**Tool Auswahl**

Die Grundlage für diese Bachelorarbeit ist das Verhalten der MySQL – Datenbank in Bezug auf die unterschiedlichen Aspekte, die im Rahmen dieser Arbeit behandelt werden. Der folgende Abschnitt beschäftigt sich mit Umsetzung, um dieses Verhalten messbar und veranschaulich mithilfe von Grafiken zu machen. Damit wir die Kennzahlen für bestimmte Abfragen an die MySQL – Datenbank bestimmen können, brauchen wir ein zentrales Tool. Dieses Tool ist dafür verantwortlich ist die Benchmark - Tests durchzuführen.

Meine Entscheidung ist dabei schlussendlich auf Sysbench gefallen. Sysbench ist ein Open-Source-Tool, das ein skriptfähiges, multi-threaded Benchmark-Tool ist, das auf LuaJIT basiert. Es wird hauptsächlich für Datenbankbenchmarks verwendet, kann jedoch auch dazu eingesetzt werden, beliebig komplexe Arbeitslasten zu erstellen, die keinen Datenbankserver erfordern. Sysbench analysiert dabei Metriken, wie unter anderem Transaktionen pro Sekunde, Latenz und Anzahl an Threads. Dabei kann man genauer spezifizieren, wie oft diese Metriken geloggt werden sollen. Sysbench ist dabei nicht auf ein einziges Datenbanksystem eingeschränkt, sondern man kann sich zwischen vielen unterschiedlichen System entscheiden.

Im Zuge der Wahl des Benchmark – Tools habe ich auch andere Benchmarking-Tools betrachtet, wie beispielsweise Benchbase oder mybench. Im Vergleich zu diesen Tools bietet Sysbench jedoch die Vorteile der höheren Skriptfähigkeit und Flexibilität. Damit ist gemeint, dass bei Sysbench das erste Projekt mit mehr Aufwand verbunden ist als bei den Alternativen. Wenn man aber ein Projekt erstmal erstellt hat, dann ist es sehr individuell und man kann schnelle Änderungen hervornehmen. In dem Kapitel (TODO (Daniel): Kapitel mit Join Typ) betrachten wir ein beispielhaftes Projekt mit Sysbench, bei dem der Einfluss von unterschiedlichen Datentypen als Join-Operator zwischen zwei Tabellen verglichen wird. Wenn wir später die Performance von unterschiedliche Index-Typen betrachten, dann müssen wir nur an wenigen Stellen Veränderungen durchführen, die in dem Kapitel genauer besprochen werden.

Ein weiterer Vorteil von Sysbench ist, dass es als de facto Standard im Bereich der Datenbankbenchmarks angesehen wird. Durch diese Position gibt es viele aktive Nutzer und dadurch bedingt viele verfügbaren Ressourcen. Vorteile der anderen Tools sind jedoch die weniger präzise Steuerung der Ergebnisraten und der Transaktionen von Sysbench. Zudem ist Sysbench auf das Minimale beschränkt, was den Output angeht, da es, wie schon erwähnt, nur eine Reihe von Log-Dateien gibt und die Visualisierung der Ergebnisse muss vom Benutzer selbst mithilfe von anderen Tools umgesetzt werden. Anders sieht dies bei dem Tool mybench aus, da es dort die Möglichkeit gibt in Echtzeit umfassende Abbildungen zu betrachten. Obwohl dieses Feature sehr hilfreich ist, bin ich nach Abwägung der Vor- und Nachteile zu dem Entschluss gekommen, dass die einfachere Bedienung und die Tatsache, dass Sysbench der de facto Standard ist, für mich überwiegen, weshalb ich mich für Sysbench entschieden habe.

Nichtsdestotrotz kann nicht komplett auf Graphen verzichtet werden, da Entwicklungen im Laufe einer Zeitmessung in einem Kurvenverlauf deutlich besser zu erkennen sind als in einer Log - Datei. Anhand der reinen Zahlen lassen sich unter Umständen Trends von zwei oder etwas mehr unterschiedliche Messungen erkennen, aber besonders wiederkehrende Trends werden aus der schriftlichen Form nicht schnell ersichtbar. Ganz anders sieht dies bei Graphen mit einer Zeitachse aus. Dort werden sofort Trends ersichtlich und auch der Vergleich zwischen den unterschiedlichen Messungen erfolgt deutlich besser.

Um die Kennzahlen, die mithilfe von Sysbench ermittelt worden sind, in eine grafische Darstellung umzuwandeln, gibt es unterschiedliche Tools, die wiederum einige Vor - und Nachteile mit sich bringen. Das erste mögliche Tool stellt Gnuplot dar, mit dem sich CSV – Dateien sehr gut darstellen lassen. Wenn man aber nur bestimmte Spalten aus der Tabelle darstellen lassen, dann kommt man schnell an seine Grenzen. Deshalb habe ich mich für eine anpassungsfähigere Alternative entschieden, denn die Transformationen und die grafische Darstellung wird mithilfe eines Python Scripts umgesetzt. Für die grafische Darstellung sind dabei die Libraries pandas (//TODO(Daniel): find source) und matplotlib.pyplot (//TODO(Daniel): find source) verantwortlich.

**Einführung in die Tools**

Als allererstes muss der MySQL – Server gestartet sein. Dabei ist es egal, ob dies lokal auf dem Rechner oder über einen Docker in eines GitHub CI/CD-Workflows erfolgt. Das Wichtigste dabei ist es, dass man sich die Zugangsdaten, bestehend aus User - und Passwortdaten, zwischen speichert, da diese gebraucht werden, um den Benchmarktest mit Sysbench zu starten. Nachdem das RDBMS gestartet worden ist, muss zudem eine Datenbank erstellt werden. Dies könnte unter anderem so aussehen:

**Beispiel 1: database.sql**

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

**Auflistung der Argumente**

**Erklärung des Vorgehen anhand des Join\_Typ - Beispiels**

In dem vorausgegangenen Kapitel (TODO (Daniel): add name + reference) wurde das Tool Sysbench und seine Funktionsweise anhand eines Demo - Projekts näher erläutert. Damit die Reihenfolge und die Bedeutungen der unterschiedlichen Methoden (prepare → run → cleanup) sowie die Vorgehensweise zur Erstellung unserer Grafiken deutlich geworden. Das bisherige Problem ist aber, dass wir bei dem dargelegten Beispiel keine Kontrolle über die getesteten Daten haben. Wenn man sich die Logs genauer anschaut, dann sieht man, dass man über die Parameter an den Sysbench – Befehl die Anzahl der erstellten Tabellen und eingefügten Datensätze von außen steuern kann, aber die genaue Implementierung können wir auf diese Weise nicht steuern. Genau für diese Anwendungsfälle gibt es die Möglichkeit ein Lua - Skript als Parameter beim Sysbench - Aufruf mit anzugeben. In diesen Lua - Dateien können die Implementierungen der einzelnen Methoden selbstständig gewählt werden.

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

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

**01\_Create\_Table\_Kunde**

**02\_Create\_Table\_Bestellung**

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:

**03\_Prepare\_Query**

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

**04\_CleanUp\_Query**

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

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

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

**05\_Insert\_Query**

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.

**06\_Select\_Query**

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:

local length = 10

Die Zeile mit importierten Länge:

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

Den Orchestrator - Script ruft man wie folgt auf:

**07\_Orchestrator\_Command**

Die Parameter haben folgende Funktion:

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

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

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

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

Wenn wir den Befehl ausführen, wird der Output-Ordner an der gewünschten Stelle erstellt. In diesem Ordner werden verschiedene Grafiken generiert, die die Ergebnisse visualisieren. Dabei gibt es zwei unterschiedliche Arten von Grafiken. Die erste Art von Grafik ist ein Zeitreihendiagramm, welches auf der X-Achse den zeitlichen Verlauf zeigt. Auf der Y-Achse werden in einigen Diagrammen die unterschiedlichen Metriken für jedes einzelne Skript dargestellt, während andere Diagramme die Werte einer bestimmten Metrik auf der Y-Achse zeigen und dabei die Ergebnisse verschiedener Skripte vergleichen. Dadurch können beispielsweise die Metriken "Reads" und "Writes" analysiert werden, um herauszufinden, welches Skript in diesen Bereichen besser abschneidet. Danach wird der Output - Ordner erstellt und die Spaltenüberschriften in die CSV – Dateien geschrieben. Danach beginnt erst das eigentliche Durchgehen der unterschiedlichen Skripte, die unter dem Argument -script angegeben wurden. Zunächst schreibt man die einzelnen Dateien nach dem obigen Schema (TODO Daniel) auf, denn als Argument wurde nur der oberste Ordner angegeben. Als nächstes kommt eine Fallunterscheidung, die überprüft, ob dieses Skript exportierte Variable nutzt oder nicht. Für den Fall, dass keine Variablen exportiert werden (z.B. int\_queries) wird einfach die Prepare – Funktion aufgerufen, dann process\_script\_benchmark und anschließend die Cleanup - Funktion. Wenn aber Variablen exportiert werden, dann müssen weitere Zwischenschritte umgesetzt werden. Und zwar müssen alle Kombinationen zwischen den verschiedenen exportierten Variablen generiert werden. Wenn es drei Variablen gibt, von denen 2 jeweils 2 Werte und eine letzte nur einen Wert hat, dann gibt es 2 \* 2 \* 1 = 4 unterschiedliche Kombinationen. Als nächstes muss man für alle diese Kombinationen die Schritte ausführen, die man schon bei der Variante ohne exportierte Variable ausgeführt hat und dabei darf man nicht vergessen die Variablen an sich zu exportieren.

**08\_Main\_Loop.sh**

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

**09\_Process\_Script\_Benchmark.sh**

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

**10\_Run\_Benchmark.sh**

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

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

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

**PNGs**

Aus den Grafiken, die für ein Skript alle Metriken veranschaulichen, kann man möglicherweise Datenfehler erkennen. So springt bei Abbildung (int\_queries.png) die Latenz bei einigen Messpunkten von 0 ms auf einen deutlich erhöhten Wert und danach wieder auf 0 ms zurück. Ansonsten aber sind die anderen Metriken auf einem konstanten Level, und es gibt wenige Schwankungen. Bei der Abbildung (varchar\_queries\_length\_1.png) sieht dies sehr ähnlich aus, und auch dort schwankt die Latenz etwas mehr. Wenn wir jetzt die drei Skripte miteinander vergleichen wollen, können wir die Abbildungen Reads.png und Writes.png heranziehen. Was die Lesegeschwindigkeit angeht, kann man erkennen, dass int\_queries am meisten Reads hat, als Nächstes kommt varchar\_queries\_length\_1 und dann varchar\_queries\_length\_64. Damit sind die Abfragen, wie wir erwartet haben, bei int\_queries am schnellsten, und je länger der String wird, desto langsamer werden die Abfragen. Bei den Schreibgeschwindigkeiten sieht das schon etwas anders aus, wobei es hier zunächst bei allen eine langsamere Startphase gibt. Anschließend an diesen Cold Start liegt das Niveau von int\_queries am höchsten, also auch am schnellsten. Die beiden varchar\_queries sind hier aber überraschenderweise auf einem ähnlichen Niveau. Bei der Abbildung (statistics.png) kann man die Effekte der Lese- und Schreibgeschwindigkeiten auch erkennen. Es fällt auch auf, dass, anders als bei Reads, Writes, Queries, die Latenz bei schnelleren Queries geringer ist und nicht der höchste Wert der Beste ist.

**Github Actions**

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

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

**11\_Pattern.json**

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

**12\_Workflow.yaml**

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

Zum einen kann man die Dependencies, die man installieren muss, mithilfe von GitHub Cache (<https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows#comparing-artifacts-and-dependency-caching>) cachen. Dies bietet sich besonders deswegen an, weil sich die Abhängigkeiten über die Workflows hinweg kaum verändern. Wenn sich etwas verändert, dann kann man beispielweise die „requirements.txt“ verändern und schon werden einmalig alle Dependencies neu installiert und dann im Cache gespeichert. Wenn sich das zu dem nächsten Workflow nichts an den Abhängigkeiten verändert, wird der Cache gefunden und heruntergeladen. Wenn man jetzt die Abhängigkeiten herunterladen will, dann wird der heruntergeladene Cache benutzt. Der Zeitgewinn in unserem Beispiel ist nur sehr gering und liegt bei nur wenigen Sekunden pro Workflow.

Deutlich mehr Zeit und Ressourcen kann man aber sparen, wenn man zwischen zwei unterschiedlichen Arten von Dateien unterscheidet. Denn zum einen gibt es Dateien, die die Ergebnisse von allen Skripten beeinflussen. Dazu gehören das Workflow - Skript und die JSON - Datei, aber auch das Orchestrator - Skript und die darin verwendeten Python - Skripte. Die Ordner an sich, die in der JSON angegeben werden, die beeinflussen nur sich selbst und nicht die anderen Skripte. Beispielweise wenn ich in Projekt A die Anzahl an Zeilen ändere, die 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 neudurchgeführt 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 einen 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 den jeweiligen Skript aus der JSON benötigt werden. Diese beiden Hashs kann zusammen mit den Testtypen kombinieren, damit bekommt die folgende Struktur für den Namen:

NAME="${{ matrix.test-type }}-${{ env.hash }}-${{ env.general\_hash }}"

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

Die letzte Frage, die sich stellt, ist wo man die Ordner mit den berechneten Namen speichert und beim nächsten Run wieder herunterlädt. 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 (https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows#comparing-artifacts-and-dependency-caching). Eine andere interessante Lösung kann auch das Nutzen von expliziten Branches nur für die Sicherung der Dateien seien. Das Problem ist hier, dass es manchmal durch bestimmtes Timing zu Problemen beim Pushen kommen kann, da zufällig ein anderer paralleler Workflow in der Zeit zwischen Rebase, Commit und Push den Code verändert hat, wodurch nach verhindertem Push erneut ein Rebase durchgeführt werden muss. Außerdem muss man dafür der GitHub Action Schreibberichtigungen geben. Des Weiteren eignen sich auch Cloud - Speicherlösungen sehr gut, um die Ordner zu speichern und wieder herunterzuladen. Dazu gehören von Google Cloud Storage (GCS), AWS S3 oder MS Azure Storage, die sich zusammen mit GitHub Artifacts am besten eignen.