

**HNU Working Paper**

Nr. 54

Thomas Bauer

**Auswirkungen von auf künstlicher Intelligenz basierender  
Generierung von SQL-Abfragen auf die Lehre**

9 / 2025

## **Abstrakt**

Heutzutage ist es möglich, sich mittels Künstlicher Intelligenz automatisch Programmcode generieren zu lassen. Dies gilt auch für Datenbankabfragen, die in der Sprache SQL erstellt werden. Diese Möglichkeit wirft die Frage auf, ob es überhaupt noch notwendig ist, SQL im Rahmen von Datenbank-Vorlesungen zu lehren. Außerdem ist zu klären, inwieweit Studierende die SQL-Generierung zur Erstellung von Prüfungsergebnissen nutzen können bzw. wie Prüfungen gestaltet sein müssen, um entsprechende Betrugsversuche zu vermeiden. Hierzu wird in dieser Arbeit der Stand der Forschung zum Thema SQL-Generierung dargestellt. Außerdem werden die Ergebnisse von Experimenten beschrieben, in welchen die Qualität von automatisch generierten SQL-Abfragen ermittelt wurde.

Freie Schlagwörter: Datenbanken, SQL, Künstliche Intelligenz, Large Language Model

JEL-Klassifikation: L86

## Inhaltsverzeichnis

Motivation .....	4
Literaturanalyse .....	4
NL2SQL in der Lehre .....	5
Analyse der Fähigkeit zur Generierung von SQL-Abfragen .....	7
Verbesserung von NL2SQL .....	8
Fazit .....	9
Aufgabenstellungen für Datenbank-Vorlesungen .....	10
Verwendete Szenarien .....	11
Durchführung der Experimente .....	13
Auffällige Einzelergebnisse .....	16
Unzulängliche Lösungen .....	16
Sonstige Auffälligkeiten .....	32
Fazit .....	34
Schlussfolgerungen .....	35
SQL-Generierung in der betrieblichen Praxis .....	35
Auswirkungen auf die Lehre .....	36
Zusammenfassung, Limitationen und Ausblick .....	37
Literatur .....	38
Anhang A: Übersicht über die Aufgabenstellungen .....	41
Anhang B: Mögliche Lösungen für die Aufgabenstellungen .....	44

## Motivation

Künstliche Intelligenz (KI) ist inzwischen sehr gut in der Lage, Texte zu generieren. Mit einem entsprechenden Large Language Model (LLM) kann auch Programmcode generiert werden. Im Kontext von Datenbanken sind dies Abfragen in SQL. Es wird hierbei eine natürlichsprachliche Aufgabenstellung vorgegeben und zusätzlich das Datenbank-Schema, das die zugrunde liegenden Datenbank-Tabellen beschreibt. Das LLM generiert daraus eine Abfrage in der Sprache SQL (Natural Language to SQL: NL2SQL).

In dieser Arbeit wird untersucht, wie gut die Fähigkeit heutiger LLMs hierbei sind. Dabei liegt der Fokus auf Auswirkungen, die sich auf die Lehre zum Thema Datenbanken ergeben. Die Forschungsfrage lautet also: Welche Qualität haben aktuell automatisch generierte SQL-Abfragen und welche Konsequenzen ergeben sich daraus für die Lehre?

Hierbei werden die folgenden beiden Teilaspekte betrachtet:

1) Korrektheit: Für den Fall, dass es mittels eines LLM möglich ist, stets (d.h. für alle Aufgabenstellungen) eine (fast) zu 100% korrekt SQL-Abfrage zu generieren, besteht keine Notwendigkeit mehr, SQL-Kenntnisse zu lehren. Diese hätten dann allenfalls den Zweck, die generelle Funktionsweise von Datenbanksystemen besser zu verstehen. Falls jedoch manchmal oder für bestimmte Aufgabenstellungen oder Datenbank-Schemata fehlerhafte SQL-Abfragen generiert werden, sind weiterhin SQL-Kenntnisse erforderlich, z.B. zum Überprüfen und Verbessern der generierten SQL-Anweisungen.

2) Prüfungsgestaltung: Wenn LLMs lediglich in der Lage sind, relativ häufig korrekte SQL-Abfragen zu generieren, dann sollten diese (wie erwähnt) nicht unbesehen verwendet werden. Dennoch bieten sie Studierenden die Möglichkeit, ohne ausreichende SQL-Kenntnisse und bei geringem Aufwand, Lösungen für Prüfungsaufgaben zu erzeugen. Sind viele der generierten SQL-Abfragen korrekt oder beinhalten nur kleinere Fehler, so wird mit dieser Vorgehensweise eine recht gute Note erzielt werden. In diesem Fall sollten Prüfungen so gestaltet sein, dass durch dieses Betrugspotential kein großer Vorteil gegenüber Studierenden erzielt werden kann, die ihre Prüfungsergebnisse selbst erstellen.

Im nachfolgenden Abschnitt wird wissenschaftliche Literatur zum Thema NL2SQL dargestellt. Darauf folgt eine Beschreibung der durchgeführten Experimente und deren Ergebnisse. Diese haben das Ziel, die heutige Qualität von NL2SQL in Szenarien zu ermitteln, die bzgl. Komplexität des Datenbank-Schemas und Schwierigkeitsgrad üblichen Lehr- und Prüfungsinhalten entsprechen. Darauf folgt ein Abschnitt, der Schlussfolgerungen für die Lehre zieht. Der Betrag schließt mit einer kurzen Zusammenfassung der wichtigsten Erkenntnisse.

## Literaturanalyse

Im Gesamtthema der Generierung von SQL-Abfragen, ausgehend von einer in natürlicher Sprache vorgegebenen Problemstellung (NL2SQL), findet aktuell viel Forschung statt. Dies ist auch daran erkennbar, dass die meisten der nachfolgend dargestellten Veröffentlichungen recht neu sind. Bei einigen Arbeiten handelt es sich sogar um Vorab-Veröffentlichungen. Diese wurden berücksichtigt, obwohl ihre Qualität noch nicht durch einen Peer-Review-Prozess bestätigt ist, um so einen sehr aktuellen Stand der Forschung darstellen zu können.

Im Folgenden wird zuerst Literatur zu dem Thema NL2SQL im Kontext von Lehre dargestellt. Das darauf folgende Unterkapitel beschäftigt sich mit der Bewertung des aktuellen Standes der

SQL-Generierung. Schließlich werden noch Ansätze vorgestellt, die das Ziel haben, NL2SQL selbst zu verbessern.

## ***NL2SQL in der Lehre***

Da in dieser Arbeit insbesondere die Auswirkungen von NL2SQL auf die zukünftige Lehre untersucht werden sollen, wurde gezielt nach Literatur zu diesem Thema gesucht.<sup>1</sup> Dabei wurden keine Arbeiten gefunden, die zu erwartende Veränderungen der erforderlichen Kompetenzen von Studierenden oder zukünftig (noch) notwendige Lehrinhalte beschreiben. Allerdings erwähnen einige Autoren (Dixit and Gajjam, 2024; Ganesan et al., 2024; Hong and Storey, 2023; Taipalus and Seppänen, 2020) von nachfolgend dargestellten Arbeiten die Wichtigkeit von SQL-Kenntnissen für Studierende, allerdings ohne speziell hierzu eine Studie durchgeführt zu haben. Da sich diese Arbeiten jedoch mit dem generellen Thema „KI und Datenbanken“ beschäftigen, impliziert dies, dass die Autoren diese Kenntnisse auch weiterhin für erforderlich halten.

Zudem ergab die erwähnte Literatursuche sehr viele Veröffentlichungen, die sich mit dem Lernen eines neuronalen Netzes bzw. eines LLMs mittels Trainingsdaten beschäftigen. Diese Art des „Lernens“ ist im vorliegenden Kontext nicht relevant, weshalb entsprechende Arbeiten im Folgenden nicht dargestellt werden.

Es wurden Arbeiten zu den nachfolgend beschriebenen Themen identifiziert:

(Taipalus and Seppänen, 2020) bietet einen ausführlichen Überblick über Arbeiten zum Thema SQL-Ausbildung. Die Arbeit enthält jedoch keine Aussagen zu den Themen generative KI, NL2SQL oder den durch diese Technologie resultierenden Auswirkungen auf erforderliche SQL-Kenntnisse.

Einige Arbeiten beschäftigen sich mit dem Einsatz von KI oder generell von Softwarelösungen beim Unterrichten von SQL:

(Prakash et al., 2024) verwendet nicht ausschließlich ein Standard-Produkt (wie z.B. ChatGPT) zur Generierung von SQL-Abfragen, sondern schlägt hierfür eine speziell entwickelte Architektur vor. Mit dieser können sich Studierende SQL-Abfragen generieren lassen. Sie können aber auch selbst solche Abfragen erstellen und an das Tool übergeben. Dieses führt sie dann aus und analysiert das Ergebnis. Ist letzteres falsch, werden den Studierenden Quizfragen zum betroffenen Thema gestellt.

(Zhang, 2025) nutzt eine KI zur Generierung von SQL-Abfragen im Rahmen einer Lehrveranstaltung. Dazu erhalten die Studierenden verschiedene Aufgaben (einschließlich dem Ändern und Abfragen von Daten mittels SQL), welche sie selbst lösen sollen und zu denen sie sich zusätzlich eine Lösung von der KI generieren lassen sollen. Die Studierenden lernen hierbei unterschiedliche Vorgehensweisen kennen und können diese kritisch vergleichen.

(Carr et al., 2023) beschäftigt sich mit der Problemstellung, dass es schwierig ist, von Studierenden erstellte SQL-Abfragen automatisch auf Korrektheit zu überprüfen, weil auch korrekte SQL-Abfragen sehr stark von der Musterlösung abweichen können. Deshalb wird ChatGPT verwendet, um weitere Lösungen für SQL-Abfragen aus den natürlichsprachlichen

---

<sup>1</sup> Hierzu wurde eine Suche in Google-Scholar durchgeführt, wobei nach NL2SQL jeweils in Kombination mit education, teaching, student, Lehre oder Unterricht gesucht wurde. Außerdem wurde nach „SQL generation“ in Kombination mit einem der letzteren Begriffe gesucht. Schließlich wurde auch noch eine Suche nach den Begriffen SQL education artificial intelligence durchgeführt.

Aufgabenstellungen zu generieren. Diese werden verwendet, um sie mit den Lösungen der Studierenden zu vergleichen. In der Arbeit wurde erkannt, dass diese Vorgehensweise gute Ergebnisse liefert, so dass sie in der Lehre eingesetzt werden kann.

(Hong and Storey, 2023) skizziert einen Ansatz, bei dem ChatGPT, zum Zweck der Lehre, u.a. SQL-Statements generieren und den Studierenden die zugrunde liegende Idee erläutern soll. Allerdings wird in dieser Arbeit ein erst kürzlich gestartetes Projekt beschrieben, so dass noch keine fundierten Erkenntnisse vorliegen. So ist z.B. noch unklar, wie der Erfolg dieser Vorgehensweise gemessen werden kann und wie sich die Rolle eines Lehrenden dadurch verändert.

(Dixit and Gajjam, 2024) möchte das Lernen von SQL mittels des KI-Tools AI2SQL.io verbessern. Dieses übernimmt die Rolle eines Tutors, indem es den Studierenden während ihrer SQL-Übungen Feedback gibt. Diese Automatisierung erlaubt es (auch bei einer großen Gruppengröße), für jede einzelne Übungsaufgabe sofort Feedback z.B. bzgl. Fehlern zu geben. Es wird also ein individuelles Feedback möglich und zugleich der erforderliche Zeitaufwand für Lehrende reduziert.

Auch in (Matek et al., 2017) wird eine KI als Tutor eingesetzt. Diese generiert Hinweise auf Basis von Lösungen, welche die Studierenden vorheriger Jahrgänge erstellt haben. Die aktuellen Studierenden erstellen SQL-Abfragen (für dieselben Übungsaufgaben) und können diese wahlweise ausführen oder die KI nach einem Hinweis fragen. Dieser besteht darin, dass die aktuelle SQL-Abfrage von der KI angepasst wird. Diesen Hinweis können die Studierenden benutzen oder aber auch ignorieren, um so zu einer korrekten Lösung zu gelangen.

Auch das in (Ganesan et al., 2024) vorgestellte Tool SQLearn prüft von Studierenden erstellte SQL-Abfragen auf Korrektheit. Allerdings wird hierfür kein KI-basierter Ansatz verwendet, sondern die Ähnlichkeit zu einer Referenz-Abfrage (der Musterlösung) wird automatisch bewertet. Eine Einschränkung dieses Ansatzes im Vergleich zu den zuvor vorgestellten KI-basierten Ansätzen ist deshalb, dass lediglich das Feedback „korrekt“, „falsch“ oder „teilweise korrekt“ möglich ist.

(Steinberger and Wedam, 2025) stellt einen Ansatz vor, der es ermöglicht, mittels einer KI, Material für Übungen automatisch zu generieren. Dieses umfasst Datenbanken mit Beispieldaten, Übungsaufgaben und Musterlösungen. Das Ziel dieses Ansatzes ist, den Arbeitsaufwand für Lehrende zu reduzieren. Einen ähnlichen Ansatz verfolgt (Zhang, 2025). Dort wird vorgeschlagen, sich Unterrichtsmaterial, wie z.B. Quizfragen, Übungsaufgaben und Vorlesungsfolien von einer KI generieren zu lassen. Generell spart ein solches Vorgehen sicherlich Aufwand ein, ist aber recht unabhängig von dem in diesem Bericht behandelten Thema SQL, so dass auf andere ähnliche Ansätze hier nicht eingegangen wird.

(Farinetti and Cagliero, 2025) verwendet kein spezielles Tool, sondern die Studierenden sollten direkt mit ChatGPT arbeiten: Sie sollen natürlichsprachliche Aufgabenstellungen entwickeln, bei denen ChatGPT fehlerhafte Lösungen für die zugehörige SQL-Abfrage generiert. Außerdem erstellen die Studierenden selbst eine korrekte Lösung für die SQL-Abfragen. Ziele dieses Experiments sind, dass sich Studierende mit komplizierten Abfragen beschäftigen (da die KI bei solchen eher Fehler macht) und sie sich bewusst werden, dass die KI-generierten Lösungen durchaus fehlerhaft sein können. Im durchgeführten Experiment waren in letzteren sogar mehr Fehler als in den von den Studierenden manuell erstellten Lösungen. Insgesamt

soll der Gamification-Aspekt motivationssteigernd wirken, weshalb das Experiment auf diese Art und Weise gestaltet wurde.

(Gaitantzi and Kazanidis, 2025) analysiert 31 Studien zum Thema KI-Unterstützung bei der Lehre zum Thema Software-Entwicklung (und hierbei teilweise auch mit dem Thema SQL oder zumindest Datenbanken). In den analysierten Studien wurden KI-Tools für die bereits genannten Zwecke eingesetzt: zur Generierung von Programmcode (d.h. SQL-Abfragen), als automatisierter Tutor und zur Generierung von Lehrmaterial. Die Studien erkennen einen Nutzen durch die Verwendung von KI, z.B. weil (auch bei großen Gruppen) ein persönliches sowie sofortiges Feedback möglich wird. Allerdings werden auch Probleme der Ansätze benannt, wie z.B. Fehler im generierten Inhalt oder eine entstehende Abhängigkeit der Studierenden von der KI.

## **Analyse der Fähigkeit zur Generierung von SQL-Abfragen**

Im Folgenden werden Arbeiten vorgestellt, welche die heutige Qualität von NL2SQL bewerten. Zu diesem Zweck existieren auch einige Benchmarks, die aus einer Sammlung von Beispieltabellen und natürlichsprachlichen Abfragen an diese bestehen.

Kernaussage von (Floratou et al., 2024) ist, dass das Problem der SQL-Generierung (NL2SQL) aktuell noch nicht gelöst ist. Die Arbeit benennt konkrete Probleme hierbei, wie z.B. sehr viele Tabellen und Spalten in einer realistischen (d.h. praxisnahen) Datenbank. Hierzu wird als Beispiel das Microsoft-interne Data-Warehouse für Finanzdaten mit 632 Tabellen und über 4000 Spalten (weitere in Views) vorgestellt. Ein solch komplexes Szenario ist für Lehr-Beispiele und -Übungen natürlich nicht sinnvoll, zeigt jedoch, dass bei dem Thema noch viele ungelöste Probleme bestehen. Zudem werden als weitere Schwierigkeiten in der Praxis genannt, dass Abfragen in natürlicher Sprache oft nicht eindeutig sind und Endbenutzer die Semantik (d.h. tatsächliche Bedeutung) der Tabellen und deren Spalten nicht kennen, was jedoch für eine exakte Formulierung von natürlichsprachlichen Abfragen erforderlich ist. All dies führte in dieser Arbeit zu einer durchschnittlichen Korrektheit von nur 22,7% bei der Generierung von SQL-Abfragen.

Ähnliche Probleme werden auch in (Liu et al., 2025) und (Mohammadjafari et al., 2025) genannt.

In (Nascimento et al., 2025) werden Experimente vorgestellt, die mit einer realen Datenbank (d.h. aus der Praxis) durchgeführt wurden. Die mit GPT-4 generierten SQL-Abfragen waren dabei nur zu 41% korrekt.

(Liu et al., 2023) untersucht die Qualität generierter SQL-Abfragen für den Fall, dass dem LLM im Prompt keine besondere Unterstützung (z.B. durch Beispiele) gegeben wird (Zero-Shot). ChatGPT erreichte bei diesen Tests eine Korrektheit von max. 76,6%, wobei diese allerdings für andere Arten von Problemstellungen deutlich schlechter war. Das LLM, das bei dem Test am besten abgeschnitten hat, erreichte 84,1% Korrektheit. Auffällig war, dass dieses seinen besten Korrektheitswert bei anderen Problemstellungen erreichte als ChatGPT. Als generelle Erkenntnis lässt sich festhalten, dass keine der LLMs eine nahezu absolute Korrektheit erreicht hat.

(Sun et al., 2023) untersucht die Qualität generierter SQL-Abfragen für verschiedene LLMs, wobei auch verschiedene Arten des Abbaus der Prompts berücksichtigt werden. Das beste Ergebnis erzielte hierbei ChatGPT 3.5, wobei die Korrektheit abhängig vom Prompt-Aufbau im

Intervall 31,1% bis 70,9% lag. Eine solche Abhängigkeit wurde auch für andere LLMs festgestellt, wobei bei diesen teilweise ein anderer Prompt-Aufbau optimal war, als bei ChatGPT. Auch (Campos et al., 2025) ermittelt mittels Experimenten die Korrektheit von generierten SQL-Abfragen. Allerdings sind diese sehr einfach strukturiert, da die Datenbank aus nur einer einzigen Tabelle besteht (mit jedoch sehr vielen Zeilen). Die ermittelte Korrektheit lag, abhängig vom LLM-Modell, zwischen 10% und 96%, d.h. auch bei solch einfachen Abfragen lässt sich keine absolute (d.h. sichere) Korrektheit erreichen. Um die Korrektheit zu messen, wurde in einem ersten Schritt verglichen, ob die generierte Abfrage exakt der manuell erstellten Soll-Abfrage entspricht. Falls dies nicht zutrifft, wurden in einem zweiten Schritt beide Abfragen ausgeführt und die Ergebnisse auf Gleichheit geprüft. Ein solches Vorgehen ist natürlich nur bei einer ausreichend großen Anzahl von Beispieldaten sinnvoll, weil sonst häufig der Fall auftritt, dass die Ergebnisse einer (leicht) falschen Abfrage zufällig identisch sind.

In (Pornphol and Chittayasothorn, 2023) wird der spezielle Aspekt der Vollständigkeit untersucht: Eine Datenbank-Abfragesprache gilt als vollständig, wenn sie mindestens die Mächtigkeit der Relationenalgebra aufweist. Als Relationenalgebra wurden die von Codd in seiner Originalpublikation vorgeschlagenen 8 Operationen definiert. Anhand von 5 Beispielen wurde festgestellt, dass ChatGPT korrekte SQL-Abfragen erstellen kann, welche alle Operationen abdecken. Dies lässt die Aussage zu, dass ChatGPT bzgl. der generierten SQL-Abfragen vollständig ist.

Zur Bewertung von generierten SQL-Abfragen existieren verschiedene Benchmarks wie BIRD (Li et al., 2023), Spider (Yu et al., 2019), UNITE (Lan et al., 2023), TrustSQL (Lee et al., 2024) und TPC-DS (Poess and Floyd, 2000). Letzter umfasst Abfragen, die aus dem Bereich „Decision-Making“ stammen und deshalb zu besonders komplexen SQL-Abfragen führen. Dieser Benchmark wurde in (Ma et al., 2024) verwendet, um die Abfrage-Generierung von 11 LLMs zu bewerten. Das Ergebnis zeigt, dass hierbei selbst die besten LLMs lediglich eine Korrektheit von 33% erreichen. (Lee et al., 2024) zeigt unter Verwendung des Benchmarks TrustSQL, dass heutige NL2SQL-Methoden noch nicht zufriedenstellend funktionieren. Die begrenzte Aussagekraft von Benchmarks wird in (Ganti et al., 2024) aufgezeigt, da in Protokolldateien enthaltene reale Abfragen eine um ca. 30% geringere Korrektheit aufweisen, als die Abfragen der Benchmarks. Als Gründe hierfür wurden Tabellen mit einer großen Anzahl an Spalten, nicht eindeutige (aber realistische bzw. praxisnahe) Abfragen, sowie komplexe oder verschachtelte resultierende SQL-Abfragen identifiziert.

(Kim et al., 2020) zeigt für mehrere Benchmarks und unterschiedliche LLMs die begrenzte Aussagekraft der erzielten Ergebnisse auf. Die Bewertung der Korrektheit der generierten SQL-Abfragen muss, wegen der großen Anzahl an ausgeführten Testfällen, normalerweise automatisch erfolgen. Hierfür existieren unterschiedliche Methoden, die aber alle zu Falschbewertungen führen können. Durch Experimente wird gezeigt, dass sich je nach gewählter Methode deutliche Fehler ergeben, d.h. ein konkret angegebener Prozentsatz für die Korrektheit sollte nicht „überinterpretiert“ werden.

## **Verbesserung von NL2SQL**

Einige Ansätze verfolgen das Ziel, die Qualität von generierten SQL-Abfragen zu verbessern. (Shi et al., 2024) bietet hierzu einen Überblick, der u.a. Ansätze zum Prompt-Engineering oder Finetuning von Trainingsmethoden, -daten und zur Modellevaluation beinhaltet. Diese Ansätze



werden nicht durch Experimente verglichen, so dass keine unmittelbare Aussage zu deren Leistungsfähigkeit gemacht wird. Stattdessen wird ein inhaltlicher Vergleich durchgeführt. (Liu et al., 2025) bietet ebenfalls einen sehr umfassenden Überblick über Methoden zur SQL-Generierung, wobei auch detailliert auf technische Aspekte von deren Umsetzung eingegangen wird. Dies ist auch der Fokus von (Qin et al., 2022).

(Zhang et al., 2024) untersucht verschiedener Einzelaufgaben von NL2SQL in Bezug z.B. auf Qualität und Ausführungsgeschwindigkeit der generierten SQL-Statements.

(Gao et al., 2023) betrachtet verschiedene Methoden zum Prompt-Engineering im Kontext der SQL-Generierung. Auf dieser Basis wird eine „In-Context-Learning“-Methode für LLMs entwickelt. Hierbei werden in dem Prompt einige Beispiele bereitgestellt, aus denen das LLM lernt. Durch Experimente wird bestätigt, dass dies die Korrektheit der generierten SQL-Abfragen verbessern kann.

In (Dong et al., 2023) wird eine Methode vorgestellt, die speziell Schwachstellen von ChatGPT bei der Generierung von SQL-Abfragen beheben soll: Das Prompt wird auf eine vorgegebene Weise strukturiert und dabei nur für diese Abfrage relevante Tabellen und Spalten angegeben bzw. bzgl. ihrer Relevanz bewertet. Zusätzlich werden Hinweise bereitgestellt, z.B. welche Spalten nicht benötigt werden, oder dass ein Left Outer Join nur verwendet werden soll, wenn tatsächlich ein Outer Join benötigt wird. Schließlich werden mehrere SQL-Abfragen durch ChatGPT generiert und ausgeführt, woraufhin die beste ausgewählt wird.

(Mohammadjafari et al., 2025) bewertet verschiedene NL2SQL-Ansätze bzgl. der Korrektheit der erzeugten SQL-Statements und deren Effizienz (d.h. Ausführungsgeschwindigkeit und notwendige Ressourcen). Als Ergebnis wird unter anderem die Verwendung von Knowledge-Graphen empfohlen. Diese definieren die Beziehungen zwischen Entitäten, Tabellen und Attributen und helfen bei der Generierung von SQL-Statements. Allerdings ist die Erstellung eines solchen Knowledge-Graphen für den Benutzer eine aufwendige Aufgabe. Auch (Nascimento et al., 2025) verwendet solche Graphen. Durch Experimente wurde bestätigt, dass sich dadurch die Korrektheit der generierten SQL-Abfragen verbessern lässt. Allerdings wurde dadurch und durch weitere Maßnahmen lediglich eine Korrektheit von 93% erreicht, d.h. eine absolute Korrektheit der generierten Ergebnisse war nicht möglich.

Bei dem Ansatz von (Biswal et al., 2024) wird aus der Benutzeranfrage eine SQL-Query generiert und ausgeführt. Zusätzlich zum Abfrageergebnis wird dann die natürlich-sprachige Anfrage verwendet, um eine Antwort für den Benutzer zu generieren. Mit diesem Ansatz lassen sich zusätzliche Typen von Benutzeranfragen beantworten.

(Bhaskar et al., 2023) untersucht Szenarien, in denen die für eine bestimmte Abfrage zu generierende SQL-Abfrage nicht eindeutig ist. Dieser Fall kann z.B. aufgrund uneindeutiger Namen von Tabellen oder Spalten sowie aufgrund verwirrender Beziehungen auftreten. Idealerweise sollten dann alle möglicherweise gewünschten SQL-Abfragen generiert und dem Benutzer bereitgestellt werden. Durchgeführte Tests ergaben jedoch, dass dies bei heutigen LLMs nicht der Fall ist. Deswegen wurde in dieser Arbeit ein neuartiger Algorithmus entwickelt, um diesen Fall besser handhaben zu können.

## **Fazit**

Zusammenfassend lässt sich feststellen, dass die Problemstellung der Generierung von SQL-Abfragen aktuell nicht gelöst ist. Daraus lässt sich schließen, dass es für Mitarbeiter aktuell

noch notwendig ist, korrekte SQL-Abfragen erstellen zu können. Die teilweise hohe Fehlerquote bei NL2SQL lässt vermuten, dass dies auch noch längere Zeit gelten wird. Zumindest müssen die Mitarbeiter weiterhin fähig sein, die generierten SQL-Abfragen auf ihre Korrektheit hin zu überprüfen. Dies erfordert sehr ähnliche SQL-Kenntnisse, wie die Erstellung solcher Abfragen. Deswegen ist es weiterhin erforderlich, dass SQL in Lehrveranstaltungen zum Thema Datenbanken enthalten ist. Es konnte jedoch keine wissenschaftliche Literatur identifiziert werden, die Aussagen zu diesem Thema machen, hier existiert also eine Forschungslücke.

Die Ergebnisse von NL2SQL-Bewertungen mittels Benchmarks fallen sehr unterschiedlich aus. Zudem wird in einigen Arbeiten betont, dass deren Aussagekraft begrenzt ist. Eine Bewertung von LLMs mittels Benchmarks ist für die vorliegende Forschungsfrage ohnehin wenig hilfreich, weil ein bestimmter Prozentsatz für die Korrektheit keine direkten Rückschlüsse auf erforderliche Lehrinhalte zulässt. Stattdessen sind solche Benchmarks eher nützlich, um neu entwickelte Ansätze für NL2SQL bewerten und mit existierenden vergleichen zu können. Da eine Entwicklung neuer Ansätze nicht unser Ziel ist, werden im Folgenden keine Benchmarks verwendet.

Ansätze zur Verbesserung der generierten SQL-Abfragen, wie z.B. eine spezielle Prompt-Gestaltung (z.B. In-Context-Learning) oder die Erstellung von Knowledge-Graphen sind zu aufwendig, um in einer Vorlesung im Grundstudium behandelt zu werden. Aufgrund des Aufwands kann vermutet werden, dass solche Verfahren von Studierenden wohl kaum genutzt werden, wenn einige SQL-Abfragen z.B. im Rahmen einer Projektarbeit zu erstellen sind. Ähnliches gibt wohl auch für Mitarbeiter in Unternehmen, die spontan eine einzelne oder einige wenige SQL-Abfragen, z.B. zur Berechnung von Statistiken, erstellen sollen. Deswegen wird auch dieses Thema hier nicht weiter verfolgt.

Wie bereits beim Thema Benchmarks erwähnt, ist die Verbesserung der internen Funktionalität von LLMs bei der Generierung von SQL-Abfragen im Kontext von Lehre ebenfalls nicht relevant.

Deshalb werden im Folgenden nur einige wenige Tests mit unterschiedlichen LLMs durchgeführt, die sich allerdings am Schwierigkeitsgrad von Aufgabenstellungen aus der Lehre orientieren. Es ist dabei nicht das Ziel, die Korrektheit dieser LLMs so detailliert zu vergleichen, dass sich die Ergebnisse verallgemeinern lassen würden. Stattdessen soll lediglich überprüft werden, ob diese tatsächlich keine 100%ige Korrektheit für solch einfache Problemstellungen erreichen.

## Aufgabenstellungen für Datenbank-Vorlesungen

In diesem Abschnitt werden die Ergebnisse einiger Experimente vorgestellt. Hierbei ist das Ziel, zu untersuchen, welche Ergebnisse Studierende erzielen werden, wenn sie sich ihre Aufgabenstellungen (z.B. aus Projektarbeiten oder Prüfungen) von einer KI generieren lassen. Deshalb wurde auch nicht versucht, speziell für SQL geeignete KI-Plattformen oder Vorgehensweisen zu verwenden, da diese den Studierenden wohl nicht bekannt sind.

Stattdessen wurden die folgenden „populären“ (z.B. aus Medien bekannten) LLMs getestet:

1. Microsoft Copilot in der von der HNU lizenzierten Version. Diese verwendete zum Zeitpunkt der Tests ChatGPT in der Version GPT4

2. ChatGPT in der über openai.com öffentlich und kostenlos zugänglichen Version. Dies war bei Durchführung der Experimente die Version GPT-4-turbo
3. DeepSeek in der Version V3

Auch der Schwierigkeitsgrad der Aufgabenstellungen und die Komplexität des zugrunde liegenden Datenbank-Schemas orientieren sich an dem, was in einer (ersten) Vorlesung zum Thema Datenbanken üblicherweise zu erwarten ist. Es sind also keine extrem komplexen Abfragen zu erstellen und die Datenbank enthält nicht besonders viele Tabellen und Spalten. Dementsprechend generieren die LLMs deutlich bessere Ergebnisse, als die in der Literatur erwähnten, z.B. in einem realistischen betrieblichen Umfeld oder bei komplexen Benchmarks (vgl. Abschnitt „Literaturanalyse“).

## **Verwendete Szenarien**

Wie bereits erwähnt, soll die Komplexität der verwendeten Datenbanken dem entsprechen, was in einer Vorlesung zum Thema Datenbanken zu erwarten ist. Im ersten Szenario werden Kunden, sowie ihre Adressen, Bestellung, Rechnungen etc. gespeichert. Dieses stammt (ebenso wie die Abfragen 1 bis 18, s.u.) aus einer Datenbankvorlesung, welche im 1. bzw. 2. Semester in „Wirtschaftsinformatik-artigen“ Studiengängen an der HNU angeboten wird. Abb. 1a zeigt das entsprechende Entity-Relationship-Diagramm (um die Bedeutung der Inhalte ohne längere textuelle Erklärung verständlich zu machen). Abb. 1b stellt die daraus resultierenden Datenbanktabellen als EER-Diagramm in MySQL-Workbench dar (Primärschlüssel sind mit einem Schlüssel-Symbol markiert, Fremdschlüssel haben ein rotes Symbol).

Das in diesen Vorlesungen verwendete Szenario enthält keine Entitätstypen, zwischen denen es mehrere Beziehungen gibt. Außerdem ist keine (rekursive) Beziehung eines Entitätstyps mit sich selbst enthalten. Dies schränkt die Menge an möglichen Aufgabenstellungen ein. Deshalb wurden auch Experimente mit einem zweiten Szenario durchgeführt, in dem u.a. Mitarbeiter, die Abteilung, in der sie arbeiten bzw. die sie leiten, sowie Standorte gespeichert werden. Abb. 2a stellt das Szenario dieser Firmen-Datenbank wieder als Entity-Relationship-Diagramm dar und Abb. 2b die Tabellen als EER-Diagramm. Dieses Szenario erlaubt komplexere Aufgabenstellungen, in welchen z.B. die Beziehungen „arbeitet in“ und „leitet“ gemeinsam oder sogar mehrfach benutzt werden müssen. Außerdem werden Abfragen möglich, welche Rekursion erfordern (in SQL eine sog. Common Table Expression: CTE), z.B. um auch indirekte Vorgesetzte zu berechnen. Solche Aufgabenstellungen und -arten sind für Datenbank-Vorlesungen durchaus angemessen, allerdings eher in reinen Informatik-Studiengängen. Eine entsprechende Komplexität von SQL-Abfragen ist auch in einem betrieblichen Umfeld selbst für Berufseinsteiger realistisch.

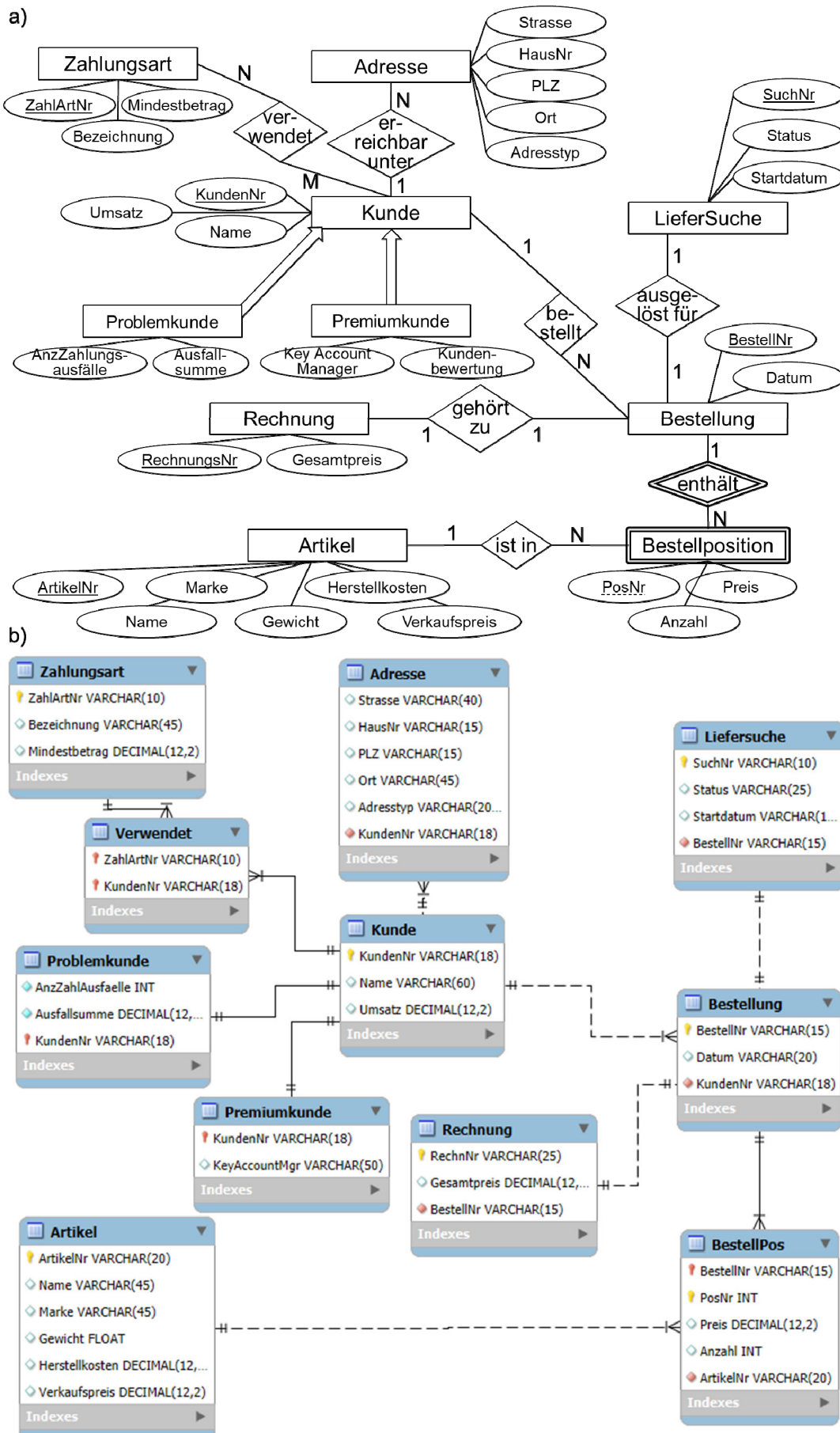
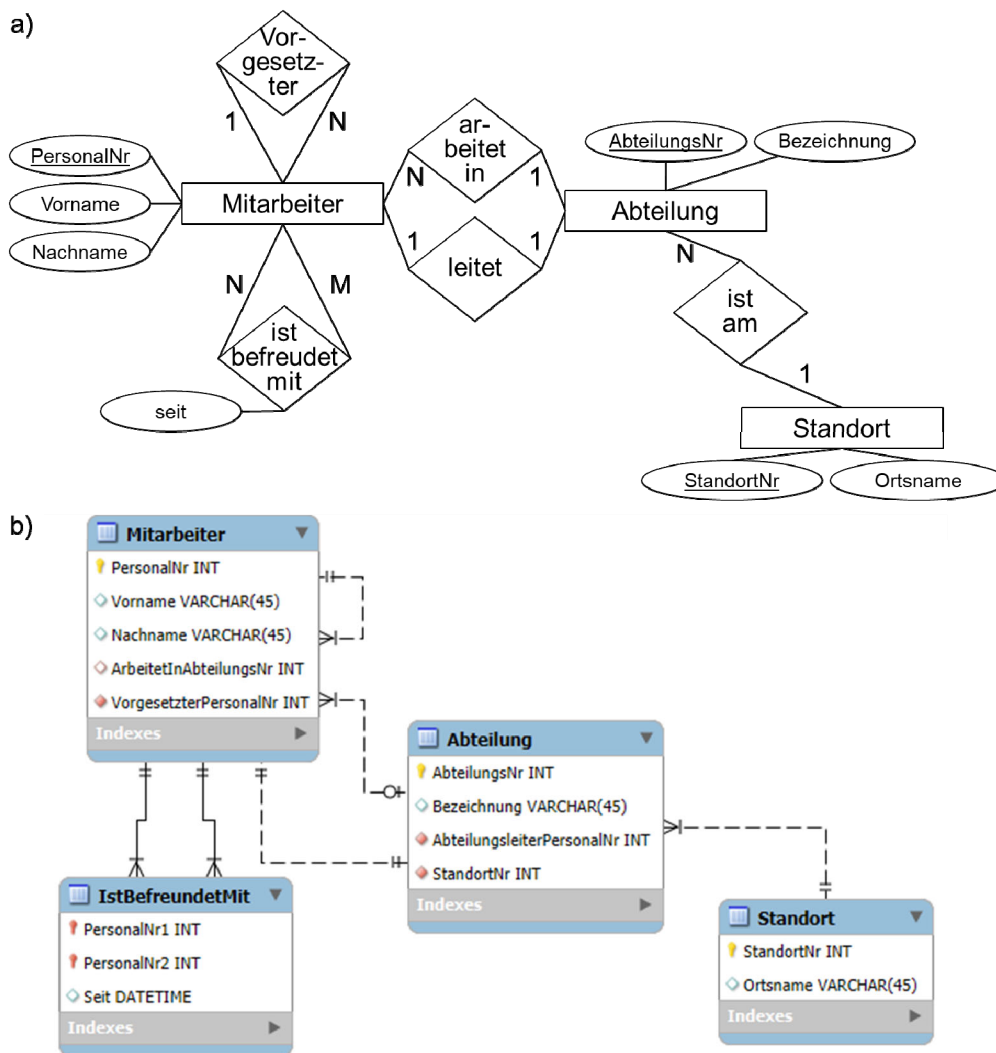


Abb. 1: a) Entity-Relationship-Diagramm und b) Datenbank-Tabellen der Kunden-Datenbank



**Abb. 2:** a) Entity-Relationship-Diagramm und b) Datenbank-Tabellen der Firmen-Datenbank

## Durchführung der Experimente

Zur Beantwortung der Frage, ob Studierende bzw. Berufseinsteiger aktuell noch eigene SQL-Kenntnisse benötigen, soll untersucht werden, ob KIs bereits in der Lage sind, SQL-Abfragen völlig fehlerfrei zu generieren. Nur wenn dies der Fall ist, kann man sich „quasi blind“ (d.h. ohne eigene Kompetenzen zum Thema SQL) auf ein generiertes Ergebnis verlassen. Dementsprechend ist irrelevant, ob eine generierte Abfrage für eine bestimmte Problemstellung in 99% der Fälle korrekt ist, oder nur in 80% - die Studierenden müssen in beiden Fällen in der Lage sein, eine generierte Abfrage zu überprüfen und ggf. selbst eine korrekte Abfrage zu erstellen. Sie benötigen also tiefgehende SQL-Kenntnisse, d.h. dieses Thema muss weiterhin in Datenbank-Vorlesungen gelehrt werden.

Da eine solche quantitative Bewertung der Qualität von generierten SQL-Abfragen also nicht hilfreich ist, wurde auf einen großen „Stichprobenumfang“ verzichtet: Das bedeutet, für eine Aufgabenstellung wurden nicht sehr viele Lösungen automatisch von der KI generiert und diese ebenfalls automatisch auf Korrektheit geprüft, um so einen Prozentsatz für die Korrektheit zu ermitteln. Stattdessen wurden die Lösungen (ähnlich wie bei der Korrektur einer

Klausur) vom Autor einzeln manuell geprüft. Diese Vorgehensweise wird als „manual matching“ bezeichnet. In (Kim et al., 2020) wird festgestellt, dass dies viel Aufwand erfordert, aber Falschbewertungen vermeidet, die bei den anderen (automatisch durchführbaren) Bewertungsmethoden auftreten können. Ein Nachteil ist, dass keine quantitativen Aussagen möglich sind, welche aber (wie bereits diskutiert) in unserem Kontext auch nicht erforderlich sind. Damit eine KI sinnvolle SQL-Abfragen generieren kann, müssen ihr die Namen der zugrundeliegenden Tabellen und deren Spalten bekannt sein. Diese könnten prinzipiell mittels eines „Prosa-Textes“ oder einer speziellen Syntax beschrieben werden. Hier wurde jedoch der Ansatz gewählt, die CREATE-TABLE-Statements als SQL-Befehle zu verwenden. Gründe hierfür sind, dass diese den Studierenden in Projektarbeiten üblicherweise ohnehin vorliegen, so dass es naheliegend ist, dass auch sie diese im Prompt verwenden. Außerdem wird diese Vorgehensweise vermutlich auch häufig in den Lerndaten der LLMs vorkommen, weil auch hier diese SQL-Befehle ohnehin vorhanden sein werden. Ganz konkret wurde deshalb aus den bereits erwähnten EER-Diagrammen durch MySQL-Workbench automatisch ein entsprechendes Skript generiert. Außerdem wurde davor und danach ein erklärender kurzer Text eingefügt, um die Aufgabenstellung im Prompt zu verdeutlichen. Schließlich endet der Prompt mit der konkreten Aufgabenstellung. Damit ergibt sich folgender Gesamtaufbau (wobei die CREATE-TABLE-Statements für die weiteren Tabellen weggelassen wurden, um die Lesbarkeit zu erhöhen):

Eine relationale Datenbank besteht aus folgenden als DDL beschriebenen Tabellen.

```
-- Schema KundenDB
```

```
CREATE SCHEMA IF NOT EXISTS `KundenDB` DEFAULT CHARACTER SET utf8 ;
USE `KundenDB` ;
```

```
-- Table `KundenDB`.`Kunde`
```

```
CREATE TABLE IF NOT EXISTS `KundenDB`.`Kunde` (
  `KundenNr` VARCHAR(18) NOT NULL,
  `Name` VARCHAR(60) NULL,
  `Umsatz` DECIMAL(12,2) NULL,
  PRIMARY KEY (`KundenNr`))
ENGINE = InnoDB;
```

```
-- Table `KundenDB`.`Bestellung`
```

```
CREATE TABLE ...
```

Erstelle eine SQL-Anfrage für folgende Problemstellung.

Übersicht über die Bestellungen mit zugehörigen Detaildaten.

Auszugebende Ergebnisspalten: BestellNr, Kundenname, Anzahl Bestellpositionen, Preis des teuersten Artikels

Der Anhang A enthält eine Liste aller Aufgabenstellungen. Wie im obigen Beispiel sind hierbei auch jeweils die auszugebenden Ergebnisspalten angegeben, weil sich diese nicht aus der inhaltlichen Beschreibung der Aufgabenstellung ergeben (d.h. die SELECT-Klausel ist ansonsten nicht eindeutig definiert). Die Aufgabenstellungen 1 bis 21 verwenden die in Abb. 1 dargestellte Kunden-Datenbank, ab der Nummer 22 wird die Firmen-Datenbank (vgl. Abb. 2) verwendet. Wurde nicht sofort eine korrekte Lösung generiert, so wurde die entsprechende Abfrage wiederholt ausgeführt.

Abb. 3 enthält eine Kurzbeschreibung der zu erwartenden Lösungen, die vollständigen Lösungen sind als SQL-Statement in Anhang B dargestellt. Selbstverständlich existieren auch andere korrekte Lösungen, wie z.B. die Verwendung einer Sub-Query anstatt einem INNER bzw. OUTER JOIN (für Abfrage 13 sind im Anhang B exemplarisch drei unterschiedliche Lösungen angegeben). Außerdem stellt Abb. 3 einen Überblick über die Abfrageergebnisse dar. Die Beschriftungen „korrekt“ (dunkelgrün) bedeuten hierbei, dass das entsprechende LLM sofort eine korrekte Lösung generiert hat. Diese Fälle sind für die weitere Analyse eher uninteressant, so dass nicht weiter auf sie eingegangen wird. Alle anderen (d.h. zumindest teilweise falschen) Lösungen werden im nächsten Unterabschnitt detailliert diskutiert. Im Falle von „noch korrekt“ wurde die Lösung zwar als korrekt gewertet, aber mit Einschränkungen. Dies betrifft z.B. Lösungen, in denen eine bestimmte Tabelle unnötigerweise

Nr	Kurzbeschreibung der Lösung	1. Copilot	2. ChatGPT	3. DeepSeek
1	nur 1 Tabelle benötigt, WHERE mit Ähnlichkeitsuche (d.h. LIKE)	korrekt	korrekt	korrekt
2	2 Tabellen (d.h. 1 INNER JOIN) und ORDER BY	korrekt	korrekt	korrekt
3	dasselbe mit OUTER JOIN	korrekt	korrekt	korrekt
4	dasselbe, aber nur Zeilen ohne Join-Partner (d.h. WHERE mit IS NULL)	korrekt	manchmal	korrekt
5	1 JOIN, GROUP BY und Aggregationsfunktion COUNT	korrekt	korrekt	korrekt
6	3 Tabellen (d.h. 2 x JOIN), Aggregation ohne GROUP BY	korrekt	korrekt	korrekt
7	UNION mit je 1 JOIN in beiden Teilanfragen	korrekt	korrekt	korrekt
8	3 Tabellen, WHERE mit LIKE, GROUP BY, mehrere Aggregationen	korrekt	korrekt	korrekt
9	4 Tabellen (d.h. 3x JOIN), jeweils OUTER JOIN	korrekt	korrekt	korrekt
10	3 Tabelle, GROUP BY, 2 Aggregationen	korrekt	korrekt	korrekt
11	SubQuery mit derselben Tabelle wie in Hauptquery	noch korrekt	noch korrekt	noch korrekt
12	3 Tabellen, GROUP BY, HAVING-Klausel, keine Aggregation im SELECT	korrekt	korrekt	korrekt
13	2x OUTER JOIN und IS NULL oder SubQuery oder HAVING	korrekt	korrekt	korrekt
14	3 Tabellen, GROUP BY, HAVING mit Multiplikation	korrekt	korrekt	korrekt
15	UNION mit 1x bzw. 2x JOIN und WHERE in den Teilanfragen	korrekt	korrekt	korrekt
16	SubQuery mit 1 JOIN und WHERE	falsch	korrekt	korrekt
17	6 Tabellen (d.h. 5x JOIN), WHERE	korrekt	korrekt	korrekt
18	SubQuery mit 1 JOIN und WHERE	falsch	korrekt	korrekt
19	SubQuery mit 2x JOIN und WHERE	falsch	korrekt	korrekt
20	SubQuery mit 2x JOIN und WHERE, zusätzl. 1x JOIN in Haupt-Query	falsch	korrekt	noch korrekt
21	SubQuery mit 2x JOIN und WHERE, zusätzl. 2x JOIN in Haupt-Query	korrekt	falsch	korrekt
22	JOIN (mit richtiger Beziehung) und WHERE	korrekt	korrekt	korrekt
23	JOIN (mit der anderen Beziehung) und WHERE	korrekt	korrekt	korrekt
24	rekursive Bez. genutzt (2x JOIN) & Bez. leitet (1 JOIN), WHERE	korrekt	manchmal	noch korrekt
25	rekursive Bez. genutzt (2x JOIN) & Bez. arbeitetIn (1 JOIN), WHERE	noch korrekt	teilweise	korrekt
26	zusätzlich JOIN mit Tabelle Standort	korrekt	manchmal	korrekt
27	rekursive Bez. und Bez. arbeitetIn & leitet genutzt (2x JOIN), WHERE	falsch	korrekt	teilweise
28	rekursive Bez. und Bez. arbeitetIn 2 mal genutzt (2x JOIN), WHERE	noch korrekt	falsch	korrekt
29	rekursive Query (Common Table Expression: CTE) für Tab. Mitarbeiter	korrekt	manchmal	korrekt
30	zusätzlich JOIN mit Tabelle Abteilung	falsch	falsch	korrekt
31	zusätzlich JOIN mit Tabelle Standort	falsch	manchmal	korrekt
32	rekursive Bez. genutzt (2x JOIN) & Bez. leitet & Vorgesetzter, WHERE	teilweise	teilweise	manchmal
33	rekursive Bez. genutzt (2x JOIN) & 2x Bez. leitet, WHERE	falsch	korrekt	manchmal

**Abb. 3:** Überblick über die Korrektheit der generierten Lösungen für unterschiedliche LLMs

verwendet wurde oder weil (ohne dass dies so vorgegeben ist) angenommen wurde, dass Einträge in der Tabelle IstBefreundetMit (vgl. Abb. 2b) „inhaltlich doppelt“ vorkommen (d.h. etwa eine Zeile mit PersonalNr1 = 27 und PersonalNr1 = 88 sowie zusätzlich eine weitere Zeile mit PersonalNr1 = 88 und PersonalNr1 = 27 – dies kann in einer Datenbank so realisiert sein, dies muss aber nicht der Fall sein).

Als „teilweise“ korrekt wurden z.B. Lösungen gewertet, bei denen sich außer den korrekten Ergebniszeilen auch falsche ergeben. So sind in Abfrage 26 die Standorte gefragt, an denen Freunde von Peter Müller arbeiten. ChatGPT hat jedoch eine Lösung generiert, bei der außer diesen auch der Standort von Peter Müller selbst im Ergebnis enthalten ist.

Bei Einträgen, die mit „manchmal“ beschriftet sind, hat das LLM zuerst eine falsche Lösung generiert. Eine wiederholte Ausführung ergab dann allerdings ein korrektes SQL-Statement. Hierbei wurde die Aufgabenstellung teilweise jedoch umformuliert, um die Verständlichkeit zu verbessern.

Bei „falsch“ markierten Einträgen wurden auch im Wiederholungsfall falsche Lösungen generiert.

## ***Auffällige Einzelergebnisse***

Im Folgenden wird auf alle nicht „perfekt gelösten“ Fälle eingegangen, d.h. auf diejenigen, die in Abb. 3 nicht mit korrekt markiert sind. Außerdem werden einige sonstige Auffälligkeiten der von den LLMs generierten Ergebnisse dargestellt.

## ***Unzulängliche Lösungen***

Die nicht fehlerfreien Lösungen sind im Folgenden geordnet nach LLM dargestellt (und nicht nach Aufgabenstellung), da manche Aufgabenstellung nur von einem einzigen LLM falsch gelöst wurden und sich zudem die Art der Fehler selbst bei derselben Aufgabenstellung bei verschiedenen LLMs stark unterscheidet.

### ***1. Microsoft Copilot (auf Basis ChatGPT in der Version GPT4)***

Dieses Produkt hat die meisten völlig falschen SQL-Abfragen generiert. Im Folgenden wird zusätzlich auf die teilweise oder geringfügig falschen Ergebnisse eingegangen. Um die Übersichtlichkeit zu erhöhen ist dabei stets die jeweilige Aufgaben-Nummer (vgl. Abb. 3 sowie Anhang A und B) sowie die Aufgabenstellung (in verkürzter Form) angegeben. Einige Experimente sind dabei ausführlicher und in verschiedenen Varianten dargestellt, um das Verhalten des LLM bei einem unterschiedlich detaillierten Prompt zu erläutern.

#### ***11. Der Kunde (oder die Kunden) mit dem größten Umsatz***

Die ursprüngliche Aufgabenstellung lautete hier „Der Kunde mit dem größten Umsatz“. Hierfür wurde die unten dargestellte Lösung generiert. Diese funktioniert zwar, aber nur wenn es exakt einen einzigen Kunden mit diesem Maximalumsatz gibt. Das „LIMIT 1“ führt nämlich dazu, dass nur eine Zeile ausgegeben wird. Durch die Sortierung ist das dann tatsächlich der Kunde mit dem größten Umsatz – oder eben einer dieser Kunden, falls es mehrere solche gibt.



```
SELECT k.KundenNr, k.Name, k.Umsatz  
FROM Kunde k  
ORDER BY k.Umsatz DESC  
LIMIT 1
```

Weil die LIMIT-Klausel nicht von allen Produkten unterstützt wird, wurde Copilot im Prompt zudem aufgefordert, eine Lösung zu generieren, die auch in Microsoft Access funktioniert. Daraufhin entstand die unten dargestellte Lösung (mit TOP 1 anstatt LIMIT 1), welche auch nur einen einzigen Kunden ausgibt.

```
SELECT TOP 1 Kunde.KundenNr, Kunde.Name, Kunde.Umsatz  
FROM Kunde  
ORDER BY Kunde.Umsatz DESC
```

Nach einer Klarstellung der Aufgabenstellung („Der Kunde (oder die Kunden) mit ...“, s.o.) wurde eine korrekte Lösung generiert. Allerdings ist es für einen Benutzer des LLMs evtl. nicht einfach, zu erkennen, dass die Aufgabenstellung nicht ideal formuliert war und deshalb eine möglicherweise falsche Lösung entstanden ist.

**16. Bestellungen, in denen kein Artikel mit einem Gewicht von 1715 (kg) bestellt wurde.**

Hierfür wurde folgende Lösung generiert (die Einrückungen wurden nachträglich eingefügt, um die einzelnen SQL-Klauseln besser erkennbar zu machen). Die Aufgabenstellung wurde 5 mal ausgeführt, wobei jedes Mal folgendes falsche Ergebnis generiert wurde:

```
SELECT b.BestellNr, b.Datum, b.KundenNr  
FROM Bestellung b LEFT JOIN BestellPos bp ON b.BestellNr = bp.BestellNr  
LEFT JOIN Artikel a ON bp.ArtikelNr = a.ArtikelNr AND a.Gewicht = 1715  
WHERE a.ArtikelNr IS NULL
```

Um diese Aufgabenstellung zu lösen, sollte eigentlich eine Sub-Query verwendet werden (vgl. Anhang B). Dies ist hier nicht der Fall. Die Lösung ist aus folgendem Grund falsch: Angenommen eine Bestellung hat 2 Bestellpositionen (d.h. 2 Zeilen in der Tabelle BestellPos) hat, mit den Artikeln A2 (mit 1715 kg) und A3 (mit einem anderen Gewicht). Wegen dem Artikel A2 mit 1715 kg sollte diese Bestellung nicht im Ergebnis sein. Allerdings findet die Bestellposition mit dem Artikel A3 keinen Join-Partner in der Tabelle Artikel, weil die Join-Bedingung „a.Gewicht = 1715“ nicht erfüllt ist. Wegen dem OUTER JOIN werden die Felder aus der Tabelle Artikel mit NULL-Werten aufgefüllt, so dass die WHERE-Klausel erfüllt ist und diese Bestellung fälschlicherweise ausgegeben wird.

**a. Variante: Bestellungen, in denen kein Artikel mit einem Gewicht über 1715 (kg) bestellt wurde**

Wird die Aufgabenstellung so modifiziert, dass (anstatt Artikel mit 1715 kg) solche mit einem größeren Gewicht nicht bestellt werden dürfen, dann wird eine korrekte Lösung generiert. Dies ist erstaunlich, weil sich der SQL-Befehl nur durch ein > anstatt einem = unterscheidet.

**b. Zusätzliche Erklärung: Bestellungen, in denen kein Artikel mit einem Gewicht von 1715 (kg) bestellt wurde. Achtung: Das ist etwas anderes als Bestellungen, in denen auch ein Artikel mit einem anderen Gewicht als 1715 bestellt wurde.**

Diese Erklärung beschreibt den Fehlerfall der ursprünglichen Lösung. Trotzdem generiert dieses LLM dieselbe falsche SQL-Abfrage.

**c. Zusätzliche Erklärung und Beispiel: Bestellungen, in denen kein Artikel mit einem Gewicht von 1715 (kg) bestellt wurde. Achtung: Das ist etwas anderes als Bestellungen, in denen auch**

*ein Artikel mit einem anderen Gewicht als 1715 bestellt wurde. Beispielsweise gehört eine Bestellung für einen Artikel mit 1715 kg und einem Artikel mit 2000 kg nicht in das Ergebnis.* Hier wurde eine korrekte Lösung generiert. Jedoch erfordert sowohl die Erklärung als auch das Beispiel große Kompetenzen bzgl. Erkennung des Fehlers und eines konkreten Problemfalls. Es sind also sehr gute SQL-Kenntnisse erforderlich (vermutlich ist es einfacher, die SQL-Anweisung selbst zu erstellen).

**d. Weiteres erklärendes Beispiel:** *Bestellungen, in denen kein Artikel mit einem Gewicht von 1715 (kg) bestellt wurde. Achtung: Das ist etwas anderes als Bestellungen, in denen auch ein Artikel mit einem anderen Gewicht als 1715 bestellt wurde. Beispielsweise gehört eine Bestellung für einen Artikel mit 1715 kg und einem Artikel mit 2000 kg nicht in das Ergebnis. Eine Bestellung ganz ohne Bestellpositionen (und damit Artikel) gehört hingegen schon in das Ergebnis.*

Daraufhin wurde eine korrekte Lösung generiert, die aber seltsamerweise ein GROUP BY und HAVING verwendet, sowie ein nicht von allen Datenbank-Management-Systemen unterstütztes CASE:

```
SELECT b.BestellNr, b.Datum, b.KundenNr
FROM Bestellung b LEFT JOIN BestellPos bp ON b.BestellNr = bp.BestellNr
      LEFT JOIN Artikel a ON bp.ArtikelNr = a.ArtikelNr
GROUP BY b.BestellNr, b.Datum, b.KundenNr
HAVING SUM(CASE WHEN a.Gewicht = 1715 THEN 1 ELSE 0 END) = 0
```

**18. Kunden, die die Zahlungsart Kreditkarte nie verwendet haben.**

Dies führte zu einer falschen Lösung, weil die Bezeichnung Kreditkarte als ZahlArtNr interpretiert wurde. Um die Aufgabenstellung eindeutiger zu machen, wurde sie wie folgt abgewandelt:

*Kunden, die die Zahlungsart mit der Bezeichnung Kreditkarte nie verwendet haben.*

Eine korrekte Lösung sollte wieder eine Sub-Query verwenden. Die generierte Lösung war falsch und hat wieder denselben Aufbau wie bei Aufgabenstellung 16:

```
SELECT k.KundenNr, k.Name
FROM Kunde k LEFT JOIN Verwendet v
      ON k.KundenNr = v.KundenNr AND v.ZahlArtNr = 'Kreditkarte'
WHERE v.KundenNr IS NULL
```

Es wurde wieder untersucht, inwieweit zusätzliche Erklärungen und Beispiele hilfreich sind: Es genügte bereits eine Erklärung (analog 16b), um eine korrekte Lösung zu erhalten. Dasselbe gilt für Erklärung und Beispiel (analog 16c). Wenn zusätzlich ein dem Fall 16d entsprechender Erklärungstext angegeben wird (... Ein Kunde ganz ohne Zahlungsarten hingegen schon), dann entsteht wieder die oben dargestellte falsche Lösung. Diese Vorgehensweise beim Erstellen eines Prompt ist also nicht zuverlässig.

**19. Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben.**

Die Aufgabenstellung ist ähnlich zu Nr. 16, wobei jedoch ein zusätzlicher Join mit der Tabelle Bestellung in der Sub-Query erforderlich ist. Außerdem muss die Tabelle Kunde verwendet werden, weil diese die auszugebenden Spalten enthält. Diese Tabelle ist in der generierten Lösung zwar vorhanden, aber ansonsten wurde derselbe Fehler wie bei Nr. 16 gemacht:

```
SELECT k.KundenNr, k.Name, k.Umsatz
FROM Kunde k LEFT JOIN Bestellung b ON k.KundenNr = b.KundenNr
      LEFT JOIN BestellPos bp ON b.BestellNr = bp.BestellNr
      LEFT JOIN Artikel a ON bp.ArtikelNr = a.ArtikelNr AND a.Gewicht = 1715
WHERE a.ArtikelNr IS NULL
```

**20. Die Adressen derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben.**

Auch dies folgt demselben Muster, wobei zusätzlich ein Join in der Haupt-Query nötig ist (mit der Tabelle Adresse). Die generierte Lösung enthält wieder denselben Fehler:

```
SELECT DISTINCT a.Strasse, a.HausNr, a.PLZ, a.Ort
FROM KundenDB.Adresse a JOIN KundenDB.Kunde k ON a.KundenNr = k.KundenNr
      LEFT JOIN KundenDB.Bestellung b ON k.KundenNr = b.KundenNr
      LEFT JOIN KundenDB.BestellPos bp ON b.BestellNr = bp.BestellNr
      LEFT JOIN KundenDB.Artikel ar ON bp.ArtikelNr = ar.ArtikelNr AND ar.Gewicht = 1715
WHERE ar.ArtikelNr IS NULL
```

Auch die Aufgabenstellung 21 ist hierzu sehr ähnlich, wobei sogar 2 Join-Operationen in der Haupt-Query nötig sind. Erstaunlicherweise wurde hierfür jedoch eine korrekte Lösung generiert.

**25. Alle Abteilungen, in denen Freunde des Mitarbeiters Peter Müller arbeiten**

Hier wurde die nachfolgend angegebene Lösung generiert. Diese ist nur dann korrekt, wenn alle Freundschaftsbeziehungen „doppelt“ in der Tabelle IstBefreundetMit gespeichert sind. Das bedeutet, dass außer einer Zeile mit PersonalNr1 = X und PersonalNr2 = Y zusätzlich eine Zeile mit PersonalNr1 = Y und PersonalNr2 = X vorhanden sein muss. Es ist zwar möglich, diese Tabelle so zu befüllen, das muss aber nicht der Fall sein und war bei dem im Prompt abgegebenen Schema auch nicht so vorgegeben. Deswegen wird diese Lösung lediglich als „noch korrekt“ gewertet.

```
SELECT DISTINCT A.AbtellungsNr, A.Bezeichnung
FROM Abteilung A JOIN Mitarbeiter M ON A.AbtellungsNr = M.ArbeitetInAbtellungsNr
      JOIN IstBefreundetMit B ON M.PersonalNr = B.PersonalNr2
      JOIN Mitarbeiter P ON B.PersonalNr1 = P.PersonalNr
WHERE P.Vorname = 'Peter' AND P.Nachname = 'Müller'
```

**27. Alle Abteilungsleiter von Freunden des Mitarbeiters Peter Müller**

Die hierfür generierte (nachfolgend dargestellte) Lösung funktioniert nicht. Korrekt realisiert ist, dass m (Peter Müller) mit m1 befreundet ist. Die (letzten beiden) Join-Bedingungen m1.PersonalNr = a.AbtellungsleiterPersonalNr und a.AbtellungsleiterPersonalNr = m2.PersonalNr sorgen zusammen aber dafür, dass m1 und m2 identisch sind. Korrekt wäre, die Abteilung zu ermitteln, in der m2 arbeitet (nicht: leitet). Deshalb müsste die erste dieser Bedingungen m1.ArbeitetInAbtellungsNr = a.AbtellungsNr lauten (vgl. Anhang B).

```
SELECT DISTINCT a.AbleitungsleiterPersonalNr AS PersonalNr, m2.Vorname, m2.Nachname
FROM AbteilungsDB.Mitarbeiter m JOIN AbteilungsDB.IstBefreundetMit f
    ON m.PersonalNr = f.PersonalNr1 OR m.PersonalNr = f.PersonalNr2
JOIN AbteilungsDB.Mitarbeiter m1
    ON (m1.PersonalNr = f.PersonalNr1 OR m1.PersonalNr = f.PersonalNr2)
    AND m1.PersonalNr != m.PersonalNr
JOIN AbteilungsDB.Ableitung a ON m1.PersonalNr = a.AbleitungsleiterPersonalNr
JOIN AbteilungsDB.Mitarbeiter m2 ON a.AbleitungsleiterPersonalNr = m2.PersonalNr
WHERE m.Vorname = 'Peter' AND m.Nachname = 'Müller'
```

**28. Alle Mitarbeiter, die in der selben Abteilung arbeiten wie ein Freund von Peter Müller**

Die generierte Lösung ist nicht falsch. Sie ist allerdings sehr seltsam, weil nach dem Join mit der Tabelle Mitarbeiter p keine Join-Bedingung folgt, die eine Verknüpfung mit einer anderen Tabelle realisiert. Zudem wird die im Join davor angegebene Tabelle Abteilung a ansonsten überhaupt nicht verwendet, so dass dieser Join unnötig ist und ersatzlos gestrichen werden kann.

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM Mitarbeiter m JOIN Abteilung a ON m.ArbeitetInAbteilungsNr = a.AbleitungsNr
JOIN Mitarbeiter p ON p.Nachname = 'Müller' AND p.Vorname = 'Peter'
JOIN IstBefreundetMit f ON (f.PersonalNr1 = p.PersonalNr
    OR f.PersonalNr2 = p.PersonalNr)
JOIN Mitarbeiter mf ON (mf.PersonalNr = f.PersonalNr1
    OR mf.PersonalNr = f.PersonalNr2) AND mf.PersonalNr != p.PersonalNr
WHERE m.ArbeitetInAbteilungsNr = mf.ArbeitetInAbteilungsNr
```

**30. Die Abteilungen, in denen Vorgesetzte von Peter Müller arbeiten, auch indirekt Vorgesetzte**

Hier wurde die unten dargestellte, inhaltlich falsche Abfrage generiert. Der rekursive Teil ermittelt (korrekt) die Personalnummern von Mitarbeitern und ihren Vorgesetzten. Der Teil danach ist jedoch falsch, weil wegen der Join-Bedingung a.AbleitungsleiterPersonalNr = v.PersonalNr Abteilungen ausgegeben, die von einem (indirekten) Vorgesetzten von Peter Müller geleitet werden. Es sind aber Abteilungen gesucht, in denen diese Personen arbeiten. Hierzu müsste also das Attribut a.ArbeitetInAbteilungsNr verwendet werden. Damit dieses verwendbar wird, wäre ein weiterer Join (mit der Tabelle Mitarbeiter) notwendig, d.h. dieser fehlt in der vorgeschlagenen Lösung.

Zudem ist in v.PersonalNr auch Peter Müller selbst enthalten. Es selbst ist in der Ausgabe aber nicht zu berücksichtigen, sondern nur seine Vorgesetzten. Deshalb darf diese Spalte in der letzten Join-Bedingung nicht verwendet werden, sondern es muss mit VorgesetzterPersonalNr verglichen werden (für den Join mit der o.g. Tabelle Mitarbeiter).

Dieser Fehler ist (insb. für eher unerfahrene Studierende) vermutlich schwer zu finden, weil die Abfrage syntaktisch korrekt ist und auf den ersten Blick plausibel klingt. Es ist aber etwas anderes, dass eine Person (auch wenn sie Vorgesetzter ist) in einer Abteilung arbeitet, als wenn diese Person sie leitet.

WITH RECURSIVE Vorgesetzte AS (

```
SELECT m.PersonalNr, m.VorgesetzterPersonalNr
FROM AbteilungsDB.Mitarbeiter m
WHERE m.Nachname = 'Müller' AND m.Vorname = 'Peter'
UNION
SELECT m.PersonalNr, m.VorgesetzterPersonalNr
FROM AbteilungsDB.Mitarbeiter m INNER JOIN Vorgesetzte v
ON m.PersonalNr = v.VorgesetzterPersonalNr
```

)

```
SELECT DISTINCT a.AbteilungsNr, a.Bezeichnung
FROM AbteilungsDB.Abteilung a JOIN Vorgesetzte v
ON a.AbteilungsleiterPersonalNr = v.PersonalNr
```

**31. Die Abteilungen und die Standorte an denen Vorgesetzte von Peter Müller arbeiten, auch indirekt Vorgesetzte**

Hier wurde derselbe Fehler gemacht, wie bei der vorherigen Aufgabenstellung, d.h. die letzte Join-Bedingung enthält wieder beide bereits beschriebenen Fehler.

WITH RECURSIVE Vorgesetzte AS (

```
SELECT m.PersonalNr, m.VorgesetzterPersonalNr
FROM AbteilungsDB.Mitarbeiter m
WHERE m.Vorname = 'Peter' AND m.Nachname = 'Müller'
UNION ALL
SELECT m.PersonalNr, m.VorgesetzterPersonalNr
FROM AbteilungsDB.Mitarbeiter m
INNER JOIN Vorgesetzte v ON m.PersonalNr = v.VorgesetzterPersonalNr
```

)

```
SELECT a.AbteilungsNr, a.Bezeichnung, s.StandortNr, s.Ortsname
FROM AbteilungsDB.Abteilung a INNER JOIN AbteilungsDB.Standort s
ON a.StandortNr = s.StandortNr
INNER JOIN Vorgesetzte v ON a.AbteilungsleiterPersonalNr = v.PersonalNr
```

**32. Die Abteilungsleiter, die mit einem Vorgesetzten von Peter Müller befreundet sind**

Die generierte Lösung enthält einen eher kleinen Fehler: Der Mitarbeiter m ist mit pm befreundet (dem Vorgesetzten von Peter Müller). Jedoch kann auch m = pm sein (z.B. falls bei beiden OR-Bedingungen die PersonalNr1 verwendet wird). Also werden nicht nur Freunde des Vorgesetzten, sondern auch dieser Vorgesetzte selbst ausgegeben. In der SQL-Abfrage fehlt die im Anhang B mit AND verknüpfte Bedingung, die dafür sorgt, dass die PersonalNr von m und pm unterschiedlich sind.

```
SELECT DISTINCT a.AbteilungsleiterPersonalNr AS PersonalNr, m.Vorname, m.Nachname
FROM AbteilungsDB.Abteilung a JOIN AbteilungsDB.Mitarbeiter m
ON a.AbteilungsleiterPersonalNr = m.PersonalNr
JOIN AbteilungsDB.IstBefreundetMit ib ON m.PersonalNr = ib.PersonalNr1
OR m.PersonalNr = ib.PersonalNr2
JOIN AbteilungsDB.Mitarbeiter pm ON (ib.PersonalNr1 = pm.PersonalNr
OR ib.PersonalNr2 = pm.PersonalNr)
```

```
WHERE pm.PersonalNr IN (  
    SELECT VorgesetzterPersonalNr  
    FROM AbteilungsDB.Mitarbeiter  
    WHERE Vorname = 'Peter' AND Nachname = 'Müller' )
```

**33. Die Abteilungsleiter, die mit dem Abteilungsleiter von Peter Müller befreundet sind**

Hier ist die generierte Lösung falsch: Wegen 1. Join-Bedingung ist a1 die Abteilung, die Peter Müller (m1) leitet. Dass er Abteilungsleiter ist, wird in der Aufgabenstellung jedoch gar nicht gefordert, sondern es geht um die Abteilung, in der er arbeitet. Deshalb müsste (anstatt der PersonalNr) die AbteilungsNr in der Join-Bedingung verwendet werden, also die Bedingung m1.ArbeitetInAbteilungsNr = a1.AbteilungsNr.

```
SELECT DISTINCT m2.PersonalNr, m2.Vorname, m2.Nachname  
FROM AbteilungsDB.Mitarbeiter m1 JOIN AbteilungsDB.Abteilung a1  
    ON m1.PersonalNr = a1.AbteilungsleiterPersonalNr  
JOIN AbteilungsDB.IstBefreundetMit b  
    ON a1.AbteilungsleiterPersonalNr = b.PersonalNr1  
    OR a1.AbteilungsleiterPersonalNr = b.PersonalNr2  
JOIN AbteilungsDB.Abteilung a2  
    ON (b.PersonalNr1 = a2.AbteilungsleiterPersonalNr  
    OR b.PersonalNr2 = a2.AbteilungsleiterPersonalNr)  
    AND a2.AbteilungsleiterPersonalNr != a1.AbteilungsleiterPersonalNr  
JOIN AbteilungsDB.Mitarbeiter m2 ON a2.AbteilungsleiterPersonalNr = m2.PersonalNr  
WHERE m1.Vorname = 'Peter' AND m1.Nachname = 'Müller'
```

**2. ChatGPT (GPT-4-turbo)**

Im Folgenden sind die nicht vollständig korrekt generierten SQL-Abfragen von ChatGPT dargestellt. Dabei ist zu beachten, dass mit GPT-4-turbo eine andere Version des LLMs zugrunde liegt als bei den zuvor dargestellten Lösungen von Microsoft Copilot.

**4. Alle Bestellungen sortiert nach BestellNr, aber nur Bestellungen, für die (noch) keine Rechnung existiert**

Hier wurde die die Sortierung vergessen:

```
SELECT b.BestellNr, b.Datum AS 'Bestell-Datum'  
FROM Bestellung b LEFT JOIN Rechnung r ON b.BestellNr = r.BestellNr  
WHERE r.RechnNr IS NULL
```

Bei der Wiederholung des Tests wurde jedoch eine korrekte Lösung generiert, d.h. die ORDER-BY-Klausel ist vorhanden.

**11. Der Kunde (oder die Kunden) mit dem größten Umsatz**

Ebenso wie Microsoft Copilot wurde mit der ursprünglichen Aufgabenstellung („Der Kunde mit ...“) eine Lösung generiert, die nur funktioniert, wenn es nur einen einzigen solchen Kunden gibt. Grund ist wieder das „LIMIT 1“, aufgrund dessen nur eine Zeile ausgegeben wird.

```
SELECT k.KundenNr, k.Name, k.Umsatz  
FROM Kunde k  
ORDER BY k.Umsatz DESC  
LIMIT 1
```

Die Klarstellung der Aufgabenstellung führte auch hier zu einer korrekten Lösung.



**21. Die Zahlungsarten derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben**

Die nachfolgend dargestellte generierte Lösung enthält in der Sub-Query einen unnötigen Join mit der Tabelle Kunde, weil das zurückgegebene Attribut KundenNr ist auch in der Tabelle Bestellung enthalten ist.

In der Haupt-Query werden Joins mit den Tabellen Bestellung, BestellPos und Artikel durchgeführt. Dies führt zu einem unvollständigen (d.h. falschen) Ergebnis, weil dann Kunden (und damit deren Zahlungsarten) nicht berücksichtigt werden, denen gar keine Bestellung, BestellPos oder Artikel zugeordnet sind. Die Zahlungsarten solcher Kunden gehören jedoch in das Ergebnis, weil z.B. Kunden ohne eine Bestellung sicher keinen Artikel mit 1715 kg Gewicht bestellt haben.

```
SELECT DISTINCT Z.ZahlArtNr, Z.Bezeichnung
FROM KundenDB.Zahlungsart Z JOIN KundenDB.Verwendet V ON Z.ZahlArtNr = V.ZahlArtNr
      JOIN KundenDB.Kunde K ON V.KundenNr = K.KundenNr
      JOIN KundenDB.Bestellung B ON K.KundenNr = B.KundenNr
      JOIN KundenDB.BestellPos BP ON B.BestellNr = BP.BestellNr
      JOIN KundenDB.Artikel A ON BP.ArtikelNr = A.ArtikelNr
WHERE K.KundenNr NOT IN (
      SELECT DISTINCT K2.KundenNr
      FROM KundenDB.Kunde K2 JOIN KundenDB.Bestellung B2
            ON K2.KundenNr = B2.KundenNr
            JOIN KundenDB.BestellPos BP2 ON B2.BestellNr = BP2.BestellNr
            JOIN KundenDB.Artikel A2 ON BP2.ArtikelNr = A2.ArtikelNr
      WHERE A2.Gewicht = 1715 )
```

**a. Zusätzliche Erklärung:** Die Zahlungsarten derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben. Achtung: Zahlungsarten von Kunden, die gar keine Bestellung oder eine Bestellung ohne Bestellpositionen haben, gehören in das Ergebnis, weil diese Kunden dann ja keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben.

Ebenso wie beim Copilot wurde getestet, ob der eben beschriebene Fehler durch obigen Erklärungstext vermieden werden kann. Dies führte jedoch zu der nachfolgend dargestellten falschen SQL-Abfrage. Diese berücksichtigt auch Kunden, die irgendeinen Artikel mit einem anderen Gewicht als 1715 bestellt haben, auch wenn sie zusätzlich einen Artikel mit 1715 kg bestellt haben – also eigentlich explizit aus dem Ergebnis ausgeschlossen werden sollen.

```
SELECT DISTINCT z.ZahlArtNr, z.Bezeichnung
FROM Zahlungsart z JOIN Verwendet v ON z.ZahlArtNr = v.ZahlArtNr
      LEFT JOIN Kunde k ON v.KundenNr = k.KundenNr
      LEFT JOIN Bestellung b ON k.KundenNr = b.KundenNr
      LEFT JOIN BestellPos bp ON b.BestellNr = bp.BestellNr
      LEFT JOIN Artikel a ON bp.ArtikelNr = a.ArtikelNr
WHERE (a.Gewicht IS NULL OR a.Gewicht <> 1715)
```

**b. Zusätzliche Erklärung und Beispiel:** Die Zahlungsarten derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben. Achtung: Zahlungsarten von Kunden, die gar keine Bestellung oder eine Bestellung ohne Bestellpositionen haben, gehören in das Ergebnis, weil diese Kunden dann ja keinen Artikel mit einem Gewicht von 1715 (kg) bestellt

*haben. Kunden, die Artikel mit einem anderen Gewicht und zusätzlich Artikel mit 1715 (kg) bestellt haben, dürfen nicht berücksichtigt werden.*

Damit wurde ein korrektes Ergebnis generiert. Jedoch muss wieder festgestellt werden, dass die Fehlererkennung und geeignete Formulierung von Erklärungen und Beispielen schwierig ist und unerfahrene Anwender vermutlich überfordert.

```
SELECT DISTINCT z.ZahlArtNr, z.Bezeichnung
FROM Zahlungsart z JOIN Verwendet v ON z.ZahlArtNr = v.ZahlArtNr
      JOIN Kunde k ON v.KundenNr = k.KundenNr
WHERE k.KundenNr NOT IN (
      SELECT DISTINCT b.KundenNr
      FROM Bestellung b JOIN BestellPos bp ON b.BestellNr = bp.BestellNr
            JOIN Artikel a ON bp.ArtikelNr = a.ArtikelNr
      WHERE a.Gewicht = 1715 )
```

#### **24. Alle Abteilungen, die Freunde des Mitarbeiters Peter Müller leiten**

Die generierte Lösung ist wegen der letzten Join-Bedingung A.AnteilungsleiterPersonalNr = M.PersonalNr falsch: Aufgrund der WHERE-Klausel ist M = Peter Müller. Durch die erwähnte Join-Bedingung ist er Abteilungsleiter der Abteilung a. Es wird also die von ihm geleitete Abteilung ausgegeben – Aufgabenstellung war jedoch, Abteilungen zu ermitteln, die seine Freunde leiten. Für eine korrekte Lösung hätte in der letzten Join-Bedingung (anstatt M.) IFM.PersonalNr1/2 verwendet werden müssen.

```
SELECT A.AnteilungsNr, A.Bezeichnung
FROM AbteilungsDB.Mitarbeiter M JOIN AbteilungsDB.IstBefreundetMit IFM
      ON M.PersonalNr = IFM.PersonalNr1 OR M.PersonalNr = IFM.PersonalNr2
      JOIN AbteilungsDB.Anteilung A ON A.AnteilungsleiterPersonalNr = M.PersonalNr
WHERE (M.Vorname = 'Peter' AND M.Nachname = 'Müller')
```

Die Wiederholung des Tests führte zu einer fast richtigen Lösung. Diese unten dargestellte SQL-Abfrage führt 2 Joins mit der Tabelle Mitarbeiter durch (als m und p). Dies ist nicht falsch, aber unnötig (vgl. Anhang B).

In den Join-Bedingungen fehlt jedoch eine Bedingung, die verhindert, dass Peter Müller (p) als sein eigener Freund (m, der Abteilungsleiter) betrachtet wird (z.B., weil in beiden mit OR verknüpften Join-Bedingungen ihm.PersonalNr1 verwendet wird). Die Lösung ist also nur teilweise korrekt.

```
SELECT a.AnteilungsNr, a.Bezeichnung
FROM AbteilungsDB.Anteilung a JOIN AbteilungsDB.Mitarbeiter m
      ON a.AnteilungsleiterPersonalNr = m.PersonalNr
      JOIN AbteilungsDB.IstBefreundetMit ihm ON (m.PersonalNr = ihm.PersonalNr1
            OR m.PersonalNr = ihm.PersonalNr2)
      JOIN AbteilungsDB.Mitarbeiter p ON (ihm.PersonalNr1 = p.PersonalNr
            OR ihm.PersonalNr2 = p.PersonalNr)
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

#### **25. Alle Abteilungen, in denen Freunde des Mitarbeiters Peter Müller arbeiten**

Hier wurde derselbe Fehler gemacht, wie bei der vorherigen Aufgabe. d.h. Peter Müller erfüllt selbst die Join-Bedingungen: Da eine entsprechende Bedingung fehlt (vgl. AND in Anhang B),



kann er außer p auch m sein, so dass fälschlicherweise auch die Abteilung ausgegeben wird, in der Peter Müller selbst arbeitet – auch wenn dort kein Freund von ihm arbeitet.

```
SELECT DISTINCT a.AbtellungsNr, a.Bezeichnung
FROM AbteilungsDB.Abtellung a JOIN AbteilungsDB.Mitarbeiter m
      ON m.ArbeitetInAbteilungsNr = a.AbtellungsNr
      JOIN AbteilungsDB.IstBefreundetMit bf ON (bf.PersonalNr1 = m.PersonalNr
      OR bf.PersonalNr2 = m.PersonalNr)
      JOIN AbteilungsDB.Mitarbeiter p ON (p.PersonalNr = bf.PersonalNr1
      OR p.PersonalNr = bf.PersonalNr2)
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

## **26. Die Standorte von Abteilungen, in denen Freunde des Mitarbeiters Peter Müller arbeiten**

Hier sind die Join-Bedingungen unsinnig. Es wird zwar ein Join der Tabelle m (Peter Müller) mit IstBefreundetMit (ibm) durchgeführt, aber ibm ansonsten nicht verwendet. Stattdessen bewirkt die Bedingung a.AbtellungsNr = m.ArbeitetInAbteilungsNr, dass a die Abteilung ist, in der Peter Müller arbeitet, so dass sein eigener Standort ausgegeben wird.

```
SELECT DISTINCT s.StandortNr, s.Ortsname
FROM AbteilungsDB.Mitarbeiter m JOIN AbteilungsDB.IstBefreundetMit ibm
      ON m.PersonalNr = ibm.PersonalNr1
      OR m.PersonalNr = ibm.PersonalNr2
      JOIN AbteilungsDB.Abtellung a ON (a.AbtellungsNr = m.ArbeitetInAbteilungsNr)
      JOIN AbteilungsDB.Standort s ON s.StandortNr = a.StandortNr
WHERE m.Vorname = 'Peter' AND m.Nachname = 'Müller'
```

Die Wiederholung des Tests führte zu einer korrekten Lösung. Wie nachfolgend dargestellt, enthält diese jedoch ein CASE-Konstrukt, das nicht von allen Datenbanksystemen unterstützt wird.

```
SELECT DISTINCT s.StandortNr, s.Ortsname
FROM Mitarbeiter peter JOIN IstBefreundetMit f
      ON peter.PersonalNr = f.PersonalNr1 OR peter.PersonalNr = f.PersonalNr2
      JOIN Mitarbeiter freund
      ON freund.PersonalNr = CASE
      WHEN f.PersonalNr1 = peter.PersonalNr THEN f.PersonalNr2
      ELSE f.PersonalNr1
      END
      JOIN Abteilung a ON freund.ArbeitetInAbteilungsNr = a.AbtellungsNr
      JOIN Standort s ON a.StandortNr = s.StandortNr
WHERE peter.Vorname = 'Peter' AND peter.Nachname = 'Müller'
```

## **28. Alle Mitarbeiter, die in der selben Abteilung arbeiten wie ein Freund von Peter Müller**

Die generierte Abfrage ist falsch, weil der Mitarbeiter m ausgegeben wird. Dieser ist wegen den ersten beiden Join-Bedingungen mit Peter Müller (pm) befreundet. Es sollen aber nicht seine Freunde ausgegeben werden, sondern deren Abteilungskollegen.

Die 1. Bedingung in der WHERE-Klausel führt dazu, dass pm = Peter Müller ist. Eine weitere WHERE-Klausel in der Sub-Query mit derselben Bedingung ist unsinnig.

```
SELECT m.PersonalNr, m.Vorname, m.Nachname
FROM AbteilungsDB.Mitarbeiter m JOIN AbteilungsDB.IstBefreundetMit ibm
    ON m.PersonalNr = ibm.PersonalNr1 OR m.PersonalNr = ibm.PersonalNr2
JOIN AbteilungsDB.Mitarbeiter pm
    ON (ibm.PersonalNr1 = pm.PersonalNr OR ibm.PersonalNr2 = pm.PersonalNr)
WHERE pm.Vorname = 'Peter' AND pm.Nachname = 'Müller'
    AND m.ArbeitetInAbteilungsNr = ( SELECT ArbeitenInAbteilungsNr
    FROM AbteilungsDB.Mitarbeiter
    WHERE Vorname = 'Peter' AND Nachname = 'Müller')
    AND m.PersonalNr != pm.PersonalNr
```

Eine wiederholte Ausführung führte zu einer anderen falschen Lösung: Der ausgegebene Mitarbeiter m ist wegen der ersten beiden Join-Bedingungen mit peter (d.h. Peter Müller) befreundet. Die Aufgabenstellung „derselben Abteilung“ wird also nicht erfüllt, der letzte Join ändert daran nichts, weil die Tabelle Abteilung (a) ansonsten nicht verwendet wird.

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM AbteilungsDB.Mitarbeiter m JOIN AbteilungsDB.IstBefreundetMit ibm
    ON m.PersonalNr = ibm.PersonalNr1 OR m.PersonalNr = ibm.PersonalNr2
JOIN AbteilungsDB.Mitarbeiter peter ON (ibm.PersonalNr1 = peter.PersonalNr
    OR ibm.PersonalNr2 = peter.PersonalNr)
JOIN AbteilungsDB.Abteilung a ON m.ArbeitetInAbteilungsNr = a.AbteilungsNr
WHERE peter.Vorname = 'Peter' AND peter.Nachname = 'Müller'
    AND m.PersonalNr != peter.PersonalNr
```

## **29. Alle Vorgesetzten von Peter Müller, auch indirekt Vorgesetzte**

Die generierte Lösung enthält mehrere kleinere Fehler. So ist REKURSIVE mit K statt C geschrieben, weswegen die SQL-Abfrage syntaktisch falsch und damit nicht ausführbar ist. Ein inhaltlicher Fehler ist, dass im 1. und 2. SELECT die Spalte VorgesetzterPersonalNr fehlt. Deshalb ist die 1. Join-Bedingung falsch, weil es die Spalte VorgesetzterPersonalNr in der Tabelle Vorgesetzte (v) gar nicht gibt.

Schließlich ist die Sub-Query eine sehr umständliche Lösung. Es wäre viel einfacher, Peter Müller direkt aufgrund seines Namens in der Tabelle v auszuschließen.

```
WITH REKURSIVE Vorgesetzte AS (
    -- Basis: Peter Müller ist der Startpunkt
    SELECT m.PersonalNr, m.Vorname, m.Nachname
    FROM AbteilungsDB.Mitarbeiter m
    WHERE m.Vorname = 'Peter' AND m.Nachname = 'Müller'
    UNION ALL
    -- Rekursion: Vorgesetzte der gefundenen Personen
    SELECT m.PersonalNr, m.Vorname, m.Nachname
    FROM AbteilungsDB.Mitarbeiter m JOIN Vorgesetzte v
        ON m.PersonalNr = v.VorgesetzterPersonalNr
)
```

-- Auswahl der endgültigen Vorgesetzten

```
SELECT DISTINCT v.PersonalNr, v.Vorname, v.Nachname
FROM Vorgesetzte v
WHERE v.PersonalNr != ( SELECT PersonalNr
                        FROM AbteilungsDB.Mitarbeiter
                        WHERE Vorname = 'Peter' AND Nachname = 'Müller')
```

Eine Wiederholung des Tests führte zu folgender korrekten Lösung:

```
WITH RECURSIVE Vorgesetzte AS (
    -- Start mit Peter Müller, um seine direkten Vorgesetzten zu finden
    SELECT PersonalNr, Vorname, Nachname, VorgesetzterPersonalNr
    FROM AbteilungsDB.Mitarbeiter
    WHERE Vorname = 'Peter' AND Nachname = 'Müller'
    UNION ALL
    -- Rekursive Abfrage: alle Vorgesetzten der bereits gefundenen Vorgesetzten
    SELECT m.PersonalNr, m.Vorname, m.Nachname, m.VorgesetzterPersonalNr
    FROM AbteilungsDB.Mitarbeiter m INNER JOIN Vorgesetzte v
    ON m.PersonalNr = v.VorgesetzterPersonalNr
)
SELECT PersonalNr, Vorname, Nachname
FROM Vorgesetzte
```

Bei einem 3. Test fehlte wieder die Spalte VorgesetzterPersonalNr (wie beim 1. Test), aber zumindest RECURSIVE war korrekt geschrieben.

**30. Die Abteilungen, in denen Vorgesetzte von Peter Müller arbeiten, auch indirekt Vorgesetzte**

Hier wurde eine falsche Lösung generiert, in der die indirekten Vorgesetzte fehlen, weil gar keine Rekursion (CTE) in der Abfrage genutzt wird.

Zudem ist der Mitarbeiter v (Peter Müller) der Vorgesetzte der Mitarbeiter m (m.VorgesetzterPersonalNr = v.PersonalNr), deren Abteilungen ausgegeben werden. Korrekt wäre jedoch, die Abteilungen der Vorgesetzten auszugeben, und nicht die der Untergeordneten von Peter Müller.

```
SELECT DISTINCT a.AbcilungsNr, a.Bezeichnung
FROM AbteilungsDB.Abcilung a JOIN AbteilungsDB.Mitarbeiter m
    ON a.AbcilungsNr = m.ArbeitetInAbteilungsNr
    JOIN AbteilungsDB.Mitarbeiter v ON m.VorgesetzterPersonalNr = v.PersonalNr
WHERE v.Vorname = 'Peter' AND v.Nachname = 'Müller'
```

Die Wiederholung des Tests führte wieder zu einer Abfrage ohne Rekursion (CTE). Jedoch werden jetzt genau 2 Stufen der Vorgesetztenbeziehung verwendet, d.h. der direkte Vorgesetzte und alle Vorgesetzten ab der Stufe 3 fehlen. Außerdem wurde Peter Meier wieder als Vorgesetzter verwendet und nicht als Untergeordneter.

```
SELECT DISTINCT a.AbcilungsNr, a.Bezeichnung
FROM AbteilungsDB.Abcilung a JOIN AbteilungsDB.Mitarbeiter m
    ON m.ArbeitetInAbteilungsNr = a.AbcilungsNr
    JOIN AbteilungsDB.Mitarbeiter v1 ON v1.PersonalNr = m.VorgesetzterPersonalNr
    JOIN AbteilungsDB.Mitarbeiter v2 ON v2.PersonalNr = v1.VorgesetzterPersonalNr
WHERE v2.Vorname = 'Peter' AND v2.Nachname = 'Müller'
```

**31. Die Abteilungen und die Standorte an denen Vorgesetzte von Peter Müller arbeiten, auch indirekt Vorgesetzte**

Hier verhielt sich ChatGPT sehr seltsam, weil es keine eindeutige Antwort lieferte, sondern die nachfolgend dargestellten beiden „Reaktionen“. Dabei ist die Reaktion 1 korrekt. Die Reaktion 2 ist falsch, weil die Spalte AbteilungsleiterPersonalNr (in der WHERE-Klausel) verwendet wird, d.h. es werden die Abteilungsleiter der Vorgesetzten berücksichtigt, nicht die Vorgesetzten selbst.

**Reaktion 1**

```
WITH RECURSIVE Vorgesetzte AS (  
    -- Starte mit dem direkten Vorgesetzten von Peter Müller  
    SELECT m.VorgesetzterPersonalNr  
    FROM AbteilungsDB.Mitarbeiter m  
    WHERE m.Vorname = 'Peter' AND m.Nachname = 'Müller'  
    UNION ALL  
    -- Rekursive Suche nach allen indirekten Vorgesetzten  
    SELECT m.VorgesetzterPersonalNr  
    FROM AbteilungsDB.Mitarbeiter m JOIN Vorgesetzte v  
        ON m.PersonalNr = v.VorgesetzterPersonalNr  
)  
SELECT a.AbdteilungsNr, a.Bezeichnung, s.StandortNr, s.Ortsname  
FROM AbteilungsDB.Mitarbeiter m JOIN AbteilungsDB.Abdteilung a  
    ON m.ArbeitetInAbteilungsNr = a.AbdteilungsNr  
    JOIN AbteilungsDB.Standort s ON a.StandortNr = s.StandortNr  
WHERE m.PersonalNr IN (SELECT VorgesetzterPersonalNr FROM Vorgesetzte)
```

**Reaktion 2**

```
WITH RECURSIVE Vorgesetzte AS (  
    -- Basisfall: Direkter Vorgesetzter von Peter Müller  
    SELECT m.VorgesetzterPersonalNr  
    FROM AbteilungsDB.Mitarbeiter AS m  
    WHERE m.Vorname = 'Peter' AND m.Nachname = 'Müller'  
    UNION  
    -- Rekursiver Fall: Finde alle indirekten Vorgesetzten  
    SELECT m.VorgesetzterPersonalNr  
    FROM AbteilungsDB.Mitarbeiter AS m  
    INNER JOIN Vorgesetzte v ON m.PersonalNr = v.VorgesetzterPersonalNr  
)  
SELECT a.AbdteilungsNr, a.Bezeichnung, s.StandortNr, s.Ortsname  
FROM AbteilungsDB.Abdteilung AS a JOIN AbteilungsDB.Standort AS s  
    ON a.StandortNr = s.StandortNr  
WHERE a.AbdteilungsleiterPersonalNr IN (    SELECT VorgesetzterPersonalNr  
        FROM Vorgesetzte)
```

**32. Die Abteilungsleiter, die mit einem Vorgesetzten von Peter Müller befreundet sind**

Die generierte Lösung ist fast korrekt, wobei zusätzlich Vorgesetzte von Peter Müller berücksichtigt werden. Gewünscht war jedoch nur die Ausgabe der Abteilungsleiter von dessen

Freunden. Grund für den Fehler ist, dass (ähnlich wie bei der Aufgabenstellung 25) die Bedingung fehlt, dass die Mitarbeiter m und v nicht identisch sein dürfen (vgl. AND in Anhang B).

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM AbteilungsDB.Mitarbeiter AS m JOIN AbteilungsDB.Abteilung AS a
      ON m.PersonalNr = a.AbteilungsleiterPersonalNr
      JOIN AbteilungsDB.IstBefreundetMit AS ib
            ON m.PersonalNr = ib.PersonalNr1 OR m.PersonalNr = ib.PersonalNr2
      JOIN AbteilungsDB.Mitarbeiter AS v
            ON (ib.PersonalNr1 = v.PersonalNr OR ib.PersonalNr2 = v.PersonalNr)
WHERE v.PersonalNr IN (
      SELECT VorgesetzterPersonalNr
      FROM AbteilungsDB.Mitarbeiter
      WHERE Vorname = 'Peter' AND Nachname = 'Müller' )
```

### 3. DeepSeek (Version V3)

Dieses LLM hat in den durchgeführten Tests die meisten bereits beim ersten Versuch völlig korrekten Lösungen generiert. Bei falschen Lösungen führte eine Wiederholung des Tests schließlich zumindest zu einer teilweise (d.h. fast) korrekten Lösung.

#### 11. Der Kunde (oder die Kunden) mit dem größten Umsatz

Auch hier wurde bei der ursprünglichen Aufgabenstellung („Der Kunde mit ...“) eine Abfrage mit „LIMIT 1“ generiert (s.u.), die nur eine einzige Zeile ausgibt. Dieser Mangel war nach der Klarstellung der Aufgabenstellung verschwunden.

```
SELECT KundenNr, Name, Umsatz
FROM Kunde
ORDER BY Umsatz DESC
LIMIT 1
```

#### 20. Die Adressen derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben

Die generierte Lösung ist zwar korrekt, verwendet jedoch unnötigerweise die Tabelle Kunde in der Sub-Query. Da die KundenNr auch in der Tabelle Bestellung enthalten ist (b.KundenNr, vgl. Join-Bedingung), könnte auch diese von der Sub-Query zurückgegeben werden.

```
SELECT DISTINCT a.Strasse, a.HausNr AS HNr, a.PLZ, a.Ort
FROM KundenDB.Adresse a JOIN KundenDB.Kunde k ON a.KundenNr = k.KundenNr
WHERE k.KundenNr NOT IN (
      SELECT DISTINCT k.KundenNr
      FROM KundenDB.Kunde k JOIN KundenDB.Bestellung b ON k.KundenNr = b.KundenNr
            JOIN KundenDB.BestellPos bp ON b.BestellNr = bp.BestellNr
            JOIN KundenDB.Artikel ar ON bp.ArtikelNr = ar.ArtikelNr
      WHERE ar.Gewicht = 1715 )
```

#### 24. Alle Abteilungen, die Freunde des Mitarbeiters Peter Müller leiten

Wie z.B. bereits bei der Aufgabenstellung 25 beim Microsoft Copilot beschrieben, ist die unten dargestellte Lösung nur dann korrekt, wenn alle Freundschaftsbeziehungen „doppelt“ in der

Tabelle IstBefreundetMit eingetragen sind (was aber so nicht vorgegeben war). Die generierte Lösung wurde deshalb wieder nur als „noch korrekt“ gewertet.

```
SELECT A.AbteilungsNr, A.Bezeichnung
FROM Abteilung A JOIN Mitarbeiter M ON A.AbteilungsleiterPersonalNr = M.PersonalNr
      JOIN IstBefreundetMit F ON M.PersonalNr = F.PersonalNr2
      JOIN Mitarbeiter PM ON F.PersonalNr1 = PM.PersonalNr
WHERE PM.Vorname = 'Peter' AND PM.Nachname = 'Müller'
```

Bei den nächsten beiden Aufgabenstellungen wurde jeweils eine Lösung generiert, bei der dieser Fehler vermieden wurde (obwohl die Aufgabenstellungen teilweise komplizierter sind). Es existiert also keine „Zuverlässigkeit“ bzgl. der Qualität der generierten SQL-Abfragen.

### **27. Alle Abteilungsleiter von Freunden des Mitarbeiters Peter Müller**

Die generierte Lösung ist wegen der verschachtelten Sub-Queries recht verwirrend. Außerdem wird unnötigerweise (vgl. Anhang B) doppelt mit dem Namen Peter Müller verglichen. Inhaltlich ist die SQL-Abfrage fast korrekt. Der einzige Fehler ist, dass Peter Müller als sein eigener Freund betrachtet wird. Deswegen wird fälschlicherweise zusätzlich sein Abteilungsleiter ausgegeben, und nicht nur die seiner Freunde. Denselben Fehler machte z.B. auch Microsoft Copilot bei der Aufgabenstellung 32 und ChatGPT bei Nr. 25 und 32.

```
SELECT m.PersonalNr, m.Vorname, m.Nachname
FROM Mitarbeiter m JOIN Abteilung a ON m.PersonalNr = a.AbteilungsleiterPersonalNr
WHERE a.AbteilungsNr IN (
    SELECT ma.ArbeitetInAbteilungsNr
    FROM Mitarbeiter ma JOIN IstBefreundetMit f
        ON ma.PersonalNr = f.PersonalNr1 OR ma.PersonalNr = f.PersonalNr2
    WHERE (f.PersonalNr1 =
        (SELECT PersonalNr
         FROM Mitarbeiter
         WHERE Vorname = 'Peter' AND Nachname = 'Müller')
    OR f.PersonalNr2 =
        (SELECT PersonalNr
         FROM Mitarbeiter
         WHERE Vorname = 'Peter' AND Nachname = 'Müller')
    )
)
```

### **32. Die Abteilungsleiter, die mit einem Vorgesetzten von Peter Müller befreundet sind**

Die zuerst generierte Lösung ist fehlerhaft, weil auch der Vorgesetzte von Peter Müller ausgegeben wird. Grund hierfür ist, dass die Mitarbeiter m und v identisch sein können, z.B. wenn bei beiden Join-Bedingungen PersonalNr1 verwendet wird. Es fehlt also wieder eine mit AND verknüpfte Bedingung, die dies explizit ausschließt (vgl. Anhang B).

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM Mitarbeiter m JOIN Abteilung a ON m.PersonalNr = a.AbteilungsleiterPersonalNr
      JOIN IstBefreundetMit b
        ON m.PersonalNr = b.PersonalNr1 OR m.PersonalNr = b.PersonalNr2
      JOIN Mitarbeiter v ON (b.PersonalNr1 = v.PersonalNr OR b.PersonalNr2 = v.PersonalNr)
      JOIN Mitarbeiter p ON v.VorgesetzterPersonalNr = p.PersonalNr
```

WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'

Die erneute Ausführung dieses Tests führte zu eine SQL-Abfrage, bei der dieser Fehler nicht auftritt. Stattdessen wird jetzt aber wieder angenommen (vgl. Nr. 24), dass alle Freundschaftsbeziehungen doppelt in der Tabelle IstBefreundetMit eingetragen sind. Deswegen ist diese Lösung lediglich als noch korrekt zu betrachten.

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM Mitarbeiter m JOIN Abteilung a ON m.PersonalNr = a.AbleitungsleiterPersonalNr
      JOIN IstBefreundetMit ibm ON m.PersonalNr = ibm.PersonalNr2
      JOIN Mitarbeiter vorgesetzter ON ibm.PersonalNr1 = vorgesetzter.PersonalNr
      JOIN Mitarbeiter peter ON vorgesetzter.PersonalNr = peter.VorgesetzterPersonalNr
WHERE peter.Vorname = 'Peter' AND peter.Nachname = 'Müller'
```

Eine dritte Ausführung dieses Tests führte zu einer Lösung, mit derselben Unzulänglichkeit, wie beim zweiten Versuch.

### **33. Die Abteilungsleiter, die mit dem Abteilungsleiter von Peter Müller befreundet sind**

Beim ersten Test wurde eine falsche Lösung generiert. Bei dieser ermittelt die 1. Sub-Query die PersonalNr vom Abteilungsleiter von Peter Müller und gibt diese zurück. Das zugehörige IN sorgt dafür, dass die AbteilungsleiterPersonalNr der Abteilung a diesen Wert hat. Da es also derselbe Abteilungsleiter sein muss, ist a die Abteilung, der Peter Müller zugeordnet ist. Durch den 1. Join werden die Abteilungskollegen von Peter Müller ausgegeben. Die eigentlich notwendige Beziehung IstBefreundetMit wird nicht sinnvoll verwendet. Dies ist daran zu erkennen, dass nur ein einziger Fremdschlüssel dieser N:M-Beziehung verwendet wird (in der Join-Bedingung mit dem OR, d.h. entweder b.PersonalNr1 oder b.PersonalNr2). Der jeweils andere Fremdschlüssel kommt in gar keiner Bedingung vor, so dass die daraus resultierenden Freunde überhaupt nicht ermittelt werden.

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM Mitarbeiter m JOIN Abteilung a ON m.PersonalNr = a.AbleitungsleiterPersonalNr
      JOIN IstBefreundetMit b ON a.AbleitungsleiterPersonalNr = b.PersonalNr1
      OR a.AbleitungsleiterPersonalNr = b.PersonalNr2
WHERE a.AbleitungsleiterPersonalNr IN (
      SELECT a2.AbleitungsleiterPersonalNr
      FROM Mitarbeiter m2 JOIN Abteilung a2
      ON m2.ArbeitetInAbteilungsNr = a2.AbleitungsNr
      WHERE m2.Vorname = 'Peter' AND m2.Nachname = 'Müller' )
AND m.PersonalNr <> (
      SELECT a2.AbleitungsleiterPersonalNr
      FROM Mitarbeiter m2 JOIN Abteilung a2
      ON m2.ArbeitetInAbteilungsNr = a2.AbleitungsNr
      WHERE m2.Vorname = 'Peter' AND m2.Nachname = 'Müller' )
```

Eine wiederholte Ausführung dieses Tests führte zu einer korrekten Lösung. Hier wird sogar explizit berücksichtigt, dass Freundschaftsbeziehungen evtl. nicht doppelt in der Tabelle IstBefreundetMit eingetragen sind. Dies erfolgt durch das OR und die ersten beiden Sub-Queries. Außerdem wird durch das != (≠) und die 3. Sub-Query der Abteilungsleiter von Peter Müller aus dem Ergebnis ausgeschlossen.



Allerdings ist auch diese Lösung (durch die vielen Sub-Queries und den mehrfachen Vergleich mit dem Namen Peter Müller) verwirrender und komplexer als die in Anhang B dargestellte SQL-Abfrage.

```
SELECT DISTINCT m.PersonalNr, m.Vorname, m.Nachname
FROM Mitarbeiter m JOIN Abteilung a ON m.PersonalNr = a.AbleitungsleiterPersonalNr
      JOIN IstBefreundetMit b ON a.AbleitungsleiterPersonalNr = b.PersonalNr1
      OR a.AbleitungsleiterPersonalNr = b.PersonalNr2
WHERE (
  b.PersonalNr1 = (
    SELECT a2.AbleitungsleiterPersonalNr
    FROM Mitarbeiter m2 JOIN Abteilung a2
      ON m2.ArbeitetInAbteilungsNr = a2.AbleitungsNr
    WHERE m2.Vorname = 'Peter' AND m2.Nachname = 'Müller' )
  OR b.PersonalNr2 = (
    SELECT a2.AbleitungsleiterPersonalNr
    FROM Mitarbeiter m2 JOIN Abteilung a2
      ON m2.ArbeitetInAbteilungsNr = a2.AbleitungsNr
    WHERE m2.Vorname = 'Peter' AND m2.Nachname = 'Müller' )
  )
AND m.PersonalNr != (
  SELECT a2.AbleitungsleiterPersonalNr
  FROM Mitarbeiter m2 JOIN Abteilung a2
    ON m2.ArbeitetInAbteilungsNr = a2.AbleitungsNr
  WHERE m2.Vorname = 'Peter' AND m2.Nachname = 'Müller' )
```

## ***Sonstige Auffälligkeiten***

Im Folgenden sind einige Auffälligkeiten der generierten Ergebnisse dargestellt, die unabhängig von der reinen Korrektheit der SQL-Abfragen sind.

### ***1. Erklärungen durch das LLM***

Alle getesteten LLMs haben, außer der SQL-Abfrage selbst, jeweils einen Text generiert, der deren Funktionsweise erklärt. Diese Erklärungen können für den Benutzer hilfreich sein, um die Lösung zu besser verstehen. Im Idealfall hilft dies Studierenden sogar, zukünftig für ähnliche Problemstellungen selbst eine entsprechende SQL-Abfrage zu erstellen. Es ist also dadurch ein gewisser „Lerneffekt“ möglich. Nachfolgend ist exemplarisch die von ChatGPT erstellte Erklärung für die Aufgabenstellung 27 dargestellt:

Um die Abteilungsleiter von Freunden des Mitarbeiters Peter Müller zu ermitteln, müssen wir folgende Schritte in der SQL-Abfrage durchführen:

1. **Mitarbeiter suchen:** Zuerst müssen wir den PersonalNr des Mitarbeiters „Peter Müller“ herausfinden.
2. **Freunde von Peter Müller finden:** Wir ermitteln alle Freunde, die mit „Peter Müller“ über die Tabelle IstBefreundetMit verknüpft sind.



3. **Abteilungsleiter der Freunde finden:** Für jeden Freund suchen wir den Abteilungsleiter der jeweiligen Abteilung, in der der Freund arbeitet.
4. **Abteilungsleiter-Informationen anzeigen:** Zuletzt holen wir die Daten der Abteilungsleiter, also deren PersonalNr, Vorname, und Nachname.

Die SQL-Abfrage sieht wie folgt aus:

```
SELECT DISTINCT A.AbteilungsleiterPersonalNr AS PersonalNr, M.Vorname, M.Nachname
FROM AbteilungsDB.Mitarbeiter P JOIN AbteilungsDB.IstBefreundetMit IB
    ON P.PersonalNr = IB.PersonalNr1 OR P.PersonalNr = IB.PersonalNr2
JOIN AbteilungsDB.Mitarbeiter F
    ON (IB.PersonalNr1 = F.PersonalNr AND P.PersonalNr != IB.PersonalNr1)
    OR (IB.PersonalNr2 = F.PersonalNr AND P.PersonalNr != IB.PersonalNr2)
JOIN AbteilungsDB.Abteilung A ON F.ArbeitetInAbteilungsNr = A.AbteilungsNr
JOIN AbteilungsDB.Mitarbeiter M ON A.AbteilungsleiterPersonalNr = M.PersonalNr
WHERE P.Vorname = 'Peter' AND P.Nachname = 'Müller';
```

#### Erklärung:

- Zuerst suchen wir nach dem Mitarbeiter „Peter Müller“ in der Tabelle Mitarbeiter (abgefragt über P).
- Wir verbinden die Tabelle IstBefreundetMit, um alle Freunde von Peter Müller zu finden (über die Verknüpfung der PersonalNr von „Peter Müller“ mit den PersonalNr1 oder PersonalNr2).
- Anschließend verknüpfen wir die Tabelle Abteilung, um für jeden Freund die zugehörige Abteilung zu finden.
- Schließlich holen wir die Daten des Abteilungsleiters (über die AbteilungsleiterPersonalNr aus der Tabelle Abteilung), indem wir auf die Tabelle Mitarbeiter zugreifen.

Die Ergebnismenge enthält dann die PersonalNr, den Vorname und den Nachname der Abteilungsleiter, die für die Freunde von Peter Müller zuständig sind.

Allerdings waren auch die falschen Lösungen mit einer teilweise „plausibel klingenden“ Erklärung versehen. So erstellte z.B. Microsoft Copilot für die Aufgabenstellung 19 den nachfolgend dargestellten Erklärungstext. Die SQL-Abfrage ist – wie bereits im vorherigen Abschnitt erläutert – jedoch falsch, und der Erklärungstext führt möglicherweise dazu, dass ein Benutzer sie für plausibel und deshalb die SQL-Abfrage für korrekt hält. Dies gilt umso mehr, als dass der Erklärungstext keinen deutlich erkennbaren „roten Faden“ besitzt, so dass aufgrund von Begründungen kein „Denkfehler“ (z.B. verletzte Kausalität) erkennbar wäre.

Um die Kunden zu finden, die keinen Artikel mit einem Gewicht von 1715 kg bestellt haben, kannst du eine SQL-Abfrage verwenden, die eine Kombination aus LEFT JOIN und WHERE-Klauseln nutzt, um die gewünschten Ergebnisse zu filtern. Hier ist ein Beispiel für eine solche Abfrage:

```
SELECT k.KundenNr, k.Name, k.Umsatz
FROM Kunde k LEFT JOIN Bestellung b ON k.KundenNr = b.KundenNr
    LEFT JOIN BestellPos bp ON b.BestellNr = bp.BestellNr
    LEFT JOIN Artikel a ON bp.ArtikelNr = a.ArtikelNr AND a.Gewicht = 1715
WHERE a.ArtikelNr IS NULL;
```

Diese Abfrage funktioniert wie folgt:

1. LEFT JOIN verbindet die Tabellen Kunde, Bestellung, BestellPos und Artikel.
  2. Die Bedingung AND a.Gewicht = 1715 stellt sicher, dass nur Artikel mit einem Gewicht von 1715 kg in die Verbindung einbezogen werden.
  3. Die WHERE-Klausel a.ArtikelNr IS NULL filtert die Ergebnisse, um nur die Kunden anzuzeigen, die keinen Artikel mit einem Gewicht von 1715 kg bestellt haben.
- Falls du noch weitere Fragen hast oder Hilfe benötigst, lass es mich wissen! 😊

## 2. Verwirrende Lösungen

Die LLMs generierten teilweise sehr verwirrende Lösungen. Wie bereits im vorherigen Abschnitt erläutert, wurden z.B. sehr viele Sub-Queries verwendet, teilweise sogar bei Aufgabenstellungen, die problemlos ganz ohne Sub-Query lösbar sind. Dies war unter anderem der Fall bei ChatGPT bei der Aufgabenstellung 29 und bei DeepSeek bei den Aufgabenstellungen 33 und 27. Bei der letzteren wurden sogar ineinander verschachtelte Sub-Queries verwendet. In diesen war zudem auch noch der Vergleich mit dem vorgegebenen Namen Peter Müller doppelt realisiert (d.h. in zwei dieser Sub-Queries).

Selbstverständlich können auch solch komplexe Lösungen von einem Datenbanksystem ausgeführt werden. Problematisch ist im hier betrachteten Kontext jedoch, wenn Studierende versuchen, die generierte SQL-Abfrage als Muster für diese Art von Aufgabenstellung zu erlernen. Aufgrund der unnötig komplexen SQL-Abfrage wird es für sie schwierig sein, im Falle einer ähnlichen Aufgabenstellung, eine korrekte Lösung auf dieser Basis zu erstellen.

## 3. Sonstiges

Häufig wurde in den generierten SQL-Abfragen eine „Langform“ für Namen von Tabellen verwendet, z.B. KundenDB.Adresse. Dieses Format (also Datenbankname.Tabellenname) ist in MySQL erlaubt und die Vorgabe der von MySQL-Workbench generierten CREATE-TABLE-Befehle im Prompt kann eine Ursache hierfür sein. Dieses Tabellennamen-Format funktioniert jedoch bei anderen Datenbanksystemen nicht (z.B. Microsoft Access). Außerdem ist es auffällig, weil es so in (normalen) Datenbank-Vorlesung nicht gelehrt wird und zudem in Lehrbüchern zu SQL so nicht verwendet wird. Dies kann also ein Indiz dafür darstellen, dass eine SQL-Abfrage (z.B. in einer Prüfung) nicht vom Studierenden erstellt, sondern mittels einer KI generiert wurde.

Ähnlich ist es mit der Tatsache, dass häufig Lösungen mit dem Schlüsselwort JOIN anstatt INNER JOIN generiert wurden. Auch das wird in Vorlesungen oft nicht so gelehrt und kann bei manchen Datenbanksystemen (z.B. Microsoft Access) so nicht verwendet werden.

Schließlich haben alle LLMs für die Aufgabenstellung 11 bei der Originalformulierung (der Kunde mit dem größten Umsatz) eine Lösung mit „limit 1“ generiert. Dies ist keine Operation der Relationenalgebra, wird so üblicherweise nicht gelehrt und wird ebenfalls von manchen Datenbanksystemen nicht unterstützt.

## Fazit

Bei den durchgeführten Tests hing die Qualität der generierten SQL-Abfragen von dem jeweils verwendeten LLM ab (wie in Abb. 3 an den Hintergrundfarben leicht erkennbar ist). Jedoch ist die Anzahl der getesteten Aufgabenstellung und auch die Anzahl der jeweiligen Wiederholungen der Tests zu gering, um die Qualität der LLMs verallgemeinerbar vergleichen zu können. Dies war auch nicht das Ziel der Experimente. Es konnte jedoch gezeigt werden, dass

alle untersuchten LLMs nicht absolut fehlerfrei fähig sind, eine SQL-Abfrage für eine gegebene Aufgabenstellung zu generieren. Es entstanden sogar falsche Lösungen bei der sehr einfach strukturierten Kunden-Datenbank und auch bei recht einfachen Aufgabenstellungen (aus einer Einführungsvorlesung zum Thema Datenbanken).

Bei dem komplexeren Schema der Firmen-Datenbank wurden deutlich mehr falsche Abfragen generiert. Fehler waren unter anderem eine falsche Verwendung der rekursiven Beziehungen (z.B. IstBefreundetMit) oder die Benutzung der falschen Beziehung, wenn mehrere zwischen denselben Tabellen existieren. Dass in diesen Fällen mehr Fehler auftreten ist auch nachvollziehbar, weil mehrere Fremdschlüssel-Attribute existieren, so dass ein falsches verwendet werden kann. Zudem waren die resultierenden Join-Bedingungen bei manchen Aufgabenstellungen komplexer. Entsprechende Szenarien entsprechen (wie bei der Literaturanalyse dargestellt) aber eher dem, was in einem realen betrieblichen Umfeld zu erwarten ist (dann evtl. aber mit noch deutlich mehr Tabellen und Spalten).

Zusammenfassend lässt sich feststellen, dass die Generierung von SQL-Abfragen keineswegs fehlerfrei erfolgt ist. Deshalb sind deren Kontrolle und ggf. Überarbeitung durch den Benutzer erforderlich. Andererseits wurden (insb. durch DeepSeek) so gute Abfragen generiert, dass deren Verwendung in einer Prüfung zu einem besseren Ergebnis führen kann, als deren selbstständige Erstellung (zumindest bei Studierenden mit schlechten SQL-Kenntnissen).

## Schlussfolgerungen

Im Folgenden werden einige Konsequenzen abgeleitet, sowohl für die Generierung von SQL-Abfragen im beruflichen Umfeld als auch für die Lehre.

### ***SQL-Generierung in der betrieblichen Praxis***

Folgende Vorgehensweisen für die Erstellung von SQL-Abfragen sind theoretisch denkbar:

1. Generierung von SQL-Anweisungen ohne deren Kontrolle:

Die durchgeführten Tests basierten allesamt auf Datenbank-Schemata und Aufgabenstellungen, die, verglichen mit einem beruflichen Umfeld, recht einfach waren. Dennoch wurden völlig falsche oder unzulängliche SQL-Abfragen generiert. Auch die Literaturanalyse bestätigt diesen Sachverhalt. Daraus folgt, dass die Verwendung von automatisch generierten Ergebnissen ohne weitere Kontrolle nicht sinnvoll ist.

2. Generierung von SQL-Anweisungen mit anschließender Kontrolle:

Eine akzeptable Möglichkeit ist, sich für eine gegebene Problemstellung die zugehörige SQL-Abfrage generieren zu lassen und dann zu überprüfen. Hierfür sind ähnliche SQL-Kenntnisse erforderlich, wie für die selbstständige Erstellung der Abfrage. Evtl. lässt sich so jedoch Arbeitszeit einsparen. Allerdings birgt diese Vorgehensweise die Gefahr, dass Fehler im generierten Programm-Code übersehen werden, weil der Benutzer nicht zuerst selbst über die Problemstellung nachgedacht hat. Dann kann eine falsche Lösung durchaus plausibel wirken. Die (im vorherigen Abschnitt bereits diskutierten) Erklärungstexte des LLMs können zusätzlich dazu führen, dass man die erläuterte Funktionsweise einer falschen Lösung plausibel findet. Durch die teilweise sehr verwirrenden SQL-Abfragen, welche die LLMs generieren, wird es für den Benutzer zusätzlich erschwert, Fehler als solche zu erkennen.

### 3. Manuelle Erstellung und anschließende SQL-Generierung zur Überprüfung:

Vermutlich ist es die sicherste Vorgehensweise, eine Lösung zuerst manuell zu erstellen. Danach kann eine generierte SQL-Abfrage genutzt werden, um so nach Fehlern in der eigenen Lösung zu suchen. Beispielsweise kann die in einer generierten SQL-Abfrage enthaltene Bedingung „AND p.PersonalNr <> AbtLeiter.PersonalNr“ (vgl. Aufgabenstellung 24 im Anhang B) dazu führen, dass der Benutzer erkennt, dass der Abteilungsleiter (AbtLeiter) ansonsten mit der Person (p) identisch sein kann. Ähnlich kann hier wegen der generierten Oder-Verknüpfung „OR AbtLeiter.PersonalNr = IstBefreundetMit.PersonalNr2“ erkannt werden, dass Freundschaften nicht doppelt in der Tabelle IstBefreundetMit gespeichert sind (oder eben doch, weshalb diese Bedingung weggelassen werden kann, um so eine bessere Ausführungsgeschwindigkeit zu erzielen). Idealerweise wird die selbst erstellte SQL-Abfrage mit mehreren generierten Lösungen verglichen, wobei hierfür unterschiedliche LLMs verwendet werden sollten.

Die akzeptablen Vorgehensweisen 2 und 3 haben gemeinsam, dass hierfür gute SQL-Kenntnisse erforderlich sind.

## **Auswirkungen auf die Lehre**

Im Folgenden werden einige Aspekte dargestellt, welche die zukünftige Gestaltung von Lehrveranstaltungen und Prüfungen betreffen.

### 1. SQL weiterhin Thema in der Lehre:

Eine Erkenntnis dieser Arbeit ist, dass Mitarbeiter weiterhin SQL-Kompetenzen benötigen. Der Grund dafür ist, dass von einer KI generierte Lösungen zwar häufig korrekt sind, man sich aber nicht darauf verlassen kann. Obwohl insb. DeepSeek häufig korrekte SQL-Abfragen erstellt hat, war dies nicht bei jeder Wiederholung desselben Experiments der Fall, d.h. teilweise erst bei der 2. Durchführung des Tests. Da die Tests unabhängig voneinander durchgeführt wurden, kann natürlich ebenso die erste Lösung korrekt sein und die nachfolgend durchgeführten Tests zu einem falschen Ergebnis führen. Zudem entstand bei einer geringfügig veränderten Aufgabenstellung beim selben LLM manchmal eine fehlerhafte (z.B. DeepSeek Aufgabe 24) und manchmal eine korrekte Lösung (Aufgabe 25 und 26). Der Benutzer muss die generierten SQL-Abfragen also verstehen und überprüfen können. Hierfür sind gute SQL-Kenntnisse erforderlich, weil die Fehler teilweise schwer zu finden sind (wie u.a. bei Microsoft Copilot für Aufgabe 30 diskutiert).

Dasselbe gilt für die Vorgehensweise, das Prompt um Erklärungen und Beispiele anzureichern. Die generierten Ergebnisse müssen dennoch überprüft werden, weil diese Vorgehensweise zwar manchmal zu einer fehlerfreien Lösung führt (vgl. Copilot Aufgabe 16b. bis d.), ein zusätzliches Beispiel aber auch in einer falschen Lösung resultieren kann (vgl. Copilot Aufgabe 18). Auch sehr ausführliche Erklärungen und Beispiele führen also nicht zuverlässig zu einer korrekten SQL-Abfrage. Deshalb ist es weiterhin erforderlich, dass das Erstellen von SQL-Abfragen im Rahmen von Datenbank-Vorlesungen gelehrt wird. Evtl. sollte hier jedoch der Aspekt des Verstehens und Überprüfens von SQL-Abfragen gestärkt werden.

### 2. Unterstützung des Lernens durch LLMs:

Alle getesteten LLMs haben zusätzlich zur geforderten SQL-Abfrage auch Erklärungstexte generiert. Diese können helfen, den generierten SQL-Code zu verstehen und damit das

Erlernen von SQL unterstützen. Allerdings kann ein Erklärungstext bei einer falsch generierten SQL-Abfrage auch nachteilig sein, da dieser dann meistens ebenfalls inhaltlich falsch sein wird. Dies ist für Studierende evtl. schwer erkennbar, so dass sie annehmen, die Lösung und die Begründung seien korrekt. Es kann sich dann nachteilig auf den Lernerfolg auswirken, wenn falsche Lösungen als Muster für diese Art von Problemstellung verinnerlicht werden.

Ähnlich negativ für den Lerneffekt ist, dass die LLMs teilweise unnötig komplexe SQL-Abfragen generieren. Werden solche Lösungen zum Lernen verwendet, dann versuchen Studierende sich diese als Muster für eine bestimmte Art von Aufgabenstellung einzuprägen. Bei sehr komplexen SQL-Abfragen ist das jedoch schwierig und später wird es ein Problem darstellen, eine Lösung für eine ähnliche Aufgabenstellung korrekt zu reproduzieren. Es ist einfacher, kürzere und gut strukturierte Lösungen (vgl. Anhang B) zu verstehen, wie sie z.B. in einem Lehrbuch vermittelt werden.

### 3. Auswirkungen auf Prüfungen:

Die generierten Ergebnisse (insb. bei DeepSeek) sind recht gut. Deshalb kann es für Studierende mit beschränkten SQL-Kenntnissen vorteilhaft sein (d.h. es führt zu einer besseren Note), sich SQL-Abfragen von einem LLM generieren zu lassen, anstatt selbst eine Lösung zu erstellen. Dies gilt sogar, wenn sie die von der KI erstellten SQL-Abfragen überhaupt nicht verstehen und deshalb auch nicht überprüfen oder verbessern können.

Daraus folgt, dass eine außerhalb eines überwachten Prüfungsraums abgelegte Leistung (z.B. eine Projektarbeit, Ausarbeitung, o.ä.) keinen größeren Einfluss auf die resultierende Zeugnisnote haben sollte.

## Zusammenfassung, Limitationen und Ausblick

In dieser Arbeit wurde wissenschaftliche Literatur zum Thema SQL-Generierung mittels eines LLMs analysiert. Außerdem wurden mit drei aktuell populären LLMs hierzu Tests durchgeführt. Die Erkenntnis aus beidem ist, dass die Qualität von generierten SQL-Abfragen heutzutage zwar gut ist, aber diese keinesfalls zuverlässig fehlerfrei sind. Ziel war es, Auswirkungen auf die zukünftig für Hochschulabsolventen erforderlichen Fähigkeiten und die Gestaltung von Lehrveranstaltungen zum Thema Datenbanken zu ermitteln. Da die entsprechenden Schlussfolgerungen im vorherigen Abschnitt bereits ausführlich erläutert wurden, werden nachfolgend nur die wichtigsten Ergebnisse kurz zusammengefasst:

- SQL muss weiterhin ein Bestandteil der Datenbank-Lehre sein, weil generierte SQL-Abfragen zumindest überprüft und ggf. korrigiert werden müssen.
- Von LLMs generierte SQL-Abfragen können Studierenden beim Lernen helfen. Da LLMs auch falsche oder verwirrende SQL-Abfragen generieren, birgt diese Vorgehensweise aber auch Risiken.
- Weil generierte SQL-Abfragen teilweise eine gute Qualität haben, sollten entsprechende Prüfungen nur einem überwachten Prüfungsraum durchgeführt werden bzw. außerhalb dieses Raums erstellte Prüfungsleistungen allenfalls einen kleinen Einfluss auf die Gesamtnote haben.

Die mit den LLMs durchgeführten Tests lassen keinen verallgemeinerbaren oder quantitativen Vergleich von deren Fähigkeit zur SQL-Generierung zu. Hierfür wurden zu wenige Wiederholungen für eine Aufgabenstellung durchgeführt und zudem zu wenige unterschiedliche

Aufgabenstellungen getestet. Außerdem ist es möglich, dass bei einem anderen Szenario oder aus einer anderen Formulierung der Aufgabenstellungen ein abweichendes Ergebnis resultiert. Ziel war lediglich, einen ersten Eindruck bzgl. der Leistungsfähigkeit verschiedener LLMs zu gewinnen. Die Schlussfolgerungen für die Gestaltung zukünftiger Lehrveranstaltungen sind nicht formal belegbar und nicht mittels einer Studie überprüft. Evtl. sind sie auch teilweise durch die eigene Lehrerfahrung des Autors geprägt und damit nicht objektiv.

Der von einem LLM generierte Text ist stets abhängig von den verwendeten Lerndaten. Da dem Autor unbekannt ist, ob die verwendeten Szenarien in diesen besonders häufig oder besonders selten vorkommen, könnten zukünftig noch Tests mit anderen Szenarien durchgeführt werden. Außerdem wäre es möglich, zukünftig ähnliche Experimente für andere Datenbank-Themen durchzuführen, wie z.B. die Generierung eines Entity-Relationship-Diagramm aus einer vorgegebenen Szenariobeschreibung, die Ableitung der daraus resultierenden Datenbanktabellen oder die Generierung eines Programms, das SQL-Anweisungen enthält (z.B. ein Java-Programm mit JDBC-Anweisungen).

## Literatur

- Bhaskar, A., Tomar, T., Sathe, A., Sarawagi, S., 2023. Benchmarking and Improving Text-to-SQL Generation under Ambiguity. arXiv:2310.13659. <https://doi.org/10.48550/arXiv.2310.13659>
- Biswal, A., Patel, L., Jha, S., Kamsetty, A., Liu, S., Gonzalez, J.E., Guestrin, C., Zaharia, M., 2024. Text2SQL is Not Enough: Unifying AI and Databases with TAG. arXiv:2408.14717. <https://doi.org/10.48550/arXiv.2408.14717>
- Campos, J., García, G., A. De Sousa, J., Corseuil, E., Izquierdo, Y., Lemos, M., Casanova, M., 2025. Text-to-SQL Experiments with Engineering Data Extracted from CAD Files, in: Proc. 27th Int. Conf. on Enterprise Information Systems, Y. Porto, pp. 343–350. <https://doi.org/10.5220/0013436800003929>
- Carr, N., Shawon, F.R., Jamil, H.M., 2023. An Experiment on Leveraging ChatGPT for Online Teaching and Assessment of Database Students, in: Proc. IEEE Conf. on Teaching, Assessment and Learning for Engineering. pp. 1–8. <https://doi.org/10.1109/TALE56641.2023.10398239>
- Dixit, R., Gajjam, N., 2024. AI2SQL.io: Empowering SQL Learning and SocialChange Through IT-Enabled Smart Tutoring in Computer Science and Engineering, in: Proc. Int. Conf. on Artificial Intelligence and Quantum Computation-Based Sensor Application. pp. 1–5. <https://doi.org/10.1109/ICAQSA64000.2024.10882332>
- Dong, X., Zhang, C., Ge, Y., Mao, Y., Gao, Y., Chen, Lu, Lin, J., Lou, D., 2023. C3: Zero-shot Text-to-SQL with ChatGPT. arXiv:2307.07306. <https://doi.org/10.48550/arXiv.2307.07306>
- Farinetti, L., Cagliero, L., 2025. A Critical Approach to ChatGPT: An Experience in SQL Learning, in: Proc. 56th ACM Technical Symposium on Computer Science Education. ACM, Pittsburgh, pp. 318–324. <https://doi.org/10.1145/3641554.3701932>
- Floratou, A., Psallidas, F., Zhao, F., Deep, S., Hagleither, G., Cahoon, J., Alotaibi, R., Henkel, J., Singla, A., van Grootel, A., Deng, K., Lin, K., Campos, M., Emani, V., Pandit, V., Wang, W., Curino, C., 2024. NL2SQL is a Solved Problem... Not!, in: Proc. Conf. on Innovative Data Systems Research.
- Gaitantzi, A., Kazanidis, I., 2025. The Role of Artificial Intelligence in Computer Science Education: A Systematic Review with a Focus on Database Instruction. Applied Sciences 15, 3960. <https://doi.org/10.3390/app15073960>
- Ganesan, S., Gong, T., Lee, J., 2024. SQLearn: Automated SQL Statement Assessment using Structure-based Analysis, in: Proc. 55th ACM Technical Symposium on Computer Science Education. ACM, Portland, pp. 1644–1645. <https://doi.org/10.1145/3626253.3635607>

- Ganti, M., Orr, L., Wu, S., 2024. Evaluating Text-to-SQL Model Failures on Real-World Data, in: Proc. IEEE 40th Int. Conf. on Data Engineering. <https://doi.org/10.1109/ICDE60146.2024.00456>
- Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., Zhou, J., 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. arXiv:2308.15363. <https://doi.org/10.48550/arXiv.2308.15363>
- Hong, S.H., Storey, V.C., 2023. Teaching SQL Using ChatGPT, in: Proc. 29th Americas Conf. on Information Systems. Panama.
- Kim, H., So, B.-H., Han, W.-S., Lee, H., 2020. Natural Language to SQL: Where are we Today? Proc. VLDB Endowment 13, 1737–1750. <https://doi.org/10.14778/3401960.3401970>
- Lan, W., Wang, Z., Chauhan, A., Zhu, H., Li, A., Guo, J., Zhang, S., Hang, C.-W., Lilien, J., Hu, Y., Pan, L., Dong, M., Wang, J., Jiang, J., Ash, S., Castelli, V., Ng, P., Xiang, B., 2023. UNITE: A Unified Benchmark for Text-to-SQL Evaluation. arXiv:2305.16265. <https://doi.org/10.48550/arXiv.2305.16265>
- Lee, G., Chay, W., Cho, S., Choi, E., 2024. TrustSQL: Benchmarking Text-to-SQL Reliability with Penalty-Based Scoring. arXiv:2403.15879. <https://doi.org/10.48550/arXiv.2403.15879>
- Li, J., Hui, B., Qu, G., Yang, J., Li, Binhua, Li, Bowen, Wang, B., Qin, B., Geng, R., Huo, N., Zhou, X., Chenhao, M., Li, G., Chang, K., Huang, F., Cheng, R., Li, Y., 2023. Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. Advances in Neural Information Processing Systems 36, 42330–42357.
- Liu, A., Hu, X., Wen, L., Yu, P.S., 2023. A Comprehensive Evaluation of ChatGPT's Zero-shot Text-to-SQL Capability. arXiv:2303.13547. <https://doi.org/10.48550/arXiv.2303.13547>
- Liu, X., Shen, S., Li, B., Ma, P., Jiang, R., Zhang, Y., Fan, J., Li, G., Tang, N., Luo, Y., 2025. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? <https://doi.org/10.48550/arXiv.2408.05109>
- Ma, L., Pu, K., Zhu, Y., 2024. Evaluating LLMs for Text-to-SQL Generation With Complex SQL Workload. arXiv:2407.19517. <https://doi.org/10.48550/arXiv.2407.19517>
- Matek, T., Zrnc, A., Lavbič, D., 2017. Learning SQL with Artificial Intelligent Aided Approach. Int. Journal of Information and Education Technology 7, 803–808. <https://doi.org/10.18178/ijiet.2017.7.11.976>
- Mohammadjafari, A., Maida, A.S., Gottumukkala, R., 2025. From Natural Language to SQL: Review of LLM-based Text-to-SQL Systems. arXiv:2410.01066. <https://doi.org/10.48550/arXiv.2410.01066>
- Nascimento, E., Avila, C., Izquierdo, Y., García, G., Andrade, L., Facina, M., Lemos, M., Casanova, M., 2025. On the Text-to-SQL Task Supported by Database Keyword Search, in: Proc. 27th Int. Conf. on Enterprise Information Systems. Porto, pp. 173–180. <https://doi.org/10.5220/0013126300003929>
- Poess, M., Floyd, C., 2000. New TPC Benchmarks for Decision Support and Web Commerce. ACM SIGMOD Record 29, 64–71. <https://doi.org/10.1145/369275.369291>
- Pornphol, P., Chittayasothorn, S., 2023. Verification of Relational Database Languages Codes Generated by ChatGPT, in: Proc. 4th Asia Service Sciences and Software Engineering Conference. ACM, Aizu-Wakamatsu City, pp. 17–22. <https://doi.org/10.1145/3634814.3634817>
- Prakash, K., Rao, S., Hamza, R., Lukich, J., Chaudhari, V., Nandi, A., 2024. Integrating LLMs into Database Systems Education, in: Proc. 3rd Int. Workshop on Data Systems Education: Bridging Education Practice with Education Research. ACM, Santiago, pp. 33–39. <https://doi.org/10.1145/3663649.3664371>
- Qin, B., Hui, B., Wang, L., Yang, M., Li, J., Li, B., Geng, R., Cao, R., Sun, J., Si, L., Huang, F., Li, Y., 2022. A Survey on Text-to-SQL Parsing: Concepts, Methods, and Future Directions. arXiv:2208.13629. <https://doi.org/10.48550/arXiv.2208.13629>
- Shi, L., Tang, Z., Zhang, N., Zhang, X., Yang, Z., 2024. A Survey on Employing Large Language Models for Text-to-SQL Tasks. arXiv:2407.15186. <https://doi.org/10.48550/arXiv.2407.15186>

- Steinberger, C., Wedam, A., 2025. Facilitate Flipped SQL-Classrooms with Text-Generating AI, in: Auer, M.E., Rüttemann, T. (Eds.), Futureproofing Engineering Education for Global Responsibility. Springer Nature Switzerland, pp. 451–461. [https://doi.org/10.1007/978-3-031-85652-5\\_45](https://doi.org/10.1007/978-3-031-85652-5_45)
- Sun, S., Zhang, Y., Yan, J., Gao, Y., Ong, D., Chen, B., Su, J., 2023. Battle of the Large Language Models: Dolly vs LLaMA vs Vicuna vs Guanaco vs Bard vs ChatGPT - A Text-to-SQL Parsing Comparison. arXiv:2310.10190. <https://doi.org/10.48550/arXiv.2310.10190>
- Taipalus, T., Seppänen, V., 2020. SQL Education: A Systematic Mapping Study and Future Research Agenda. ACM Transactions on Computing Education 20, 1–33. <https://doi.org/10.1145/3398377>
- Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D., 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887. <https://doi.org/10.48550/arXiv.1809.08887>
- Zhang, B., Ye, Y., Du, G., Hu, X., Li, Zhishuai, Yang, S., Liu, C.H., Zhao, R., Li, Ziyue, Mao, H., 2024. Benchmarking the Text-to-SQL Capability of Large Language Models: A Comprehensive Evaluation. arXiv:2403.02951. <https://doi.org/10.48550/arXiv.2403.02951>
- Zhang, X., 2025. Teaching Tip Incorporating AI Tools Into Database Classes. Journal of Information Systems Education 36, 37–52. <https://doi.org/10.62273/GKZI2477>



## **Anhang A: Übersicht über die Aufgabenstellungen**

### **Auf Basis der Kunden-Datenbank (vgl. Abb. 1):**

1.  
Anzeige aller Zahlungsarten, bei denen ein „a“ in der Bezeichnung vorkommt.  
Auszugebende Ergebnisspalten: ZahlArtNr, Bezeichnung
2.  
Alle Bestellungen und die zugehörige Rechnungsnummer, sortiert nach BestellNr.  
Auszugebende Ergebnisspalten: BestellNr, Bestell-Datum, RechnungsNr
3.  
Alle Bestellungen und die zugehörige Rechnungsnummer, sortiert nach BestellNr, inkl. der Bestellungen, für die (noch) keine Rechnung existiert.  
Auszugebende Ergebnisspalten: BestellNr, Bestell-Datum, RechnungsNr
4.  
Alle Bestellungen sortiert nach BestellNr, aber nur Bestellungen, für die (noch) keine Rechnung existiert.  
Auszugebende Ergebnisspalten: BestellNr, Bestell-Datum
5.  
Anzahl der Bestellungen je Artikel.  
Auszugebende Ergebnisspalten: ArtikelNr, Artikelname, Anzahl Bestellungen
6.  
Preis (aus Tabelle Bestellposition) des teuersten Artikels, den AVIS bestellt hat.  
Auszugebende Ergebnisspalten: Preis, das Ergebnis soll nur 1 Zeile enthalten
7.  
Alle besonderen Kunden, d.h. alle Problemkunden und alle Premiumkunden – aber nicht die „normalen“ Kunden.  
Auszugebende Ergebnisspalten: KundenNr, Name, Anzahl Zahlungsausfälle (bei Problemkunden) bzw. Kundenbewertung (bei Problemkunden)
8.  
Errechnetes Gesamtgewicht und Gesamtherstellkosten aufgeschlüsselt nach Bestelldatum für Bestellungen aus dem Jahr 2024.  
Auszugebende Ergebnisspalten: Bestelldatum, Gesamtgewicht, Gesamtherstellkosten
9.  
Alle Kundendaten, auch für Kunden bei denen einige Daten undefiniert sind.  
Auszugebende Ergebnisspalten: KundenNr, Kundenname, Ort, PLZ, Key-Account-Mgr, Anzahl Zahlungsausfälle
10.  
Übersicht über die Bestellungen mit zugehörigen Detaildaten.  
Auszugebende Ergebnisspalten: BestellNr, Kundenname, Anzahl Bestellpositionen, Preis des teuersten Artikels

11.

Der Kunde (oder die Kunden) mit dem größten Umsatz.

Auszugebende Ergebnisspalten: Ergebnisspalten: KundenNr, Name, Umsatz

12.

Alle Kunden, die mind. 2 Bestellpositionen haben.

Auszugebende Ergebnisspalten: KundenNr, Name

13.

Alle Kunden, ohne Bestellpositionen.

Auszugebende Ergebnisspalten: KundenNr, Name

14.

Alle Kunden, die mind. für 100.000€ bestellt haben, basierend auf Preis & Anzahl aus BestellPos.

Auszugebende Ergebnisspalten: KundenNr, Name

15.

Zahlungsarten, die ein Kunde mit mehr als 2 Mio € Umsatz benutzt hat, sowie zusätzlich Artikel, die in einer Bestellposition als PosNr 1 oder 2 bestellt wurden.

Auszugebende Ergebnisspalten: Nr, Bezeichnung bzw. Name

16.

Bestellungen, in denen kein Artikel mit einem Gewicht von 1715 (kg) bestellt wurde.

Auszugebende Ergebnisspalten: BestellNr, Datum, KundenNr

17.

Alle Zahlungsarten von Kunden, die den Artikel smart4two bestellt haben.

Auszugebende Ergebnisspalten: ZahlArtNr, (Zahlungsart-)Bezeichnung, (Kunden)Name

18.

Kunden, die die Zahlungsart Kreditkarte nie verwendet haben.

Auszugebende Ergebnisspalten: KundenNr, Name

19.

Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben.

Auszugebende Ergebnisspalten: KundenNr, Name, Umsatz

20.

Die Adressen derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben.

Auszugebende Ergebnisspalten: Strasse, HNr, PLZ, Ort

21.

Die Zahlungsarten derjenigen Kunden, die keinen Artikel mit einem Gewicht von 1715 (kg) bestellt haben.

Auszugebende Ergebnisspalten: ZahlungsartNr, Bezeichnung

## **Auf Basis der Firmen-Datenbank (vgl. Abb. 2):**

22.

Der Abteilungsleiter der Abteilung "Marketing".

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

23.

Alle Mitarbeiter, die in der Abteilung "Marketing" angestellt sind.

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

24.

Alle Abteilungen, die Freunde des Mitarbeiters Peter Müller leiten.

Auszugebende Ergebnisspalten: AbteilungsNr, Bezeichnung

25.

Alle Abteilungen, in denen Freunde des Mitarbeiters Peter Müller arbeiten.

Auszugebende Ergebnisspalten: AbteilungsNr, Bezeichnung

26.

Die Standorte von Abteilungen, in denen Freunde des Mitarbeiters Peter Müller arbeiten.

Auszugebende Ergebnisspalten: StandortNr, Ortsname

27.

Alle Abteilungsleiter von Freunden des Mitarbeiters Peter Müller.

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

28.

Alle Mitarbeiter, die in der selben Abteilung arbeiten wie ein Freund von Peter Müller.

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

29.

Alle Vorgesetzten von Peter Müller, auch indirekt Vorgesetzte.

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

30.

Die Abteilungen, in denen Vorgesetzte von Peter Müller arbeiten, auch indirekt Vorgesetzte.

Auszugebende Ergebnisspalten: AbteilungsNr, Bezeichnung

31.

Die Abteilungen und die Standorte, an denen Vorgesetzte von Peter Müller arbeiten, auch indirekt Vorgesetzte.

Auszugebende Ergebnisspalten: AbteilungsNr, Bezeichnung, StandortNr, Ortsname

32.

Die Abteilungsleiter, die mit einem Vorgesetzten von Peter Müller befreundet sind

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

33.

Die Abteilungsleiter, die mit dem Abteilungsleiter von Peter Müller befreundet sind

Auszugebende Ergebnisspalten: PersonalNr, Vorname, Nachname

## Anhang B: Mögliche Lösungen für die Aufgabenstellungen

**Auf Basis der Kunden-Datenbank (vgl. Abb. 1):**

1.

```
SELECT ZahlArtNr, Bezeichnung  
FROM Zahlungsart  
WHERE Bezeichnung LIKE '%a%'
```

2.

```
SELECT Bestellung.BestellNr, Bestellung.Datum, Rechnung.RechnungsNr  
FROM Bestellung INNER JOIN Rechnung  
    ON Bestellung.BestellNr = Rechnung.BestellNr  
ORDER BY Bestellung.BestellNr
```

3.

```
SELECT Bestellung.BestellNr, Bestellung.Datum, Rechnung.RechnungsNr  
FROM Bestellung LEFT OUTER JOIN Rechnung  
    ON Bestellung.BestellNr = Rechnung.BestellNr  
ORDER BY Bestellung.BestellNr
```

4.

```
SELECT Bestellung.BestellNr, Bestellung.Datum  
FROM Bestellung LEFT OUTER JOIN Rechnung  
    ON Bestellung.BestellNr = Rechnung.BestellNr  
WHERE Rechnung.RechnungsNr IS NULL  
ORDER BY Bestellung.BestellNr
```

5.

```
SELECT Artikel.ArtikelNr, Artikel.Name, COUNT(BestellNr) AS AnzahlBestellungen  
FROM BestellPos INNER JOIN Artikel  
    ON BestellPos.ArtikelNr = Artikel.ArtikelNr  
GROUP BY Artikel.ArtikelNr, Artikel.Name
```

6.

```
SELECT MAX(Bestellpos.Preis) AS PreisTeuersterArtikel  
FROM (Bestellpos INNER JOIN Bestellung ON Bestellpos.BestellNr = Bestellung.BestellNr)  
    INNER JOIN Kunde ON Bestellung.KundenNr = Kunde.KundenNr  
WHERE Name = 'AVIS'
```

7.

```
SELECT Kunde.KundenNr, Name  
FROM Premiumkunde INNER JOIN Kunde ON Premiumkunde.KundenNr = Kunde.KundenNr  
UNION  
SELECT Kunde.KundenNr, Name  
FROM Problemkunde INNER JOIN Kunde ON Problemkunde.KundenNr = Kunde.KundenNr
```

8.

```
SELECT Datum, SUM(Gewicht * Anzahl) AS Gesamtgewicht,  
       SUM(Herstellkosten*Anzahl) AS Gesamtherstellkosten  
FROM (Bestellung INNER JOIN Bestellpos ON Bestellung.BestellNr = Bestellpos.BestellNr)  
     INNER JOIN Artikel ON Artikel.ArtikelNr = Bestellpos.ArtikelNr  
WHERE Datum LIKE '%2024'  
GROUP BY Datum
```

9.

```
SELECT Kunde.KundenNr, Kunde.Name, Ort, PLZ, KeyAccountMgr, AnzZahlungsausfälle  
FROM ((Kunde LEFT OUTER JOIN Adresse ON Kunde.KundenNr = Adresse.KundenNr)  
     LEFT OUTER JOIN Premiumkunde ON Kunde.KundenNr = Premiumkunde.KundenNr)  
     LEFT OUTER JOIN Problemkunde ON Kunde.KundenNr = Problemkunde.KundenNr
```

10.

```
SELECT Bestellung.BestellNr, Kunde.Name, COUNT(Bestellpos.PosNr) AS AnzahlPositionen,  
       MAX(Preis) AS PreisTeuersterArtikel  
FROM (Bestellung INNER JOIN Kunde ON Bestellung.KundenNr = Kunde.KundenNr)  
     INNER JOIN Bestellpos ON Bestellpos.BestellNr = Bestellung.BestellNr  
GROUP BY Bestellung.BestellNr, Kunde.Name
```

11.

```
SELECT KundenNr, Name, Umsatz  
FROM Kunde  
WHERE Umsatz = (SELECT MAX(Umsatz) FROM Kunde)
```

12.

```
SELECT Kunde.KundenNr, Name  
FROM (Kunde INNER JOIN Bestellung ON Kunde.KundenNr=Bestellung.KundenNr)  
     INNER JOIN BestellPos ON Bestellung.BestellNr=BestellPos.BestellNr  
GROUP BY Kunde.KundenNr, Name  
HAVING COUNT(*) >= 2
```

13.

```
SELECT Kunde.KundenNr, Name  
FROM (Kunde LEFT OUTER JOIN Bestellung ON Kunde.KundenNr = Bestellung.KundenNr)  
     LEFT OUTER JOIN BestellPos ON Bestellung.BestellNr = BestellPos.BestellNr  
WHERE PosNr IS NULL
```

oder:

```
SELECT Kunde.KundenNr, Name  
FROM (Kunde LEFT OUTER JOIN Bestellung ON Kunde.KundenNr = Bestellung.KundenNr)  
WHERE Bestellung.BestellNr NOT IN (SELECT BestellNr FROM BestellPos)
```

oder:

```
SELECT Kunde.KundenNr, Name
FROM (Kunde LEFT OUTER JOIN Bestellung ON Kunde.KundenNr = Bestellung.KundenNr)
     LEFT OUTER JOIN BestellPos ON Bestellung.BestellNr = BestellPos.BestellNr
GROUP BY Kunde.KundenNr, Name
HAVING COUNT(PosNr) = 0
```

14.

```
SELECT Kunde.KundenNr, Name, SUM (Preis * Anzahl) AS Gesamtpreis
FROM (Kunde INNER JOIN Bestellung ON Kunde.KundenNr = Bestellung.KundenNr)
     INNER JOIN BestellPos ON Bestellung.BestellNr = BestellPos.BestellNr
GROUP BY Kunde.KundenNr, Name
HAVING SUM (Preis * Anzahl) >= 100000
```

15.

```
SELECT Zahlungsart.ZahlArtNr, Bezeichnung
FROM (Zahlungsart INNER JOIN verwendet ON Zahlungsart.ZahlArtNr = verwendet.ZahlartNr)
     INNER JOIN Kunde ON verwendet.KundenNr = Kunde.KundenNr
WHERE Umsatz > 2000000
UNION
SELECT Artikel.ArtikelNr, Name
FROM Artikel INNER JOIN BestellPos ON Artikel.ArtikelNr = BestellPos.ArtikelNr
WHERE PosNr <= 2
```

16.

```
SELECT *
FROM Bestellung
WHERE BestellNr NOT IN
    (    SELECT BestellNr
      FROM BestellPos INNER JOIN Artikel ON Artikel.ArtikelNr = BestellPos.ArtikelNr
      WHERE Gewicht = 1715 )
```

17.

```
SELECT Zahlungsart.ZahlArtNr, Zahlungsart.Bezeichnung, Kunde.Name
FROM (((Zahlungsart INNER JOIN Verwendet
      ON Zahlungsart.ZahlArtNr = Verwendet.ZahlArtNr)
  INNER JOIN Kunde ON Verwendet.KundenNr = Kunde.KundenNr)
  INNER JOIN Bestellung ON Kunde.KundenNr = Bestellung.KundenNr)
  INNER JOIN BestellPos ON Bestellung.BestellNr = BestellPos.BestellNr)
  INNER JOIN Artikel ON BestellPos.ArtikelNr = Artikel.ArtikelNr
WHERE Artikel.Name = 'C320'
```

18.

```
SELECT KundenNr, Name
FROM Kunde
WHERE KundenNr NOT IN
    (    SELECT KundenNr
        FROM verwendet INNER JOIN Zahlungsart
            ON verwendet.ZahlArtNr = Zahlungsart.ZahlArtNr
        WHERE Bezeichnung = 'Kreditkarte' )
```

19.

```
SELECT KundenNr, Name, Umsatz
FROM Kunde
WHERE KundenNr NOT IN (
    SELECT KundenNr
    FROM (Bestellung INNER JOIN BestellPos
        ON Bestellung.BestellNr = BestellPos.BestellNr)
        INNER JOIN Artikel ON BestellPos.ArtikelNr = Artikel.ArtikelNr
    WHERE Gewicht = 1715 )
```

20.

```
SELECT DISTINCT Strasse, HNr, PLZ, Ort
FROM Kunde INNER JOIN Adresse ON Kunde.KundenNr = Adresse.KundenNr
WHERE Kunde.KundenNr NOT IN (
    SELECT KundenNr
    FROM (Bestellung INNER JOIN BestellPos
        ON Bestellung.BestellNr = BestellPos.BestellNr)
        INNER JOIN Artikel ON BestellPos.ArtikelNr = Artikel.ArtikelNr
    WHERE Gewicht = 1715 )
```

21.

```
SELECT DISTINCT Zahlungsart.ZahlArtNr, Bezeichnung
FROM (Kunde INNER JOIN verwendet ON Kunde.KundenNr = verwendet.KundenNr)
    INNER JOIN Zahlungsart ON verwendet.ZahlArtNr = Zahlungsart.ZahlArtNr
WHERE Kunde.KundenNr NOT IN (
    SELECT KundenNr
    FROM (Bestellung INNER JOIN BestellPos
        ON Bestellung.BestellNr = BestellPos.BestellNr)
        INNER JOIN Artikel ON BestellPos.ArtikelNr = Artikel.ArtikelNr
    WHERE Gewicht = 1715 )
```

**Auf Basis der Firmen-Datenbank (vgl. Abb. 2):**

22.

```
SELECT Mitarbeiter.PersonalNr, Vorname, Nachname
FROM Abteilung INNER JOIN Mitarbeiter
    ON Abteilung.AbstellungsleiterPersonalNr = Mitarbeiter.PersonalNr
WHERE Bezeichnung = 'Marketing'
```

23.

```
SELECT Mitarbeiter.PersonalNr, Vorname, Nachname
FROM Abteilung INNER JOIN Mitarbeiter
      ON Abteilung.AnteilungsNr = Mitarbeiter.ArbeitetInAnteilungsNr
WHERE Bezeichnung = 'Marketing'
```

24.

```
SELECT Abteilung.AnteilungsNr, Abteilung.Bezeichnung
FROM Abteilung INNER JOIN Mitarbeiter AS AbtLeiter
      ON Abteilung.AnteilungsleiterPersonalNr = AbtLeiter.PersonalNr
      INNER JOIN IstBefreundetMit ON (AbtLeiter.PersonalNr = IstBefreundetMit.PersonalNr1
      OR AbtLeiter.PersonalNr = IstBefreundetMit.PersonalNr2)
      INNER JOIN Mitarbeiter AS p ON (IstBefreundetMit.PersonalNr1 = p.PersonalNr
      OR IstBefreundetMit.PersonalNr2 = p.PersonalNr)
      AND p.PersonalNr <> AbtLeiter.PersonalNr
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

25.

```
SELECT DISTINCT Abteilung.AnteilungsNr, Abteilung.Bezeichnung
FROM Mitarbeiter AS p INNER JOIN IstBefreundetMit
      ON p.PersonalNr = IstBefreundetMit.PersonalNr1
      OR p.PersonalNr = IstBefreundetMit.PersonalNr2
      INNER JOIN Mitarbeiter AS freund
      ON (IstBefreundetMit.PersonalNr1 = freund.PersonalNr
      OR IstBefreundetMit.PersonalNr2 = freund.PersonalNr)
      AND freund.PersonalNr <> p.PersonalNr
      INNER JOIN Abteilung ON freund.ArbeitetInAnteilungsNr = Abteilung.AnteilungsNr
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

26.

```
SELECT DISTINCT Abteilung.AnteilungsNr, Abteilung.Bezeichnung
FROM Mitarbeiter AS p INNER JOIN IstBefreundetMit
      ON p.PersonalNr = IstBefreundetMit.PersonalNr1
      OR p.PersonalNr = IstBefreundetMit.PersonalNr2
      INNER JOIN Mitarbeiter AS freund
      ON (IstBefreundetMit.PersonalNr1 = freund.PersonalNr
      OR IstBefreundetMit.PersonalNr2 = freund.PersonalNr)
      AND freund.PersonalNr <> p.PersonalNr
      INNER JOIN Abteilung ON freund.ArbeitetInAnteilungsNr = Abteilung.AnteilungsNr
      INNER JOIN Standort ON Abteilung.StandortNr = Standort.StandortNr
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```



27.

```
SELECT DISTINCT AbtLeiter.PersonalNr, AbtLeiter.Vorname, AbtLeiter.Nachname
FROM Mitarbeiter AS p INNER JOIN IstBefreundetMit
    ON p.PersonalNr = IstBefreundetMit.PersonalNr1
    OR p.PersonalNr = IstBefreundetMit.PersonalNr2
INNER JOIN Mitarbeiter AS freund
    ON (IstBefreundetMit.PersonalNr1 = freund.PersonalNr
    OR IstBefreundetMit.PersonalNr2 = freund.PersonalNr)
    AND freund.PersonalNr <> p.PersonalNr
INNER JOIN Abteilung ON freund.ArbeitetInAbteilungsNr = Abteilung.AbteilungsNr
INNER JOIN Mitarbeiter AS AbtLeiter
    ON Abteilung.AbteilungsleiterPersonalNr = AbtLeiter.PersonalNr
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

28.

```
SELECT DISTINCT MitarbInSelberAbt.PersonalNr,
    MitarbInSelberAbt.Vorname, MitarbInSelberAbt.Nachname
FROM Mitarbeiter AS p INNER JOIN IstBefreundetMit
    ON p.PersonalNr = IstBefreundetMit.PersonalNr1
    OR p.PersonalNr = IstBefreundetMit.PersonalNr2
INNER JOIN Mitarbeiter AS freund
    ON (IstBefreundetMit.PersonalNr1 = freund.PersonalNr
    OR IstBefreundetMit.PersonalNr2 = freund.PersonalNr)
    AND freund.PersonalNr <> p.PersonalNr
INNER JOIN Abteilung ON freund.ArbeitetInAbteilungsNr = Abteilung.AbteilungsNr
INNER JOIN Mitarbeiter AS MitarbInSelberAbt
    ON Abteilung.AbteilungsNr = MitarbInSelberAbt.ArbeitetInAbteilungsNr
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

29.

```
WITH RECURSIVE Vorgesetzte AS (
    SELECT Mitarbeiter.PersonalNr, Mitarbeiter.Vorname,
        Mitarbeiter.Nachname, Mitarbeiter.VorgesetzterPersonalNr
    FROM Mitarbeiter
    WHERE Mitarbeiter.Vorname = 'Peter' AND Mitarbeiter.Nachname = 'Müller'
    UNION ALL
    SELECT Mitarbeiter.PersonalNr, Mitarbeiter.Vorname, Mitarbeiter.Nachname,
        Mitarbeiter.VorgesetzterPersonalNr
    FROM Mitarbeiter INNER JOIN Vorgesetzte
        ON Mitarbeiter.PersonalNr = Vorgesetzte.VorgesetzterPersonalNr
)
SELECT PersonalNr, Vorname, Nachname
FROM Vorgesetzte
```

30.

```
WITH RECURSIVE Vorgesetzte AS (  
    SELECT Mitarbeiter.PersonalNr, Mitarbeiter.VorgesetzterPersonalNr,  
           Mitarbeiter.ArbeitetInAbteilungsNr  
    FROM Mitarbeiter  
    WHERE Mitarbeiter.Vorname = 'Peter' AND Mitarbeiter.Nachname = 'Müller'  
    UNION ALL  
    SELECT Mitarbeiter.PersonalNr, Mitarbeiter.VorgesetzterPersonalNr,  
           Mitarbeiter.ArbeitetInAbteilungsNr  
    FROM Mitarbeiter INNER JOIN Vorgesetzte  
           ON Mitarbeiter.PersonalNr = Vorgesetzte.VorgesetzterPersonalNr  
)  
SELECT DISTINCT Abteilung.AbteilungsNr, Abteilung.Bezeichnung  
FROM Abteilung INNER JOIN Vorgesetzte
```

31.

```
WITH RECURSIVE Vorgesetzte AS (  
    SELECT Mitarbeiter.PersonalNr, Mitarbeiter.VorgesetzterPersonalNr  
    FROM Mitarbeiter  
    WHERE Mitarbeiter.Vorname = 'Peter' AND Mitarbeiter.Nachname = 'Müller'  
    UNION ALL  
    SELECT Mitarbeiter.PersonalNr, Mitarbeiter.VorgesetzterPersonalNr  
    FROM Mitarbeiter INNER JOIN Vorgesetzte  
           ON Mitarbeiter.PersonalNr = Vorgesetzte.VorgesetzterPersonalNr  
)  
SELECT DISTINCT Abteilung.AbteilungsNr, Abteilung.Bezeichnung, Standort.StandortNr,  
           Standort.Ortsname  
FROM Vorgesetzte INNER JOIN Mitarbeiter  
           ON Vorgesetzte.VorgesetzterPersonalNr = Mitarbeiter.PersonalNr  
INNER JOIN Abteilung ON Mitarbeiter.ArbeitetInAbteilungsNr = Abteilung.AbteilungsNr  
INNER JOIN Standort ON Abteilung.StandortNr = Standort.StandortNr;
```

32.

```
SELECT DISTINCT AbtLeiter.PersonalNr, AbtLeiter.Vorname, AbtLeiter.Nachname  
FROM Mitarbeiter AS AbtLeiter INNER JOIN Abteilung  
           ON AbtLeiter.PersonalNr = Abteilung.AbteilungsleiterPersonalNr  
INNER JOIN IstBefreundetMit ON (AbtLeiter.PersonalNr = IstBefreundetMit.PersonalNr1  
           OR AbtLeiter.PersonalNr = IstBefreundetMit.PersonalNr2)  
INNER JOIN Mitarbeiter AS Vorgesetzter  
           ON (IstBefreundetMit.PersonalNr1 = Vorgesetzter.PersonalNr  
           OR IstBefreundetMit.PersonalNr2 = Vorgesetzter.PersonalNr)  
           AND AbtLeiter.PersonalNr <> Vorgesetzter.PersonalNr  
INNER JOIN Mitarbeiter AS p ON Vorgesetzter.PersonalNr = p.VorgesetzterPersonalNr  
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```

33.

```
SELECT DISTINCT AbtLeiter.PersonalNr, AbtLeiter.Vorname, AbtLeiter.Nachname
FROM Mitarbeiter AS p INNER JOIN Abteilung AS Abt1
    ON p.ArbeitetInAbteilungsNr = Abt1.AbteilungsNr
    INNER JOIN IstBefreundetMit
        ON (Abt1.AbteilungsleiterPersonalNr = IstBefreundetMit.PersonalNr1
            OR Abt1.AbteilungsleiterPersonalNr = IstBefreundetMit.PersonalNr2)
    INNER JOIN Abteilung AS Abt2
        ON (IstBefreundetMit.PersonalNr1 = Abt2.AbteilungsleiterPersonalNr
            OR IstBefreundetMit.PersonalNr2 = Abt2.AbteilungsleiterPersonalNr)
        AND Abt1.AbteilungsleiterPersonalNr <> Abt2.AbteilungsleiterPersonalNr
    INNER JOIN Mitarbeiter AS AbtLeiter
        ON Abt2.AbteilungsleiterPersonalNr = AbtLeiter.PersonalNr
WHERE p.Vorname = 'Peter' AND p.Nachname = 'Müller'
```