SQL

MySQL – DB5 – VELOCÉBELLO

ITA 12 – Aps, Fabian

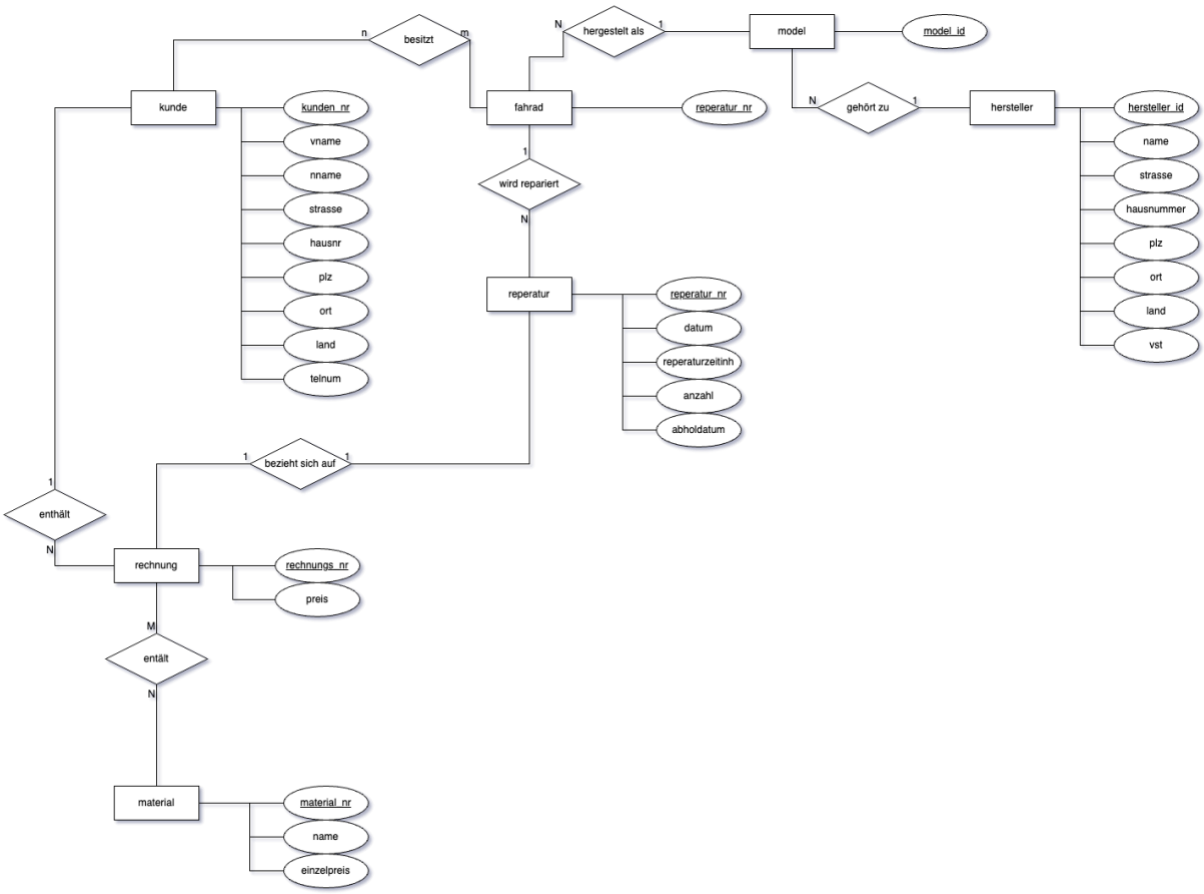
Inhalt

ER-Diagramm in Chen-Notation

Definition

Ein ER-Diagramm (Entity-Relationship-Diagramm) in Chen-Notation ist eine grafische Darstellung der Datenbankstruktur, die die Entitäten und ihre Beziehungen in einem relationalen Datenbankschema zeigt. In MySQL wird die Chen-Notation verwendet, um die Struktur der Datenbank visuell zu modellieren. Entitäten werden durch Rechtecke dargestellt, Beziehungen zwischen Entitäten werden durch Linien mit verschiedenen Notationen dargestellt, wie zum Beispiel "1:n" für eine einseitige Beziehung oder "m:n" für eine viele-zu-viele-Beziehung. Attribute werden innerhalb der Entitäten mit Ellipsen dargestellt und Primärschlüssel werden durch Unterstreichung gekennzeichnet. Diese Diagramme helfen Entwicklern und Datenbankadministratoren, das Datenmodell zu verstehen und zu planen, bevor sie es in die Datenbank implementieren.

Beispiel



Relationenmodell

Definition

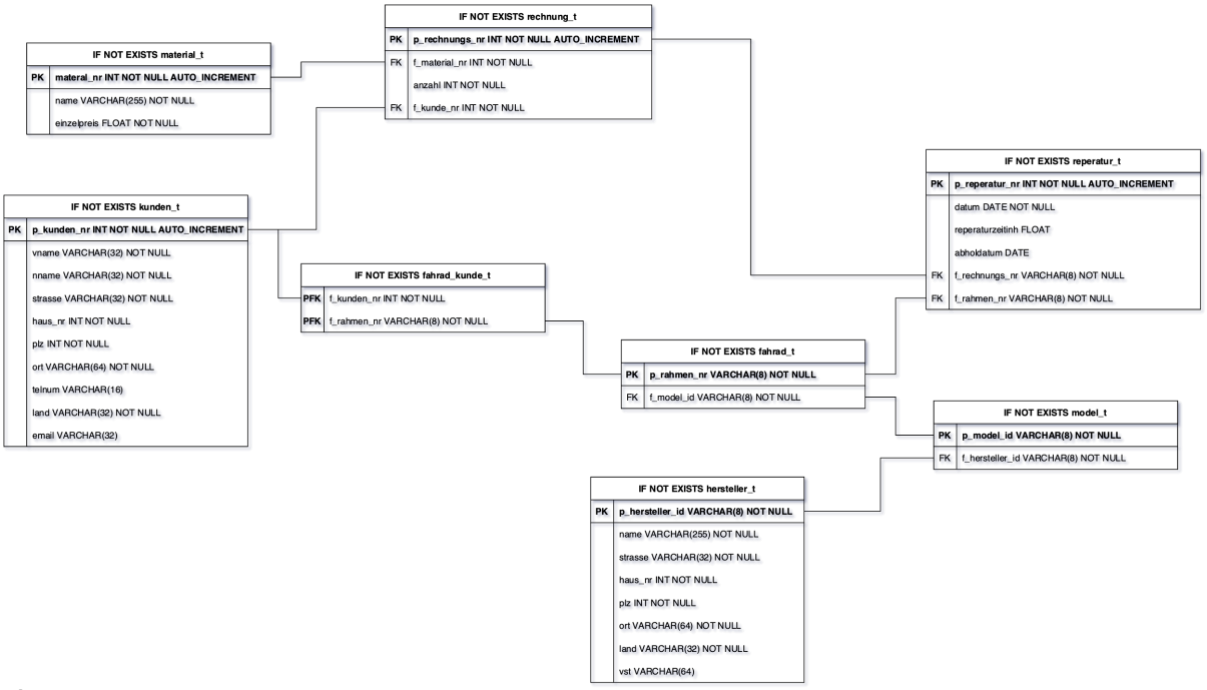
Das Relationenmodell ist ein mathematisches Konzept zur Darstellung von Daten in relationalen Datenbanken. Es wurde von Edgar F. Codd in den 1970er Jahren entwickelt und ist seitdem eine grundlegende Grundlage für moderne Datenbanksysteme. Im Relationenmodell werden Daten in Tabellen mit Zeilen und Spalten organisiert, wobei jede Zeile einen Datensatz darstellt und jede Spalte ein Attribut oder Merkmal dieses Datensatzes enthält.

Die Struktur des Relationenmodells basiert auf mathematischen Konzepten der Mengenlehre. Eine Relation entspricht einer Tabelle in einer relationalen Datenbank und repräsentiert eine Menge von Tupeln oder Datensätzen. Jedes Tupel besteht aus einer geordneten Liste von Attributwerten, wobei jede Spalte einem bestimmten Attribut entspricht.

Zentral für das Relationenmodell ist der Begriff der Schlüssel, der dazu dient, einzelne Tupel eindeutig zu identifizieren. Ein Primärschlüssel ist ein Attribut oder eine Kombination von Attributen, die jeden Datensatz eindeutig identifizieren. Darüber hinaus können in Beziehungen zwischen Tabellen definiert werden, um die Verknüpfung von Daten zwischen verschiedenen Tabellen zu ermöglichen.

Das Relationenmodell bietet eine klare und strukturierte Methode zur Organisation von Daten, die Flexibilität, Effizienz und Datenintegrität in Datenbankanwendungen ermöglicht. Es ist ein fundamentales Konzept in der Datenbanktheorie und bildet die Grundlage für die meisten modernen Datenbankmanagementsysteme (DBMS).

Beispiel



1. bis 3. Normalform

Definition

Die 1., 2. und 3. Normalform sind wichtige Konzepte im Bereich der Datenbanknormalisierung, die sicherstellen, dass eine Datenbank effizient und konsistent strukturiert ist, um Redundanzen und Anomalien zu minimieren. Hier sind die Definitionen dieser Normalformen:

Normalform (1NF):

Die 1. Normalform fordert, dass alle Attribute in einer Tabelle atomar sind, das heißt, jedes Attribut enthält nur einen einzigen Wert, und es gibt keine wiederholten Gruppen von Attributen. Jedes Feld in einer Tabelle sollte atomar sein und keine wiederholten oder mehrdeutigen Werte enthalten.

Normalform (2NF):

Die 2. Normalform beseitigt Redundanzen, die durch Teilschlüsselabhängigkeiten entstehen können. Eine Tabelle ist in 2NF, wenn sie in 1NF ist und jedes Nichtschlüsselattribut voll funktional von jedem Kandidatenschlüssel abhängt. Mit anderen Worten, es sollten keine abhängigen Teilmengen von Daten in derselben Tabelle vorhanden sein.

Normalform (3NF):

Die 3. Normalform beseitigt Transitive Abhängigkeiten zwischen Nichtschlüsselattributen, indem sie sicherstellt, dass jedes Nichtschlüsselattribut direkt von einem Schlüsselattribut abhängt, nicht jedoch von einem anderen Nichtschlüsselattribut. Eine Tabelle ist in 3NF, wenn sie in 2NF ist und keine Transitiven Abhängigkeiten zwischen Nichtschlüsselattributen existieren.

Diese Normalformen sind wichtige Schritte bei der Gestaltung von Datenbanken, um die Datenkonsistenz, Effizienz und Strukturierung zu verbessern und sicherzustellen, dass die Datenbank keine Anomalien aufweist.

Normalisierung

Normalisierung ist ein wichtiger Prozess in der Datenbankgestaltung, der darauf abzielt, eine Datenbank in eine effiziente und konsistente Struktur zu bringen, um Redundanzen und Anomalien zu minimieren. Der Normalisierungsprozess wird in mehrere Stufen oder Normalformen unterteilt, die sicherstellen, dass die Datenbank optimal strukturiert ist.

Hier sind die grundlegenden Schritte der Normalisierung:

Erste Normalform (1NF):

Jedes Attribut enthält nur einen Wert (atomare Werte).

Es gibt keine wiederholten Gruppen von Attributen.

Jedes Feld in einer Tabelle sollte atomar sein und keine wiederholten oder mehrdeutigen Werte enthalten.

Zweite Normalform (2NF):

Die Tabelle ist in 1NF.

Jedes Nichtschlüsselattribut ist funktional abhängig von jedem Kandidatenschlüssel.

Beseitigt Redundanzen, die durch Teilschlüsselabhängigkeiten entstehen können.

Dritte Normalform (3NF):

Die Tabelle ist in 2NF.

Es gibt keine Transitiven Abhängigkeiten zwischen Nichtschlüsselattributen.

Jedes Nichtschlüsselattribut ist direkt von einem Schlüsselattribut abhängig, nicht jedoch von einem anderen Nichtschlüsselattribut.

Weitere Normalformen wie die Boyce-Codd-Normalform (BCNF) und die Vierte Normalform (4NF) können angewendet werden, um noch komplexere Anomalien zu eliminieren. Die Normalisierung ist ein iterativer Prozess, der durchgeführt wird, während die Datenbankstruktur entwickelt oder modifiziert wird, um sicherzustellen, dass die Datenbank effizient, konsistent und leicht wartbar ist.

Änderungs- / Lösch/- Einfüge-Anomalie

Änderungs-, Lösch- und Einfüge-Anomalien sind Probleme, die in unzureichend normalisierten Datenbanken auftreten können. Diese Anomalien treten auf, wenn die Struktur der Datenbank nicht angemessen gestaltet ist und können zu Inkonsistenzen und Problemen bei der Datenmanipulation führen. Hier sind die Definitionen dieser Anomalien:

Änderungsanomalie:

Eine Änderungsanomalie tritt auf, wenn Änderungen an den Daten einer Entität schwierig, fehleranfällig oder sogar unmöglich sind, ohne dass dabei inkonsistente oder unerwartete Ergebnisse entstehen.

Zum Beispiel: In einer unnormalisierten Tabelle, in der dieselbe Information in mehreren Tupeln dupliziert wird, kann die Aktualisierung dieser Information an verschiedenen Stellen inkonsistente Daten hinterlassen oder erfordert eine wiederholte Aktualisierung an mehreren Stellen.

Löschanomalie:

Eine Löschanomalie tritt auf, wenn das Löschen von Daten eines Tupels unerwartete Datenverluste oder eine ungewollte Entfernung anderer relevanter Daten zur Folge hat.

Zum Beispiel: Wenn ein Datensatz gelöscht wird, der auch Informationen enthält, die für andere Datensätze relevant sind, gehen diese zusätzlichen Informationen verloren, was zu unerwarteten Lücken oder Inkonsistenzen in der Datenbank führt.

Einfügeanomalie:

Eine Einfügeanomalie tritt auf, wenn es unmöglich ist, bestimmte Daten einzufügen, ohne gleichzeitig unnötige oder inkonsistente Daten einzufügen.

Zum Beispiel: Wenn eine Tabelle keine Datensätze enthält, die von anderen Datensätzen abhängen, wird das Einfügen neuer Daten in die Tabelle unmöglich, da mindestens ein nicht verwendetes Attribut eingefügt werden muss, was zu inkonsistenten Daten führen kann.

Diese Anomalien werden durch eine unzureichende Normalisierung der Datenbankstruktur verursacht und können durch den Prozess der Normalisierung behoben werden, der sicherstellt, dass die Datenbank effizient strukturiert ist und Redundanzen sowie Inkonsistenzen minimiert werden.

Beispiele

Änderungsanomalie:

Angenommen, Sie haben eine unnormalisierte Tabelle für Kundenbestellungen, in der Kundeninformationen in jedem Auftragsdatensatz wiederholt werden. Wenn ein Kunde seine Kontaktinformationen ändert, müssen Sie alle Auftragsdatensätze des Kunden aktualisieren, um sicherzustellen, dass die Informationen konsistent bleiben. Wenn Sie dies vergessen oder falsch machen, bleiben einige Auftragsdatensätze mit veralteten Kundendaten bestehen, während andere aktualisiert wurden, was zu Inkonsistenzen führt.

Löschanomalie:

Nehmen wir an, Sie haben eine Tabelle für Mitarbeiter, die auch Informationen zu den von ihnen betreuten Projekten enthält. Wenn ein Mitarbeiter das Unternehmen verlässt und sein Datensatz gelöscht wird, gehen auch die Projektinformationen verloren, die ausschließlich mit diesem Mitarbeiter verbunden sind. Dadurch gehen potenziell wichtige Projektdaten verloren, obwohl sie unabhängig von den Mitarbeitern sind.

Einfügeanomalie:

Angenommen, Sie haben eine Tabelle für Klassenräume, die auch die Kursinformationen enthält, die in jedem Raum unterrichtet werden. Wenn ein neuer Klassenraum erstellt wird, kann dies nur geschehen, wenn gleichzeitig mindestens ein Kursraum zugewiesen wird. Dies führt dazu, dass Daten in die Tabelle eingefügt werden müssen, auch wenn keine Kurse für diesen Raum geplant sind, was zu unnötigen oder inkonsistenten Daten führt.

Diese Beispiele verdeutlichen, wie unzureichend normalisierte Datenbankstrukturen zu verschiedenen Arten von Anomalien führen können, die die Datenkonsistenz und -integrität beeinträchtigen können. Durch eine angemessene Normalisierung können solche Anomalien vermieden oder behoben werden.

Referentielle Integrität

Referentielle Integrität ist ein Konzept in der Datenbanktheorie, das sicherstellt, dass Beziehungen zwischen Tabellen konsistent und gültig sind. Dies wird erreicht, indem Regeln und Einschränkungen definiert werden, die die Konsistenz der referentiellen Beziehungen innerhalb einer Datenbank erzwingen. Das Konzept der referentiellen Integrität beinhaltet normalerweise die folgenden Aspekte:

Fremdschlüsselconstraint:

Eine häufig verwendete Methode zur Implementierung der referentiellen Integrität ist die Verwendung von Fremdschlüsseln. Ein Fremdschlüssel ist ein Attribut oder eine Gruppe von Attributen in einer Tabelle, die auf den Primärschlüssel einer anderen Tabelle verweisen.

Durch die Definition eines Fremdschlüssels in einer Tabelle wird eine Beziehung zwischen den Tabellen hergestellt. Der Fremdschlüssel in der Kindertabelle verweist auf den Primärschlüssel in der Elterntabelle, wodurch eine referenzielle Beziehung entsteht.

Einschränkungen und Regeln:

Referentielle Integrität wird durch das Festlegen von Einschränkungen und Regeln auf Datenbankebene erreicht. Typische Einschränkungen umfassen CASCADE, RESTRICT, SET NULL und SET DEFAULT.

CASCADE: Aktualisierungen oder Löschungen in der Elterntabelle werden auf die Kindertabelle übertragen.

RESTRICT: Es wird verhindert, dass Änderungen in der Elterntabelle durchgeführt werden, wenn dadurch referenzierte Datensätze in der Kindertabelle beeinflusst würden.

SET NULL oder SET DEFAULT: Setzt den Fremdschlüsselwert in der Kindertabelle auf NULL oder einen Standardwert, wenn der entsprechende Datensatz in der Elterntabelle gelöscht oder aktualisiert wird.

Erzwingung durch das Datenbanksystem:

Moderne Datenbankmanagementsysteme bieten Mechanismen zur automatischen Überwachung und Durchsetzung der referentiellen Integrität. Dadurch werden Verstöße gegen die definierten Regeln verhindert.

Wenn versucht wird, eine Aktion durchzuführen, die die referentielle Integrität verletzt (z.B. das Löschen eines Datensatzes, auf den noch verwiesen wird), wird eine Fehlermeldung generiert und die Aktion abgelehnt.

Die Implementierung der referentiellen Integrität gewährleistet die Konsistenz und Zuverlässigkeit von Beziehungen zwischen Tabellen in einer Datenbank und trägt dazu bei, Dateninkonsistenzen zu vermeiden.

Löschweitergabe

Die Löschweitergabe ist eine Einschränkung oder Regel, die in relationalen Datenbanken definiert werden kann, um sicherzustellen, dass Änderungen oder Löschungen in einer Tabelle auch in Bezug stehende Daten in anderen Tabellen beeinflussen. Dies geschieht typischerweise über die Verwendung von Fremdschlüsseln und spezifischen Löschaktionen.

Die Löschweitergabe wird häufig verwendet, um die referentielle Integrität zu wahren und sicherzustellen, dass Daten in Beziehung stehen. Es gibt verschiedene Möglichkeiten, wie die Löschweitergabe implementiert werden kann:

CASCADE:

Bei Verwendung der CASCADE-Löschweitergabe werden alle Datensätze in abhängigen Tabellen automatisch gelöscht, wenn der entsprechende Datensatz in der primären Tabelle gelöscht wird.

Dies bedeutet, dass, wenn ein Datensatz in der Elterntabelle gelöscht wird, alle zugehörigen Datensätze in den Kindertabellen ebenfalls gelöscht werden.

SET NULL:

Mit SET NULL wird der Fremdschlüsselwert in den Kindertabellen auf NULL gesetzt, wenn der zugehörige Datensatz in der Elterntabelle gelöscht wird.

Dadurch werden die Verweise in den Kindertabellen beibehalten, aber die Integrität bleibt erhalten.

SET DEFAULT:

Ähnlich wie SET NULL wird bei SET DEFAULT der Fremdschlüsselwert in den Kindertabellen auf einen voreingestellten Standardwert gesetzt, wenn der zugehörige Datensatz in der Elterntabelle gelöscht wird.

Die Wahl der geeigneten Löschaktion hängt von den Anforderungen der Anwendung und der gewünschten Datenintegrität ab. Die Löschweitergabe ist ein wichtiger Bestandteil der referentiellen Integrität und trägt dazu bei, sicherzustellen, dass die Datenbank konsistent bleibt, wenn Datensätze gelöscht werden.

Beispiel

```
-- Erstellen der Tabellen (nur relevante Teile gezeigt)
CREATE TABLE IF NOT EXISTS `hersteller_t` (
  `p_hersteller_id` VARCHAR(8) NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`p_hersteller_id`)
);
CREATE TABLE IF NOT EXISTS `model_t` (
  `p_model_id` VARCHAR(8) NOT NULL,
  `f_hersteller_id` VARCHAR(8) NOT NULL,
  PRIMARY KEY (`p_model_id`),
  CONSTRAINT `fk_model_hersteller_id` FOREIGN KEY
(`f_hersteller_id`) REFERENCES `hersteller_t` (`p_hersteller_id`) ON
DELETE CASCADE
);
```

Datentypen

Integer-Typen:

- INT: Ganzzahl mit Vorzeichen, z.B. -2147483648 bis 2147483647.
- BIGINT: Große Ganzzahl mit Vorzeichen, z.B. -9223372036854775808 bis 9223372036854775807.
- SMALLINT: Kleine Ganzzahl mit Vorzeichen, z.B. -32768 bis 32767.
- TINYINT: Sehr kleine Ganzzahl mit Vorzeichen, z.B. -128 bis 127.

Dezimal-Typen:

- DECIMAL: Feste Punkt-Dezimalzahl, z.B. DECIMAL(10,2) für eine Zahl mit 10 Stellen insgesamt und 2 Dezimalstellen.

Fließkommazahlen-Typen:

- FLOAT: Fließkommazahl mit einfacher Genauigkeit.
- DOUBLE: Fließkommazahl mit doppelter Genauigkeit.

Zeichenfolgen-Typen:

- CHAR: Zeichenfolge fester Länge, z.B. CHAR(10) für eine Zeichenfolge mit 10 Zeichen.
- VARCHAR: Zeichenfolge variabler Länge, z.B. VARCHAR(255) für eine Zeichenfolge mit maximal 255 Zeichen.
- TEXT: Lange Zeichenfolge mit variabler Länge, z.B. TEXT für eine große Menge von Textdaten.

Datum- und Zeit-Typen:

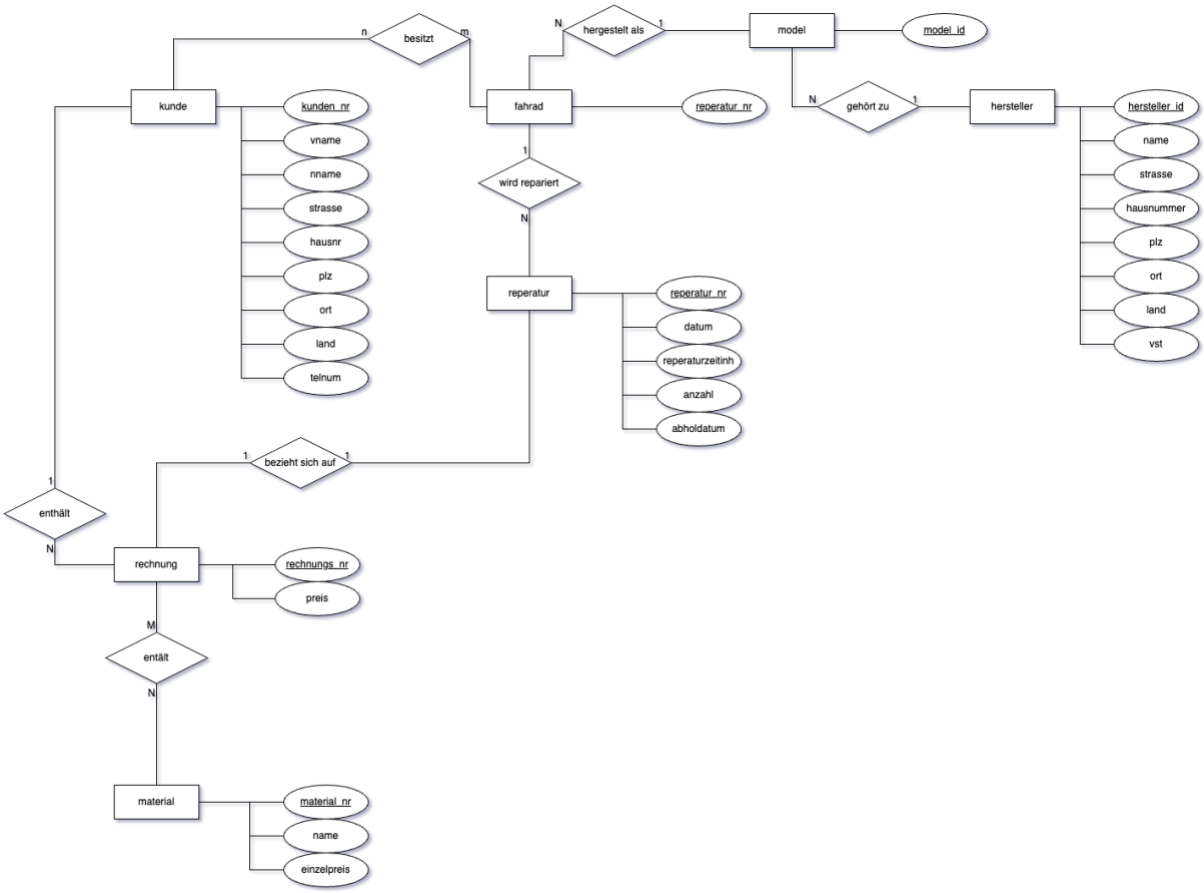
- DATE: Datumswert im Format 'YYYY-MM-DD'.
- TIME: Zeitwert im Format 'HH:MM:SS'.
- DATETIME: Kombinierter Datum- und Zeitwert im Format 'YYYY-MM-DD HH:MM:SS'.
- TIMESTAMP: Zeitstempel, der automatisch aktualisiert wird, wenn eine Zeile geändert wird.

Binäre-Typen:

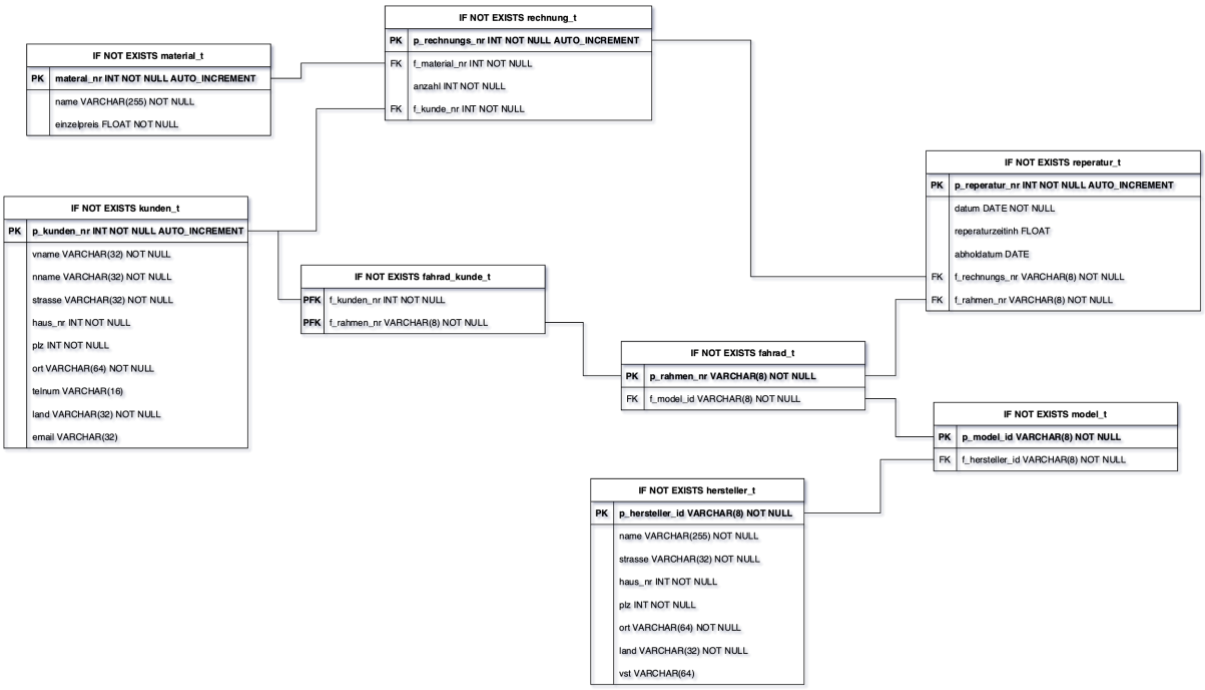
- BINARY: Binäre Zeichenfolge fester Länge.
- VARBINARY: Binäre Zeichenfolge variabler Länge.
- BLOB: Binäre große Objekte, die für die Speicherung von binären Daten wie Bilder oder Multimedia verwendet werden können.

Projekt VELOCÉBELLO

ER-Diagramm



Relationsmodel



SQL-Abfragen zum Erstellen der Tabellen

```
CREATE TABLE IF NOT EXISTS `kunden_t` (
  `p_kunden_nr` INT NOT NULL AUTO_INCREMENT,
  `vname` VARCHAR(32) NOT NULL,
  `nname` VARCHAR(32) NOT NULL,
  `strasse` VARCHAR(32) NOT NULL,
  `haus_nr` VARCHAR(4) NOT NULL,
  `plz` INT NOT NULL,
  `ort` VARCHAR(64) NOT NULL,
  `telnum` VARCHAR(16),
  `land` VARCHAR(32) NOT NULL,
  `email` VARCHAR(32),
  PRIMARY KEY (`p_kunden_nr`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_german2_ci;
-- CREATE TABLE statements for other tables (fahrad_t,
fahrad_kunde_t, model_t, hersteller_t, reparatur_t, material_t,
rechnung_t) follow similar patterns.
```

Vollständige Datenbank

SQL-Abfragen zum Einfügen der Musterdaten

```
-- Generiere 100 Dummy-Datensätze für die Tabelle 'kunden_t'
INSERT INTO `kunden_t` (`vname`, `nname`, `strasse`, `haus_nr`,
`plz`, `ort`, `telnum`, `land`, `email`)
SELECT CONCAT('Vorname', k.id),
       CONCAT('Nachname', k.id),
       CONCAT('Straße', k.id),
       k.id,
       k.id,
       CONCAT('Ort', k.id),
       CONCAT('Telefon', k.id),
       CONCAT('Land', k.id),
       CONCAT('email', k.id, '@example.com')
FROM (
    SELECT @rownum_k:=@rownum_k+1 AS id
    FROM (
        SELECT 0 UNION SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION
SELECT 4
    ) AS t1,
    (
        SELECT 0 UNION SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION
SELECT 4
    ) AS t2,
    (
        SELECT 0 UNION SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION
SELECT 4
    ) AS t3,
    (
        SELECT @rownum_k:=0
    ) AS r
) AS k;
```

Vollständige Datenbank

SQL-Abfragen zum Anpassen von Tabellen

```
ALTER TABLE `kunden_t`
ADD COLUMN `geburtsdatum` DATE AFTER `email`;
```

SQL-Abfragen für Geschäftsfälle

SELECT

```
SELECT `vname`, `nname`, `strasse`, `plz`, `ort`
FROM `kunden_t`;
```

UPDATE

```
UPDATE `kunden_t`
SET `telnum` = '+49 123456789'
WHERE `p_kunden_nr` = 1;
```

INSERT

```
INSERT INTO `kunden_t` (`vname`, `nname`, `strasse`, `haus_nr`,
`plz`, `ort`, `land`, `email`)
VALUES ('Max', 'Mustermann', 'Musterstraße', '42', '12345',
'Musterstadt', 'Deutschland', 'max@example.com');
```

DELETE

```
DELETE FROM `kunden_t`
WHERE `p_kunden_nr` = 1;
```

Komplexe SQL-Statements

SQL-Statement 1: Komplexes SELECT-Statement mit JOIN und Aggregatfunktionen

Das folgende SQL-Statement führt eine komplexe Abfrage durch, die Daten aus mehreren Tabellen kombiniert und aggregiert. Zum Beispiel:

```
SELECT k.name, COUNT(r.id) AS total_orders
FROM kunden_t k
LEFT JOIN rechnung_t r ON k.p_kunden_nr = r.f_kunde_nr
GROUP BY k.name;
```

Dieses Statement wählt den Namen jedes Kunden aus der Tabelle kunden_t aus und zählt die Anzahl der Bestellungen (Rechnungen) für jeden Kunden aus der Tabelle rechnung_t. Es verwendet einen LEFT JOIN, um Kunden ohne Bestellungen einzuschließen, und eine Gruppierung, um die Anzahl der Bestellungen für jeden Kunden zu aggregieren. Durch die Verwendung von Aggregatfunktionen und Joins können wir komplexe Informationen aus mehreren Tabellen zusammenführen und analysieren.

Die Abfrage wurde so formuliert, um alle Kunden, auch diejenigen ohne Bestellungen, zu erfassen und die Anzahl der Bestellungen für jeden Kunden zu ermitteln. Ein LEFT JOIN wird verwendet, um sicherzustellen, dass alle Kunden, unabhängig davon, ob sie Bestellungen haben oder nicht, zurückgegeben werden. Die Gruppierung nach dem Kundennamen ermöglicht es uns, die Anzahl der Bestellungen für jeden Kunden zu berechnen. Dies ermöglicht es uns, nützliche Einblicke in das Bestellverhalten unserer Kunden zu erhalten.

SQL-Statement 2: Komplexes UPDATE-Statement mit Unterabfrage

Das folgende SQL-Statement führt ein komplexes UPDATE durch, das Daten in einer Tabelle basierend auf einer Unterabfrage aktualisiert. Zum Beispiel:

```
UPDATE kunden_t
SET status = 'Premium'
WHERE p_kunden_nr IN (SELECT f_kunden_nr FROM rechnung_t GROUP BY
f_kunden_nr HAVING COUNT(*) > 10);
```

Dieses Statement aktualisiert den status in der Tabelle kunden_t auf 'Premium' für Kunden, die mehr als 10 Bestellungen in der Tabelle rechnung_t haben. Die Unterabfrage wählt die Kunden aus, die die Bedingung erfüllen (mehr als 10 Bestellungen) und gibt ihre Kundennummern zurück, die dann für das UPDATE verwendet werden. Durch die Verwendung einer Unterabfrage können wir komplexe Bedingungen auf Basis von aggregierten Daten formulieren und die betroffenen Datensätze gezielt aktualisieren.

Die Abfrage wurde so formuliert, um Kunden zu identifizieren, die eine bestimmte Anzahl von Bestellungen überschreiten, und ihren Status entsprechend zu aktualisieren. Eine Unterabfrage wird verwendet, um Kundennummern zu identifizieren, die die Bedingung erfüllen, und diese dann für das UPDATE zu verwenden. Dadurch können wir den Status der betroffenen Kunden effizient aktualisieren, basierend auf komplexen Kriterien, ohne manuell jede einzelne Zeile zu überprüfen.

Quellen

https://chat.openai.com/share/aa77b5c1-e054-4b59-b43a-14b8fe0f330f
https://chat.openai.com/share/07705990-a299-4c41-8601-86285406c2a4
https://chat.openai.com/share/ca442751-bfec-4696-8d9a-69d0ec3e3415
https://chat.openai.com/share/86e793b2-98e1-4a13-ad23-e15ad43b726f