark-Test mit Sysbench zu starten. Nachdem das RDBMS gestartet worden ist, muss zunächst eine Datenbank erstellt werden:

```
1 CREATE DATABASE sbtest;
```

Neben der erfolgreichen Erstellung der Datenbank muss ebenfalls das Tool Sysbench installiert werden. Auf einem Linux-basierten System kann dies wie folgt umgesetzt werden:

```
1 sudo apt install sysbench
```

Damit die Grafiken erstellt werden können, werden zusätzlich die Tools Gnuplot und Pandas in Kombination mit matplotlib benötigt. Die Befehle `sudo apt` und `pip` müssen je nach Betriebssystem beispielsweise durch `brew` und `pip3` ersetzt werden.

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

Damit wurden alle Abhängigkeiten eingerichtet und im nächsten Schritt muss man sich mit den Tools vertraut machen. Der Hauptfokus wird dabei auf Sysbench gerichtet. Um mit Sysbench zu interagieren, verwendet man Shell-Befehle, bei denen verschiedene Parameter angegeben werden können, um spezifische Tests durchzuführen. Die folgende Übersicht zeigt die Argumente, die übergeben werden müssen, sowie solche, die optional angegeben werden können:

- **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, hier: `sbtest`
- **time**: Laufzeit des Benchmarks in Sekunden (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)

Einige dieser genannten Parameter, wie z.B. `db-driver`, dienen dazu die Verbindung mit dem Datenbankmanagementsystem herzustellen. Mit den anderen Argumenten kann man die Einzelheiten des Benchmarks genauer bestimmen. So kann beispielsweise festgelegt werden, wie lange der Benchmark laufen soll und wie häufig die Teilergebnisse geloggt werden. Damit fehlen aber noch die Informationen über die Tabellen und die Abfragen, die getestet werden sollen. Dies kann, wie im Abschnitt 1.3 beschrieben, entweder über explizite Lua-Skripts oder mithilfe von vordefinierte Testtypen in Sysbench geregelt werden. Durch die vordefinierten Typen lässt sich schnell kontrollieren, ob die Parameter zur Datenbankverbindung korrekt sind, ohne dafür SQL-Befehle oder Lua-Skripte schreiben zu müssen. Unter anderen kann man zwischen dem reinen Einfügen von Daten (`oltp_insert`), dem Abfragen von Daten (`oltp_read_only`) oder einer Kombination aus beidem (`oltp_read_write`) wählen. Als letztes muss noch festgelegt werden, welche der folgenden Methoden ausgeführt werden soll:

- **prepare**: Bereitet die Datenbank für den Test vor, u.a. das Erstellen der Tabellen.
- **run**: Je nach Testtyp führt diese Methode die spezifizierten Operationen aus und misst dabei die Performance.
- **cleanup**: Stellt die Datenbank in ihren ursprünglichen Zustand zurück.

Die Methoden sowie der Testtyp oder der Pfad des Lua-Skripts werden am Ende der Sysbench-Befehlszeile hinzugefügt. Als Beispiel wird der Testtyp `oltp_read_write` zusammen mit der Methode `RUN` ausgewählt. Die Query könnte wie folgt aussehen, wobei die Werte für die Variablen `YOUR_USER` und `YOUR_PASSWORD` durch die tatsächlichen Benutzerdaten ersetzt werden müssen:

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

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

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

Der entstandene Fehler wird dadurch verursacht, dass die Tabelle `sbtest1` nicht erstellt worden ist. Daher muss vor der Ausführung der `RUN`-Methode zunächst die `PREPARE`-Methode durchgeführt werden. Und 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-Skript zu schreiben, das diese Aufgabe übernimmt.

Codeblock 2.1: Ausführung der Sysbench-Methoden in korrekten Reihenfolge

```
1 # Zunächst folgendes definieren : DB_HOST, DB_USER, DB_PASS, DB_NAME, TABLES, TABLE_SIZE, DURATION  
2 SYSBENCH_CONFIG="--db-driver=mysql --mysql-host=$DB_HOST --mysql-user=$DB_USER --mysql-password=$DB_PASS --mysql-db  
3   =$DB_NAME --tables=$TABLES --table-size=$TABLE_SIZE "  
4 sysbench oltp_read_write $SYSBENCH_CONFIG prepare  
5 sysbench oltp_read_write $SYSBENCH_CONFIG --time=$DURATION --threads=1 --report-interval=1 run  
6 sysbench oltp_read_write $SYSBENCH_CONFIG cleanup
```

Beim Ausführen des Skripts gibt die `run`-Methode einmal pro Sekunde die Zwischenergebnisse in der Konsole aus, da der Wert von `report_interval` auf 1 gesetzt ist. Nach erfolgreichem Abschluss wird zudem eine Gesamtstatistik mit der gemessenen Kennzahlen angezeigt. Anstatt die Ausgabe in der Konsole darzustellen, können die Werte auch in einer Log-Datei gespeichert werden. Damit wurde bisher erläutert, wie man Sysbench verwendet, jedoch fehlt noch die Erstellung von Graphen. Um diesen Prozess zu erleichtern, empfiehlt es sich, die relevanten Kennzahlen aus der Log-Datei zu extrahieren und mit passenden Spaltenüberschriften in einer CSV-Datei zu speichern. Dies kann mit dem Shell-Kommando `grep` erfolgen:

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

```
1 RAW_RESULTS_FILE="output/sysbench.log"
2 OUTPUT_FILE="output/sysbench_output.csv"
3
4 echo "Script,Time (s),Threads,TPS,QPS,Reads,Writes,Other,Latency (ms;95%),ErrPs,ReconnPs" > "$OUTPUT_FILE"
5 grep '^\\[ ' $RAW_RESULTS_FILE | while read -r line; do
6     time=$(echo $line | awk '{print $2}' | sed 's/s//')
7     threads=$(echo $line | awk -F 'thds: ' '{print $2}' | awk '{print $1}')
8     # Extract other measures
9     latency=$(echo $line | awk -F 'lat \\(ms,95%\\): ' '{print $2}' | awk '{print $1}')
10    echo "demo,$time,$threads,$tps,$qps,$reads,$writes,$other,$latency,$err_per_sec,$reconn_per_sec" >> "
        $OUTPUT_FILE"
11 done
12 echo "Results saved to $OUTPUT_FILE."
```

Mit der erstellten CSV-Datei können Graphen entweder mit Gnuplot oder der Python-Bibliothek Pandas erzeugt werden. Das Python-Skript muss die CSV-Datei als Argument erhalten und optional eine Liste spezifischer Messwerte, für die Graphen erstellt werden sollen.

Codeblock 2.3: Erstellung der Graphen aus der CSV-Datei

```
1 OUTPUT_FILE="$OUTPUT_DIR/sysbench_output.csv"
2
3 # Gnuplot
4 GNUPLOT_SCRIPT="YOUR_PATH_TO_PROJECT/plot_sysbench.gp"
5 gnuplot $GNUPLOT_SCRIPT
6
7 # Python with Library Pandas
8 PYTHON_SCRIPT="YOUR_PATH_TO_PROJECT/generatePlot.py"
9 python3 "$PYTHON_SCRIPT" "$OUTPUT_FILE"
```

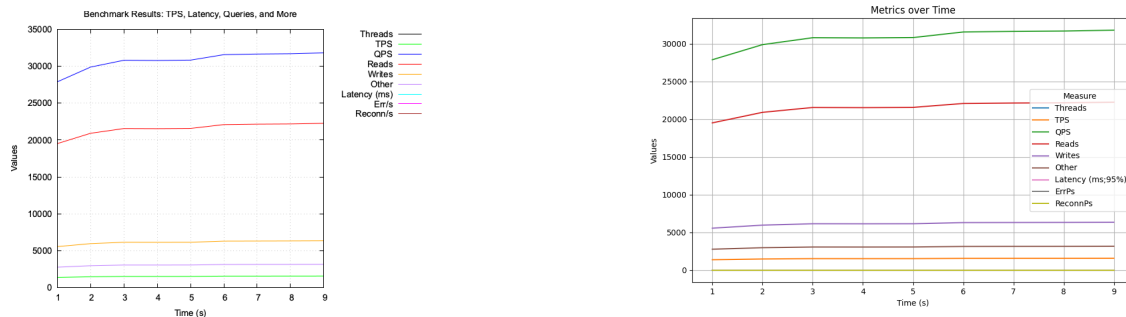


Abbildung 2.1: Grafik zeigt Erstellung mit Gnuplot (links) und Pandas (rechts)

Die Abbildung 2.1 zeigt die Ergebnisse der Grapherstellung. Die dargestellten Metriken umfassen: 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

In dem vorausgegangenen Abschnitt wurde das Tool Sysbench und seine Funktionsweise anhand eines Demo-Projekts näher erläutert. Damit ist 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, die genaue Implementierung jedoch auf diese Weise nicht veränderbar ist. Genau für diese Anwendungsfälle gibt es noch 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, ist ein Vergleich zwischen zwei verschiedenen Implementierungen erforderlich. Der Unterschied zwischen den beiden Versionen besteht darin, dass die eine Tabelle einen Primärschlüssel vom Typ INT und die andere vom Typ VARCHAR hat. 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 den Lua-Skripten mit der Spezifizierung der Tabellen begonnen, die erstellt werden sollen. Insgesamt sind vier verschiedene CREATE TABLE-Befehle erforderlich, jeweils zwei pro Variante. Sie unterscheiden sich nur im Typ des Primärschlüssels der Kundentabelle und des Fremdschlüssels in der Bestelltabelle. Für den Fall mit INT, müssen die folgenden Tabellen erstellt werden:

Codeblock 2.4: Create Table-Befehl für Kunden

```
1 CREATE TABLE KUNDEN (  
2     KUNDEN_ID      INT PRIMARY KEY,  
3     NAME           VARCHAR(255),  
4     GEBURTSTAG     DATE,  
5     ADRESSE        VARCHAR(255),  
6     STADT          VARCHAR(100),  
7     POSTLEITZAHL   VARCHAR(10),  
8     LAND           VARCHAR(100),  
9     EMAIL          VARCHAR(255) UNIQUE,  
10    TELEFONNUMMER   VARCHAR(20)  
11 );
```

Codeblock 2.5: Create Table-Befehl für Bestellung

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

Anschließend werden diese Befehle in der `prepare()`-Funktion verwendet. 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 der `Prepare`-Funktion erfolgen. Dies ist ein Auszug aus der Funktion:

Codeblock 2.6: Lua-Script für die Erstellung der Tabellen

```
1 local con = sysbench.sql.driver():connect()  
2 function prepare()  
3     local create_kunden_query = [[  
4         CREATE TABLE KUNDEN (...);  
5     ]]  
6     local create_bestellung_query = [[  
7         CREATE TABLE BESTELLUNG (...);  
8     ]]  
9  
10    con:query(create_kunden_query)  
11    con:query(create_bestellung_query)  
12    print("Tables KUNDEN und BESTELLUNG have been successfully created")  
13 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

```
1 local con = sysbench.sql.driver():connect()  
2 function cleanup()  
3     con:query("DROP TABLE IF EXISTS BESTELLUNG;")  
4     con:query("DROP TABLE IF EXISTS KUNDEN;")  
5     print("Cleanup successfully done")  
6 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 die Kundentabelle entfernt werden.

Damit 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 zudem festgelegt werden, wie viele Kunden und Bestellungen pro Kunde erstellt werden. Um sicherzustellen, dass keine Werte in den Tabellen enthalten sind, können alle Daten 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

```
1 local con = sysbench.sql.driver():connect()
2 local num_rows = 1000
3 local bestellungProKunde = 4
4
5 function delete_data()
6     con:query("DELETE FROM BESTELLUNG;")
7     con:query("DELETE FROM KUNDEN;")
8 end
9
10 function insert_data()
11     delete_data()
12     for i = 1, num_rows do
13         local kunden_id = i -- define name, geburtstag, adresse, stadt, postleitzahl, land, email, telefonnummer
14         local kunden_query = string.format([[
15             INSERT IGNORE INTO KUNDEN
16             (KUNDEN_ID, NAME, GEBURTSTAG, ADRESSE, STADT, POSTLEITZAHL, LAND, EMAIL, TELEFONNUMMER)
17             VALUES (%d, '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s');
18         ]], kunden_id, name, geburtstag, adresse, stadt, postleitzahl, land, email, telefonnummer)
19         con:query(kunden_query)
20
21         for j = 1, bestellungProKunde do
22             local bestellung_id = (i-1) * bestellungProKunde + j -- define bestelldatum, artikel_id, umsatz
23             local bestellung_query = string.format([[
24                 INSERT IGNORE INTO BESTELLUNG
25                 (BESTELLUNG_ID, BESTELLDATUM, ARTIKEL_ID, UMSATZ, FK_KUNDEN)
26                 VALUES (%d, '%s', %d, %d, %d);
27             ]], bestellung_id, bestelldatum, artikel_id, umsatz, kunden_id)
28             con:query(bestellung_query)
29         end
30     end
31 end
32
33 function event()
34     insert_data()
35 end
```

Die letzte Anweisung, die noch benötigt wird, ist die Select-Abfrage. Hierbei muss man sich Gedanken machen, welche Abfrage man stellen muss, damit die untersuchten Effekte auch tatsächlich auftreten. Da die Performance des Join-Operators untersucht wird, müssen beide Tabellen über den Fremdschlüssel verbunden werden.

Codeblock 2.9: Lua-Script für das Abfragen von Daten

```
1 local con = sysbench.sql.driver():connect()
2 function select_query()
3     local join_query = [[
4         SELECT k.STADT, SUM(b.UMSATZ) AS Total_Umsatz
5         FROM KUNDEN k
6         JOIN BESTELLUNG b ON k.NAME = b.FK_KUNDEN
7         GROUP BY k.STADT;
8     ]]
9     con:query(join_query)
10 end
11
12 function event()
13     select_query()
14 end
```

Damit sind für den Vergleich mit INT alle vier Operationen genauer definiert, aber es fehlen noch die Veränderungen für VARCHAR. 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. Man könnte nun 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:

```
1 local length = 10
```

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

```
1 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 mithilfe von 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 wieder 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 \Rightarrow enthält die prepare()- und cleanup()-Funktionen
- int_queries_insert.lua \Rightarrow enthält die insert()-Funktion
- int_queries_select.lua \Rightarrow enthält die select()-Funktion

Analog muss auch ein Ordner für die Varchar-Variante erstellt werden. Wichtig ist dabei, dass die Namen der Dateien mit dem Namen des Ordners übereinstimmen. Das Shell-Skript bedient sich dieser Struktur, führt die Lua-Skripte 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

```
1 ./sysbench_script.sh \  
2 -out "YOUR_PATH_TO_DIRECTORY/Output" \  
3 -var '{"length":[4, 16]}' \  
4 -scripts '{  
5     "YOUR_PATH_TO_DIRECTORY/Scripts/varchar_queries": {  
6         "vars": "length"  
7     },  
8     "YOUR_PATH_TO_DIRECTORY/Scripts/int_queries": {  
9         "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 wird, 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 sich beliebig viele unterschiedliche Lua-Skripts befinden, 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.

Erklärung aller möglichen Parameter des Orchestrator-Skripts:

- `-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.

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.

Im Grundlegenden arbeitet das Orchestrator-Skripts `sysbench_script.sh` 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 (siehe 2.10) definiert und überprüft. Beispielsweise wird sichergestellt, 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 verwendeten Datenbanken (unter dem Argument `-db`) und die Select-Abfragen (unter dem Argument `-selects`) des aktuellen Skripts. Daraufhin geht man in eine weitere Schleife, um die unterschiedlichen Datenbankverbindungen durchzugehen. Innerhalb dieser Schleife wird eine Methode aufgerufen, die alle Variablen vorbereitet. Zum Beispiel werden dort 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 innerhalb des für das aktuelle Skript angegebenen Ordners ein weiterer existiert oder nicht. Wenn es noch einen Ordner gibt, dann werden alle `SELECT`-Funktionen darin 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

```

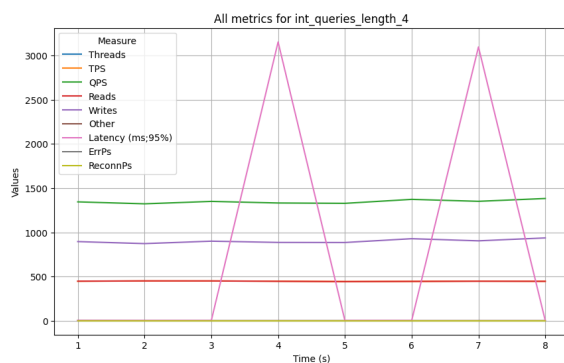
1 for SCRIPT_PATH in $SCRIPT_DIRS; do
2   DBMS=$(echo "$SCRIPTS" | jq -r --arg key "$SCRIPT_PATH" '.[key].db // ["mysql"]')
3   SELECT_QUERIES=$(echo "$SCRIPTS" | jq -r --arg key "$SCRIPT_PATH" '.[key].selects')
4   for DB in $(echo "$DBMS" | jq -r '.[0]'); do
5     prepare_variables "$SCRIPT_PATH" "$DB"
6     DB_INFO=$( [ "$DBMS_COUNT" -ne 1 ] && echo "${CUSTOM_DB_NAME}" )
7     if [ [ -n "$EXPORTED_VARS" ] ]; then
8       IFS=',' read -r -a KEYS <<< "$EXPORTED_VARS"
9
10      COMBINATIONS=$(generate_combinations "" "${KEYS[@]}")
11      while IFS=',' read -r combination; do
12        # Export key-value pairs for the current combination
13        IFS=',' read -ra key_value_pairs <<< "$combination"
14        for pair in "${key_value_pairs[@]}; do
15          export "$(echo "${pair%*=}" | tr '[:lower:]' '[:upper:]')=${pair#*=}"
16        done
17        COMBINATION_NAME=$(echo "$combination" | sed -E 's/(^),num_rows=[^,]*//g;s/^,/,;s/,,$// | tr ',' '_' |
18        tr '=' '_')
19        LOG_DIR_COMBINATION="$LOG_DIR/$COMBINATION_NAME"
20        process_script_benchmark "$DB_INFO" "$SCRIPT_PATH" "$LOG_DIR_COMBINATION" "$INSERT_SCRIPT" "
21        $SELECT_SCRIPT" "$COMBINATION_NAME"
22      done <<< "$COMBINATIONS"
23    else
24      process_script_benchmark "$DB_INFO" "$SCRIPT_PATH" "$LOG_DIR" "$INSERT_SCRIPT" "$SELECT_SCRIPT"
25    fi
26    eval $(jq -r --arg env "$DB" '.[env] | to_entries | .[] | "unset " + .key' "$ABS_PATH/envs.json")
27  done
28 done
29 # Combine csv files during runtime and end statistics and generate plots
30 python3 "$PYTHON_PATH/generateCombinedCSV.py" "$STATISTICS_FILE_TEMP" "$STATISTICS_FILE" --select_columns "
31   $STATS_SELECT_COLUMNS" --insert_columns "$STATS_INSERT_COLUMNS" --prefixes "$PREFIXES"
32 python3 "$PYTHON_PATH/generateCombinedCSV.py" "$RUNTIME_FILE_TEMP" "$RUNTIME_FILE" --select_columns "
33   $RUNTIME_SELECT_COLUMNS" --insert_columns "$RUNTIME_INSERT_COLUMNS" --prefixes "$PREFIXES"
34 python3 "$PYTHON_PATH/generatePlot.py" "$RUNTIME_FILE" "$STATISTICS_FILE"

```

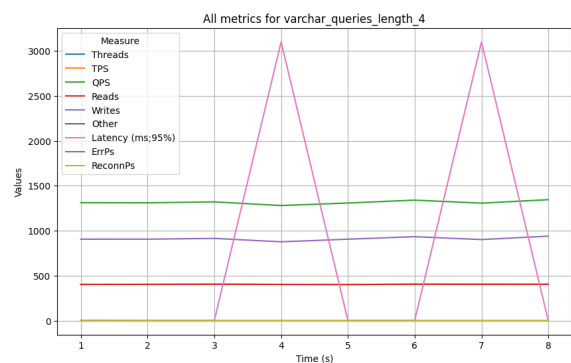
Nach dem Durchführen aller Schleifen sind alle Messwerte in CSV-Dateien gespeichert. Mithilfe von Python-Skripten müssen nun die Ergebnisse der Insert- und Select-Benchmarks aus den CSV-Dateien pro Skript vereint werden, indem die Attribute miteinander addiert werden. Als letztes werden die Graphen mithilfe von Python und Pandas erstellt.

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 beide Varianten mit der Länge 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.



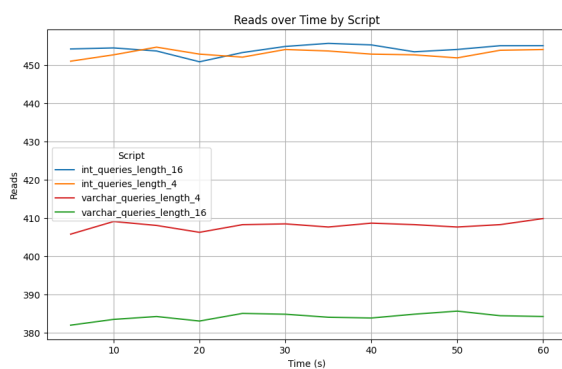
(a) int_queries_length_4



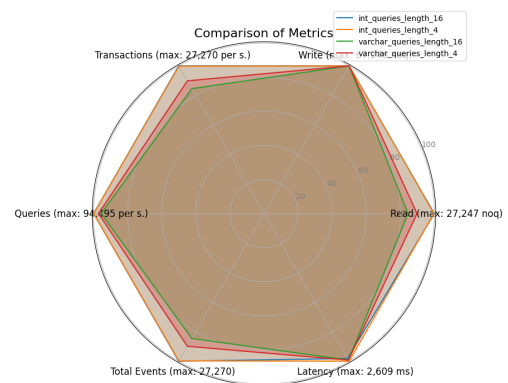
(b) varchar_queries_length_4

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 10% bessere Lese-Performance hat als VARCHAR bei einer Länge von 4. Aus dem Vergleich von den unterschiedlichen Längen mit INT kann man schließen, dass der Einfluss nicht sonderlich groß ist. In diesem Fall ist sogar die Variante mit 16 Stellen im Durchschnitt etwas schneller als die mit 4. Bei VARCHAR ist deutlich erkennbar, dass die Abfrage umso langsamer wird, je länger die Zeichenkette ist. Dennoch ist der Unterschied zwischen den Datentypen größer als innerhalb der verschiedenen Längen von VARCHAR. Es ist auch zu erkennen, dass die Werte bis auf wenige Ausnahmen sehr 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 auch bei der Latenz liegen alle Varianten nah aneinander.



(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 ausgeführt 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 Continuous Integration und Continuous Delivery (CI/CD) wurde GitHub Actions gewählt (GitHub, 2025b). 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 verfügen. Die Schritte können wiederum beliebige Shell-Befehle, Skripte oder Aktionen ausführen.

Im vorherigen Kapitel 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, hier z.B. join-type.

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

```
1 {
2   "join-type": {
3     "scripts": {
4       "./Projects/Scripts/varchar_queries": {
5         "vars": "length"
6       },
7       "./Projects/Scripts/int_queries": {
8         "vars": "length"
9       }
10    },
11    "var": {"length": [4, 16]}
12  }
13 }
```

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. Zuerst müssen, abhängig vom Projektnamen, die entsprechenden Variablen aus der JSON-Datei, die im ersten Job erstellt wurden, 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

```
1 name: Run All Benchmarks
2 on:
3   push:
4     paths: ['Projects/**', ...]
5 jobs:
6   prepare-benchmark:
7     outputs:
8       matrix: ${{ steps.set-matrix.outputs.matrix }}
9       configurations: ${{ steps.prepare-config.outputs.configurations }}
10  steps:
11    - { name: Checkout repository, uses: actions/checkout@v3 }
12    - name: Read and generate list of matrix name # echo "matrix=$matrix" >> $GITHUB_OUTPUT
13    - name: Prepare configurations for all test types
```

```

14     run: # ... export variables like test_type, dirs, var, output_dir, artifact_name as "configurations"
15 run-tests:
16   needs: prepare-benchmark
17   strategy:
18     matrix:
19       test-type: ${ fromJson(needs.prepare-benchmark.outputs.matrix) }
20   env: { TIME: 32, THREADS: 1, EVENTS: 0, REPORT_INTERVAL: 2 }
21   steps:
22     - { name: Checkout repository, uses: actions/checkout@v3 }
23     - name: Extract and save values to GitHub environment
24     - name: Install dependencies (sysbench, pandas, matplotlib)
25     - name: Start MySQL container (and wait for it to be ready)
26     run: |
27       docker run --name mysql-${ env.test_type } -d -e MYSQL_ROOT_PASSWORD=${DB_PASS} -e MYSQL_DATABASE=
28       $DB_NAME -p $DB_PORT:3306 mysql:8.0
29     - name: Run sysbench script
30     run: |
31       chmod +x Tools/Shell-Scripts/sysbench_script.sh
32       Tools/Shell-Scripts/sysbench_script.sh -out "${ env.output_dir }" \
33       -var '${ env.var }' -scripts:'${ env.dirs }'
34     - name: Stop MySQL and PostgreSQL containers
35     - name: Upload outputs # with actions/upload-artifact@v4
36 upload-combined-output:
37   needs: [prepare-benchmark, run-tests]
38   steps:
39     - name: Loop through configurations, download artifacts with artifact_name and unzip it
40     run: # ... ALL_ARTIFACTS=$(curl -s -H "Authorization: Bearer ${ secrets.GITHUB_TOKEN }" "https://api.
41       github.com/repos/${ github.repository }/actions/artifacts") ...
42     - name: Upload "Output"-folder with all downloaded benchmarks as one artifact named "combined-output"

```

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 von GitHub Caches (GitHub, 2025a) speichern. Dies bietet sich besonders deswegen an, da sich die Abhängigkeiten über die Workflows hinweg nur selten ändern. Falls sich doch etwas ändert, kann man beispielsweise die requirements.txt-Datei anpassen. Dadurch werden einmalig alle Abhängigkeiten neu installiert und anschließend im Cache abgelegt. Falls sich bis zum nächsten Workflow keine Änderungen an den Abhängigkeiten ergeben, wird der Cache automatisch 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, die in der JSON angegeben werden, die beeinflussen aber nur sich selbst und nicht die anderen Skripte. Beispielsweise, wenn in Projekt A die Anzahl an Zeilen geändert wird, die in

eine Tabelle eingefügt werden, hat dies keinen Einfluss auf das Ergebnis von Projekt B oder C. Daher würde es sich anbieten, die Benchmarks für Projekt A neu durchzuführen, während für Projekte B und C jeweils der letzte erfolgreiche Output-Ordner verwendet werden könnte. Als Endresultat könnte 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 müssen zunächst die allgemeinen Skripte sowie die benötigten Ordner mit den Lua-Skripten, die für das jeweilige Skript in der JSON-Datei erforderlich sind, gehasht werden. Diese beiden Hashes können zusammen mit den Testtypen kombiniert werden. Damit ergibt sich die folgende Struktur für den Namen:

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

Nachdem die JSON im ersten Job geladen wurde, wird nicht direkt mit dem zweiten und der damit verbundenen Installation der Abhängigkeiten fortgefahren. Stattdessen werden zunächst die unterschiedlichen Pfade gehasht und der entsprechende 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 wieder eine GitHub Cache-Lösung anbieten, aber tatsächlich sind GitHub Artifacts für das Sichern von Dateien besser geeignet (GitHub, 2025a). Eine andere mögliche Lösung wäre die Nutzung expliziter Branches ausschließlich für die Sicherung der Dateien, bei der die GitHub Action über Schreibberechtigungen verfügen muss. 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. Die Implementierung dieser Variante hat dieses Problem bestätigt. 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 in der `workflow.yaml` zu erkennen ist, wurde die Lösung mit GitHub Artifacts gewählt. 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 analysiert. Am Ende des Kapitels werden die Ergebnisse der Benchmarks betrachtet. Zunächst wird aber mit den allgemeineren Prinzipien begonnen.

3.1 Allgemeine Faktoren

In diesem Abschnitt werden die geltenden Grundsätze behandelt, die generell bei der Wahl der Datentypen beachtet werden sollten. Bevor das behandelt wird, muss geklärt werden, welche Schritte zur Auswahl von Datentypen durchgeführt werden (Schwartz et al., 2012, S. 115–145). Als erstes wird die übergeordnete Kategorie des Datentyps, wie beispielsweise numerisch, textbasiert oder zeitbezogen, festgelegt. 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. Deshalb sollte man den kleinstmöglichen Datentypen wählen, 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.“ (Winand, 2011, S. 57)

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.

Abschließend ist darauf hinzuweisen, dass MySQL eine Vielzahl von Aliasen für Datentypen unterstützt, darunter INTEGER, BOOL und NUMERIC. Obwohl diese Aliase potenziell zu Verwirrung führen können, haben sie keinen Einfluss auf 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.

3.2 Funktionsweise individueller 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 (Oracle, 2025b). 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:

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

$$\text{Unsigned: } 0 \text{ bis } 2^N - 1 \quad (3.2)$$

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:

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

Für beide Spalten wurde 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

```
1 INSERT INTO test_int (int_5, int_11) VALUES (2147483647, 2147483647);
2 INSERT INTO test_int (int_5, int_11) VALUES (-2147483648, -2147483648);
3 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. Zudem speichert DECIMAL die Zahlen in einer binären Zeichenkette mit neun Ziffern pro vier Bytes und unterstützt insgesamt bis zu 65 Ziffern.

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 ineffizienter bei Sortierungen oder Operationen auf temporären Tabellen sein kann. Deshalb sollte immer so viel Platz reserviert werden, wie tatsächlich benötigt wird.

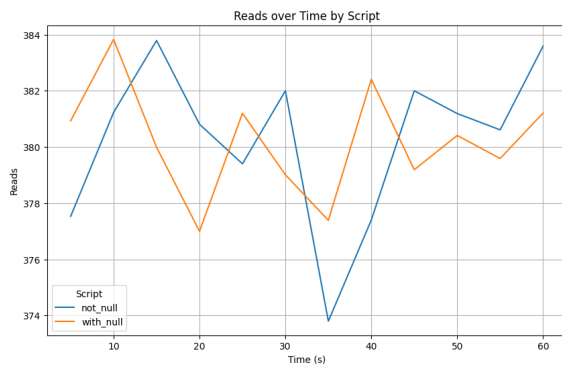
Im Gegensatz zu `VARCHAR` hat `CHAR` hingegen eine feste Länge und MySQL reserviert auch den nicht gebrauchten Platz für die angegebene Anzahl an Zeichen. Daher ist `CHAR` ideal für sehr kurze Strings oder Werte, die alle nahezu gleich lang sind, da `VARCHAR(1)` aufgrund des Längen-Bytes 2 Bytes benötigt, `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.

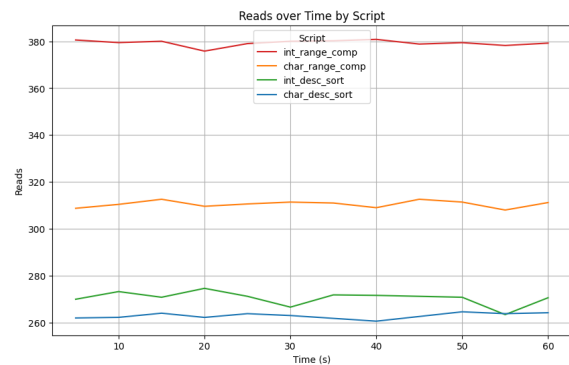
3.3 Analyse der Benchmarks

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 beim Einfügen 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 20%) 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 vor allem deswegen, weil die unterschiedlichen Werte auch aufgrund der Skalierung der y-Achse sehr stark schwanken. Als Nächstes 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.

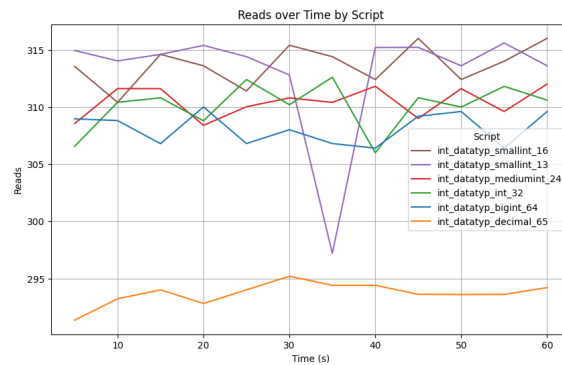
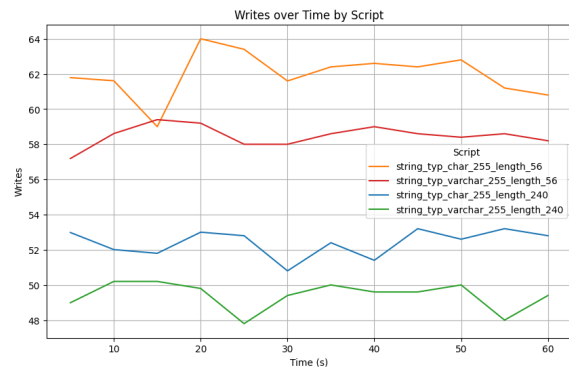
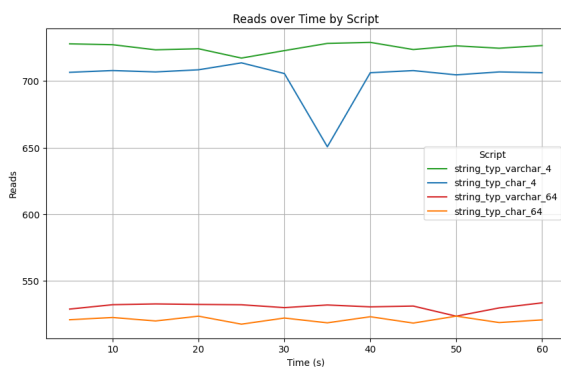


Abbildung 3.2: Vergleich von unterschiedlichen zeichenkettenbasierten Typen

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 einige 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 betrachtet. Anschließend werden die verschiedenen Arten von Indizes näher erklärt 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 Zeilen schneller zu finden. 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.

Mit zunehmender Größe der Datenbank wird das Scannen aller Tupel immer aufwendiger, weshalb Indizes eine zentrale Rolle für die Datenbank-Performance spielen. 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.

Um die Funktionsweise eines Indexes anschaulicher zu erklären, wird als Beispiel ein wissenschaftliches Fachbuch betrachtet (vgl. Schwartz et al., 2012, S. 147). 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. Ein Index kann aber nicht nur die Werte einer einzelnen Spalte enthalten, sondern auch mehrere Spalten einer Tabelle umfassen. Bei mehrspaltigen Indizes spielt die Reihenfolge der Spalten im Index eine entscheidende Rolle. Außerdem ist der Zugriff auf nicht alle Spalten bedingungslos effizient, da MySQL nur auf das linkeste Präfix des Indexes zugreifen kann. Wenn man nur das zweite Attribut eines Indexes angibt, ohne das erste zu referenzieren, kann der Index nicht direkt verwendet werden. Es ist wichtig zu beachten, dass ein Index, der über zwei Spalten definiert ist, nicht mit zwei getrennten einspaltigen Indizes gleichzusetzen ist. In diesem Fall können Abfragen nur auf einer der beiden Spalten basieren, was zu schlechterer Performance führt, wenn beide Spalten gleichzeitig abgefragt 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 (Garcia-Molina et al., 2008, S. 350–353). Das Datenbanksystem verteilt die Tupel einer Relation üblicherweise auf viele Festplattenseiten. Um die Werte eines Tupels zu prüfen, muss die gesamte Seite, auch Block genannt, in den Hauptspeicher geladen werden. Dabei erfordert es nahezu gleich viel Zeit, alle Tupel einer Seite zu prüfen, anstatt nur ein einzelnes. Aufgrund dieser Tatsache kann die Entscheidung, ob für ein bestimmtes Attribut ein Index definiert werden soll, von drei Faktoren abhängig gemacht werden. 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. Und der letzte Fall betrifft Situation, in denen Tupel nach einem Attribut geclustert sind. Da die Werte des Attributs aufeinanderfolgender gespeichert sind, müssen durch einen Index weniger Datenblöcke geladen werden.

Trotz dieser Faktoren müssen Entwickler bei der Auswahl von Schlüsseln und Indizes einen Tradeoff abwägen. 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. Dennoch kann ein Index auf ein häufig verändertes Attribut die Leistung verbessern, da einige Modifikationen zunächst eine Datenbankabfrage erfordern. Im Kapitel [Partitionen](#) wird dieses Thema erneut behandelt

Zur Entscheidungsfindung anhand einer Berechnung wird die folgende Tabelle verwendet (abgeändertes Beispiel aus Garcia-Molina et al., 2008, S. 355–357):

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

Der Schlüssel der Faktentabelle ist die Spalte Id und für die Artikel_Id sowie die Kunden_Id werden eigene Indizes erstellt, sodass insgesamt drei Indizes vorhanden sind. Als Nächstes werden Befehle benötigt, bei denen die Indizes benutzt werden (siehe 4.1). In der ersten Zeile wird nur der Kundenindex verwendet und in der zweiten nur der Artikelindex.

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;
```

Damit berechnet werden kann, ob es sinnvoll ist, die Indizes zu erstellen, müssen bestimmte Voraussetzungen festgelegt werden. Zualererst 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. Um die Seite des Indexes zu lesen, ist ein Festplattenzugriff erforderlich 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

Unter der Annahme, dass die erste Abfrage p_1 und die zweite p_2 der Zeit beansprucht, ergibt sich für I ein Anteil von $1 - p_1 - p_2$. Damit kann der Durchschnitt für den Kundenindex wie folgt berechnet (siehe letzte Zeile aus 4.1):

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

Abhängig von den Werten für p_1 und p_2 kann jede der vier Optionen die geringsten Kosten für die drei Operationen verursachen. Zum Beispiel, wenn $p_1 = p_2 = 0,1$, dann ist der Ausdruck $2 + 8p_1 + 8p_2$ am kleinsten, sodass keine Indizes bevorzugt werden würden. 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.

Um die Verantwortung für die Wahl der Indizes vom Datenbankdesigner zu übernehmen, wurden zahlreiche Tools entwickelt. Dabei optimiert das System sich selbst oder dem Entwickler werden zumindest Empfehlungen für sinnvolle Entscheidungen gegeben. Ein bewährter Ansatz zur Auswahl von Indizes ist das sogenannte Greedy-Verfahren (Garcia-Molina et al., 2008, S. 824), 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. Außerdem verwendet ihn MySQL standardmäßig für die Primary Keys (Oracle, 2025a). Die Implementierung und Nutzung des B-Baum-Indexes kann 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.“ (Ramakrishnan und Gehrke, 2002, S. 358)

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.

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.

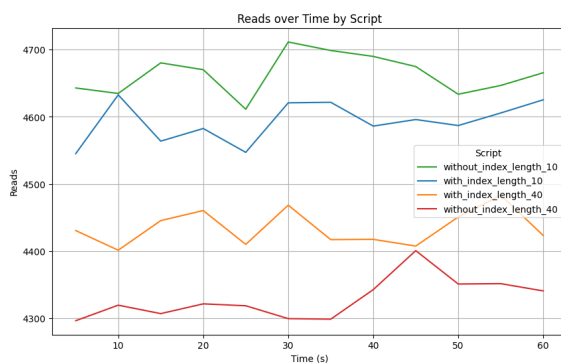
Es gibt jedoch Einschränkungen von B-Baum-Indizes, die dazu führen, dass andere Indextypen für bestimmte Szenarien besser geeignet sind. Eine Einschränkung ist, dass die Suche nicht am rechten Ende des Indexes beginnen kann. Beispielsweise ist der Beispiels-Index nicht dazu geeignet, alle Personen zu 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:

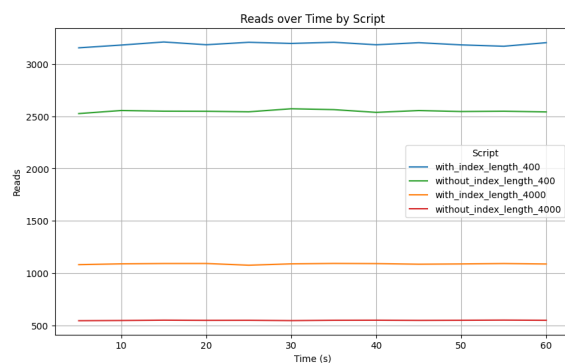
Codeblock 4.3: Definition mehrere Indizes

```
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

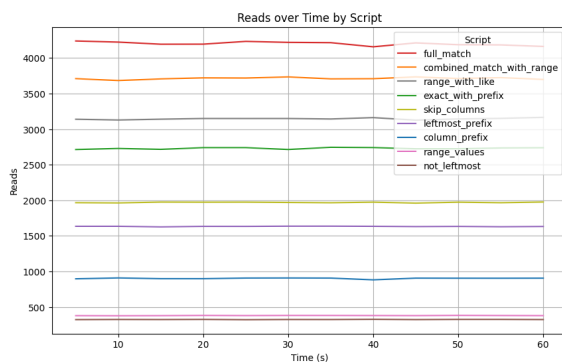
In der Abbildung 4.2a ist zu erkennen, dass bei 10 Datensätzen die Kundentabelle ohne Indizes schneller ist als die mit Indizes. Bei 40, 400 oder 4000 Zeilen (siehe 4.2a und 4.2b) wird die Wirkung der Indizes deutlich. Der Unterschied bei 40 Datensätzen ist zwar etwas geringer, aber in den anderen Fällen sind die Unterschiede noch größer. 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.

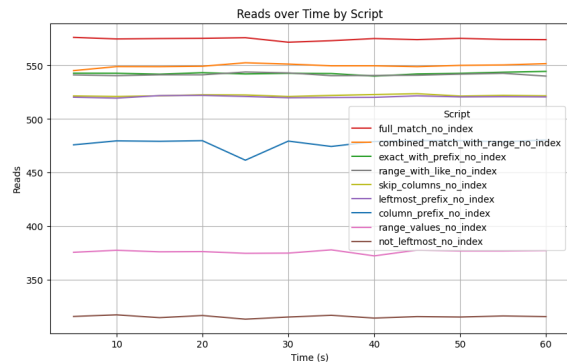
Codeblock 4.4: Unterschiedliche Where-Bedingungen für B-Tree-Index

```
1 WHERE NAME LIKE 'M%'; -- column_prefix
2 WHERE NAME = 'Müller' AND VORNAME = 'Max' AND GEBURTSTAG < '1980-01-01'; -- combined_match_with_range
3 WHERE NAME = 'Müller' AND VORNAME LIKE 'M%' ORDER BY GEBURTSTAG; -- exact_with_prefix
4 WHERE NAME = 'Müller' AND VORNAME = 'Max' AND GEBURTSTAG = '1960-01-01'; -- full_match
5 WHERE NAME = 'Müller'; -- leftmost_prefix
6 WHERE GEBURTSTAG < '1980-01-01'; -- not_leftmost
7 WHERE NAME BETWEEN 'Müller' AND 'Schulz'; -- range_values
8 WHERE NAME = 'Müller' AND VORNAME LIKE 'M%' AND GEBURTSTAG < '1980-01-01'; -- range_with_like
9 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	7.31	ja
combined_match_with_range	8	6.75	ja
range_with_like	29	5.80	ja
exact_with_prefix	51	5.03	ja
skip_columns	146	3.77	ja
leftmost_prefix	255	3.13	ja
column_prefix	517	1.88	ja
range_values	1340	1.00	nein
not_leftmost	2371	1.02	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. Damit 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 7-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. 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 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 bei Index-Wartungsoperationen mit viel Aufwand verbunden. Es gibt auch eindeutige Hash-Indizes, die 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) 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. Jedoch werden die eindeutigen Hash-Indizes nicht von der Memory-Engine in MySQL unterstützt.

Ähnlich wie der B-Baum-Index hat aber 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. Zum anderen können Hash-Indizes, anders als B-Baum-Indizes, nicht für Sortierungen verwendet werden, da die Werte nicht in einer geordneten Reihenfolge gespeichert sind. Darüber hinaus ermöglichen Hash-Indizes keine partiellen Schlüsselübereinstimmungen, da der Hash-Wert aus dem gesamten indexierten Wert berechnet wird. Bei einem Index aus den Spalten (A, B) und einer WHERE-Klausel, die nur auf A verweist, ist dies daher nicht hilfreich. Ein weiterer Nachteil von Hash-Indizes ist, dass sie keine Bereichsabfragen unterstützen und nur für Gleichheitsvergleiche wie =, <=> und IN() geeignet sind.

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.

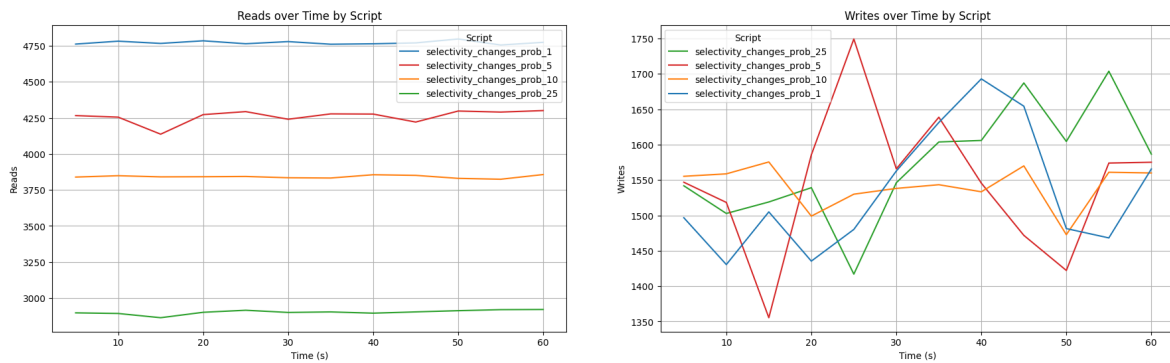


Abbildung 4.4: Vergleich der Auswirkungen von Hashkollisionen

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	2.96	ja
combined_match_with_range	9	1.16	nein
exact_with_prefix	42	1.13	nein
leftmost_prefix	204	1.22	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.

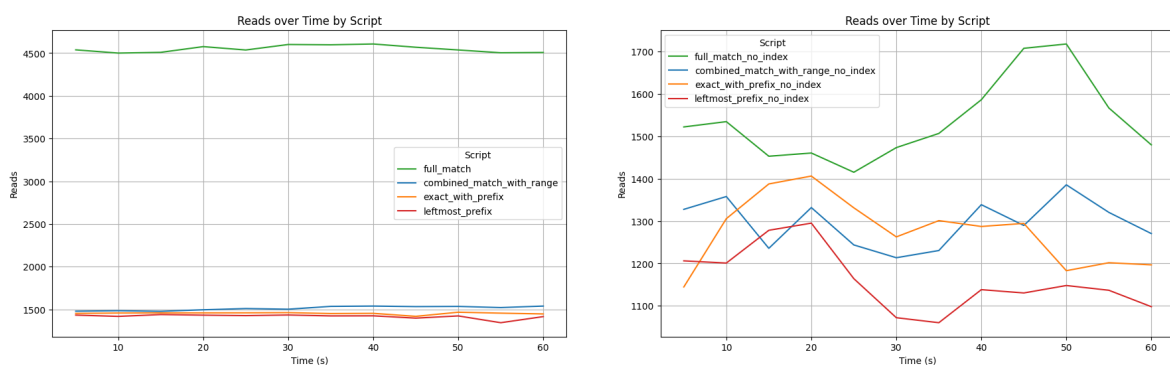
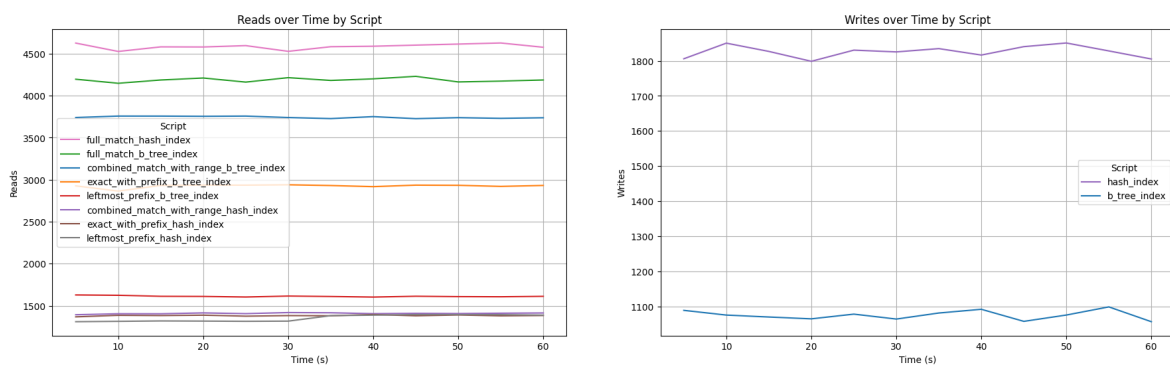


Abbildung 4.5: Grafik visualisiert Select-Queries mit (links) und ohne (rechts) Index

4.4 Vergleich zwischen 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 noch der Vergleich zwischen B-Tree- und 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. Es gibt jedoch eine weitere Klasse von SQL-Relationen, die nicht wie Tabellen physisch gespeichert werden (Garcia-Molina et al., 2008, S. 341–349, 353–366). 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.“ (Ramakrishnan und Gehrke, 2002, S. 86)

Virtuelle Sichten werden durch einen Ausdruck definiert, der einer Abfrage ähnelt. Sie können auch so abgefragt werden, als ob sie tatsächlich physisch existierten (vgl. Ramakrishnan und Gehrke, 2002, S. 87). 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 Land aggregiert werden. Die View KUNDEN_OVERVIEW hat folgende Struktur:

Codeblock 5.2: View Deklaration

```
1 CREATE VIEW KUNDEN_OVERVIEW 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;
```

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

```
1 SELECT * FROM KUNDEN_OVERVIEW
2 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

```
1 SELECT YEAR(B.BESTELLDATUM) AS Jahr, K.LAND AS Land, SUM(B.UMSATZ) AS Gesamtumsatz
2 FROM KUNDEN K
3     JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
4 GROUP BY YEAR(B.BESTELLDATUM), K.LAND
5 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. Silberschatz et al., 2011, S. 180). Die Auslösung eines Triggers kann entweder vor (BEFORE) oder nach (AFTER) einem bestimmten Ereignis erfolgen, wie INSERT, UPDATE oder DELETE. Bei Triggern auf Sichten können auch INSTEAD-OF-Trigger verwendet werden, die Änderungsversuche an der Sicht abfangen und stattdessen eine frei definierbare Aktion ausführen.

Codeblock 5.5: Allgemeine Trigger Deklaration

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

Das Problem in MySQL mit Triggern ist aber, dass sie nur auf Tabellen angewendet werden können. Später 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. Silberschatz et al., 2011, S. 173).

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. Auf diese Weise erfolgt das Einfügen der Werte in die Bestelltabelle. Materialized 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.“ (Silberschatz et al., 2011, S. 607)

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. Silberschatz et al., 2011, S. 608). Dies kann explizit mit dem folgenden Befehl durchgeführt werden:

Codeblock 5.9: Aktualisierung der materialisierten Sicht

```
1 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 Materialized View

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

Die Abfrage 5.10 könnte intern so umgeschrieben werden, dass sie nicht auf den angegebenen 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 in Oracle keine automatische Abfrageumschreibung, weshalb die Abfrage explizit auf die Tabellen zugreifen muss.

Codeblock 5.11: Select nicht mit Materialized View

```
1 SELECT Stadt, Monat, Gesamtumsatz
2 FROM KUNDEN K JOIN BESTELLUNG B ON K.KUNDEN_ID = B.FK_KUNDEN
3 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.

(Ouko, 2025). MySQL hingegen bietet gar nicht erst eine Möglichkeit an, um materialisierte Sichten nativ zu erstellen. Allerdings kann man die Funktionsweise mithilfe einer physischen Tabelle und Triggern wie folgt nachstellen nachstellen.

Zunächst muss eine physische Tabelle, z.B. mit dem Namen KUNDEN_MAT_OVERVIEW, erstellt werden. 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, da 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. Durch die Verknüpfung über KUNDEN_ID und ON DELETE CASCADE werden beim Löschen eines Kundeneintrags automatisch auch die Bestelleinträge entfernt, sodass nur die Änderungen in der Bestelltabelle für den Trigger berücksichtigt werden müssen.

In MySQL kann ein Trigger nur für einen Datenbankmanipulationsoperator gleichzeitig verwendet werden (Oracle, 2025d), weshalb für INSERT und DELETE jeweils ein Trigger definiert werden muss. Da keine Datensätze aktualisiert werden, wird aus Simplitätsgründen 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     IF EXISTS (
10         SELECT 1 FROM KUNDEN_MAT_OVERVIEW WHERE LAND = v_land AND JAHR = v_jahr
11     ) THEN
12         UPDATE KUNDEN_MAT_OVERVIEW
13         SET GESAMTUMSATZ = GESAMTUMSATZ + NEW.UMSATZ
14         WHERE LAND = v_land AND JAHR = v_jahr;
15     ELSE
16         INSERT INTO KUNDEN_MAT_OVERVIEW (JAHR, LAND, GESAMTUMSATZ)
17         VALUES (v_jahr, v_land, NEW.UMSATZ);
18     END IF;
19 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, beispielsweise 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.“ (Silberschatz et al., 2011, S. 613)

Allerdings ist die Auswahl von materialisierten Sichten deutlich komplexer 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 den 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 mit dem im Kapitel 5.2 beschriebenen Ansatz 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. Das Befüllen der Testdaten erfolgt, wie bei der virtuellen Sicht, in den ersten beiden Tabellen und bei den Select-Befehlen aus 5.13 muss der Tabellenbezeichner angepasst werden. Damit sind alle Voraussetzungen für den Vergleich zwischen keiner View, der virtuellen View und dem Ansatz mit Triggern in MySQL erfüllt.

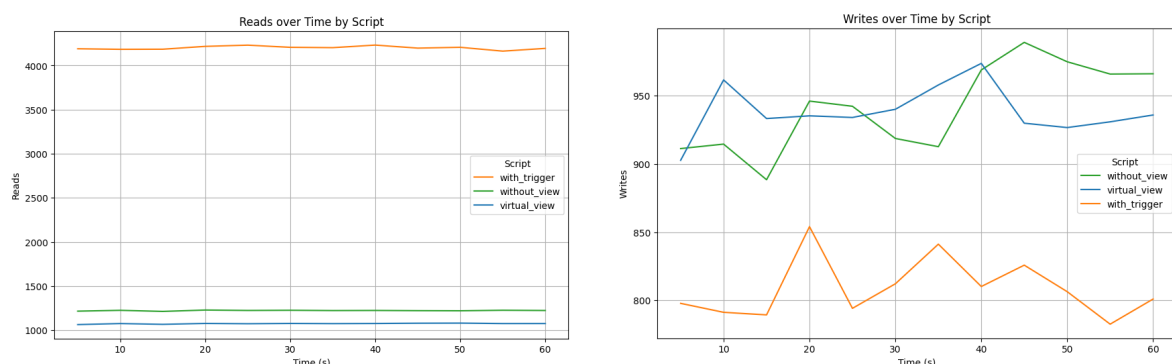


Abbildung 5.1: Vergleich zwischen keiner View, der virtuellen und Triggeransatz in MySQL

Bei den Ergebnissen fällt auf, dass die Unterschiede zwischen der virtuellen Sicht und den direkten SQL-Befehlen (`without_view`) nur minimal sind. Dennoch weist `virtual_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 4 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` durchgeführt werden müssen. Dadurch sehen wir einen deutlichen Unterschied zu den anderen beiden Ansätzen, da die Werte bei den Schreibvorgängen etwa 10–15% 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 MySQL die Logik eines Triggers direkt in der `CREATE TRIGGER`-Anweisung angeben, während in PostgreSQL ein Trigger eine separate Funktion aufruft, die die Logik enthält und mit `RETURNS TRIGGER` definiert ist. Auch die Deklaration 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.

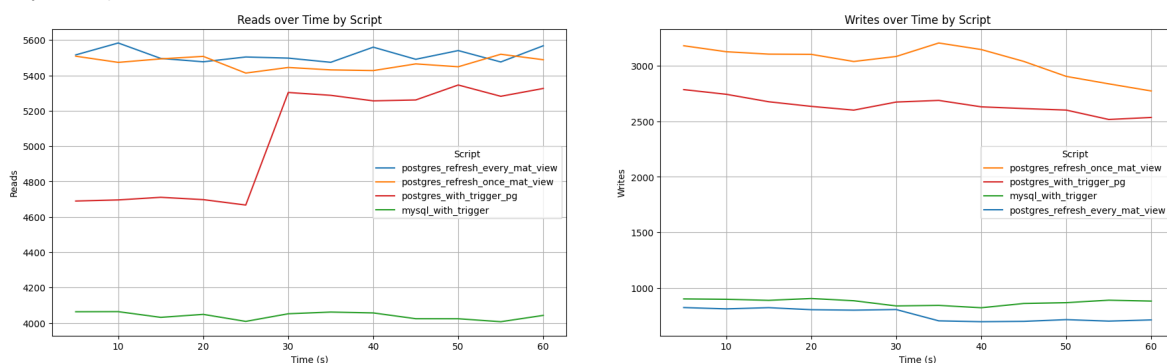


Abbildung 5.2: Vergleich zwischen Triggeransatz in MySQL und Postgres, sowie zwei nativen Implementierungen in Postgres

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. Interessant ist auch, dass der Trigger in Postgres nach etwa 25 Sekunden einen starken Performanceanstieg erfährt. Dieser Trend zeigte sich bei mehrfacher Ausführung sowohl lokal auf dem Rechner als auch im Workflow. MySQL und PostgreSQL wurden mit dem gleichen Ansatz gebenchmarkt, um die Implementierung der nativen materialisierten Sicht, die nur in PostgreSQL möglich ist, besser vergleichen zu können. 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. Damit wird noch einmal deutlich wie stark die Einfügedauer bei den materialisierten Sichten von der Anzahl an Refreshes abhängig ist, da `mat_view_refresh_every` sogar unterhalb der Performance von `with_trigger` liegt. Hingegen schneidet die einmalige Aktualisierung der Sicht im vorliegenden Beispiel besser ab als die Triggern 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 in OLTP-Systemen die Notwendigkeit besteht, häufig aggregierte Daten für analytische Auswertungen zu nutzen, erweisen sich materialisierte Sichten als 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. Kleppmann, 2017, S. 200). 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.“ (Da Silva und Tavares, 2015, S. 148)

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 7 ausführlicher erläutert wird.

Das System verwaltet Partitionen intern so, dass der Benutzer nicht bemerkt, wie genau die Daten organisiert sind (Schwartz et al., 2012, S. 265–273). 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 (Oracle, 2025c). 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 DBMS 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 einer DELETE-Abfrage 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 Einfüganfrage 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. Stattdessen legt eine mathematische Formel fest, welche Partitionen welche Kategorien von Zeilen enthalten können, wodurch nur geringer Mehraufwand verursacht wird. Zum anderen lassen sie 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).“
(Kleppmann, 2017, S. 217)

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 eine Erweiterung wie `RANGE COLUMNS` zu verwenden, die auf Funktionen in der Partitionsdefinition verzichtet. 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 Partitionierungstypen

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: Kundentabelle mit Range-Partitionierung

```
1 CREATE TABLE IF NOT EXISTS KUNDEN (  
2     KUNDEN_ID      INT NOT NULL,  
3     GEBURTSTAG     DATE NOT NULL,  
4     -- other attributes  
5     PRIMARY KEY (KUNDEN_ID, GEBURTSTAG)  
6 ) PARTITION BY RANGE (YEAR(GEBURTSTAG)) (  
7     PARTITION p1 VALUES LESS THAN (1955),  
8     PARTITION p2 VALUES LESS THAN (1960),  
9     -- other partitions  
10    PARTITION p15 VALUES LESS THAN (2025),  
11    PARTITION pmax VALUES LESS THAN MAXVALUE  
12 );
```

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

```
1 WHERE YEAR(k.GEBURTSTAG) = 1985; -- year_1985.sql  
2 WHERE k.GEBURTSTAG BETWEEN '1985-01-01' AND '1985-12-31'; -- between_1985.sql  
3 WHERE k.GEBURTSTAG = '1985-01-01'; -- first_day_1985.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: Kundentabelle mit List-Partitionierung

```
1 CREATE TABLE IF NOT EXISTS KUNDEN (
2     KUNDEN_ID      INT NOT NULL,
3     LAND           VARCHAR(100) NOT NULL,
4     -- other attributes
5     PRIMARY KEY (KUNDEN_ID, LAND)
6 ) PARTITION BY LIST COLUMNS(LAND) (
7     PARTITION p_china VALUES IN ('China'),
8     PARTITION p_india VALUES IN ('India'),
9     PARTITION p_united_states VALUES IN ('United States'),
10    -- other partitions
11    PARTITION p_thailand VALUES IN ('Thailand'),
12    PARTITION p_other VALUES IN ('Other')
13 );
```

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 garantiert 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.“ (Kleppmann, 2017, S. 203)

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.

Codeblock 6.4: Hash-Partitionierung

```
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 Auswertung der Benchmarks

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. 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 sichtbar, dass die between_1985-Query mit Partitionierung deutlich schneller ist als die anderen. Damit funktioniert das Pruning besonders gut, wenn die Geburtstage zwischen dem ersten und letzten Tage des Jahres abfragt werden. Als Nächstes kommt first_day_1985 mit der Range-Partitionierung, 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 dieser Abfrage auch nur eine Partition verwendet wurde. Etwas langsamer sind die Skripte ohne Partitionierung, die aber etwa 50% ineffizienter sind, als das schnellste Skript. Die Varianten mit YEAR() und dem ersten Tag liegen auf einem sehr ähnlichen Niveau, während die Abfrage mit dem Operator BETWEEN leicht darunter liegt. 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 die Partitionierung geringfügig schneller, jedoch sind sie nahezu gleich.

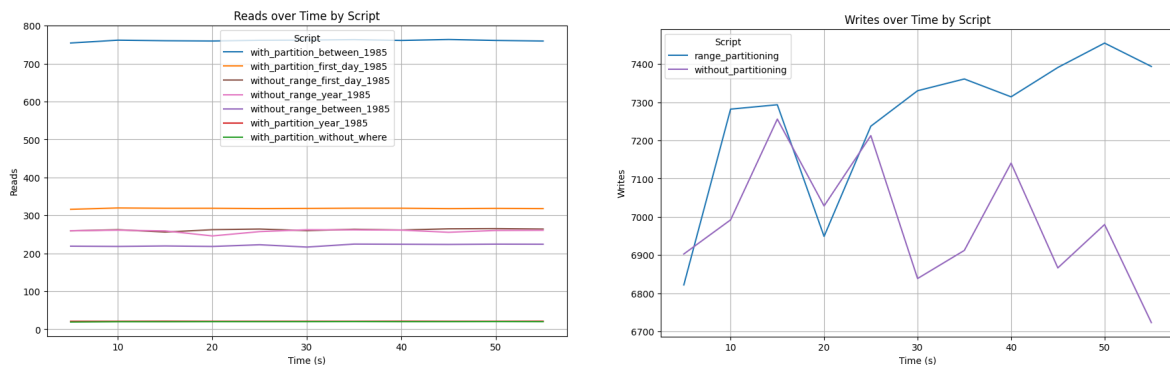


Abbildung 6.1: Vergleich zwischen der Range-Partitionierung und ohne Partition

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.

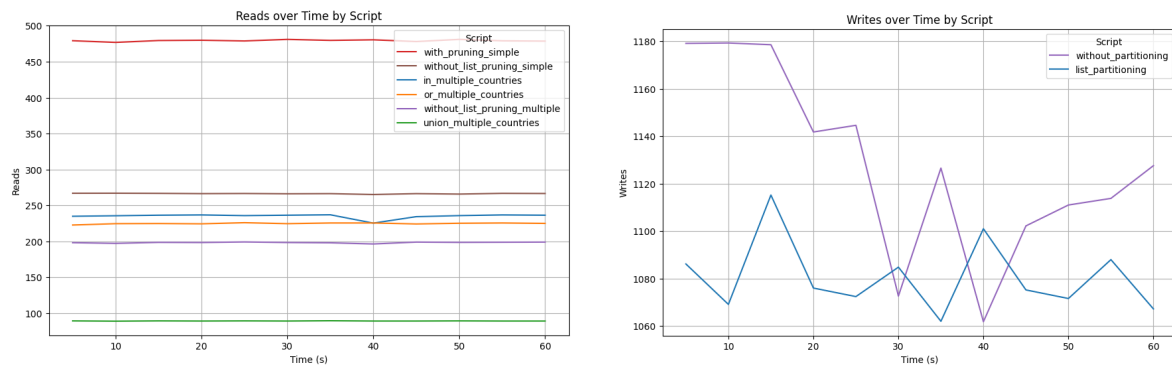


Abbildung 6.2: Vergleich zwischen der List-Partitionierung und ohne Partition

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, insbesondere wenn es um Primärschlüssel geht. Bei der Hash-Partitionierung in 6.3 fällt auf, dass die Werte der Abfragen nicht sehr konstant sind. Nur bei der Variante mit 500 Partitionen gibt es keine deutlich sichtbaren Schwankungen. Diese liegen aber nicht an dem Pruning, denn mit dem SQL-Befehl EXPLAIN sehen wir, dass immer alle 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. Bei 500 Partitionen wird der Hash-Wert mit modulo 500 gebildet, weshalb jeder 500-ste Wert in der gleichen Partition landet. 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.

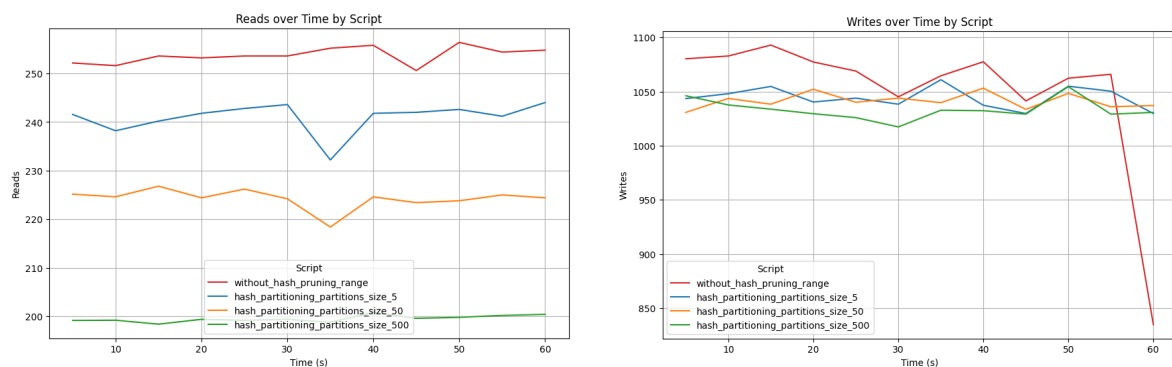


Abbildung 6.3: Vergleich zwischen der Hash-Partitionierung und ohne Partition

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. Schwartz et al., [2012](#), S. 531). 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 erläutert, 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.“ (Kleppmann, [2017](#), S. 151)

Das grundlegende Problem, das die Replikation löst, besteht darin, die Daten eines Servers mit denen eines anderen synchron zu halten. Es können sich auch mehrere Replikate mit einem einzigen Master verbinden und dessen aktuellen Zustand widerspiegeln. Sie sind aber nicht dazu geeignet, richtige Backups zu ersetzen. Außerdem lassen sich Master und Replikate in anderen Konfigurationen anordnen. Neben der klassischen 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.“ (Da Silva und Tavares, [2015](#), S. 148)

Zusätzlich sorgt sie dafür, dass die Daten näher an den Nutzern liegen, wodurch die Latenz verringert wird (vgl. Kleppmann, 2017, S. 151). 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 (Schwartz et al., 2012, S. 447–477). 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.

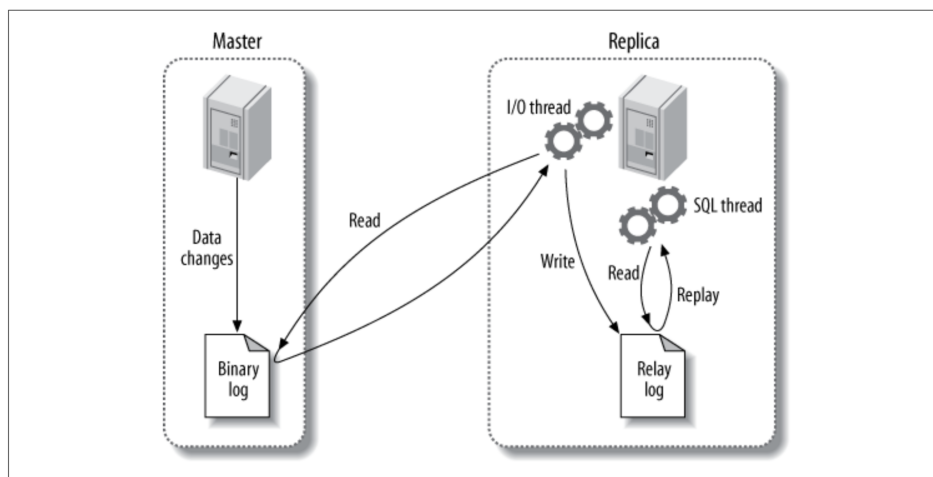


Abbildung 7.1: Darstellung der unterschiedlichen Threads

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 Quelltable scannt, aber nur drei Zeilen in der Zieltabelle bearbeitet. Bei der anweisungs-basierten 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 anweisungs-basierte 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 anweisungs-basierten 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 der Master-Replika-Architektur

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, 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 Untersuchung der Replikation

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 angepasst 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 deutlich am schnellsten ist. Die übrigen Varianten sind nur halb so schnell, liegen jedoch auf einem ähnlichen Niveau. Damit wurde auch festgestellt, dass die Select-Performance auf dem Master genauso effizient ist wie bei den Replikaten. Bei der Schreibgeschwindigkeit gibt es allerdings keinen deutlichen Unterschied zwischen den beiden Varianten, da die Replikation nur ungefähr 5% langsamer ist.

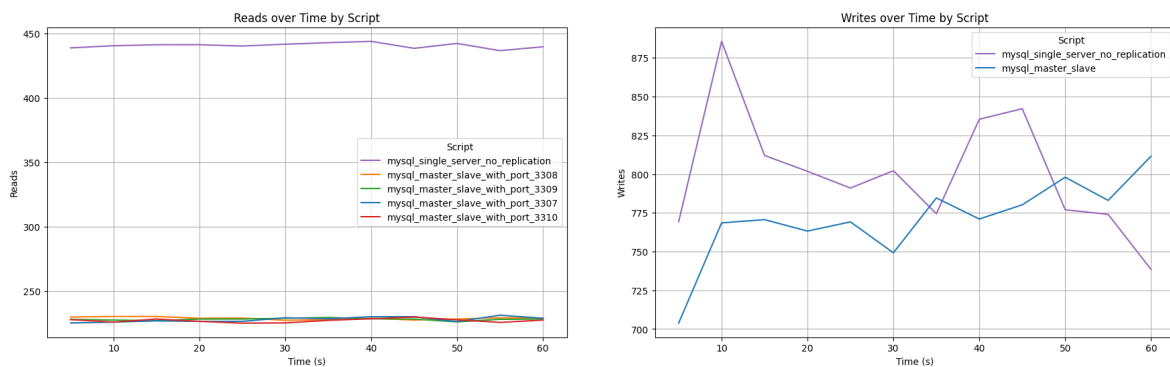


Abbildung 7.2: Vergleich zwischen Master mit 3 Replikaten und Single-Server-Ansatz

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 und der Master-Replikat-Ansatz in diesem Fall keine Vorteile bietet. 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 Replikate

je 2 Threads verarbeiten ($8/4 = 2$). Nach diesem Prinzip wird der Benchmark auch mit 16 Threads durchgeführt (siehe Abbildung 7.3a).

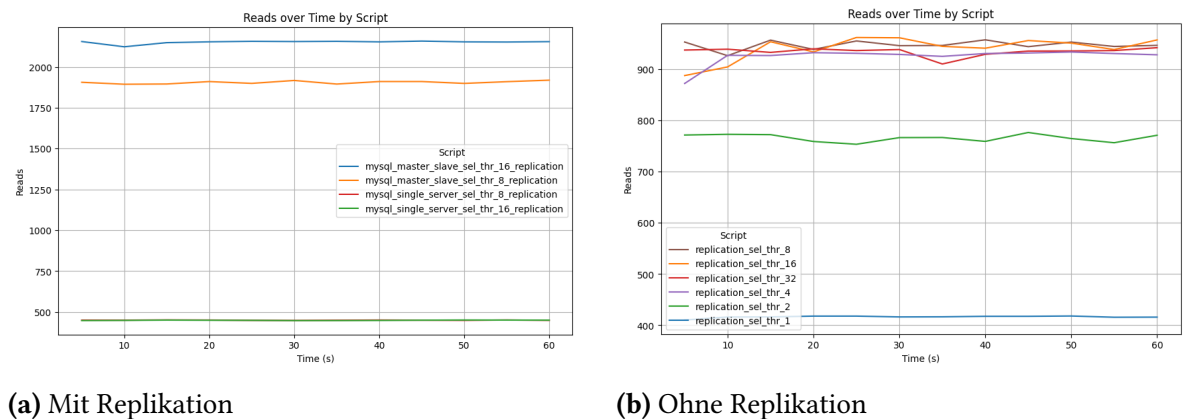


Abbildung 7.3: Vergleich von 8 Threads an Single-Server und jeweils 2 an die unters. Ports. Zunächst ist in der linken Grafik zu erkennen, dass beide Kurven des Single-Server-Ansatzes sehr ähnlich verlaufen. Auf den ersten Blick könnte das überraschen, da man erwarten würde, dass eine Verdopplung der 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

```
1 top -l 1 | grep 'CPU usage' | awk '{print $3 + $5}' %Für Mac
2 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 25%. Die Kurve mit 4 Threads weicht nur geringfügig von den anderen Threadanzahlen ab. Begründen lässt sich das durch die CPU-Auslastung. Bei einer geringeren Anzahl an Threads ist die CPU-Auslastung niedriger, was die starken Anstiege erklärt. Irgendwann ist die CPU jedoch voll ausgelastet, sodass die Performance nicht mehr weiter steigt. Dies trifft hier bei 4 bzw. 8 Threads zu, da die CPU-Auslastung dort zwischen 72 und 97% liegt.

Anzahl an Threads	Durchschnittliche CPU-Auslastung	Anzahl an Leseabfragen
1	1.73	24975
2	26.91	45972
4	72.99	55528
8	97.51	56884
16	100.00	56507
32	100.00	56133

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 gleich. 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 die 3 Replikate, am effizientesten. Danach folgt mit etwas Abstand die Version mit 8 Prozessen, die etwa 10% 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. Wenn die Threadanzahl beim Replikationsansatz weiter erhöht wird, würde auch hier irgendwann die zuvor erwähnte Grenze des Single-Server-Ansatzes erreicht werden. Der Vorteil der Replikation liegt jedoch darin, dass die erhöhte Last auf zusätzliche Replikate verteilt werden kann.

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.

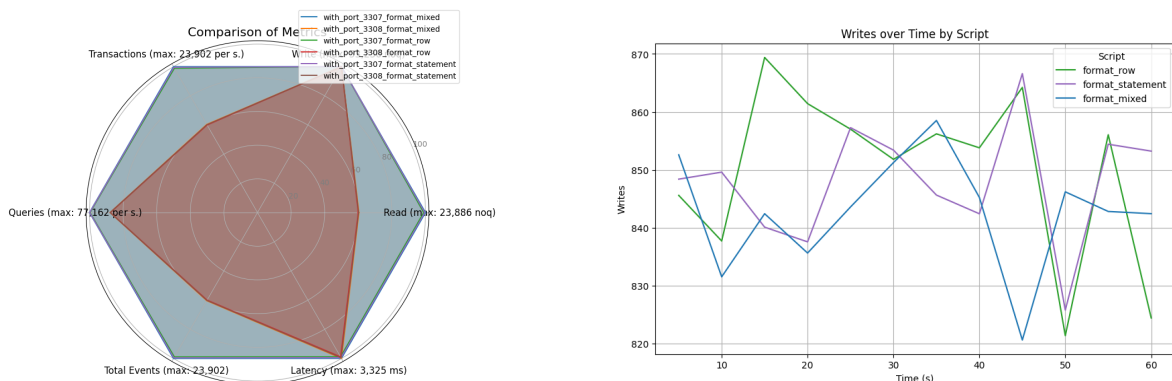


Abbildung 7.4: Vergleich zwischen den unterschiedlichen Binlog-Typen

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. Ein weiterer Leitsatz bei der Wahl der Datenformate ist, eine simplere Datenstruktur zu bevorzugen, was sich im Vergleich zeigte, da INT schneller als CHAR ist. 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 bei den Benchmarks für die Sichten festgestellt wurde, 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 sollte dieser Aspekt bei der Auswahl des DBMS berücksichtigt werden. 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. Normalerweise ist ein Merkmal, das für die Partitionierung spricht, ein natürliches Trennkriterium wie ein Zeitstempel oder geografische Regionen, da dadurch eine logische Aufteilung der Daten ermöglicht wird. Abhängig vom Trennkriterium muss man sich für einen der Partitionierungstypen entscheiden: Range, List, Hash oder Key. Der Vorteil 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. Bei der Range-Partitionierung mit einem Zeitstempel können bei einigen Operatoren unerwartete Probleme auftreten. Ein solches Beispiel stellt der YEAR()-Operator dar, der zwar dasselbe Ergebnis wie eine Bereichsabfrage liefert, jedoch nicht für das Partition-Pruning verwendet werden kann. 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. Die Hash-Partitionierung trägt zu einer gleichmäßigen Verteilung der Daten bei. Darüber hinaus wurde festgestellt, dass bei dieser sowie den anderen Typen die Komplexität der Suche innerhalb der Partitionierungsstruktur mit einer steigenden Anzahl von Partitionen zunimmt, was zu einer entsprechenden Verschlechterung der Performance führt.

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. In Bezug auf die Verteilung zeigt die Replikation Ähnlichkeiten mit den grundlegenden Konzepten von NoSQL-Datenbanken, die ebenfalls horizontale Skalierung einsetzen, um die Effizienz zu optimieren. Allerdings treten auch Nachteile beim Einfügen von Daten mit Replikation auf, da das erneute Kopieren der Daten auf die Replikate die Performance negativ beeinflusst.

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.

Literatur

- Da Silva, M. D., & Tavares, H. L. (2015). *Redis Essentials*. Packt Publishing Ltd.
- Difallah, D. E., Pavlo, A., Curino, C., & Cudré-Mauroux, P. (2013). OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4), 277–288. <http://www.vldb.org/pvldb/vol7/p277-difallah.pdf>
- Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: the complete book*. Pearson Education India.
- GitHub. (2025a). *Caching dependencies to speed up workflows*. Verfügbar 7. Januar 2025 unter <https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows#comparing-artifacts-and-dependency-caching>
- GitHub. (2025b). *Understanding GitHub Actions*. Verfügbar 20. Januar 2025 unter <https://docs.github.com/en/actions/about-github-actions/understanding-github-actions>
- Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90–95. <https://doi.org/10.1109/MCSE.2007.55>
- Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media.
- Kopytov, A. (2024). *Sysbench Github Repository*. Verfügbar 28. Oktober 2024 unter <https://github.com/akopytov/sysbench>
- Microsoft. (2024). *Database snapshots (SQL Server)*. Verfügbar 12. März 2025 unter <https://learn.microsoft.com/en-us/sql/relational-databases/databases/database-snapshots-sql-server?view=sql-server-ver16>
- Oracle. (2025a). *10.3.1 How MySQL Uses Indexes*. Verfügbar 28. Februar 2025 unter <https://dev.mysql.com/doc/refman/8.4/en/mysql-indexes.html>
- Oracle. (2025b). *13.1.2 Integer Types (Exact Value) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT*. Verfügbar 7. März 2025 unter <https://dev.mysql.com/doc/refman/8.4/en/integer-types.html>
- Oracle. (2025c). *26.6 Restrictions and Limitations on Partitioning*. Verfügbar 17. Februar 2025 unter <https://dev.mysql.com/doc/refman/8.4/en/partitioning-limitations.html>
- Oracle. (2025d). *27.3.1 Trigger Syntax and Examples*. Verfügbar 27. Januar 2025 unter <https://dev.mysql.com/doc/refman/8.4/en/trigger-syntax.html>
- Oracle. (2025e). *MySQL 8.0 Release Notes*. Verfügbar 1. März 2025 unter <https://dev.mysql.com/doc/relnotes/mysql/8.0/en/>

- Ouko, A. (2025). *SQL Materialized View: Verbesserung der Abfrageleistung*. Verfügbar 26. Januar 2025 unter <https://www.datacamp.com/de/tutorial/sql-materialized-view>
- pandas development team, T. (2020, Februar). *pandas-dev/pandas: Pandas (Version latest)*. Zenodo. <https://doi.org/10.5281/zenodo.3509134>
- Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems*. McGraw-Hill, Inc.
- Reimers, N. (2017). *Virtuelle, dezidierte und Cloud-Server: MySQL-Benchmark mittels sysbench*. Verfügbar 28. Oktober 2024 unter <https://www.webhosterwissen.de/know-how/server/mysql-benchmark-mittels-sysbench/>
- Reinboth, C. (2020). *Grundlagen der Statistik: Lagemaße – Median, Quartile, Perzentile und Modus*. Verfügbar 4. März 2025 unter <https://wissenschafts-thurm.de/grundlagen-der-statistik-lagemasse-median-quartile-perzentile-und-modus/>
- Schwartz, B., Zaitsev, P., & Tkachenko, V. (2012). *High performance MySQL: optimization, backups, and replication*. O'Reilly Media.
- Shopify. (2022a). *Detailed design documentation*. Verfügbar 28. Oktober 2024 unter <https://shopify.github.io/mybench/detailed-design-doc.html#live-monitoring-user-interface>
- Shopify. (2022b). *What is mybench?* Verfügbar 28. Oktober 2024 unter <https://shopify.github.io/mybench/introduction.html>
- Shopify. (2024). *Mybench Github Repository*. Verfügbar 28. Oktober 2024 unter <https://github.com/Shopify/mybench>
- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). *Database system concepts. Development*.
- Vogel, M. (2009). *EDV-Lexikon: Bottleneck*. Verfügbar 28. Oktober 2024 unter <https://martinvogel.de/lexikon/bottleneck.html>
- Williams, T., Kelley, C., & many others. (2010, März). *Gnuplot 4.4: an interactive plotting program*.
- Winand, M. (2011). *SQL performance explained. Development*.

Anhang

Codeblock 1: Konfiguration des Master-Replikat-Ansatzes

```
1 number_of_replicas=${1:-3}
2 docker network create mysql-network
3
4 # Start Primary Container
5 docker run -d --name mysql-primary --network mysql-network -e MYSQL_ROOT_PASSWORD=password -p 3307:3306 mysql:8 --
  server-id=1 --log-bin=mysql-bin
6 until docker exec mysql-primary mysqladmin ping -uroot -ppassword --silent &> /dev/null; do
7   echo "Waiting for MySQL Primary to be ready..."; sleep 5
8 done
9
10 # Start Replica Containers
11 for ((i=1; i<=number_of_replicas; i++)); do
12   docker run -d --name mysql-replica-${i} --network mysql-network -e MYSQL_ROOT_PASSWORD=password -p $((3307 + i))
    :3306 mysql:8 --server-id=$((i + 1)) --log-bin=mysql-bin --read-only=1
13   until docker exec mysql-replica-${i} mysqladmin ping -uroot -ppassword --silent &> /dev/null; do
14     echo "Waiting for MySQL Replica ${i} to be ready..."; sleep 5
15   done
16 done
17
18 MASTER_STATUS=$(docker exec mysql-primary mysql -uroot -ppassword -e "SHOW BINARY LOG STATUS\G")
19 BINLOG_FILE=$(echo "$MASTER_STATUS" | awk '/File:/ {print $2}')
20 BINLOG_POS=$(echo "$MASTER_STATUS" | awk '/Position:/ {print $2}')
21
22 # Setup Primary
23 echo "Configuring primary MySQL server..."
24 docker exec -i mysql-primary mysql -uroot -ppassword -e "
25 CREATE DATABASE sbtest;
26 CREATE USER 'repl'@ '%' IDENTIFIED WITH sha256_password BY 'repl_password';
27 GRANT REPLICATION SLAVE ON *.* TO 'repl'@ '%';
28 FLUSH PRIVILEGES;
29 "
30
31 # Setup Replicas
32 for ((i=1; i<=number_of_replicas; i++)); do
33   docker exec -i mysql-replica-${i} mysql -uroot -ppassword -e "
34   CHANGE REPLICATION SOURCE TO
35     SOURCE_HOST = 'mysql-primary',
36     SOURCE_USER = 'repl',
37     SOURCE_PASSWORD = 'repl_password',
38     SOURCE_LOG_FILE = '$BINLOG_FILE',
39     SOURCE_LOG_POS = $BINLOG_POS;
40   START REPLICATION;
41   "
42 done
43 echo "MySQL replication setup completed."
```


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