

BACHELORARBEIT

Entwurf 5: Performance – Optimierung von Datenbanken

vorgelegt am 28. Februar 2025
Daniel Freire Mendes

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

**HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG**

Department Informatik
Berliner Tor 7
20099 Hamburg

Zusammenfassung

TO-DO weiterverbessern: bisher nur eine erste Skizze der Zusammenfassung.

In modernen Datenbanksystemen, insbesondere bei den weit verbreiteten relationalen Datenbanken, ist die Performance ein entscheidender Faktor. Je komplexer die Abfragen und je größer das Datenvolumen, desto länger kann die Ausführungsdauer bestimmter Abfragen sein. Eine der größten Herausforderungen besteht darin, eine geeignete Strategie für das Problem auszuwählen.

Abhängig von zusätzlichen Zielen neben der Performance, wie der Verteilung der Daten, gibt es unterschiedliche Lösungskonzepte. Es geht auch darum einschätzen zu können, inwiefern sich die Veränderungen bezüglich der Lese- als auch Schreibgeschwindigkeit auswirken. Dafür benutzen wir ein Benchmark-Tool sowie verschiedene Tools zur Visualisierung der Ergebnisse.

In der Bachelorarbeit werden verschiedene Lösungsansätze, wie Datentypen, Indexierung, Views, Partitionen und Replikation näher erläutert. Zu jedem Thema wird ein Beispiel gegeben, das als Orientierung dient und die Auswirkungen der jeweiligen Änderungen verdeutlicht. Unser Fokus liegt auf dem Datenbankmanagementsystem MySQL, jedoch wird im Kapitel zu Views die native Implementierung von materialisierten Views in PostgreSQL analysiert.

Die vorgestellten Optimierungsansätze dieser Arbeit bieten wertvolle Erkenntnisse für Datenbankadministratoren, die leistungsfähige Lösungen in verschiedenen Anwendungsbereichen implementieren möchten.

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

Abstract

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

Inhaltsverzeichnis

| | |
|--|------------|
| Abbildungsverzeichnis | III |
| Tabellenverzeichnis | IV |
| 1 Einleitung | 1 |
| 1.1 Einführung in Benchmarks | 1 |
| 1.2 Measures | 1 |
| 2 Grundlagen | 2 |
| 2.1 Auswahl der Tools | 2 |
| 2.2 Einführung in die Tools | 2 |
| 2.3 Projektaufbau mit Beispiel | 2 |
| 2.4 GitHub Action | 12 |
| 2.5 Optimierungen des Workflows | 16 |
| 3 Optimierungen von Datentypen | 18 |
| 3.1 Allgemeine Faktoren | 18 |
| 3.2 Einzelne Datentypen und weitere Faktoren | 19 |
| 3.3 Analyse der Benchmarks | 22 |
| 4 Indexierung | 25 |
| 4.1 Grundlagen | 25 |
| 4.2 B-Baum-Index | 29 |
| 4.3 Hash-Index | 33 |
| 5 Views | 37 |
| 5.1 Virtuelle Views | 37 |
| 5.2 Materialisierte Views | 41 |
| 5.3 Durchführung der Benchmarks | 43 |
| 5.4 Analyse der Ergebnisse | 47 |
| 6 Partitionen | 49 |
| 6.1 Grundlagen | 49 |

| | | |
|----------|------------------------|-----------|
| 6.2 | Durchführung | 53 |
| 6.3 | Analyse | 56 |
| 7 | Replikation | 59 |
| 7.1 | Grundlagen | 59 |
| 7.2 | Durchführung | 63 |
| 7.3 | Analyse | 66 |
| 8 | Fazit | 69 |
| | Literatur | 72 |
| | Anhang | 73 |

Abbildungsverzeichnis

| | | |
|-----|--|----|
| 2.1 | Join-Typ: Skriptvergleich | 11 |
| 2.2 | Join-Typ: Metrikvergleich | 12 |
| 2.3 | Join-Typ: Hexagon-Diagramm | 12 |
| 3.1 | Datentypen: Null und Not Null | 22 |
| 3.2 | Datentypen: Int und Char | 23 |
| 3.3 | Datentypen: Numerische und zeichenkettenbasierte Typen | 24 |
| 4.1 | Binärbaum-Visualisierung | 29 |
| 4.2 | B-Tree-Indexing: Mit Index vs Ohne | 31 |
| 4.3 | B-Tree-Indexing: Unterschiedliche Selects mit Index und Ohne | 32 |
| 4.4 | Hash-Indexing: Auswirkungen von Hashkollisionen | 35 |
| 4.5 | Hash-Indexing: Unterschiedliche Abfragen mit Index und Ohne | 36 |
| 5.1 | Views: Keine View, virtuelle View und Ansatz mit Triggern | 47 |
| 5.2 | Views: Beide Triggeransätze sowie materialisierte Sicht | 48 |
| 6.1 | Range-Partitionierung: Unterschiedliche Selects mit und ohne Partition | 56 |
| 6.2 | List-Partitionierung: Unterschiedliche Abfragen mit und ohne Partition | 57 |
| 6.3 | Hash-Partitionierung: Variationen der Partitionsanzahl | 58 |
| 7.1 | Master-Replica-Visualisierung | 60 |
| 7.2 | Replikation: Master-Replica-Ansatz vs Single-Server | 67 |
| 7.3 | Replikation: Threadanzahl aufgeteilt an Master-Replica | 67 |
| 7.4 | Replikation: Unterschiedlichen Binlog-Typen | 68 |

Tabellenverzeichnis

| | | |
|-----|---|----|
| 3.1 | Ergebnis der SQL-Abfrage aus 3.2 | 20 |
| 4.1 | Performance-Vergleich | 28 |
| 4.2 | Ergebnisse der COUNT(*)-Abfragen für B-Tree-Index | 33 |
| 4.3 | Ergebnisse der COUNT(*)-Abfragen für Hash-Index | 35 |

1 Einleitung

1.1 Einführung in Benchmarks

TODO

1.2 Measures

TODO

2 Grundlagen

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

2.1 Auswahl der Tools

TODO

2.2 Einführung in die Tools

TODO

2.3 Projektaufbau mit Beispiel

In dem vorausgegangenen Kapitel [Einführung in die Tools](#) wurde das Tool Sysbench und seine Funktionsweise anhand eines Demo - Projekts näher erläutert. Damit die Reihenfolge und die Bedeutungen der unterschiedlichen Methoden (prepare → run → cleanup) sowie die Vorgehensweise zur Erstellung unserer Grafiken deutlich geworden. Das bisherige Problem ist aber, dass wir bei dem dargelegten Beispiel keine Kontrolle über die getesteten Daten haben. Wenn man sich die Logs genauer anschaut, dann sieht man, dass man über die Parameter

an den Sysbench – Befehl die Anzahl der erstellten Tabellen und eingefügten Datensätze von außen steuern kann, aber die genaue Implementierung können wir auf diese Weise nicht steuern. Genau für diese Anwendungsfälle gibt es die Möglichkeit ein Lua - Skript als Parameter beim Sysbench - Aufruf mit anzugeben. In diesen Lua - Dateien können die Implementierungen der einzelnen Methoden selbstständig gewählt werden.

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

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

Codeblock 2.1: Create Table - Befehl für Tabelle Kunden

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

Codeblock 2.2: Create Table - Befehl für Tabelle Bestellung

```
1 CREATE TABLE BESTELLUNG (
```

```

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

```

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

Codeblock 2.3: Lua - Script für die Erstellung der Tabellen

```

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

```

Wenn die Datenbank beispielsweise in einer Produktivumgebung läuft, dann wollen wir, dass die Benchmarks möglichst wenig Einfluss auf sie haben. Damit ist es das Ziel, dass die Datenbank möglichst wieder in ihrem Anfangszustand ist. Außerdem sollte der Benchmark beliebig oft nacheinander ausgeführt werden können, ohne zu Problemen zu führen. Wenn wir aber eine Tabelle erstellen und nicht wieder löschen, dann würde im nächsten Durchlauf der Create Table - Befehl scheitern. Lösen könnte man dies über Klausel „IF NOT EXISTS“ bei der Erstellung der Tabelle hinzufügen oder noch es besser ist es die Tabelle am Ende des Benchmarks einfach zu löschen. Dafür ist die `cleanup()` – Funktion vorgesehen:

Codeblock 2.4: Lua - Script für das Aufräumen

```

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

```

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

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

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

Codeblock 2.5: Lua - Script für das Einfügen von Daten

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

```

31         db_query(bestellung_query)
32     end
33 end
34 end
35
36 function event()
37     insert_data()
38 end

```

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

Codeblock 2.6: Lua - Script für das Abfragen von Daten

```

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

```

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

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

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

Analog muss auch ein Ordner für die Varchar - Vergleich erstellt werden. Als Letztes brauchen wir nur einen Orchestrator, der das korrekte Lua - Skript ausführt, die Ergebnisse in die

richtige Log-Datei schreibt und anschließend die CSV - Dateien und die Grafiken erstellt. Dieser Orchestrator ist das Shell - Skript: sysbench_script.sh.

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

Dies ist die Zeile mit der festgelegten Länge:

```
1 local length = 10
```

Die Zeile mit der importierten Länge:

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

Den Orchestrator - Script ruft man wie folgt auf:

Codeblock 2.7: Befehl zum Ausführen des Orchestrator Skripts

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

Die Parameter haben folgende Funktion:

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

Die letzte Besonderheit ist es, dass man mehrere Select – Abfragen ohne unterschiedliche Insert - Befehle definieren kann. Zu einem späteren Punkt in der Bachelorarbeit werden wir zu unterschiedliche Indextypen kommen. Um zu untersuchen, ob ein bestimmter Indextyp bei Abfragen verwendet wird, müssen wir nur unterschiedliche Selects abfragen. Die eigentlichen Tabellen und deren Datensätze müssen dabei nicht immer wieder gleich befüllt werden. Wenn wir auch unsere Ordnerstruktur mit dem Int - Query Beispiel zurückkommen, dann könnte man anstelle von `int_queries_select.lua` auch einen Ordner erstellen mit den Namen `int_queries_select`. In diesem Ordner können beliebig viele unterschiedliche Lua - Skripts sein, die select – Funktionen haben. Dadurch werden alle Select - Befehle auf der gleichen Datenbasis verglichen und so können wir im Kapitel 4.1 erkennen, wann der Index verwendet wird und wann nicht.

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

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

Wenn wir den Befehl ausführen, wird der Output-Ordner an der gewünschten Stelle erstellt. In diesem Ordner werden verschiedene Grafiken generiert, die die Ergebnisse visualisieren. Dabei gibt es zwei unterschiedliche Arten von Grafiken. Die erste Art von Grafik ist ein Zeitreihendiagramm, welches auf der x-Achse den zeitlichen Verlauf zeigt. Auf der y-Achse werden in einigen Diagrammen die unterschiedlichen Metriken für jedes einzelne Skript dargestellt, während andere Diagramme die Werte einer bestimmten Metrik auf der y-Achse zeigen und dabei die Ergebnisse verschiedener Skripte vergleichen. Dadurch können beispielsweise die Metriken `Reads` und `Writes` analysiert werden, um herauszufinden, welches Skript in diesen Bereichen besser abschneidet. Danach wird der Output - Ordner erstellt und die Spaltenüberschriften in die CSV – Dateien geschrieben. Anschließend beginnt erst das eigentliche Durchgehen der unterschiedlichen Skripte, die unter dem Argument `-script` angegeben wurden. Zunächst schreibt man die einzelnen Dateien nach dem obigen Schema (2.3) auf, denn als Argument wurde nur der oberste Ordner angegeben. Als Nächstes kommt eine Fallunterscheidung, die überprüft, ob dieses Skript exportierte Variable nutzt oder nicht. Für den Fall, dass keine Variablen exportiert werden (z.B. `int_queries`) wird einfach die `Prepare` – Funktion aufgerufen, dann `process_script_benchmark` und anschließend die `Cleanup` - Funktion. Wenn aber Variablen exportiert werden, dann müssen weitere Zwischenschritte

umgesetzt werden. Und zwar müssen alle Kombinationen zwischen den verschiedenen exportierten Variablen generiert werden. Wenn es drei Variablen gibt, von denen 2 jeweils 2 Werte und eine letzte nur einen Wert hat, dann gibt es $2 \times 2 \times 1 = 4$ unterschiedliche Kombinationen. Als Nächstes muss man für alle diese Kombinationen die Schritte ausführen, die man schon bei der Variante ohne exportierte Variable ausgeführt hat und dabei darf man nicht vergessen die Variablen an sich zu exportieren.

Codeblock 2.8: Ausschnitt aus Orchestrator Script

```

1 # Main benchmark loop
2 for INFO in "${QUERY_INFO[@]"; do
3     # Definition of the variables comes here
4
5     if [[ -n "$EXPORTED_VARS" ]]; then
6         IFS=';' read -r -a KEYS <<< "$EXPORTED_VARS"
7         combinations=$(generate_combinations "" "${KEYS[@]}")
8         # Process each combination
9         while IFS=';' read -r combination; do
10             IFS=';' read -ra key_value_pairs <<< "$combination"
11             for pair in "${key_value_pairs[@]"; do
12                 export "$(echo "${pair%*=}" | tr '[:lower:]' '[:upper:]')=${pair##*=}"
13             done
14             COMBINATION_NAME=$(echo "$combination" | sed -E 's/(^|,)num_rows=[^,]*//g;s/^,/,/;s/,/,/' | tr ',' '_' | tr
15             '=' '_')
16             LOG_DIR_KEY_VALUE="$LOG_DIR/$COMBINATION_NAME"
17             mkdir -p "$LOG_DIR_KEY_VALUE"
18
19             process_script_benchmark "$QUERY_PATH" "$LOG_DIR_KEY_VALUE" "$INSERT_SCRIPT" "$SELECT_SCRIPT" "
20             $COMBINATION_NAME"
21             done <<< "$combinations"
22         else
23             # Process normally when no keys specified
24             mkdir -p "$LOG_DIR"
25             process_script_benchmark "$QUERY_PATH" "$LOG_DIR" "$INSERT_SCRIPT" "$SELECT_SCRIPT"
26         fi
27     done

```

Die Funktion `process_script_benchmark` führt wie beim Demo - Beispiel schon erwähnt, die Methoden `prepare`, `insert`, `select` und `cleanup` durch. Außerdem überprüft sie auch, ob es sich bei dem `Select - Directory` um einen Ordner handelt oder nicht. Wenn es ein Ordner ist, dann werden alle Dateien in diesem Ordner mit Sysbench durchgeführt, wenn nicht, dann wird an den Ordner nur die Endung `.lua` hinzugefügt.

Codeblock 2.9: Methode Process Script Benchmark

```

1 process_script_benchmark() {
2     local QUERY_PATH="$1" LOG_DIR="$2" INSERT_SCRIPT="$3" SELECT_SCRIPT="$4" COMBINATION="${5:-}"
3     local SCRIPTS=()
4     local IS_FROM_SELECT_DIR=false
5
6     if [ -f "$SELECT_SCRIPT.lua" ]; then
7         # SELECT_SCRIPT is a Lua file

```

```

8   SCRIPTS=("$INSERT_SCRIPT" "$SELECT_SCRIPT.lua")
9   else
10    # SELECT_SCRIPT is a directory
11    SCRIPTS=("$INSERT_SCRIPT" "$SELECT_SCRIPT"/*)
12    IS_FROM_SELECT_DIR=true
13  fi
14
15  # Prepare benchmark
16  PREPARE_LOG_FILE="$LOG_DIR/$(basename "$QUERY_PATH")${COMBINATION:+_${COMBINATION}}_prepare.log"
17  run_benchmark "$MAIN_SCRIPT" "prepare" "$PREPARE_LOG_FILE" "" "${COMBINATION_NAME:-}"
18
19  # Select and Insert benchmark
20  for SCRIPT in "${SCRIPTS[@]"; do
21    if [ -f "$SCRIPT" ]; then
22      local SCRIPT_NAME
23      # Removed script name definition for less complexity => multiple if-cases processed
24      local RAW_RESULTS_FILE="$LOG_DIR/${SCRIPT_NAME}.log"
25      run_benchmark "$SCRIPT" "run" "$RAW_RESULTS_FILE" "$SCRIPT_NAME" "$COMBINATION"
26    fi
27  done
28
29  #Cleanup benchmark
30  run_benchmark "$MAIN_SCRIPT" "cleanup" "$LOG_DIR/$(basename "$QUERY_PATH")${COMBINATION:+_${COMBINATION}}_cleanup
31  .log"
32 }

```

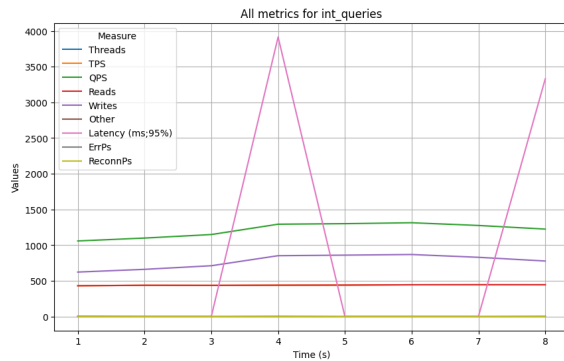
INFO: Die Shell-Ausschnitte sind zum Teil verkürzt und würden auf diese Weise nicht funktionieren.

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

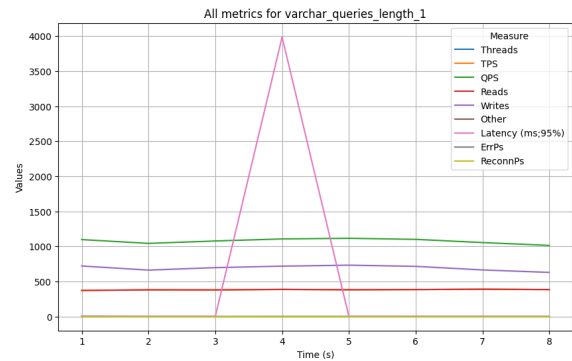
Wenn wir das komplette Skript für unser Join-Typ-Beispiel ausführen, dann ist das Endresultat ein Output-Ordner, der an angegebener Stelle erstellt wird. In diesem Ordner werden verschiedene Grafiken generiert, die die Ergebnisse visualisieren. Die erste Art stellen Zeitreihendiagramme dar, die auf der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

Die zweite Art von Grafik, die erstellt wird, ist ein Hexagon - Diagramm. Dieses verzichtet auf eine Zeitachse und fasst die Performance über den gesamten Zeitraum hinweg zusammen. Im Vergleich zur Laufzeitanalyse liefert es zusätzliche Informationen, wie etwa die Latenz

oder die Gesamtanzahl der Queries. Dadurch ist es auch möglich, dass mehrere Skripte und mehrere Kennzahlen in einer Grafik dargestellt werden können.



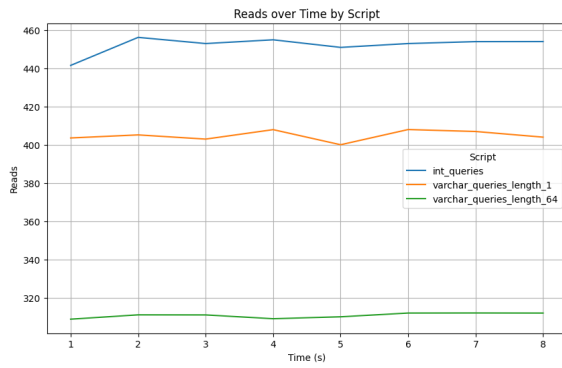
(a) int_queries



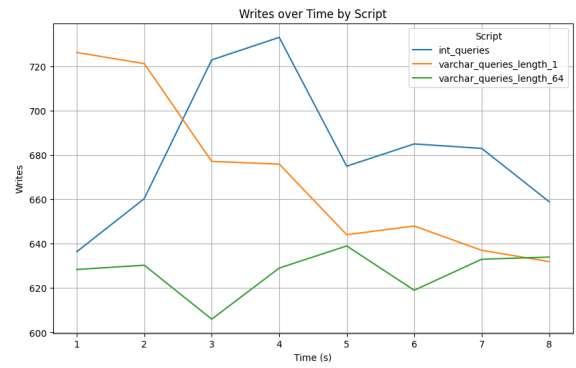
(b) varchar_queries_length_1

Abbildung 2.1: Die Grafik zeigt alle Metriken für die jeweiligen Skripte

Aus den Grafiken, die für ein Skript alle Metriken veranschaulichen, kann man möglicherweise Datenfehler erkennen. So springt bei Abbildung (int_queries.png) die Latenz bei einigen Messpunkten von 0 ms auf einen deutlich erhöhten Wert und danach wieder auf 0 ms zurück. Ansonsten aber sind die anderen Metriken auf einem konstanten Level, und es gibt wenige Schwankungen. Bei der Abbildung (varchar_queries_length_1.png) sieht dies sehr ähnlich aus, und auch dort schwankt die Latenz etwas mehr. Wenn wir jetzt die drei Skripte miteinander vergleichen wollen, können wir die Abbildungen Reads.png und Writes.png heranziehen. Was die Lesegeschwindigkeit angeht, kann man erkennen, dass int_queries am meisten Reads hat, als Nächstes kommt varchar_queries_length_1 und dann varchar_queries_length_64. Damit sind die Abfragen, wie wir erwartet haben, bei int_queries am schnellsten, und je länger der String wird, desto langsamer werden die Abfragen. Bei den Schreibgeschwindigkeiten sieht das schon etwas anders aus, wobei es hier zunächst bei allen eine langsamere Startphase gibt. Anschließend an diesen Cold Start liegt das Niveau von int_queries am höchsten, also auch am schnellsten. Die beiden varchar_queries sind hier aber überraschenderweise auf einem ähnlichen Niveau.



(a) Reads



(b) Writes

Abbildung 2.2: Die Grafik zeigt den Vergleich zwischen allen Skripten für die Metriken. Bei der Abbildung (statistics.png) kann man die Effekte der Lese- und Schreibgeschwindigkeiten auch erkennen. Es fällt auch auf, dass, anders als bei Reads, Writes, Queries, die Latenz bei schnelleren Queries geringer ist und nicht der höchste Wert der Beste ist.

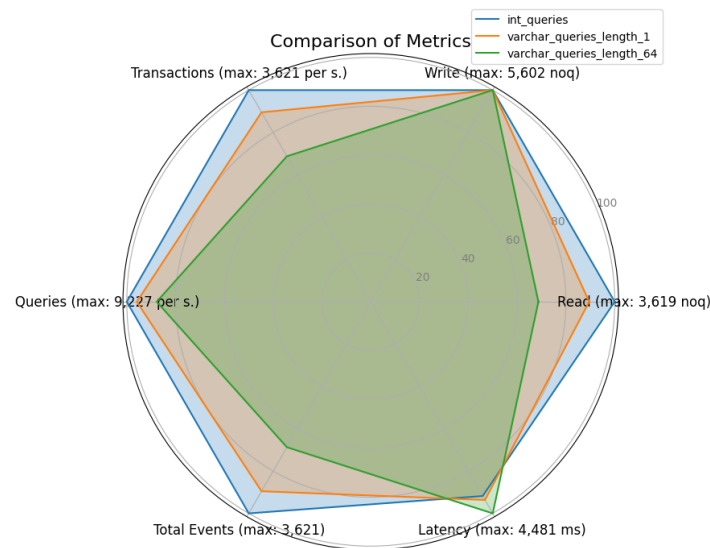


Abbildung 2.3: Darstellung der Skripte und 6 Metriken in einem Hexagon

2.4 GitHub Action

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

werden, was nicht nur zeitintensiv war, sondern auch hohe Lasten für den lokalen Rechner bedeutete. Eine Möglichkeit wäre es gewesen, die Skripte parallel durchlaufen zu lassen, um Zeit zu sparen, damit wäre aber nicht das Lastenproblem und der manuelle Aufwand gelöst worden. Eine deutlich bessere Variante ist das Automatisieren dieser Befehle unabhängig von dem lokalen Rechner auf virtuellen Maschinen in der Cloud. Als Plattform für diese Continuous Integration und Continuous Delivery (CI/CD) habe ich mich für GitHub Actions entschieden (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 man verfügt über einen oder mehrere Schritte (engl. Step). Die Schritte können wiederum beliebige Shell-Befehle, Skripte oder Aktionen ausführen.

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

Codeblock 2.10: JSON mit Konfiguration der Script

```
1 {  
2   "example-name": {  
3     "dirs": [  
4       "./example-project/Scripts/int_queries",  
5       "./example-project/Scripts/varchar_queries:length"  
6     ],  
7     "var": {"length": [1, 64]}  
8   }  
9 }
```

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

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

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

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

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

Codeblock 2.11: Ausschnitt aus der Workflow - Datei

```
1 name: Benchmark Workflow
2 on:
3   push:
4     paths:
5       - 'Projects/**'
6       - ...
7 jobs:
8   prepare-benchmark:
9     outputs:
10      matrix: ${ steps.set-matrix.outputs.matrix }
11      configurations: ${ steps.prepare-config.outputs.configurations }
12 steps:
```

```

13   - name: Checkout repository
14     uses: actions/checkout@v3
15   - name: Read and generate list of matrix name
16     run: # ... echo "matrix=$matrix" >> $GITHUB_OUTPUT
17   - name: Prepare configurations for all test types
18     run: # ... export variables like test_type, dirs, var, output_dir, artifact_name, should_run as "
        configurations"
19 run-tests:
20   needs: prepare-benchmark
21   strategy:
22     matrix:
23       test-type: ${ fromJson(needs.prepare-benchmark.outputs.matrix) }
24   env:
25     OTHER_ENVIRONMENT_VARIABLES
26   steps:
27     - name: Checkout repository
28       uses: actions/checkout@v3
29     - name: Extract and save values to GitHub environment
30       run: # ...
31     - name: Install dependencies
32       run: # install sysbench,pandas and matplotlib
33     - name: Start MySQL container
34       run: |
35       docker run --name mysql-${ env.test_type } -d -e MYSQL_ROOT_PASSWORD=$DB_PASS -e MYSQL_DATABASE=
        $DB_NAME -p $DB_PORT:3306 mysql:8.0
36       until docker exec mysql-${ env.test_type } mysqladmin --user=root --password=$DB_PASS --host=127.0.0.1
        --port=$DB_PORT ping --silent; do sleep 1; done
37       echo "MySQL is ready!"
38     - name: Run sysbench script
39       run: |
40       chmod +x Tools/Shell-Scripts/sysbench_script.sh
41       Tools/Shell-Scripts/sysbench_script.sh -out "${ env.output_dir }" -var '${ env.var }' -scripts:'${ env.dirs }'
42     - name: Upload individual outputs
43       uses: actions/upload-artifact@v4
44       with:
45         name: ${ env.artifact_name }
46         path: ${ env.output_dir }
47
48 upload-combined-output:
49   needs: [prepare-benchmark, run-tests]
50   steps:
51     - name: Loop through configurations, download artifacts with artifact_name and unzip it
52       run: # ... ALL_ARTIFACTS=$(curl -s -H "Authorization: Bearer ${ secrets.GITHUB_TOKEN }" "https://api.
        github.com/repos/${ github.repository }/actions/artifacts") ...
53     - name: Upload "Output" - folder as artifact
54       run: # ...

```

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

2.5 Optimierungen des Workflows

Zum einen kann man die zu installierenden Abhängigkeiten mithilfe des GitHub Caches (GitHub, 2025a) speichern. Dies bietet sich besonders an, da sich die Abhängigkeiten über die Workflows hinweg nur selten ändern. Falls sich doch etwas ändert, kann man beispielsweise die `requirements.txt`-Datei anpassen. Dadurch werden einmalig alle Abhängigkeiten neu installiert und anschließend im Cache abgelegt. Falls sich bis zum nächsten Workflow keine Änderungen an den Abhängigkeiten ergeben, wird der Cache automatisch genutzt. Der Zeitgewinn in unserem Beispiel ist jedoch nur gering und beträgt nur wenige Sekunden pro Workflow.

Codeblock 2.12: Speichern der Abhängigkeiten im Cache

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

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

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

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

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

Die letzte Frage lautet, wo die Ordner mit den berechneten Namen gespeichert und beim nächsten Run wieder heruntergeladen werden sollen. Zum einen kann man Lösungen in GitHub selbst verwenden. Zum einen würde sich eine GitHub Cache - Lösung wieder anbieten, aber tatsächlich sind GitHub Artifacts für das Sichern von Dateien besser geeignet (GitHub, 2025a). Eine andere interessante Lösung kann auch das Nutzen von expliziten Branches nur für die Sicherung der Dateien sein. Das Problem ist hier, dass es manchmal durch bestimmtes Timing zu Problemen beim Pushen kommen kann, da zufällig ein anderer paralleler Workflow in der Zeit zwischen Rebase, Commit und Push den Code verändert hat, wodurch nach verhindertem Push erneut ein Rebase durchgeführt werden muss. Außerdem muss man dafür der GitHub Action Schreibberechtigungen geben. Des Weiteren eignen sich auch Cloud - Speicherlösungen sehr gut, um die Ordner zu speichern und wieder herunterzuladen. Dazu gehören von Google Cloud Storage (GCS), AWS S3 oder MS Azure Storage, die sich zusammen mit GitHub Artifacts am besten eignen.

3 Optimierungen von Datentypen

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

3.1 Allgemeine Faktoren

Bei der Erstellung von Tabellen sollte man folgende Schritte für die Auswahl von Datentypen befolgen (Schwartz et al., [2012](#), pp. 115–145). Zunächst muss die allgemeine Klasse der Typen, wie beispielsweise numerisch, Zeichenketten oder zeitbezogen, festgelegt werden. Anschließend sollte der spezifische Typ ausgewählt werden. Für numerische Daten kommen beispielsweise Ganzzahlen wie INT oder Fließkommazahlen wie FLOAT und DOUBLE infrage. Die spezifischen Typen können dieselbe Art von Daten speichern, unterscheiden sich jedoch im Bereich der Werte, die sie speichern können. Auch sind sie unterschiedlich in der Precision (Genauigkeit), die sie erlauben und dem physischen Speicherplatz, den sie entweder auf der Festplatte oder im Arbeitsspeicher benötigen. Einige Datentypen haben auch spezielle Verhaltensweisen und Eigenschaften.

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

verarbeiten. Bei dem Beispiel mit Integer und Character liegt dies an den Character Sets und Sortierregeln, die den Character-Vergleich erschweren.

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

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

3.2 Einzelne Datentypen und weitere Faktoren

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

Für numerische Datentypen gibt es die Wahl zwischen Ganzzahlen und Fließkommazahlen. Die spezifischen Typen unterscheiden sich nur in der Anzahl der Bits, die sie speichern können. SMALLINT kann 16 Bits speichern, während INT 32 und BIGINT 64 Bits speichern kann. Dementsprechend verändert sich auch der mögliche Wertebereich der Zahlen, die durch den Speicherplatz abgedeckt sind. Mit den optionalen UNSIGNED - Attribute können keine negativen Werte gespeichert werden können, dafür verdoppelt sich aber die obere Grenze der Positiven. Zeitgleich bleiben der Speicherplatz und die Leistung gleich. Die Berechnung der Wertebereiche für den default, bzw. mit dem Signed-Attribut, erfolgt in 3.1 und mit dem Unsigned - Attribut in 3.2.

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

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

Hinweis: N entspricht der Anzahl der Bits.

Beispiel für 8 Bits:

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

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

Codeblock 3.1: SQL-Befehl zur Erstellung der Testtabelle

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

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

Codeblock 3.2: Inserts und Selects für Testtabelle aus 3.1

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

Tabelle 3.1: Ergebnis der SQL-Abfrage aus 3.2

| int_5 | int_11 |
|-------------|-------------|
| 2147483647 | 2147483647 |
| -2147483648 | -2147483648 |

Für den maximalen Wert von `INT` werden 32 Bits benötigt: $2^{(32-1)} - 1 = 2147483647$. `INT(5)` und `INT(11)` können beide die Grenzwerte von `INT` speichern, weshalb wir bestätigt haben,

dass die Breitenangabe keinen Einfluss auf die Speicheranforderungen hat, ansonsten hätten wir einen Fehler bei der Einfügung der Werte bekommen.

Ein spezifischer Typ für eine Festkommazahl ist `DECIMAL`, die auch für die Speicherung von Ganzzahlen geeignet ist. Außerdem kann man bei einer Festkommazahl auch die Genauigkeit angeben, da die maximale Anzahl der Ziffern vor und nach dem Dezimalpunkt definiert werden. `DECIMAL(18, 9)` beispielsweise speichert neun Ziffern vor und nach dem Dezimalpunkt und benötigt dafür 9 Bytes Speicherplatz. `DECIMAL` speichert Zahlen in einer binären Zeichenkette (binary string) mit neun Ziffern pro vier Bytes und unterstützt bis zu 65 Ziffern insgesamt.

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

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

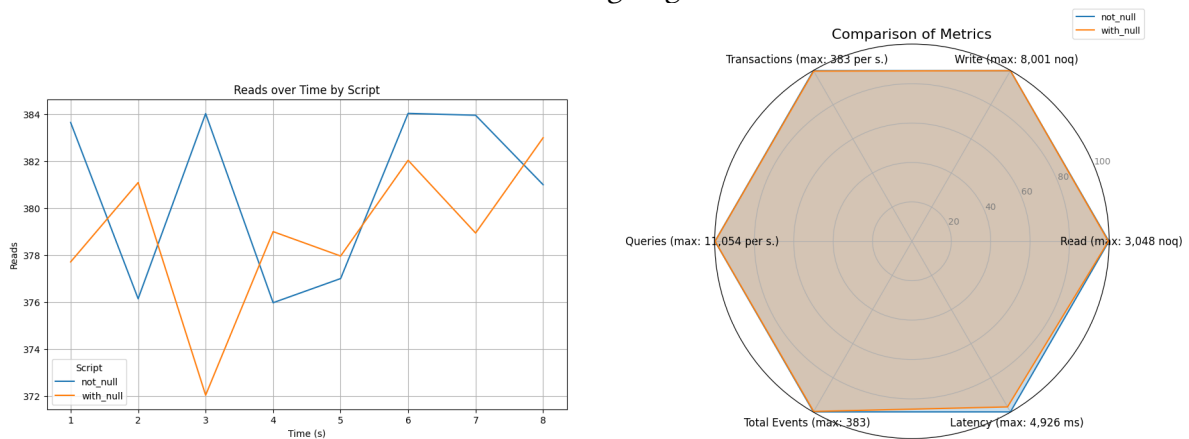
`CHAR` hingegen hat eine feste Länge und MySQL reserviert immer auch den nicht gebrauchten Platz für die angegebene Anzahl an Zeichen. Daher ist `CHAR` ideal für sehr kurze Strings oder Werte, die alle nahezu gleich lang sind, da `VARCHAR(1)` zwei Bytes aufgrund des Längen - Bytes benötigt, `CHAR(1)`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

`DATETIME` und `TIMESTAMP` können dieselbe Art von Daten speichern und beide haben dabei eine Genauigkeit von einer Sekunde. `TIMESTAMP` benötigt aber nur halb so viel Speicherplatz,

ist zeitzonenbewusst und verfügt über spezielle Auto-Update-Funktionen. Allerdings hat `TIMESTAMP` einen viel kleineren Bereich an erlaubten Werten und manchmal können seine speziellen Fähigkeiten ein Nachteil sein.

3.3 Analyse der Benchmarks

Der erste Leitsatz, den wir untersuchen, besagt, dass Spalten nach Möglichkeit als `NOT NULL` deklariert werden sollten. Für den Nachweis benutzen wir erneut die Kundentabelle (2.1). Diesmal erstellen wir eine Tabelle, bei der das Attribut `NOT NULL` für alle Spalten deklariert wird, sowie eine Tabelle ohne dieses Attribut. Wenn das Attribut nicht deklariert wird, dann können u.a. auch `NULL`-Werte in die Tabelle eingefügt werden.



(a) Readsabfragen für jeweils 2 Selects

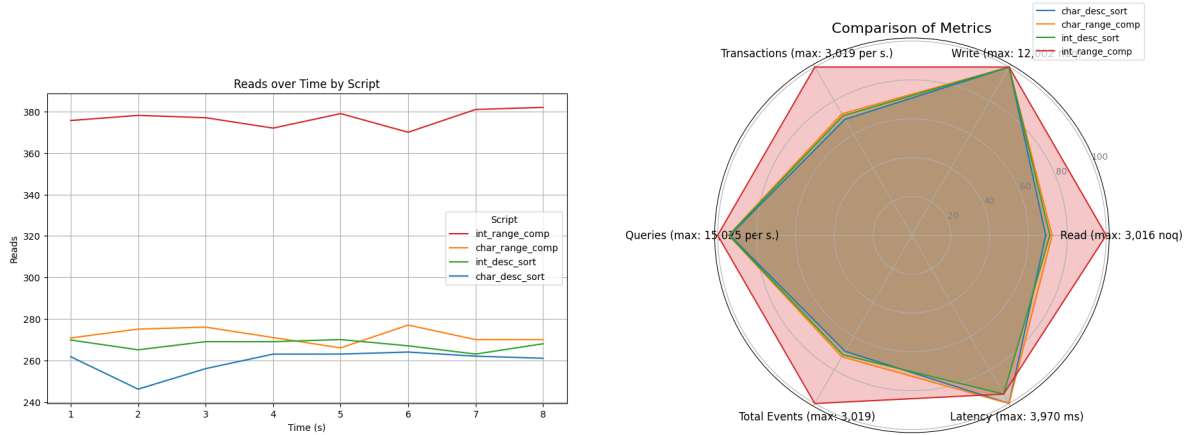
(b) Metriken über den kompletten Zeitraum

Abbildung 3.1: Vergleich von Null und Not Null

In der erstellten Grafik (siehe Abbildung 3.1a) sind die beiden Resultate der Select-Befehle zu sehen, die sowohl einfache WHERE-Klauseln als auch Count- und Group-By-Befehle enthalten. Anhand der Grafik lässt sich erkennen, dass die Werte für `NOT NULL` höher liegen als für `NULL`, bzw `WITH NULL`. Höhere Werte bedeuten, dass mehr Abfragen pro Sekunde durchgeführt werden können, was auf eine bessere Performance hindeutet. Deshalb lässt es sich sagen, dass `NOT NULL` besser performt als `NULL`, aber wenn man auf die y-Achse schaut, fällt auf, dass die Werte nicht so weit auseinanderliegen, sondern der Unterschied nur marginal ist. Daher sollten die Entscheidungen beim Datenbankentwurf nicht hauptsächlich aus Performancegründen getroffen werden, sondern vor allem aus Gründen der Datenintegrität und -konsistenz.

Um zu zeigen, dass man bei der Wahl zwischen unterschiedlichen Datentypen, den simpleren wählen sollte, haben wir erneut die Kundentabelle (2.1) benutzt. Für diesen Vergleich verändern wir jeweils den Datentyp des Schlüsselattributs der Tabelle. Zuerst erstellen wir eine

Kundentabelle mit einem INT-Primärschlüssel, danach eine mit CHAR. An den Ergebnissen im Hexagon 3.2b fällt auf, dass die Schreibbefehle in beiden Fällen etwa gleich schnell sind. Aber bei den Abfragen gibt es deutliche Unterschiede. Wenn man einen Wertebereich abfragt, dann ist INT deutlich schneller (50%) als CHAR. Bei der Sortierung hat INT ebenfalls einen Vorteil gegenüber CHAR, jedoch fallen die Abstände deutlich geringer aus (3.2a).



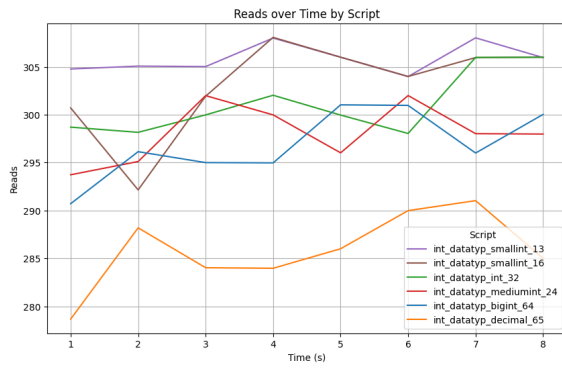
(a) Vergleich für jeweils 2 unters. Abfragen

(b) Kennzahlen über den gesamten Zeitraum

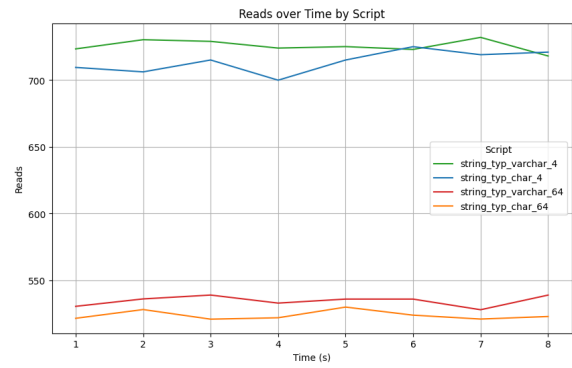
Abbildung 3.2: Vergleich von Int und Char

Als Letztes wollten wir unterschiedliche Datentypen vergleichen. Dazu verwenden wir die gleiche Tabelle wie beim Vergleich von INT und CHAR, setzen diesmal jedoch unterschiedliche numerische oder zeichenkettenbasierte Typen als Primärschlüssel ein. Beim Vergleich der numerischen Typen zeigt sich, dass DECIMAL mit deutlichem Abstand am langsamsten ist (Abbildung 3.3a). Danach folgt, wie vermutet, der nächstgrößere Datentyp BIGINT. Auffällig ist, dass der Unterschied zwischen INT, MEDIUMINT und SMALLINT kleiner ist als erwartet. Dies wird vermutlich aber daran liegen, dass wir die Abfragen nur auf eine Tabelle mit wenigen tausend Datensätzen ausgeführt haben. In der Praxis mit Hunderttausenden oder Millionen von Datensätzen ist anzunehmen, dass die Unterschiede zwischen den Typen größer wären als in unserem Vergleich.

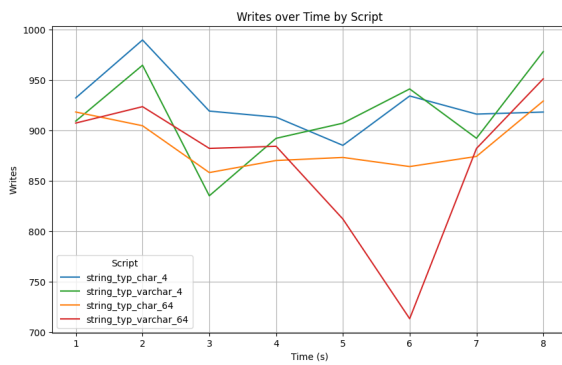
Beim Vergleich zwischen den beiden Zeichenketten-Typen CHAR und VARCHAR ist unabhängig von der Länge zu erkennen, dass VARCHAR effizienter ist als CHAR (3.3b). Im ersten Vergleich wurde jeweils eine Länge von 4 Stellen verwendet und beim zweiten Vergleich eine Länge von 64 Stellen. Bei beiden untersuchten Längen ist VARCHAR schneller als CHAR. Wenn man sich jedoch die Performance beim Einfügen von Werten anschaut, fällt auf, dass die Unterschiede eher gering sind und es gibt auch stärkere Schwankungen bei den Werten (siehe Abbildung 3.3c).



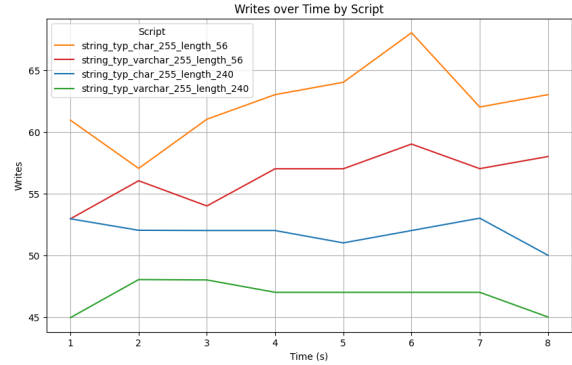
(a) Unterschiedliche numerische Typen



(b) Unterschiedliche Zeichenketten-Typen



(c) Bei gleicher Länge der Zeichen



(d) Bei unterschiedlichem Befüllungsgrad

Abbildung 3.3: Vergleich von unters. numerischen und zeichenkettenbasierten Typen

Als letzten Vergleich haben beide Zeichenketten-Typen mit der Länge von 255 Stellen definiert, aber dafür mit unterschiedlich vielen Stellen befüllt. Anschließend haben wir bei beiden Tabellen die Werte aktualisiert und wenige Stellen bei der Namen-Spalte zufällig hinzugefügt. Wenn man dies tut, dann fällt auf, dass CHAR schneller ist als VARCHAR (3.3d). Zusätzlich wird der Unterschied zwischen den beiden Typen größer, je mehr Stellen befüllt werden. Damit haben wir gezeigt, dass die Vorteile von CHAR insbesondere bei der Aktualisierung von Werten liegen, während VARCHAR bei der Selektion von Werten besser abschneidet. Der Grund hierfür ist, dass CHAR die verbleibenden Stellen mit Leerzeichen auffüllt, was zu einem höheren Speicherbedarf führt.

4 Indexierung

Das folgende Kapitel befasst sich mit der Indexierung und den damit verbundenen Performance-Optimierungen, die näher erläutert werden. Zunächst betrachten wir einige Grundlagen der Indexierung. Anschließend lernen wir die verschiedenen Arten von Indizes näher kennen und führen unterschiedliche Benchmarks mit Ihnen durch. Im letzten Schritt analysieren wir die Ergebnisse und stellen fest, welche Verwendung der Indizes am besten funktioniert.

4.1 Grundlagen

Indizes sind Datenstrukturen, die von Speicher-Engines (engl. storage engines) verwendet werden, um unter anderem Zeilen schneller zu finden (Schwartz et al., 2012, pp. 147–189). 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 wir verschiedene Indextypen kennenlernen, die nicht von allen Engines unterstützt werden.

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

Um die Funktionsweise eines Indexes anschaulicher zu erklären, betrachten wir als Beispiel ein wissenschaftliches Fachbuch. 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. Betrachten wir dazu folgendes Beispiel:

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

Wenn wir annehmen, dass es einen Index auf der Spalte KUNDEN_ID gibt, dann nutzt MySQL diesen, um Zeilen zu finden, deren KUNDEN_ID gleich 7 ist. Mit anderen Worten wird eine Suche innerhalb der Indexwerte durchgeführt und alle entsprechenden Zeilen werden zurückgegeben.

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

Um zu verstehen, wie man Indizes für eine Datenbank auswählt, ist es wichtig zu wissen, welcher Teil der Abfrage am meisten Zeit in Anspruch nimmt (Garcia-Molina, 2008, pp. 350–353). Das Datenbanksystem ist so aufgebaut, dass die Tupel einer Relation üblicherweise auf viele Seiten einer Festplatte verteilt sind, die jeweils mehrere Tausend Bytes umfassen und viele Tupel speichern. Um die Werte eines Tupels zu prüfen, muss die gesamte Seite, auch Block genannt, in den Hauptspeicher geladen werden. Dabei kostet es kaum mehr Zeit, alle Tupel einer Seite anstatt nur ein einzelnes zu prüfen.

In der Regel stellt der Schlüssel den sinnvollsten Index für eine Tabelle dar, weshalb MySQL standardmäßig den B-Tree-Index für Primary Keys verwendet (Oracle, 2025a). Die Entscheidung, ob für ein bestimmtes Attribut ein Index definiert werden soll, hängt von drei Faktoren ab: Erstens ist ein Index besonders nützlich, wenn Abfragen häufig auf ein bestimmtes Attribut zugreifen. Zweitens kann ein Index sinnvoll sein, wenn es nur wenige Tupel für einen bestimmten Wert des Attributs gibt, da dies den Festplattenzugriff bei einer Abfrage reduziert. Sobald nicht alle Blöcke geladen werden müssen, kann der Index Zeit sparen. Der letzte Fall betrifft Situationen, in denen Tupel nach einem Attribut geclustert sind. Durch einen Index können hier weniger Datenblöcke geladen werden, da die Werte des Attributs aufeinanderfolgender gespeichert sind. Mit diesen Faktoren können wir begründen, warum die Schlüssel einer Tabelle gut geeignet sind. Zum einen kommen sie oft in Abfragen vor (erster Punkt) und zum anderen enthalten sie keine doppelten Werte, da jedes Tupel einen eindeutigen Wert hat (zweiter Punkt).

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

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

Um die Vorgehensweise anhand einer beispielhaften Berechnung durchzuführen, benutzen wir die folgende Tabelle (abgeändertes Beispiel aus Garcia-Molina, 2008, pp. 355–357):

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

Der Schlüssel der Faktentabelle ist die Spalte Id und für die Attribute Artikel_Id und Kunden_Id erstellen wir jeweils einen Index. Damit haben wir inklusive des Primärschlüssels 3 unterschiedliche Indexe. Als Nächstes brauchen wir Befehle, bei denen die Indexe benutzt werden (siehe 4.1). In der ersten Zeile wird nur der Kundenindex verwendet, in der zweiten nur der Artikelindex und in der Letzten fügen wie eine Zeile ein.

Codeblock 4.1: Select-Queries für die Faktentabelle

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

Damit wir berechnen können, ob es sinnvoll ist, die Indizes zu erstellen, müssen wir bestimmte Voraussetzungen festlegen. Zuerst gehen wir davon aus, dass die Faktentabelle 10 Datenblöcke belegt und im Durchschnitt kauft jeder Kunde 3 Artikel und ein Artikel wird von 3 Kunden gekauft. Die Tupel für einen bestimmten Kunden oder Artikel sind gleichmäßig über die 10 Seiten verteilt. Trotzdem sind mit einem Index nur 3 Festplattenzugriffe erforderlich, um die durchschnittlich 3 Tupel für einen Kunden oder Artikel zu finden. Dazu ist ein Festplattenzugriff erforderlich, um die Seite des Indexes zu lesen und ein weiterer, um die modifizierte Seite zurückzuschreiben, falls eine Indexseite geändert werden muss. Haben wir keinen Index, sind 10 Festplattenzugriffe zum Lesen und zwei Festplattenzugriffe für Schreiben erforderlich. Mit diesen Bedingungen kommen wir zu folgender Kostentabelle:

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

Tabelle 4.1: Kosten der unterschiedlichen Queries in Abhängigkeit der Indizes

Die letzte Zeile aus Tabelle 4.1 gibt die durchschnittlichen Kosten einer Aktion an. Unter der Annahme, dass der Anteil der Zeit, in der wir die erste Abfrage ausführen, p_1 beträgt und der Anteil der Zeit für die zweite Query p_2 ist. Da die gesamte Zeit in einem System 100% ausmacht, beträgt der Anteil der Zeit, in der wir I ausführen: $1 - p_1 - p_2$. Den Durchschnitt für den Kundenindex berechnen wir wie folgt:

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

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 wir keine Indizes bevorzugen würden. Wenn jedoch $p_1 = 0,5$ und $p_2 = 0,1$ gelten, ergibt ein Index für die Kunden den besten Durchschnittswert.

Damit haben wir gezeigt, dass es sinnvoll ist, keinen Index zu verwenden, wenn überwiegend Einfügungen durchgeführt werden und nur sehr wenige Abfragen anfallen. Intuitiv, wenn wir viele Abfragen durchführen und die Anzahl der Abfragen, die Artikel und Kunden angeben, ungefähr gleich häufig sind, dann sind beide Indizes erwünscht. Wenn wir nur ein Typ von Query häufig verwenden, dann sollten wir nur den Index definieren, der uns bei dieser hilft.

Es gibt zahlreiche Tools, die entwickelt wurden, um die Verantwortung der Wahl der Indizes vom Datenbankdesigner zu übernehmen. Dabei optimiert das System sich selbst oder dem Designer werden zumindest Empfehlungen für sinnvolle Entscheidungen gegeben. Ein bewährter Ansatz zur Auswahl von Indizes ist das sogenannte Greedy-Verfahren (Matzer, 2019), bei dem zunächst ohne ausgewählte Indizes der Nutzen jedes Kandidaten-Index bewertet wird. Wenn es einen Index mit positivem Nutzen gibt, wird dieser ausgewählt und anschließend wird eine Neubewertung ausgeführt, wobei davon ausgegangen wird, dass der zuvor ausgewählte Index bereits verfügbar ist. Dieser Prozess wird so lange wiederholt, bis es keinen Kandidaten-Index mit positivem Nutzen mehr gibt.

4.2 B-Baum-Index

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

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

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

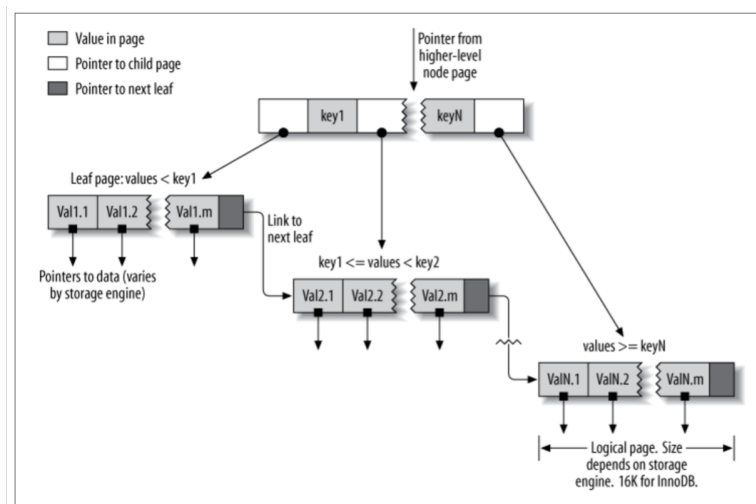


Abbildung 4.1: Binär-Baums-Darstellung (Abbildung 5–1 aus Schwartz et al., 2012, S. 149)

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

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

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

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

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

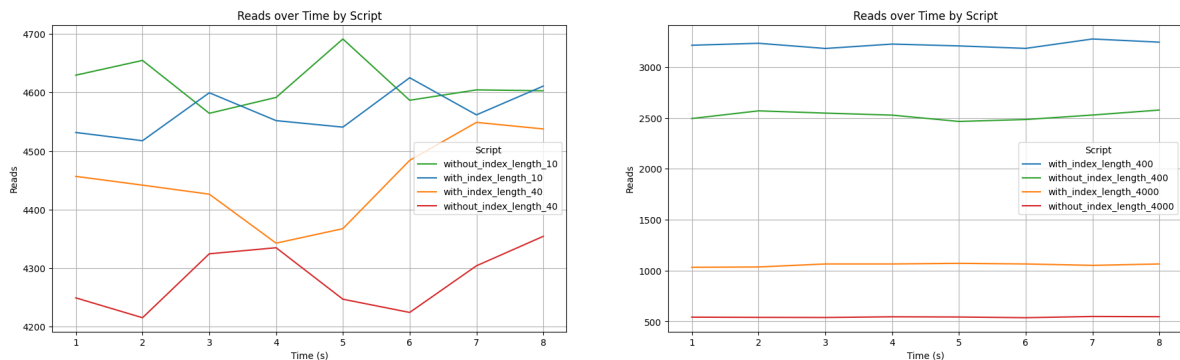
Es gibt jedoch Einschränkungen von B-Baum-Indizes, die dazu führen, dass andere Indextypen für bestimmte Szenarien besser geeignet sind. Eine Einschränkung ist, dass die Suche nicht am rechten Ende des Indexes beginnen kann. Beispielsweise ist der Beispiels-Index nicht dazu geeignet, alle Personen zu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

Im nächsten Abschnitt führen wir die Benchmarks durch, um das Verständnis für die Funktionsweise des B-Baum-Indexes zu bestätigen. Dafür erstellen wir zunächst wieder die Kundentabelle (2.1) und definieren für den ersten Vergleich folgende Indizes:

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, vergleichen wir diese Konfiguration mit einer, bei der nur die Kundentabelle ohne Indizes erstellt wird. In beiden Fällen werden eine bestimmte Anzahl an Datensätzen eingefügt. Um die Performance der Select-Abfragen zu messen, führen wir verschiedene Queries an die Datenbank aus, 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, vergleichen wir einmal 10 Datensätze mit 40 und einmal 400 mit 4000 Zeilen.



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

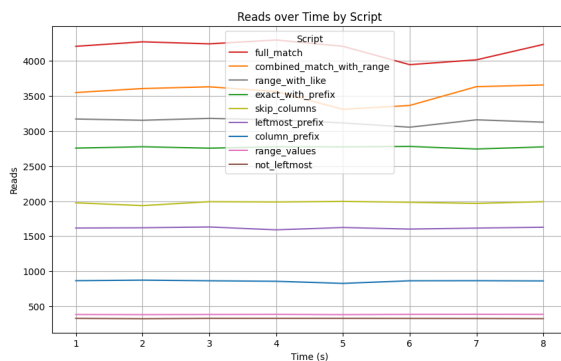
Codeblock 4.4: Unterschiedliche Where-Bedingungen für B-Tree-Index

```
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
```

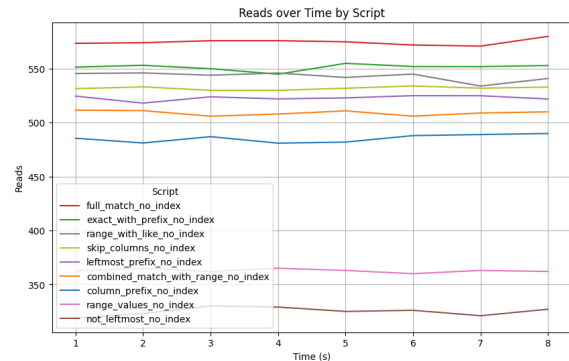
Mit dem vorherigen Benchmark können wir die Vorteile eines Indexes schon deutlich erkennen. Jetzt wollen wir aber auch die Funktionalität des B-Tree-Indexes in Bezug auf

unterschiedliche Selects untersuchen. Dazu erstellen wir erneut die Kundentabelle, aber erstellen dieses Mal nur einen Index (siehe 4.2). Anschließend befüllen wir die Tabelle mit einer festgelegten Anzahl an Datensätzen und führen unterschiedliche Select-Befehle aus (4.4).

Anhand der Grafik in Abbildung 4.3 lässt sich erkennen, bei welchen Abfragen der Index am effizientesten ist. Es lässt sich erkennen, dass `full_match` die höchsten Werte aufweist. Es folgt `combined_match_with_range`, bei dem die ersten beiden Argumente wie beim `full_match` exakt bestimmt sind, während das dritte Argument, das Geburtsdatum, eine größere Flexibilität bietet, da hier alle Kunden berücksichtigt werden, die vor dem 01. Januar 1980 geboren wurden. Im Anschluss kommt `range_with_like`, bei dem zusätzlich zum Geburtsdatum auch der Vorname mit „M“ beginnen muss, jedoch nicht exakt „Max“ sein muss. Danach folgt `exact_with_prefix`, das dem vorherigen Fall ähnlich ist, jedoch zusätzlich eine aufsteigende Sortierung nach dem Geburtsdatum aufweist. Dies zeigt, dass der B-Tree-Index auch für das Sortieren von Spalten genutzt werden kann. Es folgt `skip_columns`, bei dem der Vorname vollständig ignoriert wird. Dadurch werden mehr Zeilen zurückgegeben, was die Performance negativ beeinflusst. Anschließend kommt `leftmost_prefix`, bei dem nur nach dem Namen gefiltert wird. Bei `column_prefix`, das direkt darauf folgt, wird statt des Namens „Müller“ jeder Nachname zurückgegeben, der mit „M“ beginnt. Schließlich zeigt `range_values` alle Namen zwischen „Müller“ und „Schulz“, während bei `not_leftmost` ausschließlich nach dem Geburtsdatum gefiltert wird, ohne weitere Kriterien. Insgesamt verdeutlicht diese Analyse die entscheidende Rolle des B-Tree-Index, da die Antwortzeiten je nach Art der Abfrage erheblich variieren können.



(a) Mit Index



(b) Ohne Index

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

Diese unterschiedlichen Zeiten hängen aber nicht nur davon ab, ob der Index verwendet werden kann, sondern auch wie viele Tupel jeweils zurückgegeben werden. Wir können sehen, dass diejenigen Abfragen, die am wenigsten Zeilen zurückgeben, sind auch diejenigen sind, bei denen es die höchste Performance gibt. Aber mithilfe des EXPLAIN-Operators können

wir tatsächlich herausfinden, ob der definierte Index aus 4.2 greift oder nicht. Die Analyse von EXPLAIN zeigt, dass `range_values` und `not_leftmost` die einzigen sind, bei denen der Index nicht verwendet wird und beide sind auch die Langsamsten im Vergleich gewesen.

| Select-Query | Anzahl an Zeilen | Index benutzt? |
|---------------------------|------------------|----------------|
| full_match | 0 | ja |
| combined_match_with_range | 11 | ja |
| range_with_like | 39 | ja |
| exact_with_prefix | 65 | ja |
| skip_columns | 144 | ja |
| leftmost_prefix | 250 | ja |
| column_prefix | 509 | ja |
| range_values | 1313 | nein |
| not_leftmost | 2420 | nein |

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

4.3 Hash-Index

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

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

Wenn mehrere Werte denselben Hash-Wert besitzen, speichert der Index die Zeiger auf die Zeilen (engl. row pointers) in demselben Hash-Tabelleneintrag, typischerweise mithilfe einer verketteten Liste (z.B. einer *Linked List*). Hash-Kollisionen können die Leistung eines Hash-Index beeinträchtigen, da jeder Zeiger in der verketteten Liste durchlaufen und die entsprechenden Werte mit dem Suchwert verglichen werden müssen, um die richtigen Zeilen zu finden. Das ist auch Index-Wartungsoperationen mit viel Aufwand verbunden. Hingegen eindeutige Hash-Indizes stellen sicher, dass für jeden Hash-Wert nur ein einziger Eintrag existiert. Bei Konflikten wird ein Mechanismus wie die Open Addressing-Strategie (z.B. Linear Probing oder Quadratic Probing) eingesetzt, um Konflikte zu lösen und den Speicherplatz effizient zu verwalten. Hierbei wird versucht, Konflikte direkt innerhalb der Hash-Tabelle zu

bewältigen, anstatt auf zusätzliche Datenstrukturen wie verkettete Listen zurückzugreifen. Die eindeutigen Hash-Indizes werden nicht von der Memory-Engine in MySQL unterstützt.

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

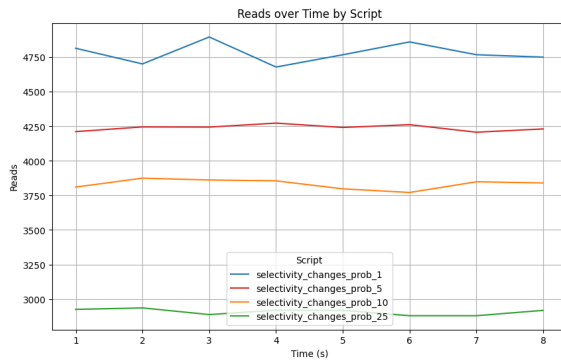
```
1 SELECT * FROM KUNDEN WHERE NAME = 'Peter';
```

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

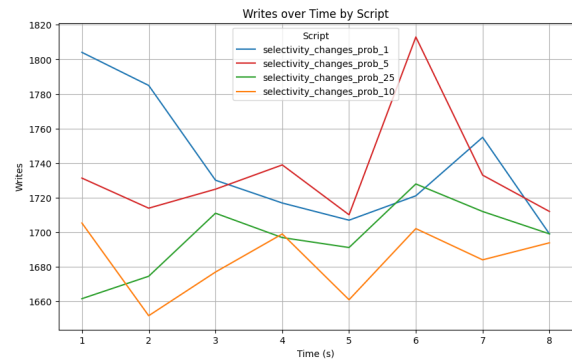
Ähnlich wie der B-Baum-Index hat auch der Hash-Index einige Einschränkungen. Zum einen enthält der Index nur Hash-Werte und Zeiger auf Zeilen (engl. row pointers), jedoch nicht die Werte selbst. Deshalb kann MySQL den Index nicht verwenden, um das Einlesen der Zeilen zu vermeiden. Allerdings erfolgt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

Als Nächstes kommen wir zu den Benchmarks mit Hash-Indizes. Dazu verwenden wir erneut die Kundentabelle und erstellen nur einen Index für die Spalte NAME. Am Ende des CREATE INDEX-Befehl müssen wir USING HASH hinzufügen, damit anstelle des standardmäßigen B-Tree-Index der Hash-Index verwendet wird. Danach befüllen wieder die Tabelle mit Testdaten.

Diesmal untersuchen wir beim ersten Benchmark den Einfluss von Hash-Kollisionen für die Performance. Um den Grad der Kollisionen zu verändern haben wir eine Variable, die die obere Grenze für die zufällige Generierung einer Zahl, die an dem Namen herangehängt wird, darstellt. Anschließend fragen wir alle Zeilen mit dem Wert Kunde_1 für die Spalte NAME ab. Wir führen die Tests mit den Kollisionswahrscheinlichkeiten von 25%, 10%, 5% und 1% durch.



(a) Unterschiede von Readsabfragen



(b) Unterschiede von Schreibbefehlen

Abbildung 4.4: Vergleich der Auswirkungen von Hashkollisionen

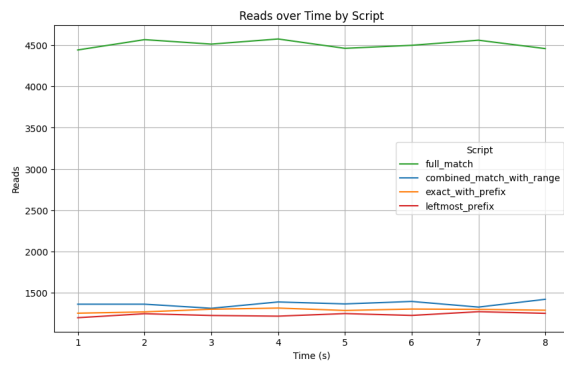
An den Ergebnissen in Abbildung 4.4 können wir sehen, dass je geringer die Wahrscheinlichkeit für eine Kollision ist, desto schneller ist die Select-Abfrage. Es fällt auch auf, dass die Unterschiede zwischen den verschiedenen Kollisionswahrscheinlichkeiten sehr groß sind. Hingegen die Einfüge-Performance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

Als zweiten Test wollen wir wieder überprüfen, ob der Index bei bestimmten Select-Queries benutzt wird oder nicht. Wie erwartbar benutzen wir wieder unsere Kundentabelle und erstellen den gleichen Index wie in Beispiel ?? . Anschließend fügen wir wieder die Testdaten ein und verwenden wieder die Select-Befehle aus 4.4. Dieses Mal benutzen wir aber nicht alle Select-Befehle, sondern nur die folgenden:

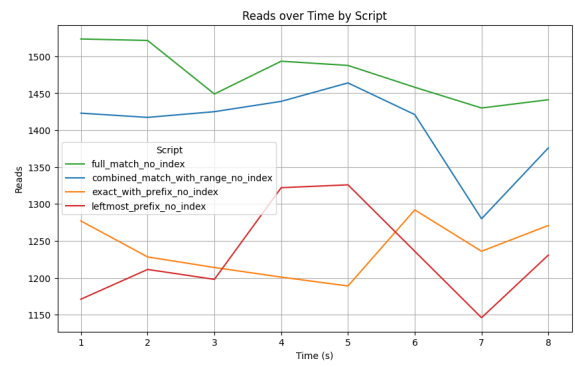
| Select-Query | Anzahl an Zeilen | Index benutzt? |
|---------------------------|------------------|----------------|
| full_match | 0 | ja |
| combined_match_with_range | 4 | nein |
| exact_with_prefix | 41 | nein |
| leftmost_prefix | 201 | nein |

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

Wie schon aus der Tabelle 4.3 ersichtlich wird, wir nur bei full_match der Index verwendet. Diese Erkenntnis können wir auch aus dem Result des Benchmarks (siehe ??) erkennen. Denn die Query ist um ein Erhebliches schneller als alle anderen (etwa um Faktor 15). Alle anderen Queries sind sehr ähnlich, was unter anderem auch der Skalierung geschuldet ist. Obwohl combined_match_with_range nur 4 Zeilen zurückgibt und leftmost_prefix hingegen 201 Datensätze, sind die Unterschiede nur sehr klein.



(a) Mit Index



(b) Ohne Index

Abbildung 4.5: Grafik visualisiert unterschiedlichen Select-Queries mit und ohne Index

5 Views

Im folgenden Kapitel betrachten wir die Performance Vorteile von Sichten (engl. Views) in SQL. Zunächst befassen wir uns mit virtuellen Sichten, ihren Vor- und Nachteilen sowie dem Verhalten bei Inserts. Danach widmen wir uns materialisierten Sichten, die physisch in der Datenbank gespeichert werden, und deren Implementierung. Dabei erklären und nutzen wir Trigger, da MySQL keine native Unterstützung für materialisierte Sichten bietet. In den letzten beiden Kapiteln betrachten wir die Durchführung der Benchmarks näher und interpretieren die entstandenen Ergebnisse.

5.1 Virtuelle Views

Grundlegend existieren Relationen, bzw. Tabellen, die durch das `CREATE TABLE` – Statement definiert werden, physisch in der Datenbank. Damit sind sie persistent, was bedeutet, dass sie dauerhaft existieren und sich nicht ändern, es sei denn, sie werden explizit durch eine SQL-Änderungsanweisung dazu aufgefordert. Dies entspricht der Dauerhaftigkeit des ACID-Prinzips Luber, 2018, die sicherstellt, dass bestätigte Transaktionen dauerhaft gespeichert bleiben und auch bei Systemausfällen nicht verloren gehen. Es gibt jedoch eine weitere Klasse von SQL-Relationen, die nicht wie Tabellen physisch gespeichert werden (Garcia-Molina, 2008, S. 341–349, 353–366, Schwartz et al., 2012, pp. 276–281). Sie werden als virtuelle Sichten bezeichnet. Virtuelle Sichten werden durch einen Ausdruck definiert, der einer Abfrage ähnelt und können auch so abgefragt werden, als ob sie tatsächlich physisch existierten. In einigen Fällen kann man sogar die Datensätze über die Sicht geändert werden.

Codeblock 5.1: Allgemeine View-Deklaration

```
1 CREATE VIEW <name> AS <definition>;
```

In dem Codeblock 5.1 sehen wir die Struktur der Definition einer View. Als Nächstes müssen wir die view-definition mit einer SQL-Abfrage ersetzen, die den Inhalt der virtuellen Sicht abbilden soll. Um dieses Vorgehen mit einem Beispiel näher zu veranschaulichen, nutzen wir die Tabellen Kunden (2.1) und Bestellung (2.2). Nun wollen wir, dass die beiden Tabellen über die KUNDEN_ID in der Sicht zusammengefügt werden, da die KUNDEN_ID zum einen der

Primärschlüssel der Kundentabelle ist und zum anderen der Fremdschlüssel in Bestellung. Um in die SQL-Abfrage noch etwas mehr Komplexität zu bekommen, wollen wir neben der Join-Operation zusätzlich den Umsatz pro Jahr und pro Stadt aggregieren. Diese Aggregation könnte beispielsweise von einem Marketingteam genutzt werden, um schwache Regionen zu identifizieren und gezielt in diesen nachzusteuern. Die View mit dem Namen KUNDEN_OVERVIEW hat folgende Struktur:

Codeblock 5.2: View 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;
```

Wenn wir die Daten dieser virtuellen Sicht abfragen wollen, dann adressieren wir den Namen in der FROM-Klausel und verlassen uns darauf, dass das Datenbankmanagementsystem die benötigten Tupel erzielt (siehe 5.3). Dabei operiert das DBMS direkt auf den Relationen, die die virtuelle Sicht definieren. In unserem Fall sind das die Kunden- und Bestelltabelle.

Codeblock 5.3: SQL-Befehl mit Sicht

```
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. Die Unterabfrage müssen wir noch mit einer Tupelvariablen ergänzen, damit wir auch Bezug auf die Tupel nehmen können. Die SQL-Abfrage aus 5.4 liefert das gleiche Ergebnis wie die aus 5.3, wenn wir unsere View wie im Beispiel 5.2 definieren. Zu dem Einfluss auf die Performance kommen wir in den späteren Unterkapiteln [Durchführung der Benchmarks](#) und [Analyse der Ergebnisse](#).

Codeblock 5.4: Select-Befehl ohne Sicht

```
1 SELECT YEAR (B.BESTELLDATUM) AS Jahr, K.STADT, 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.STADT
5 ORDER BY Jahr DESC, Gesamtumsatz DESC;
```

Man kann den Attributen einer Sicht 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 in unserem 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 (DataScientest, 2023). Die Auslösung eines Triggers kann entweder vor (BEFORE) oder nach (AFTER) einem bestimmten Ereignis erfolgen, wie INSERT, UPDATE oder DELETE. Bei Triggern auf Sichten können auch INSTEAD-OF-Trigger verwendet werden, die Änderungsversuche an der Sicht abfangen und stattdessen eine frei definierbare Aktion ausführen.

Codeblock 5.5: Allgemeine Trigger Deklaration

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

Das Problem in MySQL mit Triggern ist aber, dass sie nur auf Tabellen angewendet werden können. Später werden wir dazu im Kapitel 5.3 noch ein genaueres Beispiel betrachten. Es

bietet sich aber ein weiteres Konzept mit den Stored Procedures an, um Werte in die virtuelle Sicht einzufügen. Stored Procedures sind Funktionen, die direkt im DB-Server hinterlegt werden und wie andere integrierte Funktionen, wie z.B. round(), aufgerufen werden können.

Codeblock 5.6: Allgemeine Prozedur Deklaration

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

Damit wir für die Sicht aus unserem Beispiel (5.2) Daten einfügen können, muss die Prozedur die gleichen Parameter, wie die Spalten der View, bekommen. 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 Stadt 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.STADT = Stadt LIMIT 1), Umsatz);
6 END;
```

Jetzt können wir die Methode insert_view einfach mit dem CALL-Befehl aufrufen und die Werte für die drei Parameter in Klammern übergeben. Dadurch werden die Werte in die Bestelltabelle eingefügt. Als Bestelldatum wird stets der erste Tag des Jahres verwendet und als Kunde wird einer gewählt, der in der jeweiligen Stadt lebt.

Im Vergleich zum direkten Einfügen in die Bestelltabelle verlieren wir jedoch an Datenpräzision. Einerseits haben wir nicht das genaue Datum und andererseits fehlt die Information zur ARTIKEL_ID. Zusammengefasst lässt sich sagen, dass je nach Definition der Sicht Daten entweder direkt eingefügt oder mithilfe von Stored Procedures befüllt werden können. Es ist dabei jedoch nicht ausgeschlossen, dass in den zugrunde liegenden Tabellen zu geringerer Datenqualität kommen kann, da zum Beispiel NULL-Werte oder andere Standardwerte verwendet werden.

5.2 Materialisierte Views

Allgemein sind Sichten so definiert, dass sie eine neue Relation aus Basistabellen durch Ausführen einer Abfrage auf diesen Tabellen erstellen. Bisher haben wir Sichten ausschließlich als logische Beschreibungen von Relationen betrachtet. In bestimmten Fällen kann es jedoch aus Performancegründen sinnvoll sein, sie zu materialisieren, also die Ergebnisse physisch zu speichern. Durch die physische Speicherung verringert sich der Rechenaufwand für Abfragen, da beispielsweise in unserem Fall (siehe 5.2) der Join nicht erneut ausgeführt werden muss. Die bereits gespeicherten Ergebnisse sind direkt abrufbar. Dies führt zu einer schnelleren Antwortzeit der Query, bringt jedoch einen höheren Pflegeaufwand mit sich. Passend zu unserer virtuellen Sicht (5.2) sieht die Materialisierte wie folgt aus:

Codeblock 5.8: Materialized View

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

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

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

Die inkrementelle Aktualisierung der materialisierten Sicht ist deutlich effizienter als die ständige Neuberechnung der Sicht. Nicht jedes Datenbankmanagementsystem bietet die inkrementelle Auffrischung an. Oracle unterstützt die inkrementelle Auffrischung nativ

mit Materialized View Logs. In PostgreSQL ist eine manuelle Planung erforderlich, da eine automatische inkrementelle Aktualisierung nicht unterstützt wird (Ouko, 2025). Wenn man die Option CONCURRENTLY beim Aktualisieren einer materialisierten Sicht verwendet, können andere Prozesse weiterhin darauf zugreifen, da die bestehende Sicht erst ersetzt wird, wenn die neue Version fertiggestellt ist (siehe 5.13).

MySQL bietet gar nicht erst eine Möglichkeit an, um materialisierte Sichten nativ zu erstellen. Allerdings kann man die Funktionsweise mithilfe einer physischen Tabelle als Sicht und mit Triggern auf die zugrundeliegenden Tabellen nachstellen, die bei Aktualisierungen Änderungen an der Sichttabelle durchführen.

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

Eine materialisierte Sicht kann wie eine virtuelle Sicht in der FROM-Klausel einer Abfrage verwendet werden. In Oracle gibt es zusätzlich noch eine spezielle 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 ENABLE QUERY REWRITE für das 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 5.9: Select mit View

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


In Oracle könnte die Abfrage 5.9 intern so umgeschrieben werden, dass stattdessen die Materialized View, die die aggregierten Umsätze enthält, genutzt wird. Anders sieht dies in PostgreSQL aus, da dort diese automatische Abfrageumschreibung nicht existiert. Bei der zweiten Abfrage 5.10 wird die materialisierte View nicht verwendet, da sie nicht die Spalten Land und Monat enthält. Deshalb werden in diesem Fall die zugrunde liegenden Tabellen direkt abgefragt und die Abfrage muss explizit auf diese Tabellen zugreifen, um die gewünschten Daten zu erhalten.

Codeblock 5.10: Select nicht für View

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

Zusammengefasst lässt es sich sagen, dass die Auswahl von materialisierten Sichten deutlich komplexer ist als die von Indizes, da potenziell jede Abfrage eine Sicht definieren könnte. Damit gibt es potenziell deutlich mehr mögliche Sichten als Indexes. Es sollten aber nur Sichten erstellt werden, die mindestens eine Abfrage der erwarteten Workload verbessern, wobei Kriterien wie Relationen, Bedingungen und Attribute berücksichtigt werden. Zudem muss der Nutzen einer Sicht nicht nur anhand der Laufzeitverbesserung, sondern auch im Verhältnis zu ihrem Speicherbedarf bewertet werden, da materialisierte Sichten oft nicht nur erheblich mehr Speicherplatz beanspruchen können, sondern 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. Da MySQL nativ keine materialisierten Sichten unterstützt, verfolgen wir einen alternativen Ansatz und vergleichen diesen später mit der nativen Implementierung in PostgreSQL, einem anderen Datenbankmanagementsystem.

Zuallererst beginnen wir mit der Umsetzung der virtuellen Sicht, für die wir, wie bei den anderen Sichten auch, zunächst die Basistabellen erstellen müssen. Als Basis verwenden wir die Tabellen Kunden (2.1) und Bestellung (2.2) und erstellen die View (5.2), die wir schon in Kapitel 5.1 beschrieben haben. Es werden Testdaten für die Kunden erstellt, jedem Kunden eine festgelegte Anzahl an Bestellungen zugewiesen und beide in die jeweiligen Tabellen eingefügt. Wir fügen keine Datensätze über die virtuelle Sicht ein und sprechen die Sicht bei Select-Befehlen explizit an. Da wir die Unterschiede in der Lesegeschwindigkeit möglichst repräsentativ feststellen wollen, betrachten wir mehrere Select-Befehle auf die unterschiedlichen Spalten der virtuellen Sicht.

Codeblock 5.11: 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 sehen, führen wir deshalb einen Benchmark mithilfe der Sicht durch und bei dem anderen deklarieren wir keine Sicht und wandeln alle Befehle auf die Sicht direkt in SQL-Befehle auf die unterliegenden Tabellen um.

Als Nächstes befassen wir uns mit der materialisierten Sicht. Da MySQL, wie bereits erwähnt, nicht nativ materialisierten Views unterstützt, erstellen wir zusätzlich zur Kundentabelle eine weitere physische Tabelle namens KUNDEN_MAT_OVERVIEW. Die Tabelle KUNDEN_MAT_OVERVIEW besteht, wie die virtuelle Sicht auch, aus den Spalten JAHR, LAND und GESAMTUMSATZ, wobei die Kombination aus Jahr und Land der Schlüssel der Tabelle ist. Als Nächstes erstellen wir die Tabellen KUNDEN, BESTELLUNG und KUNDEN_MAT_OVERVIEW und befüllen weiterhin nur die ersten beiden Tabellen mit Testdaten. Nach dem Einfügen der Testdaten wollen wir wieder die Select-Performance überprüfen und ersetzen bei unseren Select-Befehlen (5.11) die virtuellen Sicht KUNDEN_OVERVIEW mit der Tabelle KUNDEN_MAT_OVERVIEW. Wenn wir die Ergebnisse der Select-Befehle betrachten, dann fällt auf, dass die Tabelle KUNDEN_MAT_OVERVIEW keine Einträge hat. Das Problem beim bisherigen Verfahren liegt daran, dass die beiden Tabellen und KUNDEN_MAT_OVERVIEW separat voneinander existierende Tabellen sind und die Einfügungen wirken sich demnach nicht auf die KUNDEN_MAT_OVERVIEW-Tabelle aus.

Dieses Problem kann man mithilfe der Definition von Triggern lösen, die ausgelöst werden, wenn sich Werte in der Bestell- oder Kundentabelle ändern. Da die beiden Tabellen über die KUNDEN_ID verknüpft sind, kann der Fremdschlüssel mit der Option ON DELETE CASCADE ausgestattet werden. Dadurch werden die Einträge, die in der Kundentabelle gelöscht werden, automatisch auch aus der Bestelltabelle entfernt und man muss nur die Änderungen in der Bestelltabelle als Auslöser für die Trigger beachten. In MySQL kann ein Trigger nur für einen Datenbankmanipulationsoperator gleichzeitig verwendet werden (Oracle, 2025c), weshalb wir für INSERT und DELETE jeweils einen Trigger definieren müssen. Da wir keine Datensätze aktualisieren, vernachlässigen wir aus Simplitätsgründen den Trigger für UPDATE.

Für unser Beispiel sieht der INSERT-Trigger wie folgt aus:

Codeblock 5.12: Insert Trigger für die Tabelle Bestellung

```
1 CREATE TRIGGER UPDATE_BESTELLUNG_MAT_OVERVIEW_AFTER_INSERT
2 AFTER INSERT ON BESTELLUNG
3 FOR EACH ROW
4 BEGIN
5     DECLARE v_land VARCHAR(255);
6     DECLARE v_jahr INT;
7     SELECT LAND INTO v_land FROM KUNDEN WHERE KUNDEN_ID = NEW.FK_KUNDEN;
8     SELECT EXTRACT(YEAR FROM NEW.BESTELLDATUM) INTO v_jahr;
9
10    IF EXISTS (
11        SELECT 1
12        FROM KUNDEN_MAT_OVERVIEW
13        WHERE LAND = v_land AND JAHR = v_jahr
14    ) THEN
15        UPDATE KUNDEN_MAT_OVERVIEW
16        SET GESAMTUMSATZ = GESAMTUMSATZ + NEW.UMSATZ
17        WHERE LAND = v_land AND JAHR = v_jahr;
18    ELSE
19        INSERT INTO KUNDEN_MAT_OVERVIEW (JAHR, LAND, GESAMTUMSATZ)
20        VALUES (v_jahr, v_land, NEW.UMSATZ);
21    END IF;
22 END;
```

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

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

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

Codeblock 5.13: Aktualisierung der materialisierten Sicht

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

Da die Umsetzung von inkrementeller Auffrischung, siehe Kapitel 5.2, in Postgres nicht unterstützt wird, muss die materialisierte Sicht immer vollständig aktualisiert werden. Den Befehl führen wir jeweils nach den INSERT und DELETE-Befehlen auf der Kundentabelle aus. Da wir den Einfluss auf die Performance des Befehls untersuchen möchten, führen wir ihn einmal nach jeder Einfügung in die Kundentabelle aus und einmal, nachdem alle Datensätze eingefügt wurden.

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

5.4 Analyse der Ergebnisse

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

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

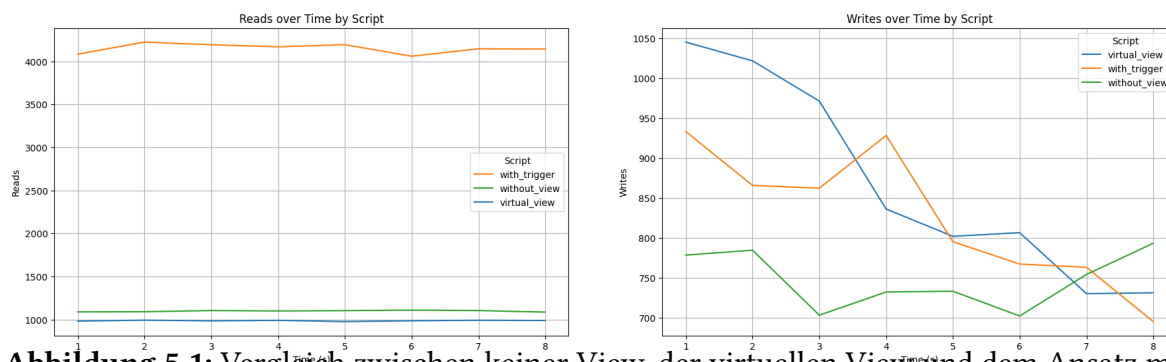


Abbildung 5.1: Vergleich zwischen keiner View, der virtuellen View und dem Ansatz mit Triggern in MySQL

Bei dem zweiten Vergleich sehen wir zuallererst einen sehr deutlichen Performanceunterschied beim Ansatz mit den Triggern zwischen Postgres und MySQL. Begründet werden kann dieser Unterschied mit den verschiedenen Vorteilen des jeweiligen Datenbankmanagementsystems und dessen Umgebung. Wir haben beide Systeme mit dem gleichen Ansatz gebenchmarkt, damit wir die Implementierung der nativen materialisierten Sicht, die nur in Postgres möglich ist, besser vergleichen zu können. Denn die Ergebnisse der nativen Implementierung sind in Bezug auf die Abfragegeschwindigkeit tatsächlich am performantesten und die Anzahl an Aktualisierungen (5.13) hat dabei keinen Einfluss. Anders hingegen sieht es bei der Einfüge-Geschwindigkeit aus, da dort die Implementierung, die nach jedem Insert-Befehl aktualisiert nicht am schnellsten, sondern am langsamsten ist. Der Vergleich zwischen den DBMS fällt wieder schwer, da die Unterschiede zwischen with_trigger und with_trigger_postgres sehr groß sind (etwa um Faktor 2–3). Damit wird noch einmal deutlich wie stark die Einfügedauer bei den materialisierten Sichten von der Anzahl an Refreshs abhängig ist, da mat_view_refresh_every unterhalb der Performance von with_trigger liegt.

In unserem Beispiel ist die einmalige Aktualisierung der Sicht besser als das Verwenden der Trigger in Postgres.

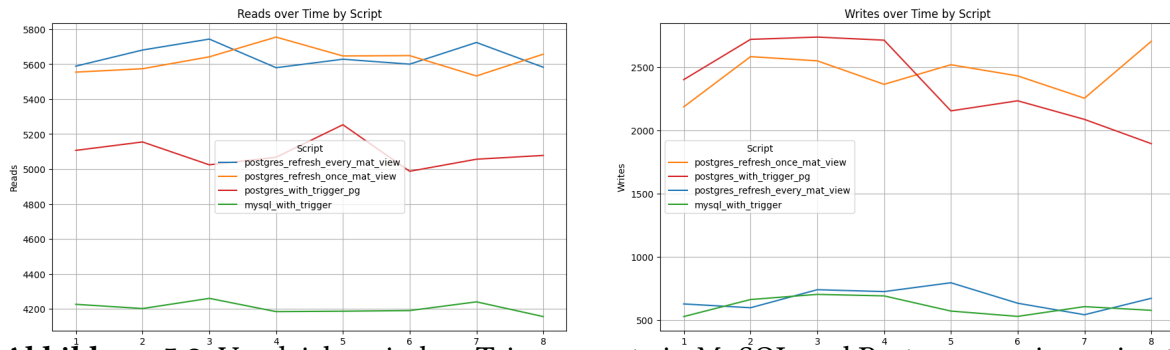


Abbildung 5.2: Vergleich zwischen Triggeransatz in MySQL und Postgres, sowie zwei nativen Implementierungen in Postgres

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

6 Partitionen

In diesem Kapitel analysieren wir die Funktionsweise von Partitionen und das Verhalten des Query-Optimizers. Zudem betrachten wir deren Verwendungszweck sowie die verschiedenen Typen. Abschließend führen wir Benchmark-Tests durch, um die jeweiligen Vor- und Nachteile zu bewerten.

6.1 Grundlagen

Zu Beginn muss zunächst geklärt werden, was eine Partition ist. Eine partitionierte Tabelle ist eine logische Einheit, die aus mehreren physischen Subtabellen besteht (Schwartz et al., 2012, pp. 265–273). Das System verwaltet die Partitionen intern, sodass der Benutzer nicht bemerkt, wie genau die Daten organisiert sind. Dadurch wirken sie für ihn wie eine Blackbox. Wir werden gleich sehen, wie man erkennen kann, welche Komponenten-Tabellen intern verwendet werden.

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 höheren Komplexität der `CREATE TABLE`- und `ALTER TABLE`-Befehle. In der Partitions-klausel selbst können auch Funktionen verwendet werden, die eine nicht-konstante, deterministische Ganzzahl zurückgeben, z.B. `YEAR()`.

Bevor wir zu den Vorteilen der Partitionierung und deren Funktionsweise kommen, klären wir noch die Einschränkungen. Zum einen müssen alle Spalten, nach denen die Partitionierung erfolgt, im Primärschlüssel oder Unique Index enthalten sein. Ansonsten ist es möglich, die Partitionen korrekt zu erstellen oder zu verwalten. Als logische Schlussfolgerung ergibt sich zusätzlicher Aufwand für die Pflege der neuen Indizes. Außerdem können alle Fremdschlüssel-Bedingungen (engl. foreign key constraints) nicht verwendet werden. Zum anderen muss man im Hinterkopf haben, dass es ein Limit an Partitionen pro Tabelle gibt. Bei älteren MySQL-Versionen liegt dieses Limit bei 1024 und seit MySQL-Version 8.0 bei 8192 Partitionen (??). Wie wir später noch sehen werden, sollte man aus verschiedenen Gründen so wenige Partitionen wie möglich definieren. Daher hat diese Einschränkung nicht die größte Relevanz, sollte jedoch im Hinterkopf behalten werden.

Da wir die Bedingungen nun geklärt haben, kommen wir als Nächstes zu der Funktionsweise. Partitionierte Tabellen bestehen aus mehreren zugrunde liegenden Tabellen, die durch Handler-Objekte repräsentiert werden. Jede Partition wird von derselben Storage Engine auf normale Weise verwaltet, aber die einzelnen Partitionen können nicht direkt angesprochen werden. Es hängt jedoch auch vom jeweiligen Datenbankmanagementsystem ab, denn in Oracle ist es wie folgt möglich:

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

Bei der Partitionierung in MySQL werden Indexe pro Partition definiert, anstatt über die gesamte Tabelle hinweg. Alle Indexe der Tabelle werden als identische Indexe auf jede Partition angewendet. Die jeweilige Implementation hängt auch hier wieder vom jeweiligen DBMS ab und kann sich unterscheiden. Beispielsweise in Oracle können Indexe und Tabellen auf flexiblere und komplexere Weise partitioniert werden. Aus Sicht der Storage Engine sind Partitionen einfach Tabellen und es ist dabei egal, ob eine Tabelle eigenständig oder Teil einer partitionierten Tabelle ist.

Der Query-Optimizer hat das Ziel, Partitionen beim Ausführen von Abfragen auszuschließen und nur diejenigen Partitionen zu untersuchen, die die gesuchten Daten enthalten. Dies ist dann immer der Fall, wenn eine WHERE-Klausel mit dem Partitionsausdruck übereinstimmt. Der Fachbegriff dafür heißt **Pruning**. Bei SELECT-Abfragen entscheidet der Query-Optimizer, ob bestimmte Partitionen ignoriert werden können und leitet die Anfragen an die Storage Engine weiter. Bei INSERT-Abfragen wird bestimmt, welche Partition die neue Zeile erhält und der Befehl wird dementsprechend übergeben. Ähnlich funktioniert es bei DELETE-Abfragen, bei denen die Löschanfrage an die passende Partition weitergegeben wird. Man sollte sich in Erinnerung rufen, dass beim Löschen einer Zeile diese zuerst lokalisiert werden muss. Bei UPDATE-Abfragen 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. Einige dieser Operationen unterstützen Pruning und andere, wie z.B. INSERT-Abfragen sind von Natur aus ausschließend (engl. self-pruned).

In der Funktionsweise des Pruning kann man auch den Hauptzweck der Partitionierung erkennen, denn wie bei der Indexierung und der Datenclustering einer Tabelle, trägt es dazu bei, große Teile der Tabelle vom Zugriff auszuschließen und zusammengehörige Zeilen nahe beieinander zu speichern. Statt Indexe zu verwenden, bietet es sich an Tabellen ohne Indexe zu erstellen und ausschließlich die Partitionierung zu nutzen, um gezielt auf die gewünschten Zeilen zuzugreifen. Wenn man sich in der Nähe der gewünschten Daten befindet, kann man von dort aus entweder das relevante Datengebiet sequentiell scannen oder es in den Speicher laden und indexieren. Zudem hat die Partitionierung einen geringen Overhead, weil es keine Datenstruktur gibt, die auf einzelne Zeilen zeigt und ständig aktualisiert werden

muss. Auch identifiziert die Partitionierung Daten nicht auf Zeilenebene und besitzt keine separate Datenstruktur. Stattdessen gibt es eine mathematische Formel, die bestimmt, welche Partitionen welche Kategorien von Zeilen enthalten können. Daraus können erhebliche Performancevorteile resultieren. Wenn wir Partitionen anstelle von Indexen verwenden, steigt die Effizienz, je mehr Partitionen durch die WHERE-Klausel in der Abfrage ausgeschlossen werden. Besonders vorteilhaft können diese sein, wenn die Tabellen sehr groß sind und nicht mehr vollständig in den Speicher passen. Außerdem sind partitionierte Daten einfacher zu verwalten, da gesamte Partitionen gelöscht oder auch wiederhergestellt werden können. Partitionierte Daten können auch physisch verteilt werden, sodass der Server mehrere Festplatten effizienter nutzen kann.

MySQL unterstützt mehrere Arten der Partitionierung. Neben der am häufigsten verwendete Art, Range-Partitionierung, bietet MySQL auch die Partitionsarten Key, Hash und List an (Oracle, 2025b). Beim RANGE-Partitionierung erfolgt die Zuordnung von Zeilen zu Partitionen basierend auf Spaltenwerten, die in einen definierten Wertebereich fallen. Für die Partitionierung einer Datumsspalte bietet sich seit MySQL 5.5 der Partitionstyp RANGE COLUMNS an, bei dem keine zusätzliche Funktion benötigt wird. Ähnlich funktioniert das LIST-Partitionierung, bei dem die Partition jedoch anhand von Spaltenwerten ausgewählt wird, die einem der vordefinierten diskreten Werte entsprechen. Beim HASH-Partitionierung wird die Partition anhand eines Werts zugewiesen, der durch einen benutzerdefinierten Ausdruck ermittelt wird. Dieser Ausdruck arbeitet mit den Spaltenwerten der Zeilen, die in die Tabelle eingefügt werden sollen und kann jede gültige MySQL-Formel umfassen, die einen ganzzahligen Wert liefert. Die Hash-Subpartitionierung hilft auch dabei, die Daten in kleinere Stücke zu zerlegen und das Problem zu entschärfen. Die KEY-Partitionierung wiederum ähnelt dem HASH-Partitionierung, jedoch werden hier nur eine oder mehrere zu bewertende Spalten übergeben, während der MySQL-Server eine eigene Hashing-Funktion nutzt. Anders als beim HASH-Partitionierung können diese Spalten auch andere Datentypen als Ganzzahlen enthalten, da die von MySQL verwendete Hashing-Funktion ein ganzzahliges Ergebnis garantiert, unabhängig vom Datentyp der Spalte.

Wenn man keinen partitionierten Schlüssel in der WHERE-Klausel angibt, dann muss der Abfrageausführungsmechanismus alle Partitionen in der Tabelle durchsuchen, was bei großen Tabellen extrem langsam sein kann. Mit dem SQL-Befehl EXPLAIN kann man überprüfen, ob der Optimierer Partitionen pruned oder nicht.

Codeblock 6.1: Unterschiedliche Ausführungspläne

```
1 EXPLAIN SELECT * FROM BESTELLUNG \G;  
2 EXPLAIN SELECT * FROM BESTELLUNG WHERE YEAR(day) = 2020\G;  
3 EXPLAIN SELECT * FROM BESTELLUNG WHERE day BETWEEN '2020-01-01' AND '2020-12-31'\G;
```

In diesem Beispiel haben wir eine Tabelle `BESTELLUNG`, bei der es für jedes Jahr eine eigene Partition gibt und Artikel nur zwischen den Jahren 2020 und 2022 bestellt wurden. Wenn wir die erste Query ausführen, dann sehen wir, dass logischerweise alle 3 Partitionen betrachtet werden. Bei der `WHERE`-Klausel für die Abfrage aus Zeile 2 würden wir erwarten, dass nur eine Partition abgefragt wird, aber wir erhalten das gleiche Ergebnis wie bei der Ersten. Daraus lässt sich schließen, dass MySQL nur Vergleiche mit den Partitionsspalten optimieren kann und das Ergebnis eines Ausdrucks nicht verwendet, selbst wenn dieser Ausdruck der Partitionsfunktion entspricht. Man kann dieses Verhalten mit dem von indexierten Spalten vergleichen, die auch im Abfrageausdruck isoliert sein müssen, damit der Index verwendet werden kann. Bei der Query aus der dritten Zeile verweist die `WHERE`-Klausel direkt auf die Partitionsspalte und nicht auf einen Ausdruck und deshalb wird nur die Partition aus dem Jahr 2020 untersucht. Der Optimierer kann Bereiche in Listen von diskreten Werten umwandeln (z.B. mit dem `IN`-Operator), auf jedes Element in der Liste prunen und während der Abfrageverarbeitung Partitionen gezielt entfernen. Wenn zum Beispiel eine partitionierte Tabelle die zweite Tabelle in einem Join ist und die Join-Bedingung der partitionierte Schlüssel ist, wird MySQL nur nach übereinstimmenden Zeilen in den relevanten Partitionen suchen. Das ist aber mit `EXPLAIN` nicht sichtbar, weil es zur Laufzeit passiert, nicht zur Abfrageoptimierungszeit. Bei der Durchführung des Benchmarks werden wir, wie hier auch, verschiedene Schreibweisen für die gleiche Abfrage auf die Performance untersuchen, damit wir den effizientesten Weg finden.

Die Effizienz von Partitionierung basiert auf zwei wichtigen Annahmen. Zum einen muss man die Suche durch das Pruning von Partitionen beim Abfragen eingrenzen können und zum anderen muss die Partitionierung selbst nicht sehr kostspielig sein. Diese Annahmen sind aber nicht immer gültig. Im Folgenden betrachten wir 3 unterschiedliche Leitsätze, um mögliche Fehler, die beim Umgang mit Partitionen auftreten können, zu vermeiden.

Zuallererst kann das Ergebnis der Partitionsfunktion `NULL` sein und das kann auch passieren, wenn das Datum als `NOT NULL` deklariert wird, da man Werte speichern kann, die kein gültiges Datum sind. Jede Zeile, deren Datum entweder `NULL` oder kein gültiges Datum ist, wird in der ersten definierten Partition gespeichert. Wenn man eine Abfrage hat, die alle Jahre außer 2020 herausfiltert, dann muss man zwei Partitionen statt nur einer durchsuchen, um die Zeilen zu finden. Diese Effekte verstärken sich mit der Größe der ersten Partition, weshalb man entweder eine dedizierte Partition für diese Sonderfälle einführen sollte oder Funktionen, wie `RANGE COLUMNS` verwenden sollte.

Wenn man einen Index definiert, der nicht mit der Partitionsklausel übereinstimmt, kann es dazu führen, dass Partitionen nicht ausgeschlossen werden können, obwohl man eigentlich davon ausgehen würde. Man sollte daher versuchen, auf nicht partitionierten Spalten keine Indexe zu erstellen, es sei denn, die Abfragen enthalten auch einen Ausdruck, der beim Pruning der Partitionen hilft. Manchmal lässt sich dieses Problem nicht auf den ersten Blick

erkennen. Angenommen eine partitionierte Tabelle ist die zweite Tabelle in einem Join und der Index, der für den Join verwendet wird, ist nicht Teil der Partitionsklausel, dann wird jede Zeile im Join jede Partition in der zweiten Tabelle durchsuchen.

Bei der Range-Partitionierung kann die Bestimmung der korrekten Partition teuer sein, weil der Server die Liste der Partitionsdefinitionen durchsucht, um die Richtige zu finden. Diese lineare Suche ist nicht sehr effizient und es steigen die Kosten, je mehr Partitionen es gibt. Das Öffnen und Sperren von Partitionen, wenn eine Abfrage auf eine partitionierte Tabelle zugreift, ist eine andere Art von Overhead pro Partition. Sie erfolgt vor dem Pruning, sodass dieser Overhead nicht pruned werden kann. Außerdem ist diese Art von Overhead unabhängig vom Partitionstyp und betrifft alle Arten von Anweisungen. Deshalb besagt der dritte Leitsatz, dass die Anzahl der definierten Partitionen begrenzt werden sollte.

Alle Partitionen sollten dieselbe Storage Engine nutzen, was Einschränkungen bei den verwendbaren Funktionen und Ausdrücken mit sich bringt. Zudem unterstützen einige Storage Engines keine Partitionierung.

6.2 Durchführung

In diesem Abschnitt wird die Durchführung der Benchmarks für die Partitionierung beschrieben. Besonders möchten wir den Effekt veranschaulichen, den die Partitionierung auf die Abfragen hat. Deshalb vergleichen wir jeweils eine partitionierte Tabelle mit einer nicht Partitionierten und stellen an beide Tabellen die gleiche Abfrage. Dieses Verfahren wenden wir auf alle Partitionsmethoden, die in dem Unterkapitel [6.1](#) angesprochen haben. Abhängig vom Typen stellen wir zusätzlich leicht unterschiedliche Abfragen.

Zunächst benutzen wir erneut als Basis die Kundentabelle ([2.1](#)) und die Bestelltabelle ([2.2](#)). Die Tabelle, die wir auf unterschiedliche Partitionen verteilen wollen, ist die Kundentabelle. Wir müssen aber noch beide Tabellen etwas anpassen, da es, wie schon erwähnt, einige 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. Die Insert-Befehle sind bei allen Typen der Partitionierung gleich und zudem mit dem Referenzfall ohne Partitionierung 100%-ig identisch. Bei den Select-Queries gibt es jedoch Unterschiede. Wichtig ist dort immer, dass die Ergebnisse der Referenzbenchmark mit denen der partitionierten Fälle übereinstimmen. Wenn man dies nicht tut, dann sind die Performancemessungen nicht miteinander vergleichbar.

Für die Range-Partitionierung wollen wir unterschiedliche Partitionen je nach Alter des Kunden. Alle fünf Jahre soll es eine neue Partition geben. Damit es zu keinen Fehlern kommt, muss hier der Geburtstag auch Teil des Primärschlüssels der Kundentabelle sein. Außerdem

muss der Attribut GEBURTSTAG bei der Bestelltabelle als Attribute hinzugefügt werden, damit das Joinen der Tabellen über den Primärschlüssel effizienter ist. Wir wollen auch die Ansätze RANGE (siehe 6.2) und RANGE COLUMNS miteinander vergleichen und beide Varianten bei der Erstellung der Tabelle benutzen.

Codeblock 6.2: 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 testen wir mehrere Select-Befehle, da je nach Art der Abfrage des Datums das Pruning besser oder schlechter funktioniert (siehe Abschnitt 6.1). Dazu joinen wir zunächst die Kundentabelle an die Bestelltabelle über die Attribute KUNDEN_ID und GEBURTSTAG. Die Testkunden werden so generiert, dass sie immer zufällig zwischen den Jahren 1950 und 2020 geboren sind. Die darauffolgenden WHERE-Bedingungen unterscheiden sich aber zwischen den unterschiedlichen Select-Befehlen.

Codeblock 6.3: Unterschiedliche WHERE-Bedingungen

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

Bei der List-Partitionierung wollen wir pro Land eine eigene Partition haben. Dafür muss die Spalte LAND Teil des Primärschlüssels sein. Beim Einfügen der Testdaten wählen wir pro Kunde ein zufälliges Land aus der Liste der 20 einwohnerreichsten Länder der Welt aus. Damit müssen wir bei der Erstellung der Tabelle auch 20 Partitionen sowie eine Zusätzliche für sonstige Werte erstellen (6.4).

Codeblock 6.4: 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  )
7  PARTITION BY LIST COLUMNS(LAND) (
8    PARTITION p_china VALUES IN ('China'),
9    PARTITION p_india VALUES IN ('India'),
10   PARTITION p_united_states VALUES IN ('United States'),
11   -- other partitions
12   PARTITION p_thailand VALUES IN ('Thailand'),
13   PARTITION p_other VALUES IN ('Other')
14 );

```

Auch bei der List-Partitionierung überprüfen wird die Performance von unterschiedlichen Select-Befehlen. Zunächst joinen wir, wie zuvor, die Kundentabelle mit der Bestelltabelle und in der WHERE-Bedingung filtern wir so, dass alle Kunden aus Deutschland kommen. Genau die gleiche Query verwenden wir beim Vergleich mit der Referenz ohne Partition. Zusätzlich wollen wir die Performance untersuchen, wenn aus der Liste an Ländern zufällig fünf ausgewählt werden und alle Kunden nur aus einem dieser fünf Länder kommen. Dazu testen wir 3 verschiedene Ansätze. Zum einen über den OR-Operator, den anderen mithilfe des IN-Operators und als Letztes fragen wir die 5 Länder, wie im ersten Beispiel, einzeln ab und verbinden die Ergebnisse der 5 Antworten mithilfe des UNION-Operators.

Bei der Hash-Partitionierung müssen wir am wenigsten verändern und nur die Zeilen aus dem Codeblock 6.5 am Ende des Create Kunden-Befehls hinzufügen. In diesem Fall testen wir auch nur eine einzige Select-Query, die wieder beide Tabellen joined und in der WHERE-Bedingung überprüft, ob die KUNDEN_ID zwischen den Werten 1000 und 2000 liegt. Für unseren Referenzfall fragen wir die Version ohne Partitionierung mit der gleichen Query ab. Damit wir mehr Werte miteinander vergleichen können, testen wir verschiedene Varianten der Hash-Partitionierung, indem wir die Anzahl der Partitionen variieren. Die Anzahl der Partitionen beträgt in unseren Benchmarks 5, 50 und 500. Einschließlich des Referenzfalls ergeben sich somit vier unterschiedliche Testfälle, die wir miteinander vergleichen.

Codeblock 6.5: Hash-Partitionierung

```

1 PARTITION BY HASH(KUNDEN_ID)
2 PARTITIONS 4;

```

Genau das Gleiche wie bei der Hash-Partitionierung machen wir auch bei der Key-Partitionierung. Dafür müssen wir nur im Codeblock 6.5 das Signalwort HASH mit KEY ersetzen. Die KEY-Partitionierung wollen wir aber nicht mit der Referenz vergleichen, sondern nur mit dem Ergebnis von HASH.

6.3 Analyse

Wie in dem Abschnitt 6.2 erklärt, führen wie pro Partitionstyp unterschiedliche Benchmark-Tests durch. Außerdem vergleichen die Ergebnisse mit jeweils einer Referenz, bei der es keine partitionierten Tabellen gibt.

Bei der Range-Partitionierung fällt auf, dass die Benutzung von RANGE oder RANGE COLUMNS keinerlei Einfluss auf die Performance hat. Damit stellt sich bei der Verwendung nicht die Frage nach der Performance, sondern nur welche Art der Definition der Nutzer bevorzugt. Es wird auch klar, dass das Pruning nur in dem Fall funktioniert, bei dem wir die Geburtstage zwischen dem ersten und letzten Tage des Jahres abfragen (Abbildung 6.1). Wenn wir die Funktion YEAR() benutzen, dann sind die Werte schlechter als in dem Fall komplett ohne Replikation. Es lässt sich als feststellen, dass Pruning mithilfe von YEAR() offensichtlich nicht funktioniert. Bei dem Fall ohne Partitionierung spielt die Verarbeitung durch YEAR() keinen Einfluss, denn es ist genauso schnell wie der direkte Aufruf über den 01. Januar des Jahres. Wenn wir die Query mit dem Primärschlüssel zwischen der Version mit Range-Partitionierung und ohne Partitionen vergleichen, dann gibt es keinen großen Unterschied, aber dennoch hat die Version ohne Partitionen einen leichten Vorteil. Der größte Unterschied ist aber definitiv, wenn Pruning erfolgreich ist, da die Query mit deutlichem Abstand schneller als bei allen anderen Select-Queries. Bei den Insert-Befehlen sieht es leicht anders aus, aber dort ist der Fall ohne Partitionierung nur minimal schneller als mit.

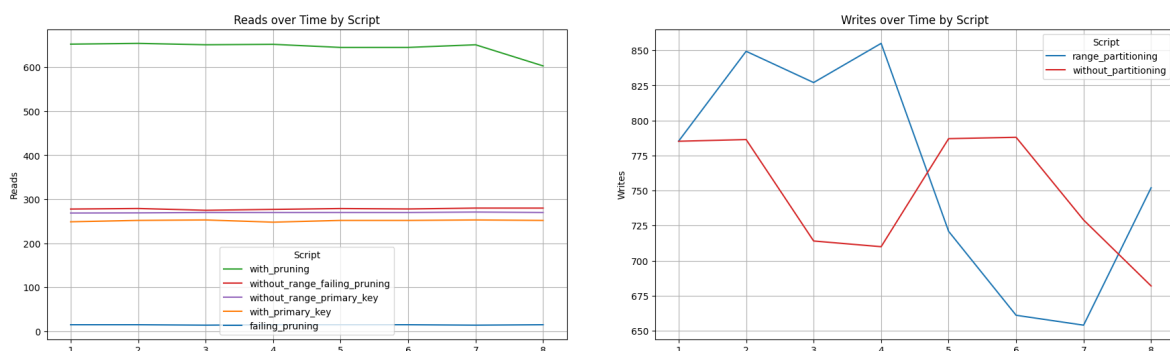


Abbildung 6.1: Vergleich zwischen der Range-Partitionierung und ohne Partition

Bei der List-Partitionierung 6.2 können wir auch einige interessante Beobachtungen machen. 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 wir statt nur eines Landes mehrere abfragen, sehen wir für die verschiedenen Operatoren unterschiedliche Ergebnisse. Die beste Performance erzielt der IN-Operator. Dicht darauf folgt der OR-Operator. Mit etwas größerem Abstand liegt der Fall ohne Partitionierung. Deutlich abgeschlagen ist die Verbindung der Ergebnisse per UNION-Operator. Die

Performance beim Einfügen der Daten ist bei der Partitionierung und dem Referenzfall sehr ähnlich, wobei letzterer einen ganz leichten Vorteil hat.

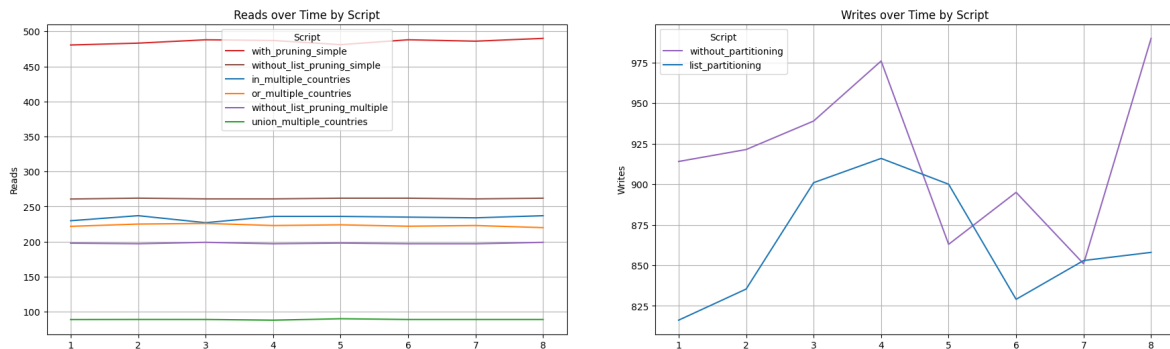


Abbildung 6.2: Vergleich zwischen der List-Partitionierung und ohne Partition

Bei der Key-Partitionierung fällt auf, dass es keinen signifikanten Performance-Unterschied zur Hash-Partitionierung gibt, wenn derselbe Datensatz und dieselbe Anzahl von Partitionen verwendet wird. Generell ist die Key-Partitionierung häufig stabiler und optimierter ist, insbesondere wenn es um Primärschlüssel geht.

Bei der Hash-Partitionierung 6.3 nehmen sich beide Select-Fälle kaum etwas. Auch hier fällt auf, dass die Werte der Abfragen sehr konstant sind. Nur bei der Variante mit 500 Partitionen gibt es deutlich sichtbare Schwankungen. Diese liegen aber nicht an dem Pruning, denn mit dem SQL-Befehl EXPLAIN sehen wir, dass alle 500 Partitionen benötigt werden und nicht keine Partitionen durch Zufall geprunt werden. Die Hash-Partitionierung berechnet aus dem Wert einer bestimmten Spalte mithilfe einer Hash-Funktion einen Hash-Wert und anhand dessen wird die Zeile einer der Partitionen zugewiesen. Da der Hash-Wert bei 500 Partitionen mit modulo 500 gebildet wird, landet jeder 500-ste Wert in der gleichen Partition. Damit ist klargestellt, dass immer alle Partitionen benutzt werden, was die folgenden Ergebnisse zeigen. Zunächst zeigt sich, dass die Abfrage ohne Partitionierung am schnellsten ist. Danach kann man der Regel folgen, dass je mehr Partitionen es gibt, desto langsamer wird die Query. Der Unterschied zwischen ohne Partitionen und 5 Partitionen ist noch überschaubar, aber bei 500 Partitionen sieht man einen sehr deutlichen Unterschied. Das bestätigt die wichtig von unserem Leitsatz, dass je mehr Partitionen benötigt werden, desto aufwendiger wird die Suche innerhalb der Partitionsstruktur, was die Performance verringert.

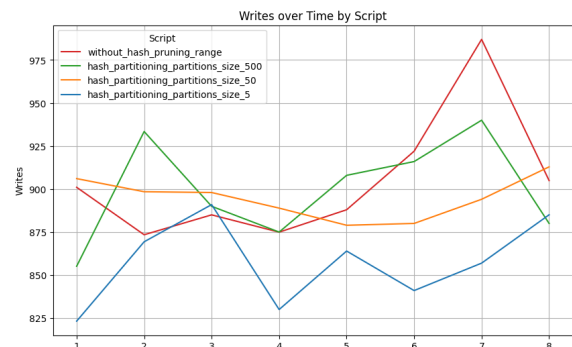
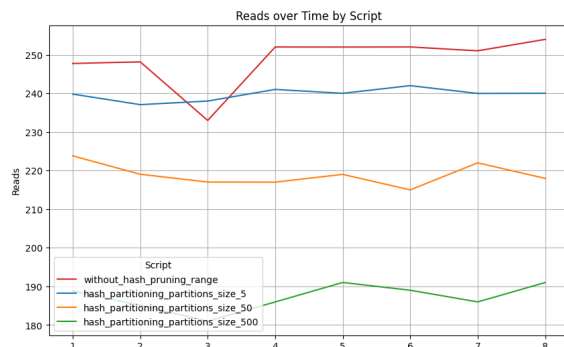


Abbildung 6.3: Vergleich zwischen der Hash-Partitionierung und ohne Partition

7 Replikation

In diesem Kapitel befassen wir uns mit dem Thema Replikation. Replikation ist die Grundlage für den Aufbau großer, leistungsstarker Anwendungen auf der Basis von MySQL. Es verfolgt dabei die sogenannten „Scale-Out“-Architektur, bei der mehrere Storage-Knoten parallelisiert arbeiten und nach außen wie ein einziges Gesamtsystem (Luber, 2023). Dadurch ist die Skalierbarkeit nahezu unbegrenzt durch das einfache Hinzufügen weiterer Speicherknoten und nicht wie bei Scale-Up durch die Systemgrenzen eines einzelnen Geräts limitiert. Es trägt allerdings auch Nachteile mit sich, zu denen wir u.a. auch später in diesem Abschnitt kommen. In schon in den vorherigen Kapiteln gehen wir zuerst auf die Grundlagen ein, dann betrachten wir die Konfiguration, die die Basis für die Benchmarks bilden und abschließend analysieren wir die Ergebnisse.

7.1 Grundlagen

Replikation ermöglicht die Konfiguration eines oder mehrere Server als Replicas eines anderen Servers, auch Master genannt (Schwartz et al., 2012, pp. 447–477). Neben der Begrifflichkeit Master-Replika, sind auch die Varianten Primary-Secondary, als auch Master-Slave üblich. Das grundlegende Problem, das die Replikation löst, besteht darin, die Daten eines Servers mit denen eines anderen synchron zu halten. Es können sich mehrere Replicas mit einem einzigen Master verbinden und mit diesem synchron bleiben. Master und Replicas können in vielen verschiedenen Konfigurationen angeordnet werden. Neben der klassischen Master-Replica-Variante können Replicas selbst als Master für weitere Replicas dienen. Zudem ist auch eine Master-Master-Kombination möglich. Das Prinzip der Replikation ist nicht nur für eine höhere Effizienz vorteilhaft, sondern auch für hohe Verfügbarkeit, Skalierbarkeit und Datenanalysen im Data Warehousing, aber sollte keine richtigen Backups ersetzen. Effizienzvorteile gibt es insbesondere durch die Lastverteilung, die Leseanfragen auf mehrere Server verteilt werden, was besonders für leselastige Anwendungen vorteilhaft ist. Außerdem kann man das Ganze mit Methoden wie Round-Robin-DNS oder Loadbalancer optimieren.

Im folgenden Abschnitt erklären wir die Funktionsweise der Replikation und betrachten dabei den einfachen Fall mit einem Master und einem oder mehreren Replicas. Unmittelbar bevor jede Transaktion, die Daten aktualisiert, auf dem Master abgeschlossen wird, zeichnet der

Master die Änderungen in seinem Binärlog (engl. binary log) auf. MySQL schreibt Transaktionen seriell im Binary-Log und teilt nach dem Schreiben der Ereignisse den Storage Engines mit, die Transaktionen zu committen. Zu diesen Änderungen können beispielsweise neu deklarierte Tabellen oder Trigger sowie Einfügeoperationen in bestehende Tabellen gehören. Im nächsten Schritt muss die Replica die Veränderungen auf dem Master mitbekommen. Dazu wird ein Worker-Thread gestartet, der als I/O-Slave-Thread bezeichnet wird, und eine Client-Verbindung zum Master geöffnet. Anschließend wird ein spezieller Prozess (binlog dump process) gestartet, der die Ereignisse aus dem Binary-Log des Masters liest. Nach dem Verarbeiten schreibt der Thread die Werte auf seine eigene Festplatte, in das sogenannte Relay-Log. Wenn er alle Ereignisse auf dem Log verarbeitet hat, geht er in einen passiven Zustand und wartet auf Aktualisierungen. Den letzten Teil des Prozesses übernimmt der SQL-Slave-Thread. Er liest und spielt Ereignisse aus dem Relay-Log ab und aktualisiert so die Daten des Replicas, sodass sie mit denen des Masters übereinstimmen. Wenn dieser Thread eine etwa gleich schnelle Verarbeitung wie der I/O-Thread, dann bleibt das Relay-Log normalerweise im Cache des Betriebssystems, sodass Relay-Logs nur sehr geringe Overhead-Kosten haben. Die Ereignisse, die der SQL-Thread ausführt, können optional zusätzlich in das eigene Binary-Log des Replicas geschrieben werden.

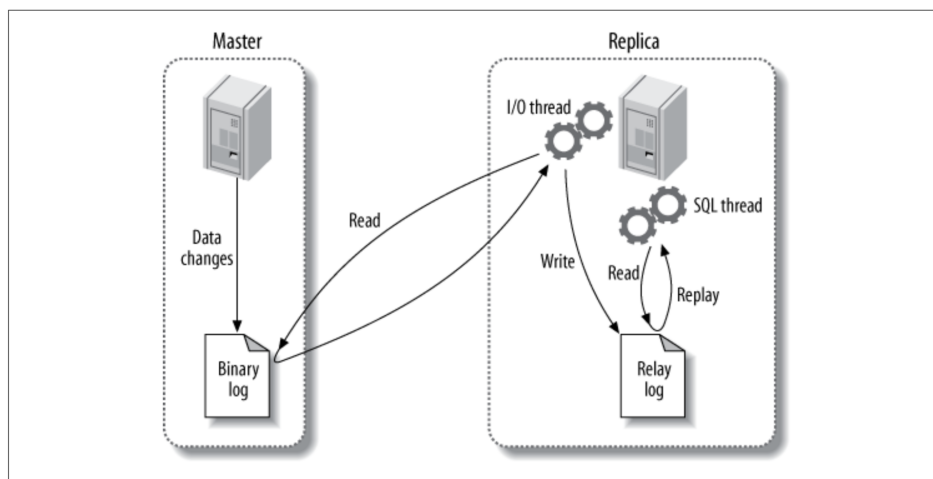


Abbildung 7.1: Darstellung der unterschiedlichen Threads

Die Abbildung 7.1 zeigt nur die beiden Replikation-Threads, die auf dem Replica laufen. Zusätzlich gibt es jedoch einen weiteren Thread auf dem Master, da die vom Replica zum Master geöffnete Verbindung einen Thread auf dem Master startet, ähnlich wie jede andere Verbindung zu einem MySQL-Server.

Diese Replikationsarchitektur entkoppelt die Prozesse des Abrufens und Abspielens von Ereignissen auf dem Replica. 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 parallel in verschiedenen Threads auf dem Master ausgeführt werden

könnten, auf dem Replica nicht parallelisiert werden können, da sie in einem einzelnen Thread ausgeführt werden. Generell gibt es keine Garantie für die Latenz auf der Replica und große Abfragen können dazu führen, dass die Replica 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.

Die Replikation fügt dem Master nur wenig Overhead hinzu. Das binäre Logging, das für ordnungsgemäße Backups und Point-in-Time-Recovery erforderlich ist, kann jedoch einen erheblichen Overhead verursachen. Abgesehen davon verursacht jede angeschlossene Replica nur geringe Last (hauptsächlich Netzwerk-I/O) auf dem Master. Das Anbinden vieler Replicas an einen Master führt einfach dazu, dass die Schreibvorgänge mehrfach ausgeführt werden, jeweils einmal auf jeder Replika. Trotzdem sollte man nicht zu leichtfertig mit der Anzahl an Replicas übertrieben, da dadurch im Wesentlichen viele Daten unnötig dupliziert werden. Bei sehr hoher Arbeitslast (z.B. 5.000 Transaktionen pro Sekunde) und vielen Replicas kann der Overhead durch das Aktivieren aller Replica-Threads jedoch erheblich werden. Replikation eignet sich gut zum Skalieren von Lesevorgängen.

Die Replikation von MySQL ist größtenteils abwärtskompatibel, d.h. dass ein neuerer Server in der Regel problemlos als Replika eines älteren Servers fungieren kann. Andersherum kann es aber zu Fehler kommen, da möglicherweise neue Funktionen oder die SQL-Syntax des neueren Servers nicht verstanden werden kann. In unserem Beispiel verwenden wir ohnehin nur eine MySQL-Version (siehe 7.2).

Es gibt zwei verschiedene Arten der Replikation, 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 eine Replica 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 und das Protokollieren und Wiederholen von den Anweisungen sollte die Replica logischerweise mit dem Master synchron halten. Außerdem sind die Binary-Log-Ereignisse in der Regel recht kompakt sind 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 Replica ausgeführt. Deshalb muss der Binary Log nicht nur der Abfragetext enthalten, sondern auch Metadaten wie den aktuellen Zeitstempel. Es gibt einige Anweisungen, die MySQL nicht korrekt replizieren kann, z.B. Abfragen, die die Funktion `CURRENT_USER()` verwenden und gespeicherte Routinen und Trigger sind ebenfalls problematisch bei dieser Art der Replikation.

Die zeilenbasierte Replikation speichert die tatsächlichen Datenänderungen im Binary-Log.

Ein großer Vorteil, der daraus folgt, ist es, dass MySQL jede Anweisung korrekt replizieren kann. Zudem können einige Änderungen mithilfe der zeilenbasierte Replikation effizienter sein, da die Replica die Abfragen, die die Zeilen auf dem Master geändert haben, nicht erneut ausführen muss. Dies ist beispielsweise der Fall, wenn eine Abfrage viele Zeilen in der Quelltable scannt, jedoch nur zu drei Zeilen in der Zieltabelle ausführt. Bei der anweisungs-basierten Replikation müsste eine Replica die Anweisung erneut ausführen, nur um ein paar Zeilen zu erstellen und bei der Zeilenbasierte ist dies effizient und trivial. Andererseits ist das folgende Ereignis deutlich günstiger mit statement-basierter Replikation zu replizieren:

Codeblock 7.1: Update-Befehl auf dem Master

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

Die Verwendung von zeilenbasierte Replikation für diese Abfrage wäre sehr teuer, da jede Zeile geändert wird und damit auch ins Binary-Log geschrieben müsste, was das Binary-Log-Ereignis extrem groß machen würde. Dies würde sowohl beim Protokollieren als auch bei der Replikation zu einer höheren Last auf dem Master führen. Die Durchführung einer Point-in-Time-Wiederherstellung ist mit einem Binary-Log im row-based Format schwieriger, aber nicht unmöglich.

Insgesamt ist die anweisungs-basierte Replikation besser geeignet, wenn das Schema auf dem Master und dem Replikat unterschiedlich ist, und kann in Szenarien eingesetzt werden, in denen Tabellen unterschiedliche, aber kompatible Datentypen, verschiedene Spaltenreihenfolgen usw. aufweisen. Außerdem vereinfacht es das Durchführen von Schema-Änderungen auf einem Replica, das später als Master verwendet werden soll, um möglicherweise die Ausfallzeit zu reduzieren. Die zeilenbasierte Replikation kann jedoch einige Operationen bei Schema-Änderungen auf einem Replica nicht handhaben. Dafür aber funktioniert sie zuverlässig mit allen SQL-Konstrukten und es gibt weniger Fälle, in denen es zu Problemen kommt. Auch stoppt sie bei Fehlern, z.B. wenn eine erwartete Zeile auf dem Replikat fehlt, was jedoch auch als Vorteil betrachtet werden kann, da es auf Inkonsistenzen hinweist. Bei der anweisungs-basierte Replikation führt ein Update auf dem Master zu keinem Fehler, wenn die Zeile auf dem Replica fehlt. Die zeilenbasierte Replikation erkennt diesen Fehler und stoppt die Replikation, was eine sofortige Überprüfung ermöglicht. Bei der anweisungs-basierten Replikation erfolgen alle Änderungen über einen bekannten Mechanismus (SQL-Anweisungen), was die Fehlersuche und das Verständnis von Problemen erleichtert, aber es erfordert auch mehr Sperren (Locks). Genau andersherum sieht es bei der Zeilenbasierte aus, da dort Nachvollziehbarkeit von Änderungen ohne die Speicherung der ursprünglichen SQL-Anweisung erschwert wird, aber dafür gibt es reduziertes Locking. Die zeilenbasierte Replikation hat auch Vorteile bei der Datenwiederherstellung, da in einigen Fällen auch alte Daten gespeichert werden, was die Wiederherstellung erleichtert. Auch benötigt sie oft

weniger CPU-intensiv, da keine komplexe SQL-Parsing- und Ausführungslogik erforderlich ist und es gibt auch Probleme bei der mehrstufigen Replikation. Denn wenn eine Anweisung mit @@binlog_format = STATEMENT auf dem Master ausgeführt wird, dann wird sie dort als Statement protokolliert. Die nachgelagerte Replicas (Second-Level-Replicas) können diese jedoch als row-basiert weiterleiten, was zu Inkonsistenzen bei der Protokollierung führt.

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. Sie können das Format auch manuell steuern, indem Sie die Session-Variable binlog_format setzen.

7.2 Durchführung

Für die Durchführung nutzen wir wieder die Kundentabelle (2.1) und die Bestelltabelle (2.2), die wir schon aus dem Beispiel aus Kapitel 2.3 kennen. Die einzige Anpassung, die wir vornehmen müssen, sind die unterschiedlichen Werte STATEMENT, ROW und MIXED, die die Variable binlog_format annehmen kann.

Codeblock 7.2: Unterschiedliche Formatimplementationen

```
1 SET SESSION binlog_format = 'STATEMENT'
2 SET SESSION binlog_format = 'ROW'
3 SET SESSION binlog_format = 'MIXED'
```

Damit können wir die Performanceunterschiede zwischen den einzelnen Arten, insbesondere für die Einfügeoperationen, vergleichen. Wie wir sehen, sind die Veränderungen an den Lua-Skripten damit minimal, aber der eigentliche Aufwand entsteht bei der Einrichtung der Replikation auf dem lokalen Rechner und in einem Workflow.

Für die Benchmarks betrachten wir das einfachste Szenario der Replikation, bestehend aus einem Master und einer beliebigen Anzahl an Replicas. Zunächst müssen wir dazu den Master und die Replikationsknoten dazu konfigurieren und anschließend dem Replikat anweisen, sich mit dem Master zu verbinden und von ihm zu replizieren. MySQL hat einige spezielle Privilegien, die es den Replikationsprozessen erst ermöglichen, zu laufen. 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.

Codeblock 7.3: Nutzererstellung und Rechtevergabe

```
1 GRANT REPLICATION SLAVE, REPLICATION CLIENT ON .
```

Wir erstellen diesen Nutzer sowohl auf dem Master als auch auf dem Replica. Der Nutzer, der zur Überwachung und Verwaltung der Replikation verwendet wird, benötigt das REPLICATION CLIENT Privileg und daher ist es einfacher, denselben Benutzer für beide Zwecke zu verwenden. Der nächste Schritt besteht darin, einige Einstellungen auf dem Master zu aktivieren. Zum einen müssen die Binärprotokollierung aktiviert und eine Server-ID angegeben werden. Wenn die Binärprotokollierung im Konfigurationsfile des Masters nicht bereits angegeben wurde, müssen Sie MySQL neu starten. Zum anderen muss explizit eine einzigartige Server-ID angegeben werden. Um zu überprüfen, ob die Binary-Logdatei auf dem Master erstellt wurde, kann man folgenden Befehl ausführen:

Codeblock 7.4: 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`: `sync_binlog=1`. Diese Option sorgt dafür, dass MySQL den Inhalt des Binary-Logs, nicht dem Relay-Log, bei jedem Transaktions-Commit auf die Festplatte synchronisiert und damit kommt es zu keinen verlorenen Log-Ereignissen im Falle eines Absturzes. Wenn diese Option deaktiviert ist, dann hat der Server etwas weniger Arbeitsaufwand, aber Binary-Log-Einträge könnten nach einem Serverabsturz beschädigt oder verloren sein. Auf einem Replikat, das nicht als Master fungieren muss, erzeugt diese Option unnötigen Overhead. Außerdem wird es empfohlen einen Basisnamen für das Binary-Log explizit anzugeben, um einheitliche Binary-Log-Namen auf allen Servern zu erstellen und Änderungen der Binary-Log-Namen zu vermeiden, falls sich der Hostname des Servers ändert. Dazu muss ein Argument für die `log_bin`-Option angegeben werden.

Auf einem Replikat ist nur der Parameter `server_id` erforderlich, wir haben zudem aber noch zwei andere optionale Konfigurationsparameter, die wir hinzufügen können. Zum einen den optionalen Konfigurationsparameter `relay_log`, der den Speicherort und den Namen des Relay-Logs angibt, sie passend dazu die Option `log_bin` für den Master. Zum anderen `log_slave_updates`, um das Replikat die replizierten Ereignisse in sein eigenes Binary-Log zu schreiben. Die Option `skip_slave_start` verhindert, dass das Replikat nach einem Absturz automatisch startet und es gibt dadurch die Möglichkeit, den Server bei Problemen zu reparieren. Die Option `read_only` verhindert, dass die meisten Benutzer nicht-temporäre Tabellen ändern. Ausnahmen sind der Replikation-SQL-Thread und Threads mit dem SUPER-Privileg. Daher sollte es auch vermieden werden, normalen Benutzerkonten das SUPER-Privileg zu geben. Die letztere Option verursacht zusätzlichen Aufwand für die Replikate, aber wir haben

gute Gründe, diese optionalen Einstellungen auf jedem Replikat hinzuzufügen. Wenn ein Replikat stark hinter dem Master zurückliegt, kann der Slave-I/O-Thread viele Relay-Logs schreiben. Der Replikation-SQL-Thread entfernt diese, sobald er mit deren Verarbeitung fertig ist (dies kann mit der Option `relay_log_purge` geändert werden). Wenn das Replikat stark im Rückstand ist, könnte der I/O-Thread tatsächlich den Speicherplatz der Festplatte füllen.

Ein Replikat kann nach einem Absturz leicht beschädigt werden, da die Relay-Logs und die `master.info`-Datei nicht absturzsicher sind.

Der nächste Schritt besteht darin, dem Replikat zu sagen, wie es sich mit dem Master verbinden kann und wie dessen Binary-Logs abspielen wird.

Codeblock 7.5: Verbindung der Replica 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 7.4 übereinstimmen. Um die eigentliche Replikation zu starten, muss man den folgenden Befehl ausführen:

Codeblock 7.6: Starten der Replikation

```
1 START SLAVE;
```

Mit dem folgenden Befehl kann man überprüfen, ob das Ganze erfolgreich war oder nicht.

Codeblock 7.7: Status der Replica

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

Die Spalten `Slave_IO_State`, `Slave_IO_Running` und `Slave_SQL_Running` zeigen an, dass die Replikationsprozesse nicht laufen oder nicht. Wenn `Seconds_Behind_Master` nicht mehr NULL ist, dann bedeutet das, dass der I/O-Thread auf ein Ereignis vom Master wartet, weil er schon alle Binary-Logs abgerufen hat. Wenn man Änderungen an dem Master vornimmt, dann sollte man beobachten, dass die verschiedenen Datei- und Positionswerte auf dem Replikat inkrementiert werden und auch die Änderungen in den Datenbanken sollten auf dem Replikat sichtbar sein. Außerdem sollten zwei Threads auf dem Replikat aktiv sein, die immer unter dem Benutzerkonto „system user“ laufen.

Bei den vorherigen Setup-Anweisungen sind wir von einer frischen Installation ausgegangen. Es gibt aber auch andere Möglichkeiten, um ein Replikat von einem anderen Server zu initialisieren und zwar das Kopieren von Daten von einem Master, das Klonen eines Replikats von einem anderen Replikat oder das Starten eines Replikats von einem aktuellen Backup. 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 [7.4](#)) sowie die Binary-Logs des Masters ab diesem Zeitpunkt. Eine kalte Kopie erfordert das Herunterfahren des Masters, um dessen Dateien zu kopieren, bevor er mit einem neuen Binary-Log neu gestartet wird, was jedoch zu Ausfallzeiten führt. Bei einer warmen Kopie können die Dateien übertragen werden, während der Server weiterhin läuft.

7.3 Analyse

Im vorherigen Abschnitt wurde erklärt, wie man die Master und vor allem die Replicas korrekt konfiguriert und den Prozess der Replikation startet. Es wurde auch erklärt, wie wir das Binlog-Format verändern können und dass wir das Lua-Skript mit der Kunden- und Bestelltabelle verwenden. Als Nächstes analysieren wir unterschiedliche Situationen, damit die Vorteile verständlicher werden.

Im ersten Vergleich wollen wir die Performanceunterschiede zwischen dem Master-Replica-Ansatz und dem Ansatz mit einem MySQL-Server feststellen. Beim Master-Replica-Ansatz verwenden wir mit ROW immer den Default-Wert des Binlog-Formats. Damit keine Fehler auftreten, müssen wir die beiden Ansätze miteinander kompatibel machen. Das Problem dabei ist, dass der Standardport von MySQL (3306) nicht gleichzeitig verwendet werden darf. Daher starten wir den Master auf Port 3307, und jede Replica erhöht diesen Wert um 1. Somit nutzt die dritte Replica den Port 3310. Die Anzahl an Replicas können wir in der `envs.json`-Datei mithilfe der Variablen `REPLICAS_COUNT` festlegen. Die Voraussetzung dafür ist, dass es lokal mindestens diese Anzahl an gestarteten und konfigurierten Replicas gibt. Innerhalb des Workflowjobs muss nichts angepasst werden, da dort der Wert von `REPLICAS_COUNT` aus der Datei genommen wird, um die exakte Anzahl an Replicas zu starten. Wichtig ist noch zu erwähnen, dass beim Master-Replica logischerweise die Insert-Befehle nur auf dem Master also Port 3307 ausgeführt werden. Die Select-Befehle werden sowohl auf dem Master als auch auf den Replicas ausgeführt.

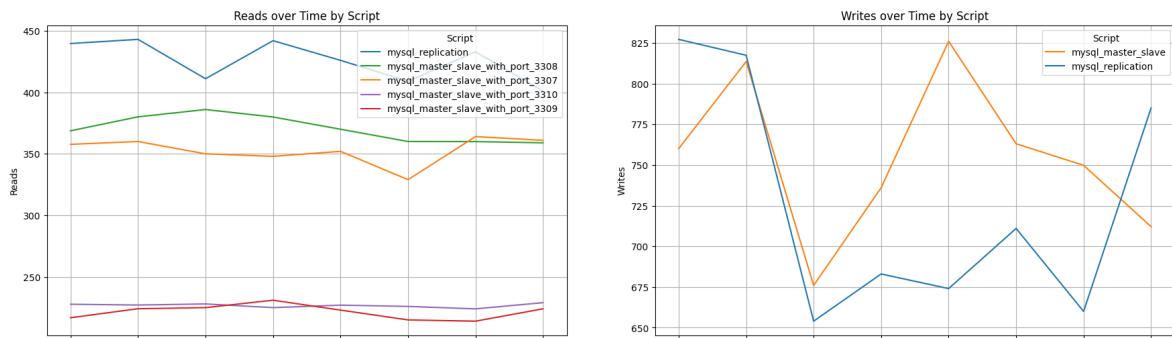


Abbildung 7.2: Vergleich zwischen dem Master mit 3 Replicas und einem normalen MySQL-Server

Wenn wir die Ergebnisse aus Abbildung 7.2 betrachten, dann fällt auf, dass die Version ohne Replikation am schnellsten ist. Danach folgen die Readabfragen auf dem Master auf Port 3307. Der Unterschied der beiden liegt bei etwa 15%. Nah an dem Ergebnis vom Master ist die erste Replica (3308). Überraschenderweise folgen danach die anderen beiden Replicas auf den Ports 3309 und 3310. Die Writegeschwindigkeit ist in beim Fall ohne und mit Replikation auf einem ähnlichen Niveau, wobei die Version mit Master-Replica etwa 10% langsamer ist. In diesem Benchmark wurde der Threads-Wert konstant auf 1 gesetzt, sodass die Leistung eines einzelnen Threads ersichtlich wird. Nun betrachten wir ein praxisnahes Szenario:

Wenn 8 Nutzer gleichzeitig und unabhängig voneinander Daten abfragen, ohne Änderungen vorzunehmen, ergibt sich eine Last von 8 parallelen Threads. Dies kann entweder von einem einzelnen MySQL-Server mit 8 Threads verarbeitet werden oder mit dem Master-Replica-Ansatz. In diesem Fall teilen sich der Master und drei Replicas die Last, sodass jeder von ihnen 2 Threads verarbeitet. Dadurch ergibt sich erneut eine Gesamtanzahl von 8 parallelen Threads ($4 \times 2 = 8$). Nach diesem Prinzip lassen wir den Benchmark für die folgende Threadanzahlen laufen: 8, 16, 32.

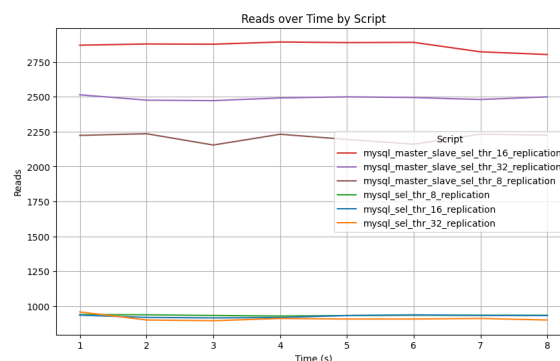


Abbildung 7.3: Vergleich von 8 Threads an Single-Server und jeweils 2 an die unters. Ports. Betrachten wir das Ergebnis aus 7.3, werden die Vorteile der Replikation deutlich. Unabhängig von der Anzahl der Threads liegen die Werte mit Replikation stets deutlich über denen des Single-Server-Ansatzes. Die Performance des Single-Servers bleibt unabhängig von der Threadanzahl nahezu konstant. Bei der Replikation zeigt sich, dass die Kombination von 16

Threads, verteilt auf den Master und 3 Replicas, am effizientesten ist. Es folgt der Benchmark mit 32 Threads, während der Vergleichstest mit 8 Threads im Vergleich zu den anderen Master-Replica-Ansätzen am ineffizientesten ist. Dennoch sind selbst die 8 Threads mit Replikation deutlich schneller als alle Varianten des Single-Server-Ansatzes.

Im letzten Vergleich verwenden wir ausschließlich den Master-Replica-Ansatz und vergleichen die unterschiedlichen Binlog-Formate, die wir aus 7.2 kennen. Um Variationen zu begrenzen, betrachten wir nur eine Replica pro Master. Daraus ergeben sich sechs unterschiedliche Leseergebnisse, da für jedes der drei Formate sowohl der Master- als auch der Replica-Port abgefragt wird. In der Grafik 7.4 können wir sehen, dass die Unterschiede kaum erkennbar sind bei verschiedenen Binlog-Formaten und Ports. Und auch die Schreibgeschwindigkeiten verhalten sich bei beiden Varianten sehr ähnlich.

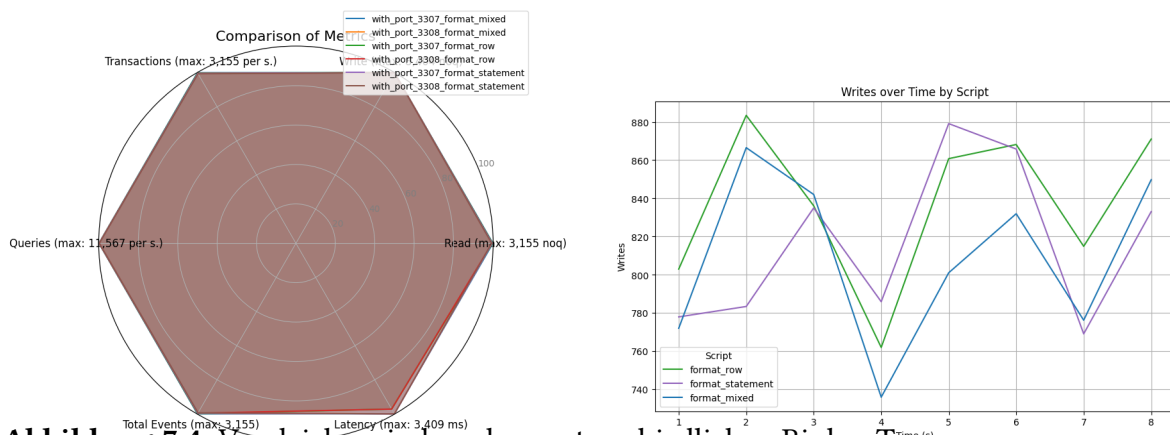


Abbildung 7.4: Vergleich zwischen dem unterschiedlichen Binlog-Typen

Die Schlussfolgerungen, die wir aus den Messungen gewinnen können, sehen dabei wie folgt aus. Wir sehen, dass durch Replikation, anders als beispielsweise mit Indexen oder Partitionen, bei 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 daher signifikante Vorteile bieten. Die Auswahl des Binlog-Formats hat bei unserem 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

Im Laufe dieser Bachelorarbeit haben wir unterschiedliche Themen behandelt. Dieses Kapitel soll dazu dienen, eine allgemeine Zusammenfassung zu erhalten.

Am Anfang der Arbeit haben wir eine Thematik zu Datentypen besprochen, die bereits beim Datenentwurf wichtig zu bedenken ist. Beim Datenbankentwurf muss man die ermittelten Anforderungen aus Interviews mit Stakeholdern verwenden, um einen konzeptionellen Entwurf, beispielsweise in Form eines ER-Modells, zu erstellen (Shortcut-Autorenteam, 2023). Danach wird aus dem konzeptionellen Entwurf ein Logischer in Form eines Relationenschemas. Zu dem logischen Entwurf gehört neben der Normalisierung von Tabellen auch die Wahl der korrekten Datentypen. Mithilfe der Benchmarks aus 3 haben wir zum einen festgestellt, dass man den kleinstmöglichen Datentypen für eine Spalte deklarieren sollte. Dazu muss man zunächst einmal festlegen, welchen Bereich an Werten abgebildet werden muss und anhand dieser Entscheidung kann man den passenden Typen wählen. Dabei ist durchaus von Vorteil, dass bei einer Verschätzung der akzeptierten Werte der Typ ohne viel Aufwand verändert werden kann. Beim Betrachten der numerischen Datentypen fiel auf, dass DECIMAL und BIGINT die ineffizientesten waren. Bei zeichenkettenbasierten Typen ist die Wahl auch einfach zu treffen, da in den meisten Fällen der Typ VARCHAR am schnellsten ist. Nur wenn eine Spalte häufiger aktualisiert als abgefragt wird, kann es sinnvoll sein, den Typ CHAR in Erwägung zu ziehen. Einer der anderen Leitsätze bei der Wahl der Datenformate ist, dass man die simple Datenstruktur wählen sollte. So war in unserem Vergleich INT in etwa 50% schneller als CHAR. Zu guter Letzt sollte auch bedacht werden, dass man nicht nur aus Performancegründen, sondern auch zur Wahrung der Datenintegrität und -konsistenz an so vielen Stellen wie möglich NOT NULL definieren sollte.

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

Zunächst haben wir bei der Indexierung gesehen, wie effektiv diese sein kann. Wir haben auch die beiden Typen B-Tree- und Hash-Indexierung miteinander verglichen und gesehen, dass der Hash-Index, wenn er bei einer Abfrage verwendet, deutlich effektiver ist, als der B-Baum-Index. Auf der anderen Seite wird der B-Baum-Index bei deutlichen mehr Abfragen eingesetzt, 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 erstellt diese. Nach einigen Tagen oder Wochen bietet es sich an, dass man eine Kontrolle durchführt und abhängig vom Resultat einige Indizes wieder entfernt und Neue hinzufügt. Ein ähnliches Vorgehen ist auch beim Einsatz von Views nützlich.

Wie die Benchmarks gezeigt haben, wirken sich virtuelle Views nicht auf die Performance aus. Dafür eignen sich virtuelle Sichten hervorragend für Gewährleistung von Rechtemanagement in einer Organisation, da die Daten nicht physisch gespeichert werden und somit keine Redundanzen entstehen. Materialisierte Sichten hingegen werden auf der Festplatte gesichert und bieten 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, wenn sie schon beim Datenbankentwurf Gedanken über Sichten macht, aber es ist auch möglich, dass sie, 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, andere sogar die inkrementelle Auffrischung, während bei einigen materialisierte Sichten durch dedizierte Tabellen in Kombination mit Triggern nachgebildet werden müssen. Bei den Tests ist jedoch deutlich geworden, dass die native Implementierung in Postgres einen klaren Performancevorteil gegenüber der Implementierung mit Triggern in MySQL bietet. Daher ist es durchaus sinnvoll, bei der Auswahl des DBMS diesen Aspekt zu berücksichtigen. Zuallerletzt muss auch hier erwähnt werden, dass die Pflege von materialisierten, nicht virtuellen Sichten einen negativen Einfluss auf die Effizienz hat.

Bei Partitionen gibt es weniger zusätzlichen Aufwand als bei Indizes oder Sichten, da die Datensätze nicht in einer einzigen Tabelle, sondern in mehreren verteilten Partitionen gespeichert werden. Wenn die Datenbankoperationen ausgeführt werden, muss zunächst die Partition oder die entsprechenden Partitionen ermittelt werden, die die angeforderten Daten enthalten. Es ist möglich, dass eine Datenbankumgebung nicht für die Partitionierung geeignet ist. Normalerweise ist ein Merkmal, das für die Partitionierung spricht, ein natürliches Trennkriterium wie beispielsweise ein Zeitstempel oder geografische Regionen, da dadurch eine logische Aufteilung der Daten ermöglicht wird. Abhängig vom Trennkriterium muss man sich für einen der Partitionstypen entscheiden: Range, List oder Hash. Der Vorteil der Partitionierung ist, dass bei einer Abfrage nur die relevanten Partitionen durchsucht werden müssen, anstatt die gesamte Tabelle zu scannen. Dies wird als Pruning bezeichnet und

steigert die Abfragegeschwindigkeit erheblich. Es gibt aber einige Probleme bei diesem Pruning, da es nicht mit allen Operatoren funktioniert. Bei der Range-Partitionierung mit einem Zeitstempel als Partitionsschlüssel kann es zu Problemen mit dem YEAR-Operator kommen. Obwohl sie dasselbe Ergebnis wie eine Bereichsabfrage nach dem ersten und letzten Tag eines Jahres liefert, kann die Variante mit YEAR nicht gepruned werden. 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 wird. Und bei der Hash-Partitionierung hat sich herausgestellt, dass mehr Partitionen benötigt werden, desto aufwendiger wird die Suche innerhalb der Partitionsstruktur und desto schlechter die Performance.

Im letzten Benchmark haben wir den Einfluss der Replikation in Form des Master-Replica-Ansatzes analysiert. Anders als bei der Partitionierung werden bei der Replikation vollständige Kopien der gesamten Datenbank auf mehreren Servern erstellt. Denn wenn Änderungen am Master vorgenommen werden, dann werden diese durch verschiedene Threads an die Replicas übertragen. Dadurch wird die Verfügbarkeit und Ausfallsicherheit erhöht, weshalb Replikation häufig in Verbindung mit Backups eingesetzt wird. Um die Performance zu testen, haben wir mit einer festgelegten Anzahl an Threads einen Sysbench-Test durchgeführt, einmal auf einem Single-Server und einmal aufgeteilt auf den Master und die Replicas. Anschließend haben die Ergebnisse der Replicas und des Masters addiert und gesehen, dass der Single-Server-Ansatz langsamer war. Genau andersherum sieht es aus, wenn man die Performance mit nur einem einzigen Thread überprüft. Die Auswahl des Binlog-Formats hat bei keinem Benchmark zu keinen Performanzgewinnen geführt. Allerdings konnten wir auch hier Nachteile beim Einfügen der Daten erkennen, da das erneute Kopieren auf die Replicas die Abfrage weniger performant machte. Replikation ist daher kein geeigneter Lösungsansatz, wenn nur ein Nutzer auf die Datenbank zugreift. Sie wird vielmehr relevant, wenn eine hohe Last auf dem System besteht, die auf mehrere Server verteilt werden muss. Damit ähnelt die Replikation den zentralen Bestandteilen von NoSQL-Datenbanken, da diese auch Daten auf mehrere Knoten verteilen (auch horizontalen Skalierung genannt).

Literatur

- DataScientest. (2023). *SQL TRIGGER zur Automatisierung deines DBMS*. Verfügbar 31. Januar 2025 unter <https://datascientest.com/de/sql-trigger-zur-automatisierung-deines-dbms>
- Garcia-Molina, H. (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>
- Luber, D.-I. (S. (2018). *Was ist ACID?* Verfügbar 26. Januar 2025 unter <https://www.bigdata-insider.de/was-ist-acid-a-776182/>
- Luber, D.-I. (S. (2023). *Was ist Scale-Out Storage?* Verfügbar 13. Februar 2025 unter <https://www.storage-insider.de/was-ist-scale-out-storage-a-c2251e6e6a1381fd5e7647120b2d2e47/>
- Matzer, M. (2019). *Der Greedy-Algorithmus*. Verfügbar 28. Februar 2025 unter <https://www.bigdata-insider.de/der-greedy-algorithmus-a-843043/>
- 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). *26.2 Partitioning Types*. Verfügbar 17. Februar 2025 unter <https://dev.mysql.com/doc/refman/8.4/en/partitioning-types.html>
- Oracle. (2025c). *27.3.1 Trigger Syntax and Examples*. Verfügbar 27. Januar 2025 unter <https://dev.mysql.com/doc/refman/8.4/en/trigger-syntax.html>
- Ouko, A. (2025). *SQL Materialized View: Verbesserung der Abfrageleistung*. Verfügbar 26. Januar 2025 unter <https://www.datacamp.com/de/tutorial/sql-materialized-view>
- Schwartz, B., Zaitsev, P., & Tkachenko, V. (2012). *High performance MySQL: optimization, backups, and replication*. „O'Reilly Media, Inc.“
- Shortcut-Autorenteam. (2023). *Grundlagen des Datenbankentwurfs, ER-Modell, Normalisierung*. Verfügbar 27. Februar 2025 unter <https://entwickler.de/datenbanken/1-grundlagen-des-datenbankentwurfs-er-modell-normalisierung>

Anhang

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

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

Eigenständigkeitserklärung

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

Performance - Optimierung von Datenbanken

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

Hamburg, 21. Dezember 1940