Performance Analyse der Optimierung von Datenbankabfragen in der HANA Calculation Engine

Projektarbeit 2

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Wirtschaftsinformatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Jared Heinrich

Abgabedatum:	26. August 2024
Bearbeitungszeitraum:	06.05.2024 - 25.08.2024
Matrikelnummer, Kurs:	5101479, WWI22SEA
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Unternehmensbetreuer:	Rainer Agelek
Wissenschaftlicher Betreuer:	Prof. Dr. Hans-Henning Pagnia

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema:

> *Performance Analyse der Optimierung von Datenbankabfragen in der HANA Calculation Engine*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, den 8. August 2024

Heinrich, Jared

Abstract

- Deutsch -

Das ist der Abstract.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CE Calculation Engine

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

2 Grundlagen

2.1 Performance Analyse

Benchmarking

Profiling

2.2 Notwendige Grundlagen aus der Graphentheorie

Definition 1 gibt eine Definition für einen gerichteten Graphen (vgl. Knebl 2021, S. 220).

> **Definition 1.** *Ein gerichteter Graph ist ein Tupel* $G = (V, E)$. V *heißt Menge der Knoten.* E *heißt Menge der gerichteten Kanten. Es gilt* $V \neq \varnothing$ *und* $E \subset V \times V \setminus \{(v, v) | v \in V\}$. *Zwischen zwei Knoten* $u, v \in V$ *gibt es eine Kante mit dem Anfangspunkt* u *und dem Endpunkt* v, *wenn* $(u, v) \in E$.

Definition 2 gibt eine Definition für einen Pfad in einem Graphen (vgl. Knebl 2021, 221f).

> **Definition 2.** *Ein Pfad* P *in* G *ist eine Folge von Knoten* $v_0, \ldots, v_n$. *Dabei muss* $\forall i \in \{0; \ldots; n-1\} : (v_i, v_{i+1}) \in E$ *gelten.* v_0 *heißt Anfangspunkt von* P, v_n *heißt Endpunkt von* P *und* n *heißt Länge von* P.

Definition 3 gibt eine Definition für Zyklen in einem gerichteten Graphen und definiert den Begriff des azyklischen Graphen (vgl. Knebl 2021, S. 222).

> **Definition 3.** *Ein Pfad heißt geschlossen, wenn* $v_0 = v_n$. *Ein geschlossener Pfad heißt einfach, wenn* $\forall i, j \in \{0; \ldots; n-1\}, i \neq j : v_i \neq v_j$ *gilt. In einem gerichteten Graphen*

> *heißt ein einfach geschlossener Pfad mit $n \geq 2$ auch Zyklus. Ein Graph heißt azyklisch, wenn er keine Zyklen besitzt.*

Definition 4 definiert den Begriff des gewichteten Graphen (vgl. Knebl 2021, S. 253).

> **Definition 4.** *G heißt gewichtet, wenn es eine Abbildung $g : E \to \mathbb{R}$ gibt, welche jeder Kante ein Gewicht zuordnet. Für $e \in E$ heißt $g(e)$ Gewicht von e.*

Definition 5 definiert die Begriffe Quelle und Senke (vgl. Knebl 2021, S. 306).

> **Definition 5.** *Ein Knoten $q \in V$ heißt Quelle, wenn $\forall p \in V : (p, q) \notin E$, q also nach Definition 6 keine Elternknoten hat. $s \in V$ heißt Senke, wenn $\forall p \in V : (s, p) \notin E$, s also nach Definition 6 keine Kindknoten hat.*

Definition 6 definiert die Begriffe Kindknoten, Elternknoten und Subknoten.

> **Definition 6.** *Für zwei Knoten c und p gilt, c ist Kindknoten von p, wenn $(p, c) \in E$. Umgekehrt gilt, wenn $(p, c) \in E$, p ist Elternknoten von c. c ist Subknoten von p, wenn $\exists P : (v_0 = p) \wedge (v_n = c)$, es also einen Pfad von p nach c gibt.*

3 Analyse des aktuellen Standes der Calculation Engine

3.1 Calculation Engine

Calculation Engine (CE) ist sowohl eine Abteilung als auch eine Softwarekomponente der SAP eigenen HANA-Datenbank . Der genaue Funktionsumfang von HANA ist in dieser Arbeit nicht weiter relevant. Eine Möglichkeit um Daten mit der HANA-Datenbank zu analysieren, sind jedoch sogenannte Kalkulationssichten. Da nur diese Kalkulationssichten in der Arbeit weiter betrachtet werden, werden sie im Folgenden auch als Datenbankabfragen bezeichnet. (Vgl. SAP 2024a) Diese Kalkulationssichten werden von der CE bearbeitet. Das Vorgehen wird in Abschnitt 3.2 genauer beschreiben.

3.2 Modelle und Datenbankabfragen

In der CE werden diese Datenbankabfragen durch Modelle dargestellt. Für jede Abfrage wird zuerst das dazugehörige Modell instanziiert. Anschließend wird dieses Modell optimiert, um die Ausführungsdauer der Abfrage zu minimieren. Diese optimierte Abfrage, wird dann auf der Datenbank ausgeführt. Für die Ausführung ist dabei jedoch nicht zwingend die CE zuständig (vgl. SAP 2024c). Allgemein kann man ein Modell als azyklischen gerichteten Graphen nach Definition 1 und Definition 3 beschreiben. Des Weiteren ist zu beachten, dass dieser Graph nur eine Quelle nach Definition 5 hat, welche auch als Abfrageknoten bezeichnet wird. Diese Definitionen reichen jedoch nicht aus, da zwischen verschiedenen Arten von Knoten unterschieden wird, welche jeweils verschiedene Informationen beinhalten. Allgemein werden die Knoten in der CE in zwei Gruppen unterschieden, Datenquellen und Sichtknoten. Es gibt verschiedene Datenquellen, zur Vereinfachung werden in dieser jedoch Arbeit nur Table-Knoten betrachtet. Deshalb werden Datenquellen im Folgenden auch Tabellenknoten genannt. Die andere Gruppe sind die Sichtknoten. Zu diesen gehören z. B. Projection, Aggregation, Join und Union.

Auch hier gibt es zwar noch Weitere, diese Arbeit beschränkt sich jedoch auf diese. Diese Modelle können zwar wie in Definition 1 beschreiben als eine Menge von Knoten und Kanten dargestellt werden, gespeichert wird jedoch eine Liste an Knoten, von welchen jedem seine Kind- und Elternknoten zugeordnet sind. Die Kindknoten werden dabei als Eingangsknoten und die Elternknoten als Ausgangsknoten bezeichnet. Jeder Knoten kann beliebig viele Ausgangsknoten haben, wobei es wie bereits beschreiben nur einen Knoten mit 0 Ausgangsknoten gibt. Die Anzahl der Eingangsknoten unterscheidet sich dabei je nach Knotentyp. Projection- und Aggregation-Knoten haben genau einen, Join-Knoten genau zwei, Union-Knoten zwei oder mehr und Table-Knoten haben keinen Eingangsknoten. Bei den Eingangsknoten kann es sich um Tabellenknoten sowie auch Sichtknoten handeln. (Vgl. SAP 2024b)

Abbildung 3.1 zeigt ein einfaches Beispielmodell, welches zur Veranschaulichung dient. Das Modell besteht aus fünf Knoten, dem Abfrageknoten, zwei Sichtknoten und zwei Tabellenknoten. In dem Knoten steht der Name des Knotens und neben ihm steht der Knotentyp. Bis auf den Abfrageknoten haben in diesem Beispiel alle Knoten eine Zahl als Namen. Der Abfrageknoten sowie alle Tabellenknoten sind zusätzlich farblich markiert. Knoten 1 ist ein Join-Knoten. Er hat zwei Eingangsknoten, den Table-Knoten 2 und den Projection-Knoten 3. Der einzige Ausgangsknoten ist in diesem Fall der Abfrageknoten, dieser ist hier vom Typ Aggregation. Der Abfrageknoten kann jedoch alternativ auch vom Typ Projection sein.

Zusätzlich zu den Eingangs- und Ausgangsknoten werden in jedem Knoten noch weitere Informationen gespeichert. Jeder Knoten beinhaltet eine Liste an Sichtattributen, diese legt fest, welche Sichtattribute dieser Knoten an seine Ausgangsknoten weitergibt. Bei einem Tabellenknoten sind die Sichtattribute gleichbedeutend mit den Spalten der Tabelle. Die Sichtattribute des Abfrageknotens sind die Attribute, welche im Ergebnis der Abfrage enthalten sind. Damit Sichtattribute umbenannt werden können, wird jedem Eingangsknoten eine Liste an Mappings zugeordnet. Hat der Knoten N einen Eingangsknoten E mit einem Mapping, dann bildet dieses ein Sichtattribut von E auf ein Sichtattribut von N ab.

Die verschiedenen Sichtknotentypen sind für verschiedene Operationen zuständig. Manche dieser Knoten haben noch zusätzliche Attribute, um die genaue Art und Weise der Operation festzulegen. Ein Join-Knoten kann genutzt werden, um die Ergebnisse der beiden Eingangsknoten zu verbinden.

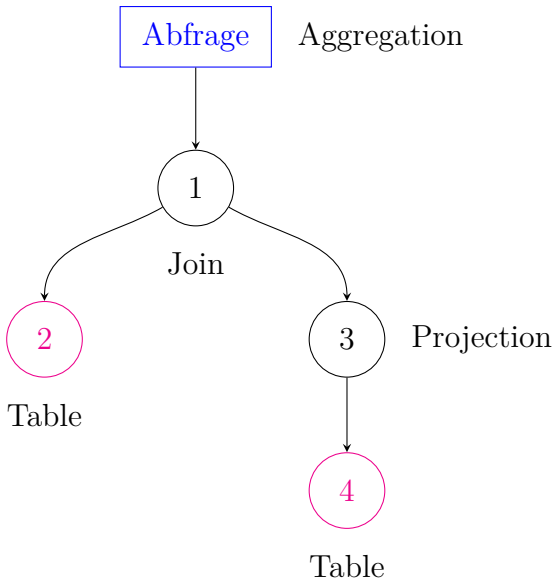


Abbildung 3.1: Darstellung eines Beispielmodells

In der CE werden bereits verschiedene Wege genutzt, um die Performance und das Verhalten der HANA-Datenbank zu analysieren.

3.3 Google Benchmark

Google Benchmark ist ein Open Source Benchmarking Tool von Google, welches es einem ermöglicht einzelne Funktionen in C++ zu benchmarken. Dazu kann man ähnlich zu den meisten Test-Frameworks drei verschiedene Codeabschnitte definieren.

Der Hauptabschnitt definiert was genau im Benchmark untersucht werden soll. Diese legt die für die Messung relevante Logik fest. Der Setup-Abschnitt wird einmal vor jeder Ausführung des Benchmarks aufgerufen. In dieser werden die Voraussetzungen für die Ausführung des Hauptabschnitts geschaffen. Man kann ihn z. B. nutzen, um Testdaten für den Benchmark zu generieren oder zu laden, da er nicht bei den Messungen beachtet wird. Der Teardown-Abschnitt wird nach jeder Ausführung des Benchmarks aufgerufen. Dieser beeinflusst, wie der Setup Abschnitt, die Messung nicht.

Des Weiteren bietet Google Benchmark die Option, einen Benchmark mehrmals mit unterschiedlichen Parametern durchzuführen. Um beispielsweise die Auswirkung der Größe des Testdatensatzes auf das Ergebnis zu beobachten.

Abbildung 3.2 zeigt eine Beispielhafte Google Benchmark Ausgabe. Benchmark ist dabei der Name des Benchmarks sowie die werte der Parameter, welche durch einen Schrägstrich getrennt sind. Time und CPU sind beide die durchschnittliche Dauer einer Ausführung des Hauptabschnitts über alle Iterationen hinweg. Der Unterschied besteht darin, welche Zeit genau gemessen wird. Time ist die tatsächlich benötigte Zeit ist, wobei Wartezeiten des Prozessors inkludiert sind. CPU ist im hingegen nur die Zeit, welche der Prozessor wirklich genutzt hat, um den Benchmark-Prozess zu bearbeiten, hierbei sind Wartezeiten also exkludiert. Dies betrifft sowohl die Wartezeiten, welche durch das Eingreifen des Scheduler auftreten (vgl. Parekh und Chaudhari 2016, S. 184), als auch Wartezeiten, welche durch z. B. Speicherzugriffe oder Ein- und Ausgabeoperationen verursacht werden. Iterations ist die Anzahl der durchgeführten Wiederholungen des Hauptabschnitts. Legt man bei der Definition des Benchmarks keine Anzahl fest, wird die Anzahl der Wiederholungen anhand der durchschnittlichen Dauer einer Iteration und der Varianz der Dauer über alle Iterationen hinweg festgelegt. (Vgl. Google 2024)

```
-----------------------------------------------------
Benchmark              Time       CPU        Iterations
-----------------------------------------------------
BM_Optimizer/8/0       213 us      212 us    3276
BM_Optimizer/16/0      271 us      271 us    2621
BM_Optimizer/32/0      389 us      389 us    1790
BM_Optimizer/64/0      739 us      738 us    1064
BM_Optimizer/128/0    1037 us     1037 us     687
BM_Optimizer/256/0    2372 us     2371 us     319
```

Abbildung 3.2: Beispielausgabe eines Google Benchmarks

Google Benchmark wird in der CE meistens genutzt, um das Verhalten der Laufzeit bestimmter Funktionen in künstlich generierten Testfällen zu vergleichen. Hierbei wird keine HANA-Instanz benötigt, da keine Operationen auf einer Datenbank ausgeführt werden, sondern nur einzelne Funktionen aufgerufen werden. Es werden folglich auch keine Testdatensätze für die Datenbank benötigt, sondern nur Daten, auf welchen man die zu analysierende Funktion aufrufen kann.

Im Vergleich zu anderen Methoden der Performance Messung, die Operationen auf einer HANA-Instanz ausführen, ist ein Google Benchmark relativ einfach. Dies liegt daran, dass sie weniger Voraussetzungen erfordern und sich lediglich auf einen kleinen Teil der Logik konzentrieren.

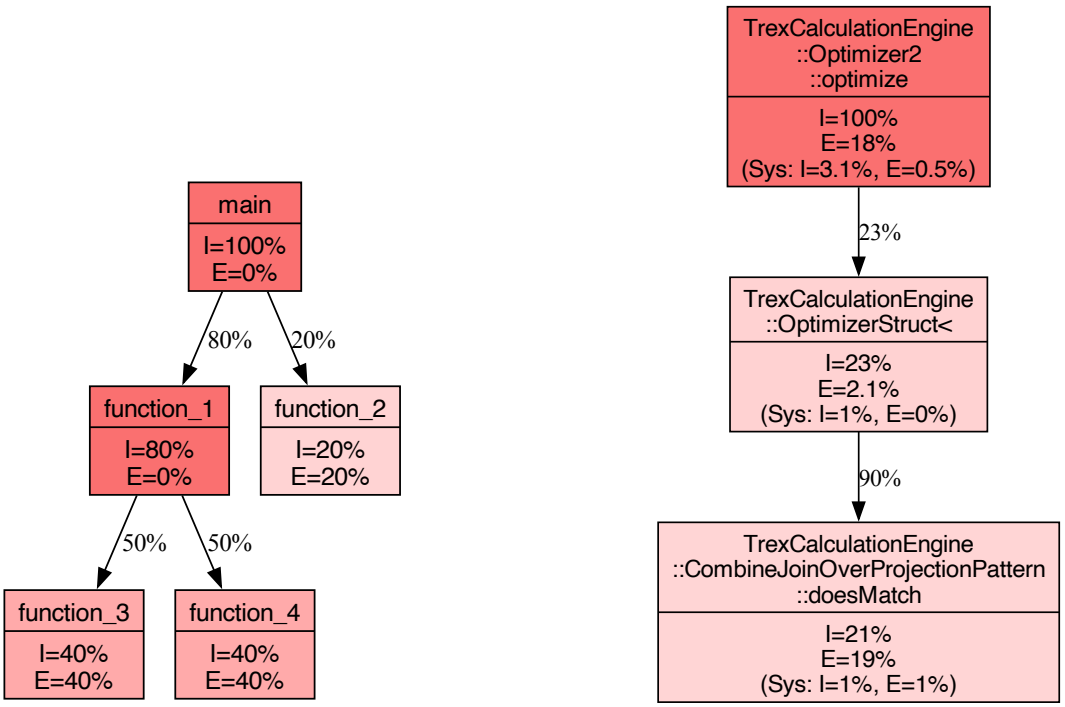
3.4 HANA-Profiler

Eine weitere Methode die Performance und das Verhalten von Software zu analysieren ist, das bereits in Abschnitt 2.1 beschriebene, Profiling.

In der CE wird hauptsächlich der „BOOSS-Profiler“, ein in HANA integrierter Profiler, verwendet. Auf diesen kann entweder manuell oder im Code zugegriffen werden. Dabei sind die wichtigsten Befehle `profiler clear` um die aktuellen Profilinginformationen zurückzusetzen, `profiler start` um den Profiler zu starten, `profiler stop` um den Profiler zu stoppen und `profiler print` um die gesammelten Profilinginformationen auszugeben.

Der Profiler kann in diesem Kontext auf zwei verschiedene Arten genutzt werden. Entweder in dem zur Laufzeit des Profilers Anfragen an eine bestehende HANA-Instanz gestellt werden. Oder der Profiler wird innerhalb eines Tests oder Benchmarks aufgerufen und es werden die dort aufgerufen Funktionen gemessen.

Die Ausgabe erzeugt dabei ist dabei zwei gewichtete gerichtete azyklische Graphen, nach Abschnitt 2.2, von denen einer die Verteilung der CPU-Zeit und der andere die Verteilung der Wartezeit, der aufgerufenen Funktionen beinhaltet. Im Folgenden werden die beiden Graphen CPU-Graph und Warte-Graph genannt. Abbildung 3.3(a) stellt eine mögliche Ausgabe des HANA-Profilers dar. Die folgende Beschreibung gilt für den CPU- als auch den Warte-Graphen. Jeder Knoten des Graphen spiegelt eine zur Laufzeit des Profilers aufgerufene Funktion wider. Jeder Knoten beinhaltet drei Informationen. Den Namen der aufgerufenen Funktion, sowie den Wert I und den Wert E. I ist der Anteil der Gesamtzeit, welcher von der Funktion des Knotens benötigt wurde, wobei die Zeiten von Funktionen, welche innerhalb der Funktion des Knotens aufgerufen wurden, inkludiert sind. E ist der Anteil der Gesamtzeit, welcher von der Funktion dieses Knotens benötigt wurde, wobei hier jedoch die Zeiten von aufgerufenen Funktion exkludiert sind. Folglich gilt für



(a) Beispielausgabe

(b) Ausschnitt aus Ausgabe für Messung

Abbildung 3.3: Mögliche Ausgaben des HANA-Profilers

alle Knoten $I \geq E$. Die Kindknoten eines Knoten K sind die Funktionen, welche von K aufgerufen wurden.

4 Konzept

Für die Untersuchung des Optimierungsalgorithmus wird ein experimenteller Ansatz statt einem analytischem gewählt, da die experimentelle Vorgehensweise es zum einen einfacher macht sehr komplexe Algorithmen zu untersuchen, zum anderen, eine experimentelle Untersuchung realitätsnähere Ergebnisse liefern kann (vgl. Bartz-Beielstein 2010, S. 3). Hierzu werden in dieser Arbeit zwei unabhängige Variablen betrachtet. Zum einen die Größe und zum anderen der Aufbau des zu optimierenden Modells, diese wurden gewählt, da sie direkt kontrolliert werden können und erwartet wird, dass sie einen großen Einfluss auf die Laufzeit haben (vgl. McGeoch 2002, S. 506). Der Aufbau wird in dieser Arbeit als Art des Modells bezeichnet. Die gemessene abhängige Variable ist die Laufzeit des Optimierungsvorgangs. Unabhängige Variablen sind Variablen, welche aktiv verändert werden, während abhängige Variablen gemessen werden (vgl. Brosius, Haas und Unkel 2022, S. 236). Damit Messergebnisse sinnvoll vergleichen werden können, darf zwischen zwei Messungen nur eine unabhängige Variable verändern werden. (vgl. Brosius, Haas und Unkel 2022, S. 236). Deshalb werden mehrere Messreihen durchgeführt, wobei innerhalb einer Messreihe der Art konstant ist, die Größe jedoch variabel ist. Für jede Art wird eine neue Messreihe begonnen. Störvariablen, also Einflussfaktoren, welche ebenfalls die abhängigen Variablen beeinflussen, jedoch während der Messung unkontrolliert auftreten, z. B. die Prozessorauslastung des Rechners, auf welchem die Messung durchgeführt wird (vgl. Brosius, Haas und Unkel 2022, S. 237). Ist der Prozessor weniger ausgelastet, dann ist die gemessene Zeit vermutlich geringer, als wenn der Prozessor stark ausgelastet ist.

Um den Einfluss dieser Störvariablen möglichst gering zu halten, wird jede Messung n mal wiederholt. Aus diesen Messwerten wird nun ein Konfidenzintervall gebildet, welches mit der Wahrscheinlichkeit $1 - \alpha$ den tatsächlichen Erwartungswert μ enthält. Dazu werden die Messwerte als eine T-verteilte Zufallsvariable X betrachtet, da n aufgrund der Dauer einer Messung nicht sehr groß gewählt werden kann. Die Varianz σ^2 von X ist dabei unbekannt und muss anhand der Stichprobe geschätzt werden, für diese Schätzung gilt: $\hat{\sigma}^2 = S^2$ (vgl Stocker und Steinke 2017, S. 528). Für das $(1 - \alpha)$-Konfidenzintervall ergibt sich deshalb nach (vgl. Stocker und Steinke 2017, S. 533):

$$[\bar{X} - t_{n-1,1-\alpha/2}\sqrt{S^2/n}, \bar{X} + t_{n-1,1-\alpha/2}\sqrt{S^2/n}]$$

Dabei ist $\bar{X}$ der Mittelwert der Stichprobe und S^2 die korrigierte Stichprobenvarianz (vgl. Stocker und Steinke 2017, S. 59, 502).

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{X})^2$$

Für die verschiedenen Arten von Modellen, werden Modelle, von reellen Abfragen betrachtet, um aus diesen, allgemeine Regeln festzulegen, mit welchen man Modelle variabler Größe aber derselben Art erzeugen kann. Die Modelle werden für die Messung künstlich erzeugt, um die unabhängigen Variablen gezielt verändern zu können. Auf die reellen Abfragen wird dabei zurückgegriffen, um die Relevanz der Messung, für reelle Szenarien zu erhöhen. (Vgl. McGeoch 2002, 500f)

Diese Messdaten werden genutzt, um festzustellen, wie sich die Laufzeit der Optimierung für Modelle bestimmter Art bei steigender Größe verhalten. Dabei ist besonders interessant, ob sich die Laufzeit zur Größe linear verhält, beziehungsweise ob sie stärker oder schwächer ansteigt. Für die Optimierung des Algorithmus sind nun die Modellarten interessant, bei welchen die Laufzeit stärker als linear ansteigt, da diese Modellarten ein besonders hohes Potenzial haben lange Laufzeiten zu verursachen. Um genauer herauszufinden, in welchem Teil des Optimierungsalgorithmus besonders viel Zeit benötigt wurde, werden diese Modelle nochmals mithilfe des in Abschnitt 2.1 beschriebenen Profilings untersucht. Dazu werden mehrere Modelle dieser Art optimiert, während der Profiler protokolliert, in welchen Methoden sich wie lange aufgehalten wurde. Anschließend muss beurteilt werden, ob die Zeit, welche in dieser Methode benötigt wird, erwartbar ist oder sie geringer sein sollte. Beziehungsweise, ob es eine Möglichkeit gibt diesen Teil des Algorithmus zu beschleunigen.

Wurde eine Optimierungsmöglichkeit gefunden und umgesetzt, kann mit einem Benchmark, welcher reelle Modelle nutzt validiert werden, ob die Veränderung eine reale Verbesserung verursacht hat.

5 Implementation der Untersuchungsmethode

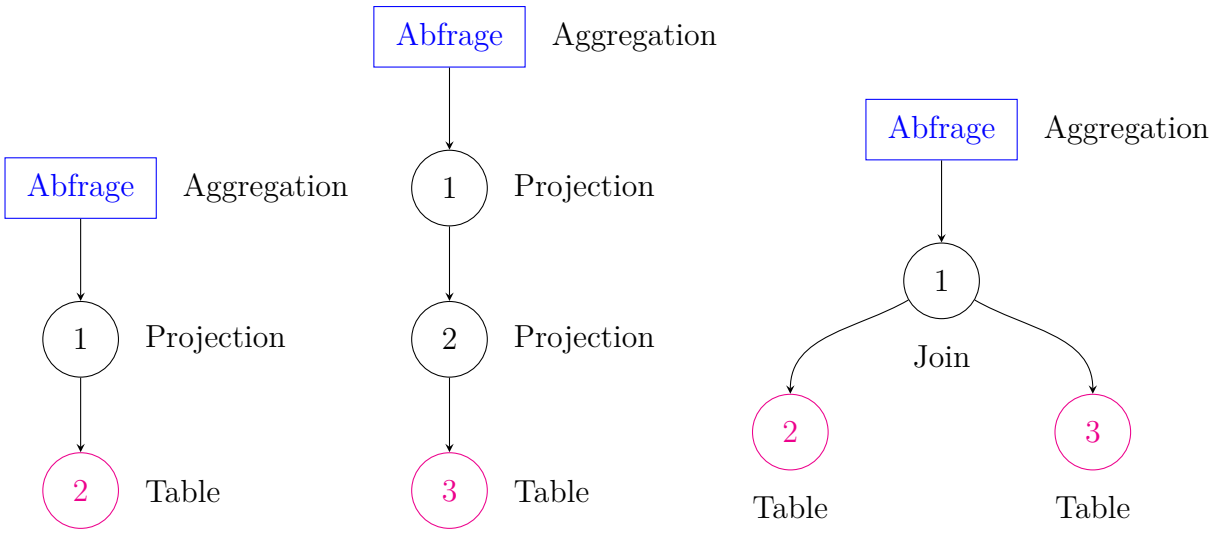
5.1 Benchmarking

Da sich die Arbeit auf die Optimierungsfunktion der CE beschränkt, kann sich auch bei den Messungen auf diese beschränkt werden. Um die Leistung einer bestimmten Funktion zu untersuchen, eignet sich, das in Abschnitt 2.1 beschriebenen, Benchmarking. Somit sind die Messungen präziser auf diese Funktion ausgerichtet und es wird Aufwand gespart, welcher auftreten würde, wenn man die Untersuchungen anhand einer HANA-Installation durchführen würde. Genauer wird das in der CE verwendete Benchmarking-Framework Google Benchmark genutzt. Dieses bietet die Möglichkeit eine Messung mit mehreren Parametern und mehreren Variationen von diesen durchzuführen. Die festgelegten unabhängigen Variablen, Größe und Art, werden durch solche Parameter widergespiegelt. Wie in Abschnitt 3.3 beschrieben, gibt Google Benchmark die Werte *CPU* und *Time* für die Laufzeit des Hauptabschnitts zurück. Für die abhängige Variable Zeit wird *Time* als Wert gewählt, da dieser auch potenzielle Wartezeiten innerhalb der Optimierungsfunktion beinhaltet. Für die beiden Parameter kann jeweils eine Menge, z. B. G für Größe und A für Art, an Werten festgelegt werden, für welche Messungen durchgeführt werden sollen.

$$G = \{2; 4; 8\} \qquad A = \{j; p\}$$

$$K = G \times A = \{(2, j); (4, j); (8, j); (2, p); (4, p); (8, p)\}$$

Das kartesische Produkt K ist eine Menge von Tupeln (g, a). Jedes Tupel spiegelt eine Kombination von Parametern wider, für welche eine Messung durchgeführt wird (vgl. Ebbinghaus 2021, S. 50). Für alle $(g, a) \in K$ wird das Modell der Art a und der Größe g



(a) Beispiel 1 (b) Beispiel 2 (c) Beispiel 3

Abbildung 5.1: Modelle gleicher und unterschiedlicher Art

optimiert und die Dauer dieser Optimierung gemessen. Dieses Modell wird im Folgendem als Modell (g, a) bezeichnet.

Die Parameter sind folgendermaßen definiert. Für jede Art wird eine Funktion definiert, welche ein Modell dieser Art zurückgibt. Die Größe ist ein Wert n, welcher an diese Funktion übergeben wird. Was genau eine Größe von n bedeutet ist dabei von Art zu Art unterschiedlich. Es kann also sein, dass das Modell $(4, j)$ mit Größe 4 und Art j mehr Knoten hat als das Modell $(4, p)$, obwohl sie dieselbe Größe haben. Das spielt jedoch keine Rolle, da diese Messungen lediglich dazu dienen, zu untersuchen, wie sich die Laufzeit bei wachsender Größe verhält. Zwei Modelle sind also von gleicher Art, wenn sie mit der gleichen Vorgehensweise erzeugt wurden. Abbildung 5.1 zeigt verschiedene Modelle. Von diesen können z. B. Abbildung 5.1(a) und Abbildung 5.1(b) von der gleichen Art sein, da diese beiden erzeugt wurden, indem ein Aggregation-Knoten, n Projection-Knoten und ein Table-Knoten hintereinander gehängt werden. Dabei kann z. B. n die Größe sein und $n + 2$ die Anzahl der Knoten. Abbildung 5.1(c) wurde nicht auf dieser Weise erzeugt, daher ist es nicht von der gleichen Art.

Code-Ausschnitt 5.1 zeigt die Definition des Benchmarks, welcher genutzt wird, um die Messungen durchzuführen. `BM_Optimizer`, abgebildet in Code-Ausschnitt 5.3 ist die Funktion, welche den Code des Benchmarks enthält. `BenchOptimizerArguments`, abgebildet

```
BENCHMARK(BM_Optimizer)
  ->Apply(BenchOptimizerArguments)
  ->Repetitions(20)
  ->Unit(benchmark::kMicrosecond);

```

Code-Ausschnitt 5.1: Benchmark Definition

in Code-Ausschnitt 5.2 legt alle Parameterpaare fest, für welche die Funktion des Benchmarks aufgerufen werden soll. `Repetitions(20)` legt die Anzahl der Wiederholungen fest und sorgt dafür, dass die Funktion für jedes dieser Paare 20-mal ausgeführt wird. `Unit(benchmark::kMicrosecond)` legt fest, dass die Einheit der Ausgabe Mikrosekunden ist.

```
enum class ModelTypes {
  PROJECTION = 0,
  JOIN = 1,
  UNION = 2,
};

static void BenchOptimizerArguments(benchmark::internal::Benchmark *b)
{
  for (int type : {
      static_cast<int>(ModelTypes::PROJECTION),
      static_cast<int>(ModelTypes::JOIN),
      static_cast<int>(ModelTypes::UNION),
      }) {
    for (int size : {8, 16, 32, 64, 128, 256, 512, 1024, 1536, 2048}) {
      //fügt {size,type} den Parameterpaaren des Benchmarks hinzu
      b->Args({size, type});
    }
  }
}

```

Code-Ausschnitt 5.2: Benchmark Parameter

Code-Ausschnitt 5.2 erzeugt das kartesische Produkt von einer Menge von Arten und einer Menge von Größen, und fügt jedes Tupel den Parameter Paaren des Benchmarks hinzu.

```
1  static void BM_Optimizer(benchmark::State &state)
2  {
3    const ltt::allocator_handle allocHandle(
4        ltt::allocator::global_allocator().createSubAllocator("test")
5        );
6    TREX_ERROR::TRexError error;
7    TRexConfig::CalcEngine::JSON::Builder builder(*allocHandle);
8    //erzeugt JSON-String für das (size,type)-Modell,
9    //abhängig von den mitgegebenen Parametern
10   CreateModel(
11       builder,
12       allocHandle,
13       state.range(0), //size
14       state.range(1) //type
15       );
16   Optimizer2 optimizer;
17   for (auto _ : state) {
18     state.PauseTiming(); //pausiert Zeitmessung
19     //erzeugt das Modell aus dem JSON-String im Builder
20     RuntimeModel model(*allocHandle);
21     THROW_ASSERT_0(
22         FactoryAPI::jsonToRuntimeModel(model, builder.toString(), error)
23         );
24     state.ResumeTiming(); //setzt Zeitmessung fort
25     optimizer.optimize(model); //führt die Optimierung aus
26   }
27 }
28
```

Code-Ausschnitt 5.3: Benchmark Hauptlogik

Code-Ausschnitt 5.3 beinhaltet die tatsächliche Logik des Benchmarks. Diese Funktion wird für jedes Parameterpaar n-mal aufgerufen, wobei n die Anzahl der Wiederholungen ist. Für die Zeitmessung wird dabei nur der Teil innerhalb der Iterationsschleife (`for (auto _ : state) {}`) beachtet. Diese wird i-mal ausgeführt, wobei i die Anzahl der Iterationen ist. i wird abhängig von der Dauer einer Iteration und der Varianz der Zeit über alle Iterationen hinweg automatisch bestimmt. Da die `optimize` Funktion die `RuntimeModel` Instanz bearbeitet ist es problematisch, wenn die Funktion mehrmals mit derselben Instanz aufgerufen wird. Deshalb wird die Modell-Instanz innerhalb der Iterationsschleife erzeugt. Um die hierfür benötigte Zeit nicht zu messen, wird die Zeitmessung davor pausiert, und danach wieder fortgesetzt.

5.2 Profiling

```
1 BENCHMARK(BM_Optimizer_Profiling)
2   ->Args({2048,modelCreationJoinWithFilter})
3   ->Unit(benchmark::kMicrosecond);
4
```

Code-Ausschnitt 5.4: Profiling Definition

Das Profiling wird ebenfalls in einem Benchmark durchgeführt, um auch hier die Messung nur auf die `optimize` Funktion zu beschränken. Die Definition von diesem ist in Code-Ausschnitt 5.4 gegeben. Der Benchmark wird einmal für das zu untersuchende Parameterpaar ausgeführt. In Code-Ausschnitt 5.4 wurde exemplarisch für die Größe `2048` und für die Art `modelCreationJoinWithFilter` gewählt.

Code-Ausschnitt 5.5 zeigt die Logik für das Profiling, diese sehr ähnlich zur Benchmarking-Logik, deshalb werden nur die Unterschiede dargelegt. Da der Benchmark für das Profiling nicht genutzt wird, um die Laufzeit die Iterationsschleife zu untersuchen, wird hier auf das Pausieren der Zeitmessung verzichtet. Stattdessen, wird das Profiling während der Initialisierung des Modells pausiert. Nachdem alle Iterationen durchlaufen wurden, werden die gesammelten Profiling-Informationen ausgegeben.

5.3 Untersuchte Parameter

Die Parameter, welche für die Messungen verwendete werden sind bereits in Code-Ausschnitt 5.2 dargestellt. In diesem Abschnitt werden jedoch die verschiedenen Modellarten noch genauer erläutert. Die Modelle für die Messungen wurde wie in Abbildung 5.2 dargestellt erzeugt. Betrachtet man nur das Submodell, so hat jedes genau eine Quelle und mindestens eine Senke. Die Quelle ist der Startknoten des Submodells. Eine der Senken ist der Endknoten des Submodells, alle übrigen Senke müssen Tabellenknoten sein. Um mehrere Submodelle aneinander zu hängen, wird der Startknoten des einen Submodells an den Endknoten des anderen gehängt. Die Anzahl der Submodelle wird dabei durch den Parameter Größe bestimmt. Der Aufbau eines Submodells wird von der Art des Modells bestimmt.

```
1  static void BM_OptimizerProfiling(benchmark::State &state)
2  {
3    const ltt::allocator_handle allocHandle(
4        ltt::allocator::global_allocator().createSubAllocator("test")
5        );
6    TREX_ERROR::TRexError error;
7    TRexConfig::CalcEngine::JSON::Builder builder(*allocHandle);
8    //erzeugt JSON-String für das (size,type)-Modell,
9    //abhängig von den mitgegebenen Parametern
10   CreateModel(
11       builder,
12       allocHandle,
13       state.range(0), //size
14       state.range(1) //type
15       );
16   Optimizer2 optimizer;
17   dbgExecuteCommand("profiler clear"); //löscht Profiling-Informationen
18   for (auto _ : state) {
19     //erzeugt das Modell aus dem JSON-String im Builder
20     RuntimeModel model(*allocHandle);
21     THROW_ASSERT_0(
22         FactoryAPI::jsonToRuntimeModel(model, builder.toString(), error)
23         );
24     dbgExecuteCommand("profiler start"); //pausiert Profiling
25     optimizer.optimize(model);
26     dbgExecuteCommand("profiler stop"); //setzt Profiling fort
27   }
28   //gibt Profiling-Informationen aus
29   dbgExecuteCommand("profiler print -o /tmp/cpu.dot,/tmp/wait.dot");
30 }
31
```

Code-Ausschnitt 5.5: Profiling Hauptlogik

Abbildung 5.3 zeigt den Aufbau der Submodelle, für die drei bereits in Code-Ausschnitt 5.2 gezeigten Modellarten. Das erste Submodell, besteht aus einem einzelnen Projection-Knoten. Dieser Knoten ist Start- und Endknoten des Submodells. Das zweite Submodell besteht aus einem Join-Knoten, einem Table-Knoten und einem Projection-Knoten. Der Join-Knoten ist der Startknoten des Submodells und hat den Table- und den Projection-Knoten als Eingangsknoten. Der Projection-Knoten ist der Endknoten des Submodells. Das dritte Submodell besteht aus einem Union-Knoten, zwei Table-Knoten und einem Projection-Knoten. Der Union-Knoten ist der Startknoten des Submodells und hat die

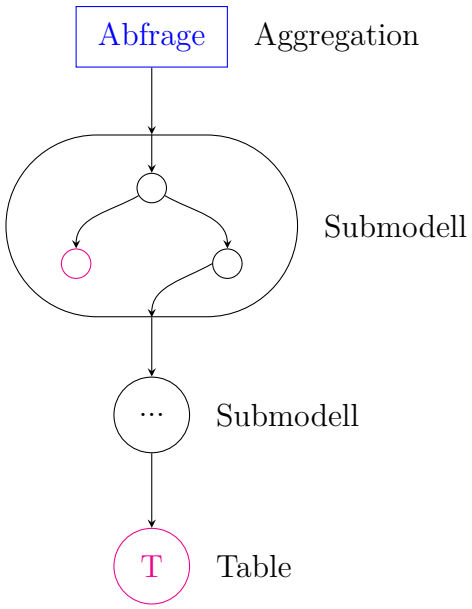


Abbildung 5.2: Darstellung der Modellgenerierung

drei übrigen Knoten als Eingangsknoten. Der Projection-Knoten ist der Endknoten des Submodells.

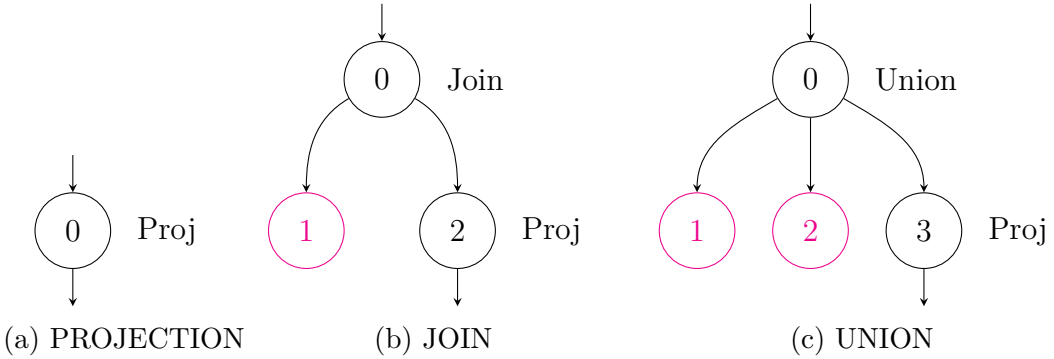


Abbildung 5.3: Darstellung der verschiedenen Modellarten

6 Umsetzung und Interpretation der Messwerte

In diesem Kapitel werden die Messwerte aus der Umsetzung dargestellt und statistisch analysiert. Für die Interpretation der Messwerte werden, wie in Tabelle 6.1 gezeigt, den verschiedenen Arten von Modellen Zahlen zugeordnet.

Modellart	**Zahl**
PROJECTION	0
JOIN	1
UNION	2

Tabelle 6.1: Zuordnung von Modellart zu Zahl

6.1 Benchmarking Ergebnisse

Abbildung 6.1 zeigt die durchschnittlichen Messwerte sowie dem 0,99-Konfidenzintervall für diese Messwerte, für alle Modellarten. Es lässt sich erkennen, dass für Modellart 0 die Zeit sich annähernd linear zur Größe verhält. Für die Modellarten 1 und 2 steigt die Steigung mit zunehmender Größe immer weiter an. Deshalb wird die Optimierung für diese Arten mithilfe von Profiling genauer betrachtet.

6.2 Profiling Ergebnisse

Abbildung 3.3(b) zeigt einen Ausschnitt aus der Profiler-Ausgabe für das Modell der Art 1 und der Größe 2048. Dieser Ausschnitt beinhaltet dabei die relevantesten Knoten der Ausgabe. Der `optimize`-Knoten hat zwar noch weitere Kindknoten. Aufgrund der geringeren Auswirkung auf die Laufzeit, wurden diese in der Grafik entfernt, um die

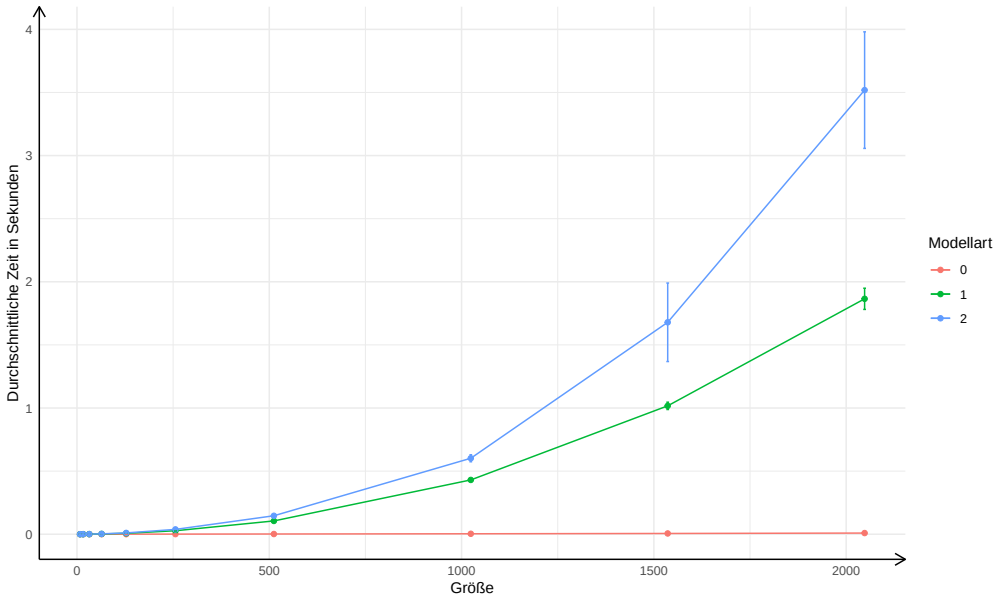


Abbildung 6.1: Durchschnittliche Zeit mit 0,99-Konfidenzintervall

Übersichtlichkeit zu erhöhen. Die relevantesten von diesen Knoten werden jedoch im folgenden weiter betrachtet. Die Ausgabe für Modellart 2 hat einen ähnlichen Aufbau.

Tabelle 6.2 stellt für ausgewählte Knoten den Anteil der von Funktion dieses Knotens benötigten Zeit von der Gesamtzeit dar. Es sind jeweils alle Messwerte für die verschiedenen Größen angegeben, um auch das Verhalten analysieren zu können. Diese Werte entsprechen dem in Abschnitt 3.4 erläutertem Wert E aus der Profiler-Ausgabe. Auffällig ist die `opptimize`-Funktion, diese benötigt sowohl bei Modellart 1 als auch bei Modellart 2 einen sehr großen Anteil der Zeit. Dieser Anteil steigt jedoch zumindest bei Modellart 2 nicht eindeutig mit der Größe. Dies bedeutet jedoch nicht, dass die benötigte Zeit in dieser Funktion ebenfalls nicht steigt, da diese Angaben prozentual zu der jeweiligen Gesamtdauer sind. Die anderen Knoten sind deshalb interessant, weil der Anteil an der Gesamtdauer mit der Größe des Modells zunimmt. Die größte Zunahme von Prozentpunkten ist dabei $32\% - 19\% = 13\%$, die größte prozentuale Steigerung ist $\frac{12\%}{1{,}5\%} - 1 = 700\%$. Aufgrund des hohen Anteils oder der großen Steigerung von diesem bieten sich alle in Tabelle 6.2 dargestellten Funktion für eine nähere Betrachtung an.

Art	**Funktion**	**2048**	**4096**	**8192**	**16384**
1	Optimizer2::optimize	18 %	22 %	24 %	24 %
1	CombineJoinOverProjectionPattern::doesMatch	19 %	26 %	30 %	32 %
1	BuildExpresionFilterPattern::doesMatchInternal	1,5 %	6,1 %	11 %	12 %
2	Optimizer2::optimize	37 %	32 %	33 %	34 %
2	BuildExpresionFilterPattern::doesMatchInternal	4,2 %	10 %	10 %	14 %

Tabelle 6.2: Anteil an der Gesamtlaufzeit ausgewählter Knoten

7 Fazit und Ausblick

A Anhang

Literaturverzeichnis

Bartz-Beielstein, T., Hrsg. (2010). *Experimental methods for the analysis of optimization algorithms*. Berlin [Heidelberg]: Springer. ISBN: 978-3-642-02537-2.

Brosius, H.-B./ A. Haas/ J. Unkel (2022). *Methoden der empirischen Kommunikationsforschung: eine Einführung*. 8., vollständig überarbeitete und erweiterte Auflage. Wiesbaden [Heidelberg]: Springer VS. ISBN: 978-3-658-34195-4 978-3-658-34194-7.

Ebbinghaus, H.-D. (2021). *Einführung in die Mengenlehre*. 5. Auflage. Berlin [Heidelberg]: Springer Spektrum. ISBN: 9783662638651.

Google (2024). *benchmark A microbenchmark support library*. URL: https://google.github.io/benchmark/ (Einsichtnahme: 10. 06. 2024).

Knebl, H. (2021). *Algorithmen und Datenstrukturen: Grundlagen und probabilistische Methoden für den Entwurf und die Analyse*. 2., aktualisierte Auflage. Wiesbaden [Heidelberg]: Springer Vieweg. ISBN: 978-3-658-32714-9 978-3-658-32713-2.

McGeoch, C. C. (2002). „Experimental Analysis of Algorithms“. In: *Handbook of Global Optimization: Volume 2*. Hrsg. von Pardalos, P. M./ Romeijn, H. E. Boston, MA: Springer US, S. 489–513. ISBN: 978-1-4757-5362-2. DOI: 10.1007/978-1-4757-5362-2_14.

Parekh, H. B./ S. Chaudhari (2016). „Improved Round Robin CPU scheduling algorithm: Round Robin, Shortest Job First and priority algorithm coupled to increase throughput and decrease waiting time and turnaround time“. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, S. 184–187. ISBN: 978-1-5090-0467-6. DOI: 10.1109/ICGTSPICC.2016.7955294.

SAP (2024a). *SAP HANA Cloud, SAP HANA Database Modeling Guide for SAP Web IDE Full-Stack. Creating Calculation Views*. URL: https://help.sap.com/docs/hana-cloud-database/sap-hana-cloud-sap-hana-database-modeling-guide-for-sap-web-ide-full-stack/creating-graphical-calculation-view (Einsichtnahme: 09. 07. 2024).

SAP (2024b). *SAP HANA Cloud, SAP HANA Database Modeling Guide for SAP Web IDE Full-Stack. Supported View Nodes for Modeling Calculation Views.* URL: https://help.sap.com/docs/hana-cloud-database/sap-hana-cloud-sap-hana-database-modeling-guide-for-sap-web-ide-full-stack/supported-view-nodes-for-modeling-calculation-views?version=2024_2_QRC (Einsichtnahme: 09. 07. 2024).

– (2024c). *SAP HANA Cloud, SAP HANA Database Performance Guide for Developers. Query Execution Engine Overview.* URL: https://help.sap.com/docs/hana-cloud-database/sap-hana-cloud-sap-hana-database-performance-guide-for-developers/query-execution-engine-overview?q=Hana%20execution%20engine (Einsichtnahme: 09. 07. 2024).

Stocker, T. C./ I. Steinke (2017). *Statistik: Grundlagen und Methodik.* Berlin, Boston: De Gruyter Oldenbourg. ISBN: 978-3-110-35388-4.