Kostenloses eBook

LERNEN SQL

Free unaffiliated eBook created from
Stack Overflow contributors.

#sql

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit SQL ... 2

Bemerkungen ... 2

Versionen ... 2

Examples ... 2

Überblick ... 2

Kapitel 2: AKTUALISIEREN ... 4

Syntax ... 4

Examples ... 4

Alle Zeilen aktualisieren ... 4

Angegebene Zeilen aktualisieren ... 4

Bestehende Werte ändern ... 4

UPDATE mit Daten aus einer anderen Tabelle ... 5

Standard SQL ... 5

SQL: 2003 ... 5

SQL Server ... 5

Aktualisierte Datensätze erfassen ... 6

Kapitel 3: Allgemeine Tabellenausdrücke ... 7

Syntax ... 7

Bemerkungen ... 7

Examples ... 7

Temporäre Abfrage ... 7

rekursiv in einem Baum aufsteigen ... 8

Werte generieren ... 9

rekursives Auflisten eines Teilbaums ... 9

Oracle CONNECT BY-Funktionalität mit rekursiven CTEs ... 10

Generieren Sie rekursiv Datumsangaben, die beispielsweise um Teamplanungen erweitert werde ... 11

Refactoring einer Abfrage zur Verwendung von Common Table-Ausdrücken ... 12

Beispiel für eine komplexe SQL mit Common Table Expression ... 13

Kapitel 4: ALTER TABELLE ... 14

Einführung ... 14
Syntax ... 14
Examples ... 14
Spalten hinzufügen ... 14
Drop Column ... 14
Drop-Einschränkung ... 14
Einschränkung hinzufügen ... 14
Spalte ändern ... 15
Primärschlüssel hinzufügen ... 15
Kapitel 5: Ansichten ... 16
Examples ... 16
Einfache ansichten ... 16
Komplexe Ansichten ... 16
Kapitel 6: Ausführungsblöcke ... 17
Examples ... 17
Verwenden von BEGIN ... END ... 17
Kapitel 7: AUSSER ... 18
Bemerkungen ... 18
Examples ... 18
Wählen Sie den Datensatz aus, außer wenn sich Werte in diesem anderen Datensatz befinden ... 18
Kapitel 8: Beispieldatenbanken und Tabellen ... 19
Examples ... 19
Auto-Shop-Datenbank ... 19
Beziehungen zwischen Tabellen ... 19
Abteilungen ... 19
Angestellte ... 20
Kunden ... 20
Autos ... 21
Bibliotheksdatenbank ... 22
Beziehungen zwischen Tabellen ... 22
Autoren ... 22
Bücher ... 23

BücherAuthors ... 24
Beispiele ... 25
Ländertabelle ... 25
Länder ... 25
Kapitel 9: BEITRETEN ... 27
Einführung ... 27
Syntax ... 27
Bemerkungen ... 27
Examples ... 27
Grundlegende explizite innere Verknüpfung ... 27
Implizite Verknüpfung ... 28
Linke äußere Verbindung ... 28
Wie funktioniert das? ... 29
Selbst beitreten ... 31
Wie funktioniert das? ... 31
CROSS JOIN ... 33
An einer Unterabfrage teilnehmen ... 34
CROSS APPLY & LATERAL JOIN ... 34
FULL JOIN ... 36
Rekursive JOINs ... 37
Unterschiede zwischen inneren und äußeren Verbindungen ... 37
Inner Join ... 39
Linke äußere Verbindung ... 39
Rechte äußere Verbindung ... 39
Volle äußere Verbindung ... 39
JOIN Terminologie: Inner, Äußer, Halb, Anti ... 39
Inner Join ... 39
Linke äußere Verbindung ... 39
Rechter äußerer Join ... 39
Voller äußerer Join ... 39
Left Semi Join ... 39

Right Semi Join ... 39
Linke Anti-Semi-Verbindung ... 39
Right Anti Semi Join ... 40
Cross Join ... 40
Self-Join ... 41
Kapitel 10: Bemerkungen ... 42
Examples ... 42
Einzeilige Kommentare ... 42
Mehrzeilige Kommentare ... 42
Kapitel 11: Bereinigen Sie Code in SQL ... 43
Einführung ... 43
Examples ... 43
Formatierung und Schreibweise von Schlüsselwörtern und Namen ... 43
Tabellen- / Spaltennamen ... 43
Schlüsselwörter ... 43
WÄHLEN * ... 43
Einrücken ... 44
Schließt sich an ... 45
Kapitel 12: Cascading Delete ... 47
Examples ... 47
ON DELETE CASCADE ... 47
Kapitel 13: Datenbank erstellen ... 49
Syntax ... 49
Examples ... 49
Datenbank erstellen ... 49
Kapitel 14: Datentypen ... 50
Examples ... 50
DECIMAL und NUMERIC ... 50
FLOAT und REAL ... 50
Ganzzahlen ... 50
Geld und Kleinigkeiten ... 50

BINARY und VARBINARY ... 51
CHAR und VARCHAR ... 51
NCHAR und NVARCHAR ... 51
EINDEUTIGE KENNUNG ... 52
Kapitel 15: DROP oder DELETE Database ... 53
Syntax ... 53
Bemerkungen ... 53
Examples ... 53
DROP-Datenbank ... 53
Kapitel 16: DROP-Tabelle ... 54
Bemerkungen ... 54
Examples ... 54
Einfacher Tropfen ... 54
Vor dem Ablegen auf Existenz prüfen ... 54
Kapitel 17: Duplikate in einem Spalten-Subset mit Detail suchen ... 55
Bemerkungen ... 55
Examples ... 55
Studenten mit demselben Namen und Geburtsdatum ... 55
Kapitel 18: EINFÜGEN ... 56
Syntax ... 56
Examples ... 56
Neue Zeile einfügen ... 56
Nur angegebene Spalten einfügen ... 56
INSERT-Daten aus einer anderen Tabelle mit SELECT ... 56
Fügen Sie mehrere Zeilen gleichzeitig ein ... 57
Kapitel 19: EXISTS-Klausel ... 58
Examples ... 58
EXISTS-Klausel ... 58
Erhalten Sie alle Kunden mit mindestens einer Bestellung ... 58
Erhalten Sie alle Kunden ohne Bestellung ... 58
Zweck ... 59

Kapitel 20: EXPLAIN und BESCHREIBEN 60
Examples 60
BESCHREIBEN Tabellenname; 60
EXPLAIN Abfrage auswählen 60
Kapitel 21: FALL 62
Einführung 62
Syntax 62
Bemerkungen 62
Examples 62
CASE in SELECT gesucht (entspricht einem booleschen Ausdruck) 62
Mit CASE COUNT die Anzahl der Zeilen in einer Spalte mit einer Bedingung übereinstimmen. 63
Abkürzung CASE in SELECT 64
CASE in einer Klausel ORDER BY 65
CASE in UPDATE verwenden 65
CASE-Verwendung für NULL-Werte, die zuletzt bestellt wurden 66
CASE in ORDER BY-Klausel zum Sortieren von Datensätzen nach dem niedrigsten Wert von 2 Spa 66
Beispieldaten 67
Abfrage 67
Ergebnisse 67
Erläuterung 67
Kapitel 22: Fensterfunktionen 69
Examples 69
Hinzufügen der insgesamt ausgewählten Zeilen zu jeder Zeile 69
Ein Flag setzen, wenn andere Zeilen eine gemeinsame Eigenschaft haben 69
Laufende Summe 70
N die aktuellsten Zeilen über mehrere Gruppierungen abrufen 71
Suchen nach Datensätzen mit der Funktion LAG () 71
Kapitel 23: Filtern Sie die Ergebnisse mit WHERE und HAVING 73
Syntax 73
Examples 73
Die WHERE-Klausel gibt nur Zeilen zurück, die ihren Kriterien entsprechen 73
Verwenden Sie IN, um Zeilen mit einem in einer Liste enthaltenen Wert zurückzugeben 73

Verwenden Sie LIKE, um übereinstimmende Zeichenfolgen und Teilzeichenfolgen zu finden ... 73
WHERE-Klausel mit NULL / NOT NULL-Werten ... 74
Verwenden Sie HAVING mit Aggregatfunktionen ... 75
Verwenden Sie ZWISCHEN, um Ergebnisse zu filtern ... 75
Gleichberechtigung ... 76
UND UND ODER ... 77
Verwenden Sie HAVING, um nach mehreren Bedingungen in einer Gruppe zu suchen ... 78
Wo EXISTEN ... 79
Kapitel 24: Fremde Schlüssel ... 80
Examples ... 80
Erstellen einer Tabelle mit einem Fremdschlüssel ... 80
Fremdschlüssel erklärt ... 80
Ein paar Tipps zur Verwendung von Fremdschlüsseln ... 81
Kapitel 25: FUNKTION ERSTELLEN ... 82
Syntax ... 82
Parameter ... 82
Bemerkungen ... 82
Examples ... 82
Erstellen Sie eine neue Funktion ... 82
Kapitel 26: Funktionen (Aggregat) ... 84
Syntax ... 84
Bemerkungen ... 84
Examples ... 85
SUMME ... 85
Bedingte Aggregation ... 85
AVG () ... 86
BEISPIELTABELLE ... 86
ABFRAGE ... 86
ERGEBNISSE ... 87
List Verkettung ... 87
MySQL ... 87
Oracle und DB2 ... 87

PostgreSQL ... 87
SQL Server ... 88
SQL Server 2016 und früher ... 88
SQL Server 2017 und SQL Azure ... 88
SQLite ... 88
Anzahl ... 89
Max ... 90
Mindest ... 90
Kapitel 27: Funktionen (Analyse) ... 91
Einführung ... 91
Syntax ... 91
Examples ... 91
FIRST_VALUE ... 91
LAST_VALUE ... 92
LAG und LEAD ... 92
PERCENT_RANK und CUME_DIST ... 93
PERCENTILE_DISC und PERCENTILE_CONT ... 94
Kapitel 28: Funktionen (Skalar / Einzelne Reihe) ... 97
Einführung ... 97
Syntax ... 97
Bemerkungen ... 97
Examples ... 98
Charakteränderungen ... 98
Datum und Uhrzeit ... 98
Konfigurations- und Konvertierungsfunktion ... 100
Logische und mathematische Funktion ... 101
SQL hat zwei logische Funktionen - CHOOSE und IIF ... 101
SQL enthält mehrere mathematische Funktionen, mit denen Sie Berechnungen zu Eingabewerten ... 102
Kapitel 29: Gespeicherte Prozeduren ... 104
Bemerkungen ... 104
Examples ... 104

Erstellen Sie eine gespeicherte Prozedur, und rufen Sie sie auf ... 104
Kapitel 30: GRANT und REVOKE ... 105
Syntax ... 105
Bemerkungen ... 105
Examples ... 105
Berechtigungen erteilen / entziehen ... 105
Kapitel 31: GRUPPIERE NACH ... 106
Einführung ... 106
Syntax ... 106
Examples ... 106
USE GROUP BY COUNT die Anzahl der Zeilen für jeden eindeutigen Eintrag in einer bestimmten ... 106
Filtern Sie die GROUP BY-Ergebnisse mit einer HAVING-Klausel ... 108
Grundlegendes GROUP BY-Beispiel ... 108
ROLAP-Aggregation (Data Mining) ... 109
Beschreibung ... 109
Beispiele ... 110
Mit Würfel ... 110
Mit aufrollen ... 111
Kapitel 32: Indizes ... 112
Einführung ... 112
Bemerkungen ... 112
Examples ... 112
Index erstellen ... 112
Gruppierte, eindeutige und sortierte Indizes ... 113
Einfügen mit einem eindeutigen Index ... 114
SAP ASE: Index löschen ... 114
Sortierter Index ... 114
Einen Index löschen oder ihn deaktivieren und neu erstellen ... 114
Eindeutiger Index, der NULL erlaubt ... 115
Index neu erstellen ... 115
Clustered-Index ... 115
Nicht gruppierter Index ... 116

Teilweiser oder gefilterter Index ... 116
Kapitel 33: Informationsschema ... 118
Examples ... 118
Grundlegende Informationsschemasuche ... 118
Kapitel 34: IN-Klausel ... 119
Examples ... 119
Einfache IN-Klausel ... 119
IN-Klausel mit einer Unterabfrage verwenden ... 119
Kapitel 35: Kennung ... 120
Einführung ... 120
Examples ... 120
Nicht zitierte Bezeichner ... 120
Kapitel 36: Kreuz anwenden, außen anwenden ... 121
Examples ... 121
GRUNDLAGEN VON CROSS APPLY und OUTER APPLY ... 121
Kapitel 37: KÜRZEN ... 124
Einführung ... 124
Syntax ... 124
Bemerkungen ... 124
Examples ... 124
Alle Zeilen aus der Employee-Tabelle entfernen ... 124
Kapitel 38: LIKE Operator ... 126
Syntax ... 126
Bemerkungen ... 126
Examples ... 126
Übereinstimmung mit offenem Muster ... 126
Einzelzeichenübereinstimmung ... 128
Übereinstimmung nach Bereich oder Satz ... 128
JEDER gegen ALLE ... 129
Suchen Sie nach einer Reihe von Zeichen ... 129
ESCAPE-Anweisung in der LIKE-Abfrage ... 129
Platzhalterzeichen ... 130

Kapitel 39: LÖSCHEN 132
Einführung 132
Syntax 132
Examples 132
LÖSCHEN Sie bestimmte Zeilen mit WHERE 132
LÖSCHEN Sie alle Zeilen 132
TRUNCATE-Klausel 132
LÖSCHEN Sie bestimmte Zeilen basierend auf Vergleichen mit anderen Tabellen 132
Kapitel 40: Löst aus 134
Examples 134
TRIGGER ERSTELLEN 134
Verwenden Sie Auslöser, um einen "Papierkorb" für gelöschte Elemente zu verwalten 134
Kapitel 41: Materialisierte Ansichten 135
Einführung 135
Examples 135
PostgreSQL-Beispiel 135
Kapitel 42: NULL 136
Einführung 136
Examples 136
Filtern nach NULL in Abfragen 136
Nullable-Spalten in Tabellen 136
Felder auf NULL aktualisieren 137
Zeilen mit NULL-Feldern einfügen 137
Kapitel 43: Primärschlüssel 138
Syntax 138
Examples 138
Erstellen eines Primärschlüssels 138
Auto Increment verwenden 138
Kapitel 44: Reihenfolge der Ausführung 140
Examples 140
Logische Reihenfolge der Abfrageverarbeitung in SQL 140

Kapitel 45: Relationale Algebra ... 142
Examples ... 142
Überblick ... 142
WÄHLEN ... 142
PROJEKT ... 143
GEBEN ... 144
NATÜRLICHER JOIN ... 144
ALIAS ... 145
TEILEN ... 145
UNION ... 145
ÜBERSCHNEIDUNG ... 145
UNTERSCHIED ... 146
UPDATE (: =) ... 146
MAL ... 146
Kapitel 46: Sequenz ... 147
Examples ... 147
Sequenz erstellen ... 147
Sequenzen verwenden ... 147
Kapitel 47: SKIP TAKE (Paginierung) ... 148
Examples ... 148
Einige Zeilen aus dem Ergebnis überspringen ... 148
Begrenzung der Anzahl der Ergebnisse ... 148
Überspringen und einige Ergebnisse aufnehmen (Paginierung) ... 149
Kapitel 48: SORTIEREN NACH ... 150
Examples ... 150
Verwenden Sie ORDER BY mit TOP, um die ersten x Zeilen basierend auf dem Wert einer Spalte ... 150
Sortierung nach mehreren Spalten ... 151
Sortierung nach Spaltennummer (statt Name) ... 151
Auftrag von Alias ... 152
Angepasste Sortierreihenfolge ... 152
Kapitel 49: SQL CURSOR ... 154

Examples ... 154
Beispiel für einen Cursor, der alle Zeilen nach Index für jede Datenbank abfragt ... 154
Kapitel 50: SQL Group By vs. Distinct ... 156
Examples ... 156
Unterschied zwischen GROUP BY und DISTINCT ... 156
Kapitel 51: SQL-Injektion ... 158
Einführung ... 158
Examples ... 158
Beispiel einer SQL-Injektion ... 158
einfache Injektionsprobe ... 159
Kapitel 52: String-Funktionen ... 161
Einführung ... 161
Syntax ... 161
Bemerkungen ... 161
Examples ... 161
Leere Räume abschneiden ... 161
Verketten ... 162
Groß- und Kleinschreibung ... 162
Unterstring ... 162
Teilt ... 163
Zeug ... 163
Länge ... 163
Ersetzen ... 164
LINKS RECHTS ... 164
UMKEHREN ... 165
ERSETZEN ... 165
REGEXP ... 165
Funktion in SQL-Abfrage ersetzen und aktualisieren ... 165
PARSENAME ... 166
INSTR ... 167
Kapitel 53: Synonyme ... 168
Examples ... 168

Synonym erstellen ... 168
Kapitel 54: TABELLE ERSTELLEN ... 169
Einführung ... 169
Syntax ... 169
Parameter ... 169
Bemerkungen ... 169
Examples ... 169
Erstellen Sie eine neue Tabelle ... 169
Tabelle erstellen aus Auswählen ... 170
Eine Tabelle duplizieren ... 170
CREATE TABLE Mit dem FOREIGN KEY ... 170
Erstellen Sie eine temporäre oder speicherinterne Tabelle ... 171
PostgreSQL und SQLite ... 171
SQL Server ... 171
Kapitel 55: Tisch Design ... 173
Bemerkungen ... 173
Examples ... 173
Eigenschaften eines gut gestalteten Tisches ... 173
Kapitel 56: Transaktionen ... 175
Bemerkungen ... 175
Examples ... 175
Einfache Transaktion ... 175
Rollback-Transaktion ... 175
Kapitel 57: UND-ODER-Operatoren ... 176
Syntax ... 176
Examples ... 176
UND ODER Beispiel ... 176
Kapitel 58: UNION / UNION ALL ... 177
Einführung ... 177
Syntax ... 177
Bemerkungen ... 177

Examples ... 177
Grundlegende UNION ALL-Abfrage ... 177
Einfache Erklärung und Beispiel ... 178
Kapitel 59: Unterabfragen ... 180
Bemerkungen ... 180
Examples ... 180
Unterabfrage in WHERE-Klausel ... 180
Unterabfrage in FROM-Klausel ... 180
Unterabfrage in SELECT-Klausel ... 180
Unterabfragen in FROM-Klausel ... 180
Unterabfragen in der WHERE-Klausel ... 181
Unterabfragen in SELECT-Klausel ... 181
Filtern Sie die Abfrageergebnisse mithilfe der Abfrage in einer anderen Tabelle ... 182
Korrelierte Unterabfragen ... 182
Kapitel 60: VERSCHMELZEN ... 183
Einführung ... 183
Examples ... 183
MERGE, um das Ziel mit der Quelle zu verbinden ... 183
MySQL: Benutzer nach Namen zählen ... 183
PostgreSQL: Benutzer nach Namen zählen ... 184
Kapitel 61: VERSUCHEN / FANGEN ... 185
Bemerkungen ... 185
Examples ... 185
Transaktion in einem TRY / CATCH ... 185
Kapitel 62: WÄHLEN ... 186
Einführung ... 186
Syntax ... 186
Bemerkungen ... 186
Examples ... 186
Verwenden Sie das Platzhalterzeichen, um alle Spalten in einer Abfrage auszuwählen ... 186
Einfache select-Anweisung ... 187
Punktnotation ... 187

Wann können Sie * , um die obige Warnung zu beachten? 188
Auswahl mit Bedingung 189
Wählen Sie einzelne Spalten aus 189
SELECT Verwenden Sie Spaltenaliasnamen 190
Alle Versionen von SQL 191
Verschiedene SQL-Versionen 191
Alle Versionen von SQL 192
Verschiedene SQL-Versionen 193
Auswahl mit sortierten Ergebnissen 194
Wählen Sie Spalten aus, die nach reservierten Schlüsselwörtern benannt sind 194
Festlegen der angegebenen Anzahl von Datensätzen 195
Auswahl mit Tabellenalias 196
Zeilen aus mehreren Tabellen auswählen 197
Auswahl mit Aggregatfunktionen 198
Durchschnittlich 198
Minimum 198
Maximal 198
Anzahl 198
Summe 199
Auswahl mit null 199
Auswahl mit CASE 199
Auswählen ohne die Tabelle zu sperren 199
Wähle eindeutig (nur eindeutige Werte) 200
Wählen Sie mit der Bedingung mehrerer Werte aus der Spalte 201
Sammelt das Ergebnis für Zeilengruppen 201
Auswahl mit mehr als 1 Bedingung. 202
Kapitel 63: XML 204
Examples 204
Abfrage vom XML-Datentyp 204
Kapitel 64: Zeilennummer 205
Syntax 205

Examples 205
Zeilennummern ohne Partitionen 205
Zeilennummern mit Partitionen 205
Alle bis auf den letzten Datensatz löschen (1 bis viele Tabelle) 205
Credits 206

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sql

It is an unofficial and free SQL ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official SQL.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit SQL

Bemerkungen

SQL ist eine strukturierte Abfragesprache, die zum Verwalten von Daten in einem relationalen Datenbanksystem verwendet wird. Verschiedene Anbieter haben die Sprache verbessert und haben verschiedene Geschmacksrichtungen für die Sprache.

Hinweis: Dieses Tag bezieht sich explizit auf den **ISO / ANSI SQL-Standard** . keine spezifische Implementierung dieses Standards.

Versionen

Ausführung	Kurzer Name	Standard	Veröffentlichungsdatum
1986	SQL-86	ANSI X3.135-1986, ISO 9075: 1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO / IEC 9075: 1989	1989-01-01
1992	SQL-92	ISO / IEC 9075: 1992	1992-01-01
1999	SQL: 1999	ISO / IEC 9075: 1999	1999-12-16
2003	SQL: 2003	ISO / IEC 9075: 2003	2003-12-15
2006	SQL: 2006	ISO / IEC 9075: 2006	2006-06-01
2008	SQL: 2008	ISO / IEC 9075: 2008	2008-07-15
2011	SQL: 2011	ISO / IEC 9075: 2011	2011-12-15
2016	SQL: 2016	ISO / IEC 9075: 2016	2016-12-01

Examples

Überblick

Structured Query Language (SQL) ist eine spezielle Programmiersprache zur Verwaltung von Daten, die in einem relationalen Datenbankverwaltungssystem (RDBMS) gespeichert sind. SQL-ähnliche Sprachen können auch in relationalen Datenstrom-Verwaltungssystemen (RDSMS) oder in Datenbanken "nicht nur SQL" (NoSQL) verwendet werden.

SQL umfasst 3 Hauptsprachen:

1. Data Definition Language (DDL): Zum Erstellen und Ändern der Datenbankstruktur.
2. Datenmanipulationssprache (DML): Zum Lesen, Einfügen, Aktualisieren und Löschen von Daten in der Datenbank.
3. Data Control Language (DCL): Zum Steuern des Zugriffs auf die in der Datenbank gespeicherten Daten.

SQL-Artikel in Wikipedia

Die wichtigsten DML-Operationen sind Create, Read, Update und Delete (kurz CRUD), die von den Anweisungen `INSERT` , `SELECT` , `UPDATE` und `DELETE` .
Es gibt auch eine (kürzlich hinzugefügte) `MERGE` Anweisung, die alle 3 Schreibvorgänge ausführen kann (INSERT, UPDATE, DELETE).

CRUD-Artikel auf Wikipedia

Viele SQL-Datenbanken sind als Client / Server-Systeme implementiert. Der Begriff "SQL Server" beschreibt eine solche Datenbank.
Zur gleichen Zeit erstellt Microsoft eine Datenbank mit dem Namen "SQL Server". Während diese Datenbank einen SQL-Dialekt spricht, sind die für diese Datenbank spezifischen Informationen in diesem Tag nicht Thema, sondern in der SQL Server-Dokumentation enthalten .

Erste Schritte mit SQL online lesen: https://riptutorial.com/de/sql/topic/184/erste-schritte-mit-sql

Kapitel 2: AKTUALISIEREN

Syntax

- UPDATE- *Tabelle*
  SET *Spaltenname* = *Wert* , *Spaltenname2* = *Wert_2* , ..., *Spaltenname_n* = *Wert_n*
  WHERE- *Bedingung* (*logischer Operator* condition_n)

Examples

Alle Zeilen aktualisieren

In diesem Beispiel wird die Cars-Tabelle aus den Beispieldatenbanken verwendet.

```
UPDATE Cars
SET Status = 'READY'
```

Diese Anweisung setzt die Spalte "Status" aller Zeilen der Tabelle "Cars" auf "READY", da sie keine `WHERE` Klausel zum Filtern des Satzes von Zeilen enthält.

Angegebene Zeilen aktualisieren

In diesem Beispiel wird die Cars-Tabelle aus den Beispieldatenbanken verwendet.

```
UPDATE
    Cars
SET
    Status = 'READY'
WHERE
    Id = 4
```

Diese Anweisung setzt den Status der Zeile "Autos" mit der ID 4 auf "BEREIT".

`WHERE` Klausel enthält einen logischen Ausdruck, der für jede Zeile ausgewertet wird. Wenn eine Zeile die Kriterien erfüllt, wird der Wert aktualisiert. Ansonsten bleibt eine Zeile unverändert.

Bestehende Werte ändern

In diesem Beispiel wird die Cars-Tabelle aus den Beispieldatenbanken verwendet.

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

Aktualisierungsvorgänge können aktuelle Werte in der aktualisierten Zeile enthalten. In diesem einfachen Beispiel wird `TotalCost` für zwei Zeilen um 100 erhöht:

- Die Gesamtkosten von Wagen Nr. 3 werden von 100 auf 200 erhöht
- Die Gesamtkosten von Wagen Nr. 4 werden von 1254 auf 1354 erhöht

Der neue Wert einer Spalte kann aus dem vorherigen Wert oder aus dem Wert einer anderen Spalte in derselben Tabelle oder einer verbundenen Tabelle abgeleitet werden.

UPDATE mit Daten aus einer anderen Tabelle

In den folgenden Beispielen wird eine `PhoneNumber` für jeden Mitarbeiter angegeben, der ebenfalls `Customer` und derzeit keine Telefonnummer in der `Employees` Tabelle festgelegt hat.

(Diese Beispiele verwenden die Employees- und Customers- Tabellen aus den Beispieldatenbanken.)

Standard SQL

Aktualisieren Sie mit einer korrelierten Unterabfrage:

```
UPDATE
    Employees
SET PhoneNumber =
    (SELECT
         c.PhoneNumber
     FROM
         Customers c
     WHERE
         c.FName = Employees.FName
         AND c.LName = Employees.LName)
WHERE Employees.PhoneNumber IS NULL
```

SQL: 2003

Update mit `MERGE` :

```
MERGE INTO
    Employees e
USING
    Customers c
ON
    e.FName = c.Fname
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
   UPDATE
      SET PhoneNumber = c.PhoneNumber
```

SQL Server

Update mit `INNER JOIN` :

```
UPDATE
    Employees
SET
    PhoneNumber = c.PhoneNumber
FROM
    Employees e
INNER JOIN Customers c
        ON e.FName = c.FName
        AND e.LName = c.LName
WHERE
    PhoneNumber IS NULL
```

Aktualisierte Datensätze erfassen

Manchmal möchte man die gerade aktualisierten Datensätze erfassen.

```
CREATE TABLE #TempUpdated(ID INT)

Update TableName SET Col1 = 42
    OUTPUT inserted.ID INTO #TempUpdated
    WHERE Id > 50
```

AKTUALISIEREN online lesen: https://riptutorial.com/de/sql/topic/321/aktualisieren

Kapitel 3: Allgemeine Tabellenausdrücke

Syntax

- WITH QueryName [(Spaltenname, ...)] AS (
  WÄHLEN ...
)
  SELECT ... FROM QueryName ...;

- WITH RECURSIVE QueryName [(Spaltenname, ...)] AS (
  WÄHLEN ...
  UNION [ALL]
  SELECT ... FROM QueryName ...
)
  SELECT ... FROM QueryName ...;

Bemerkungen

Offizielle Dokumentation: WITH-Klausel

Ein allgemeiner Tabellenausdruck ist eine temporäre Ergebnismenge und kann Ergebnis einer komplexen Unterabfrage sein. Sie wird mithilfe der WITH-Klausel definiert. CTE verbessert die Lesbarkeit und wird im Arbeitsspeicher erstellt und nicht in der TempDB-Datenbank, in der die Variablen "Temp Table" und "Table" erstellt werden.

Schlüsselkonzepte allgemeiner Tabellenausdrücke:

- Kann verwendet werden, um komplexe Abfragen aufzubrechen, insbesondere komplexe Joins und Unterabfragen.
- Ist eine Möglichkeit, eine Abfragedefinition einzukapseln.
- Bestehen Sie nur bis zur nächsten Abfrage.
- Die korrekte Verwendung kann zu Verbesserungen der Codequalität / Wartbarkeit und Geschwindigkeit führen.
- Kann verwendet werden, um auf die resultierende Tabelle mehrmals in derselben Anweisung zu verweisen (Duplizierung in SQL zu vermeiden).
- Kann eine Ansicht ersetzen, wenn die allgemeine Verwendung einer Ansicht nicht erforderlich ist; Das heißt, Sie müssen die Definition nicht in Metadaten speichern.
- Wird beim Aufruf ausgeführt, nicht bei der Definition. Wenn der CTE in einer Abfrage mehrmals verwendet wird, wird er mehrmals ausgeführt (möglicherweise mit unterschiedlichen Ergebnissen).

Examples

Temporäre Abfrage

Diese verhalten sich wie verschachtelte Unterabfragen, jedoch mit einer anderen Syntax.

```
WITH ReadyCars AS (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
)
SELECT ID, Model, TotalCost
FROM ReadyCars
ORDER BY TotalCost;
```

ICH WÜRDE	Modell	Gesamtkosten
1	Ford F-150	200
2	Ford F-150	230

Äquivalente Unterabfragesyntax

```
SELECT ID, Model, TotalCost
FROM (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
) AS ReadyCars
ORDER BY TotalCost
```

rekursiv in einem Baum aufsteigen

```
WITH RECURSIVE ManagersOfJonathon AS (
    -- start with this row
    SELECT *
    FROM Employees
    WHERE ID = 4

    UNION ALL

    -- get manager(s) of all previously selected rows
    SELECT Employees.*
    FROM Employees
    JOIN ManagersOfJonathon
        ON Employees.ID = ManagersOfJonathon.ManagerID
)
SELECT * FROM ManagersOfJonathon;
```

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId
4	Johnathon	Schmied	1212121212	**2**	1
2	John	Johnson	2468101214	**1**	1
1	James	Schmied	1234567890	NULL	1

Werte generieren

Die meisten Datenbanken verfügen nicht über eine native Methode zum Generieren einer Reihe von Zahlen für Ad-hoc-Zwecke. Bei der Rekursion können jedoch übliche Tabellenausdrücke verwendet werden, um diesen Funktionstyp zu emulieren.

Im folgenden Beispiel wird ein allgemeiner Tabellenausdruck namens `Numbers` mit einer Spalte `i` generiert, die eine Zeile für die Nummern 1-5 enthält:

```
--Give a table name `Numbers" and a column `i` to hold the numbers
WITH Numbers(i) AS (
    --Starting number/index
    SELECT 1
    --Top-level UNION ALL operator required for recursion
    UNION ALL
    --Iteration expression:
    SELECT i + 1
    --Table expression we first declared used as source for recursion
    FROM Numbers
    --Clause to define the end of the recursion
    WHERE i < 5
)
--Use the generated table expression like a regular table
SELECT i FROM Numbers;
```

ich
1
2
3
4
5

Diese Methode kann mit einem beliebigen Zahlenintervall sowie mit anderen Datentypen verwendet werden.

rekursives Auflisten eines Teilbaums

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
    -- start with this row
    SELECT 1, ID, FName, LName
    FROM Employees
    WHERE ID = 1

    UNION ALL

    -- get employees that have any of the previously selected rows as manager
    SELECT ManagedByJames.Level + 1,
           Employees.ID,
```

```
          Employees.FName,
          Employees.LName
    FROM Employees
    JOIN ManagedByJames
        ON Employees.ManagerID = ManagedByJames.ID

    ORDER BY 1 DESC   -- depth-first search
)
SELECT * FROM ManagedByJames;
```

Niveau	ICH WÜRDE	FName	LName
1	1	James	Schmied
2	2	John	Johnson
3	4	Johnathon	Schmied
2	3	Michael	Williams

Oracle CONNECT BY-Funktionalität mit rekursiven CTEs

Die CONNECT BY-Funktionalität von Oracle bietet viele nützliche und nicht triviale Funktionen, die bei rekursiven SQL-Standard-CTEs nicht integriert sind. In diesem Beispiel werden diese Features (mit einigen zusätzlichen Ergänzungen) unter Verwendung der SQL Server-Syntax repliziert. Es ist für Oracle-Entwickler am nützlichsten, wenn viele Funktionen in ihren hierarchischen Abfragen in anderen Datenbanken fehlen, aber es zeigt auch, was allgemein mit einer hierarchischen Abfrage möglich ist.

```
  WITH tbl AS (
       SELECT id, name, parent_id
         FROM mytable)
     , tbl_hierarchy AS (
       /* Anchor */
       SELECT 1 AS "LEVEL"
            --, 1 AS CONNECT_BY_ISROOT
            --, 0 AS CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , 0 AS CONNECT_BY_ISCYCLE
            , '/' + CAST(t.id   AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
            , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
            , t.id AS root_id
            , t.*
         FROM tbl t
        WHERE t.parent_id IS NULL                              -- START WITH parent_id IS NULL
       UNION ALL
       /* Recursive */
       SELECT th."LEVEL" + 1 AS "LEVEL"
            --, 0 AS CONNECT_BY_ISROOT
            --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) +
```

```
'/%' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
            , th.SYS_CONNECT_BY_PATH_id   + CAST(t.id   AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
            , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_name
            , th.root_id
            , t.*
         FROM tbl t
              JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id =
parent_id
        WHERE th.CONNECT_BY_ISCYCLE = 0)                     -- NOCYCLE
SELECT th.*
     --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
  FROM tbl_hierarchy th
       JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
 ORDER BY th.SYS_CONNECT_BY_PATH_name;                       -- ORDER SIBLINGS BY name
```

CONNECT BY-Funktionen, die oben gezeigt wurden, mit Erklärungen:

- Klauseln
  - CONNECT BY: Gibt die Beziehung an, die die Hierarchie definiert.
  - START WITH: Bestimmt die Wurzelknoten.
  - BESTELLEN VON SIBLINGS BY: Ordnet die Ergebnisse ordnungsgemäß an.
- Parameter
  - NOCYCLE: Stoppt die Verarbeitung einer Verzweigung, wenn eine Schleife erkannt wird. Gültige Hierarchien sind gerichtete azyklische Diagramme, und Zirkelverweise verletzen dieses Konstrukt.
- Operatoren
  - PRIOR: Erhält Daten vom übergeordneten Knoten des Knotens.
  - CONNECT_BY_ROOT: Ermittelt Daten vom Stamm des Knotens.
- Pseudospalten
  - LEVEL: Gibt die Entfernung des Knotens von der Wurzel an.
  - CONNECT_BY_ISLEAF: Gibt einen Knoten ohne untergeordnete Elemente an.
  - CONNECT_BY_ISCYCLE: Gibt einen Knoten mit einer Zirkelreferenz an.
- Funktionen
  - SYS_CONNECT_BY_PATH: Gibt eine abgeflachte / verkettete Darstellung des Pfads zum Knoten vom Stamm aus zurück.

Generieren Sie rekursiv Datumsangaben, die beispielsweise um Teamplanungen erweitert werden

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- Transition Sequence = Rest & Relax into Day Shift into Night Shift
-- RR (Rest & Relax) = 1
-- DS (Day Shift) = 2
-- NS (Night Shift) = 3

;WITH roster AS
(
   SELECT @DateFrom AS RosterStart, 1 AS TeamA, 2 AS TeamB, 3 AS TeamC
```

```
   UNION ALL
   SELECT DATEADD(d, @IntervalDays, RosterStart),
          CASE TeamA WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamA,
          CASE TeamB WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamB,
          CASE TeamC WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamC
   FROM roster WHERE RosterStart < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT RosterStart,
       ISNULL(LEAD(RosterStart) OVER (ORDER BY RosterStart), RosterStart + @IntervalDays) AS
RosterEnd,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

Ergebnis

Dh für Woche 1 ist TeamA auf R & R, TeamB auf Tagschicht und TeamC auf Nachtschicht.

Results | Messages

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

Refactoring einer Abfrage zur Verwendung von Common Table-Ausdrücken

Angenommen, wir möchten alle Produktkategorien mit einem Gesamtumsatz von mehr als 20 erreichen.

Hier ist eine Abfrage ohne Common Table Expressions:

```
SELECT category.description, sum(product.price) as total_sales
FROM sale
LEFT JOIN product on sale.product_id = product.id
LEFT JOIN category on product.category_id = category.id
GROUP BY category.id, category.description
HAVING sum(product.price) > 20
```

Und eine gleichwertige Abfrage mit Common Table Expressions:

```
WITH all_sales AS (
  SELECT product.price, category.id as category_id, category.description as
category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
, sales_by_category AS (
  SELECT category_description, sum(price) as total_sales
  FROM all_sales
```

```
  GROUP BY category_id, category_description
)
SELECT * from sales_by_category WHERE total_sales > 20
```

Beispiel für eine komplexe SQL mit Common Table Expression

Angenommen, wir möchten die "billigsten Produkte" aus den "Top-Kategorien" abfragen.

Hier ist ein Beispiel für eine Abfrage mit Common Table Expressions

```
-- all_sales: just a simple SELECT with all the needed JOINS
WITH all_sales AS (
  SELECT
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
-- Group by category
, sales_by_category AS (
  SELECT category_id, category_description,
  sum(product_price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
-- Filtering total_sales > 20
, top_categories AS (
  SELECT * from sales_by_category WHERE total_sales > 20
)
-- all_products: just a simple SELECT with all the needed JOINS
, all_products AS (
  SELECT
  product.id as product_id,
  product.description as product_description,
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM product
  LEFT JOIN category on product.category_id = category.id
)
-- Order by product price
, cheapest_products AS (
  SELECT * from all_products
  ORDER by product_price ASC
)
-- Simple inner join
, cheapest_products_from_top_categories AS (
  SELECT product_description, product_price
  FROM cheapest_products
  INNER JOIN top_categories ON cheapest_products.category_id = top_categories.category_id
)
--The main SELECT
SELECT * from cheapest_products_from_top_categories
```

Allgemeine Tabellenausdrücke online lesen: https://riptutorial.com/de/sql/topic/747/allgemeine-tabellenausdrucke

Kapitel 4: ALTER TABELLE

Einführung

Der ALTER-Befehl in SQL wird zum Ändern der Spalte / Einschränkung in einer Tabelle verwendet

Syntax

- ALTER TABLE [Tabellenname] ADD [Spaltenname] [Datentyp]

Examples

Spalten hinzufügen

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

Die obige Anweisung fügt Spalten mit dem Namen `StartingDate` die nicht NULL sein können, mit dem Standardwert als aktuellem Datum und `DateOfBirth` die in der Employees- Tabelle NULL sein können.

Drop Column

```
ALTER TABLE Employees
DROP COLUMN salary;
```

Dadurch werden nicht nur Informationen aus dieser Spalte gelöscht, sondern auch das Spaltengehalt der Tabellenmitarbeiter (die Spalte wird nicht mehr vorhanden).

Drop-Einschränkung

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

Dies löscht eine Einschränkung namens DefaultSalary aus der Definition der Mitarbeiter-Tabelle.

Hinweis: - Stellen **Sie** sicher, dass die Einschränkungen der Spalte gelöscht werden, bevor Sie eine Spalte löschen.

Einschränkung hinzufügen

```
ALTER TABLE Employees
```

```
ADD CONSTRAINT DefaultSalary DEFAULT ((100)) FOR [Salary]
```

Dadurch wird eine Einschränkung namens DefaultSalary hinzugefügt, die einen Standardwert von 100 für die Spalte Gehalt enthält.

Eine Einschränkung kann auf Tabellenebene hinzugefügt werden.

Arten von Einschränkungen

- Primärschlüssel - Verhindert einen doppelten Datensatz in der Tabelle
- Fremdschlüssel - verweist auf einen Primärschlüssel aus einer anderen Tabelle
- Nicht Null - verhindert, dass Nullwerte in eine Spalte eingegeben werden
- Eindeutig: Identifiziert jeden Datensatz in der Tabelle eindeutig
- Standard - gibt einen Standardwert an
- Check - begrenzt die Wertebereiche, die in einer Spalte platziert werden können

Weitere Informationen zu Einschränkungen finden Sie in der Oracle-Dokumentation .

Spalte ändern

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

Diese Abfrage ändert den Spaltendatentyp von `StartingDate` und ändert ihn von einem einfachen `date` in ein `datetime` und setzt das aktuelle Datum auf den Standardwert.

Primärschlüssel hinzufügen

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

Dadurch wird der Tabelle Employees in der Feld- `ID` ein Primärschlüssel hinzugefügt. Durch das Einfügen mehrerer Spaltennamen in die Klammern zusammen mit der ID wird ein zusammengesetzter Primärschlüssel erstellt. Wenn Sie mehr als eine Spalte hinzufügen, müssen die Spaltennamen durch Kommas getrennt werden.

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

ALTER TABELLE online lesen: https://riptutorial.com/de/sql/topic/356/alter-tabelle

Kapitel 5: Ansichten

Examples

Einfache ansichten

Eine Ansicht kann einige Zeilen aus der Basistabelle filtern oder nur einige Spalten davon projizieren:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

Wenn Sie die Ansicht auswählen:

```
select * from new_employees_details
```

Ich würde	FName	Gehalt	Anstellungsdatum
4	Johnathon	500	24-07-2016

Komplexe Ansichten

Eine Sicht kann eine sehr komplexe Abfrage sein (Aggregationen, Joins, Unterabfragen usw.). Stellen Sie einfach sicher, dass Sie für alles, was Sie auswählen, Spaltennamen hinzufügen:

```
Create VIEW dept_income AS
SELECT d.Name as DepartmentName, sum(e.salary) as TotalSalary
FROM Employees e
JOIN Departments d on e.DepartmentId = d.id
GROUP BY d.Name;
```

Jetzt können Sie aus jeder Tabelle auswählen:

```
SELECT *
FROM dept_income;
```

Abteilungsname	TotalSalary
HR	1900
Der Umsatz	600

Ansichten online lesen: https://riptutorial.com/de/sql/topic/766/ansichten

Kapitel 6: Ausführungsblöcke

Examples

Verwenden von BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
END
```

Ausführungsblöcke online lesen: https://riptutorial.com/de/sql/topic/1632/ausfuhrungsblocke

Kapitel 7: AUSSER

Bemerkungen

`EXCEPT` gibt alle unterschiedlichen Werte aus der Datenmenge links vom EXCEPT-Operator zurück, die nicht ebenfalls von der rechten Datenmenge zurückgegeben werden.

Examples

Wählen Sie den Datensatz aus, außer wenn sich Werte in diesem anderen Datensatz befinden

```
--dataset schemas must be identical
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column' UNION ALL
SELECT 'Data3' as 'Column' UNION ALL
SELECT 'Data4' as 'Column' UNION ALL
SELECT 'Data5' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
--Returns Data1, Data2, Data4, and Data5
```

AUSSER online lesen: https://riptutorial.com/de/sql/topic/4082/ausser

Kapitel 8: Beispieldatenbanken und Tabellen

Examples

Auto-Shop-Datenbank

Im folgenden Beispiel - Datenbank für ein Autohausgeschäft haben wir eine Liste von Abteilungen, Mitarbeitern, Kunden und Kundenfahrzeugen. Wir verwenden Fremdschlüssel, um Beziehungen zwischen den verschiedenen Tabellen herzustellen.

Live-Beispiel: SQL-Geige

Beziehungen zwischen Tabellen

- Jede Abteilung kann 0 oder mehr Mitarbeiter haben
- Jeder Mitarbeiter kann 0 oder 1 Manager haben
- Jeder Kunde kann 0 oder mehr Autos haben

Abteilungen

Ich würde	Name
1	HR
2	Der Umsatz
3	Technik

SQL-Anweisungen zum Erstellen der Tabelle:

```
CREATE TABLE Departments (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(25) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Departments
    ([Id], [Name])
VALUES
    (1, 'HR'),
    (2, 'Sales'),
    (3, 'Tech')
;
```

Angestellte

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellung
1	James	Schmied	1234567890	NULL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Schmied	1212121212	2	1	500	24-07-2016

SQL-Anweisungen zum Erstellen der Tabelle:

```
CREATE TABLE Employees (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    PhoneNumber VARCHAR(11),
    ManagerId INT,
    DepartmentId INT NOT NULL,
    Salary INT NOT NULL,
    HireDate DATETIME NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
    FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO Employees
    ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
    (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
    (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
    (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
    (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

Kunden

Ich würde	FName	LName	Email	Telefonnummer	PreferredContact
1	Wilhelm	Jones	william.jones@example.com	3347927472	TELEFON
2	David	Müller	dmiller@example.net	2137921892	EMAIL
3	Richard	Davis	richard0123@example.com	NULL	EMAIL

SQL-Anweisungen zum Erstellen der Tabelle:

```
CREATE TABLE Customers (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    Email varchar(100) NOT NULL,
    PhoneNumber VARCHAR(11),
    PreferredContact VARCHAR(5) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Customers
    ([Id], [FName], [LName], [Email], [PhoneNumber], [PreferredContact])
VALUES
    (1, 'William', 'Jones', 'william.jones@example.com', '3347927472', 'PHONE'),
    (2, 'David', 'Miller', 'dmiller@example.net', '2137921892', 'EMAIL'),
    (3, 'Richard', 'Davis', 'richard0123@example.com', NULL, 'EMAIL')
;
```

Autos

Ich würde	Kundennummer	Mitarbeiter-ID	Modell	Status	Gesamtkosten
1	1	2	Ford F-150	BEREIT	230
2	1	2	Ford F-150	BEREIT	200
3	2	1	Ford Mustang	WARTEN	100
4	3	3	Toyota Prius	ARBEITEN	1254

SQL-Anweisungen zum Erstellen der Tabelle:

```
CREATE TABLE Cars (
    Id INT NOT NULL AUTO_INCREMENT,
    CustomerId INT NOT NULL,
    EmployeeId INT NOT NULL,
    Model varchar(50) NOT NULL,
    Status varchar(25) NOT NULL,
    TotalCost INT NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
    FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
    ([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
    ('1', '1', '2', 'Ford F-150', 'READY', '230'),
    ('2', '1', '2', 'Ford F-150', 'READY', '200'),
    ('3', '2', '1', 'Ford Mustang', 'WAITING', '100'),
    ('4', '3', '3', 'Toyota Prius', 'WORKING', '1254')
;
```

Bibliotheksdatenbank

In dieser Beispieldatenbank für eine Bibliothek verfügen wir über die Tabellen *Authors* , *Books* und *BooksAuthors* .

Live-Beispiel: SQL-Geige

Autoren und *Bücher* werden **Basistabellen genannt** , da sie Spaltendefinitionen und Daten für die tatsächlichen Entitäten im relationalen Modell enthalten. *BooksAuthors* ist als **Beziehungstabelle bekannt** , da diese Tabelle die Beziehung zwischen der Tabelle *Books* und *Authors* definiert.

Beziehungen zwischen Tabellen

- Jeder Autor kann ein oder mehrere Bücher haben
- Jedes Buch kann einen oder mehrere Autoren haben

Autoren

(*Tabelle ansehen*)

Ich würde	Name	Land
1	JD Salinger	Vereinigte Staaten von Amerika
2	F. Scott. Fitzgerald	Vereinigte Staaten von Amerika
3	Jane Austen	Vereinigtes Königreich
4	Scott Hanselman	Vereinigte Staaten von Amerika
5	Jason N. Gaylord	Vereinigte Staaten von Amerika
6	Pranav Rastogi	Indien
7	Todd Miranda	Vereinigte Staaten von Amerika
8	Christian Wenz	Vereinigte Staaten von Amerika

SQL zum Erstellen der Tabelle:

```
CREATE TABLE Authors (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(70) NOT NULL,
    Country VARCHAR(100) NOT NULL,
    PRIMARY KEY(Id)
);
```

```
INSERT INTO Authors
    (Name, Country)
VALUES
    ('J.D. Salinger', 'USA'),
    ('F. Scott. Fitzgerald', 'USA'),
    ('Jane Austen', 'UK'),
    ('Scott Hanselman', 'USA'),
    ('Jason N. Gaylord', 'USA'),
    ('Pranav Rastogi', 'India'),
    ('Todd Miranda', 'USA'),
    ('Christian Wenz', 'USA')
;
```

Bücher

(*Tabelle ansehen*)

Ich würde	Titel
1	Der Fänger im Roggen
2	Neun Geschichten
3	Franny und Zooey
4	Der große Gatsby
5	Ausschreibung der Nacht
6	Stolz und Vorurteil
7	Professionelles ASP.NET 4.5 in C # und VB

SQL zum Erstellen der Tabelle:

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
    (Id, Title)
VALUES
    (1, 'The Catcher in the Rye'),
    (2, 'Nine Stories'),
    (3, 'Franny and Zooey'),
    (4, 'The Great Gatsby'),
    (5, 'Tender id the Night'),
    (6, 'Pride and Prejudice'),
    (7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

BücherAuthors

(*Tabelle ansehen*)

BookId	AuthorId
1	1
2	1
3	1
4	2
5	2
6	3
7	4
7	5
7	6
7	7
7	8

SQL zum Erstellen der Tabelle:

```
CREATE TABLE BooksAuthors (
    AuthorId INT NOT NULL,
    BookId  INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES Authors(Id),
    FOREIGN KEY (BookId) REFERENCES Books(Id)
);

INSERT INTO BooksAuthors
    (BookId, AuthorId)
VALUES
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 2),
    (5, 2),
    (6, 3),
    (7, 4),
    (7, 5),
    (7, 6),
    (7, 7),
    (7, 8)
;
```

Beispiele

Alle Autoren anzeigen (Live-Beispiel anzeigen):

```
SELECT * FROM Authors;
```

Alle Buchtitel anzeigen (Live-Beispiel anzeigen):

```
SELECT * FROM Books;
```

Alle Bücher und ihre Autoren anzeigen (Live-Beispiel anzeigen):

```
SELECT
  ba.AuthorId,
  a.Name AuthorName,
  ba.BookId,
  b.Title BookTitle
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
  INNER JOIN Books b ON b.id = ba.bookid
;
```

Ländertabelle

In diesem Beispiel haben wir eine Tabelle **Länder** . Eine Tabelle für Länder ist vielseitig verwendbar, insbesondere in Finanzanwendungen, die Währungen und Wechselkurse beinhalten.

Live-Beispiel: SQL-Geige

Bei einigen Marktdaten-Softwareanwendungen wie Bloomberg und Reuters müssen Sie der API entweder einen 2- oder 3-stelligen Ländercode zusammen mit dem Währungscode angeben. Daher enthält diese Beispieltabelle sowohl die 2- `ISO3 ISO ISO3` als auch die 3- `ISO3` .

Länder

(*Tabelle ansehen*)

Ich würde	ISO	ISO3	ISONumeric	Ländername	Hauptstadt	ContinentCode	Währungs
1	AU	AUS	36	Australien	Canberra	OC	AUD
2	DE	DEU	276	Deutschland	Berlin	EU	EUR
2	IM	IND	356	Indien	Neu-Delhi	WIE	INR
3	LA	LAO	418	Laos	Vientiane	WIE	Lak

Ich würde	ISO	ISO3	ISONumeric	Ländername	Hauptstadt	ContinentCode	Währungs
4	UNS	Vereinigte Staaten von Amerika	840	Vereinigte Staaten	Washington	N / A	USD
5	ZW	ZWE	716	Zimbabwe	Harare	AF	ZWL

SQL zum Erstellen der Tabelle:

```
CREATE TABLE Countries (
    Id INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    ISONumeric INT NOT NULL,
    CountryName VARCHAR(64) NOT NULL,
    Capital VARCHAR(64) NOT NULL,
    ContinentCode VARCHAR(2) NOT NULL,
    CurrencyCode VARCHAR(3) NOT NULL,
    PRIMARY KEY(Id)
)
;

INSERT INTO Countries
    (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, 'Australia', 'Canberra', 'OC', 'AUD'),
    ('DE', 'DEU', 276, 'Germany', 'Berlin', 'EU', 'EUR'),
    ('IN', 'IND', 356, 'India', 'New Delhi', 'AS', 'INR'),
    ('LA', 'LAO', 418, 'Laos', 'Vientiane', 'AS', 'LAK'),
    ('US', 'USA', 840, 'United States', 'Washington', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, 'Zimbabwe', 'Harare', 'AF', 'ZWL')
;
```

Beispieldatenbanken und Tabellen online lesen:
https://riptutorial.com/de/sql/topic/280/beispieldatenbanken-und-tabellen

Kapitel 9: BEITRETEN

Einführung

JOIN ist eine Methode zum Kombinieren (Verbinden) von Informationen aus zwei Tabellen. Das Ergebnis ist ein zusammengesetzter Satz von Spalten aus beiden Tabellen, der durch den Join-Typ (INNER / OUTER / CROSS und LEFT / RIGHT / FULL, siehe unten) definiert wird, und Join-Kriterien (wie sich die Zeilen aus beiden Tabellen beziehen).

Eine Tabelle kann mit sich selbst oder einer anderen Tabelle verbunden sein. Wenn auf Informationen aus mehr als zwei Tabellen zugegriffen werden muss, können in einer FROM-Klausel mehrere Joins angegeben werden.

Syntax

- `[ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } ] JOIN`

Bemerkungen

Joins sind, wie der Name schon sagt, eine Möglichkeit, Daten aus mehreren Tabellen gemeinsam abzufragen, wobei die Zeilen Spalten aus mehreren Tabellen anzeigen.

Examples

Grundlegende explizite innere Verknüpfung

Ein Basis-Join (auch "Inner Join" genannt) fragt Daten aus zwei Tabellen ab, deren Beziehung in einer `join` Klausel definiert ist.

Im folgenden Beispiel werden die Vornamen der Mitarbeiter (FName) aus der Tabelle Employees und der Name der Abteilung, für die sie arbeiten (Name), aus der Tabelle Departments ausgewählt:

```
SELECT Employees.FName, Departments.Name
FROM   Employees
JOIN   Departments
ON Employees.DepartmentId = Departments.Id
```

Dies würde Folgendes aus der Beispieldatenbank zurückgeben :

Mitarbeiter.FName	Abteilungen.Name
James	HR
John	HR

Mitarbeiter.FName	Abteilungen.Name
Richard	Der Umsatz

Implizite Verknüpfung

Joins können auch ausgeführt werden, indem mehrere Tabellen in der `from` Klausel mit Kommas getrennt `,` und die Beziehung zwischen ihnen in der `where` Klausel definiert wird. Diese Technik wird als impliziter Join bezeichnet (da sie eigentlich keine `join` Klausel enthält).

Alle RDBMSs unterstützen dies, von der Syntax wird jedoch normalerweise abgeraten. Die Gründe, warum es eine schlechte Idee ist, diese Syntax zu verwenden, sind:

- Es ist möglich, versehentliche Cross-Joins zu erhalten, die dann falsche Ergebnisse liefern, insbesondere wenn Sie viele Joins in der Abfrage haben.
- Wenn Sie eine Kreuzverknüpfung beabsichtigen, ist dies aus der Syntax nicht ersichtlich (schreiben Sie stattdessen CROSS JOIN aus), und jemand wird sie wahrscheinlich während der Wartung ändern.

Im folgenden Beispiel werden die Vornamen der Mitarbeiter und der Name der Abteilungen ausgewählt, für die sie tätig sind:

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DeptartmentId = d.Id
```

Dies würde Folgendes aus der Beispieldatenbank zurückgeben :

e.FName	d.Name
James	HR
John	HR
Richard	Der Umsatz

Linke äußere Verbindung

Ein Left Outer Join (auch als Left Join oder Outer Join bezeichnet) ist ein Join, der sicherstellt, dass alle Zeilen der linken Tabelle dargestellt werden. Wenn keine übereinstimmende Zeile aus der rechten Tabelle vorhanden ist, sind die entsprechenden Felder `NULL` .

Im folgenden Beispiel werden alle Abteilungen und der Vorname der Mitarbeiter ausgewählt, die in dieser Abteilung arbeiten. Abteilungen ohne Mitarbeiter werden zwar in den Ergebnissen zurückgegeben, haben jedoch NULL für den Mitarbeiternamen:

```
SELECT          Departments.Name, Employees.FName
FROM            Departments
```

```
LEFT OUTER JOIN Employees
ON              Departments.Id = Employees.DepartmentId
```

Dies würde Folgendes aus der Beispieldatenbank zurückgeben :

Abteilungen.Name	Mitarbeiter.FName
HR	James
HR	John
HR	Johnathon
Der Umsatz	Michael
Technik	NULL

Wie funktioniert das?

Die FROM-Klausel enthält zwei Tabellen:

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellung
1	James	Schmied	1234567890	NULL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Schmied	1212121212	2	1	500	24-07-2016

und

Ich würde	Name
1	HR
2	Der Umsatz
3	Technik

Zunächst wird ein *kartesisches* Produkt aus den beiden Tabellen erstellt, die eine Zwischentabelle enthalten.
Die Datensätze, die die Join-Kriterien erfüllen (*Departments.Id = Employees.DepartmentId*), sind fett hervorgehoben. Diese werden an die nächste Stufe der Abfrage übergeben.

Da dies ein LEFT-OUTER-JOIN ist, werden alle Datensätze von der LEFT-Seite des Joins (Abteilungen) zurückgegeben, während alle Datensätze auf der RECHTEN Seite eine NULL-Markierung erhalten, wenn sie nicht den Join-Kriterien entsprechen. In der folgenden Tabelle wird **Tech** mit `NULL`

Ich würde	Name	Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId
1	**HR**	**1**	**James**	**Schmied**	**1234567890**	**NULL**	**1**
1	**HR**	**2**	**John**	**Johnson**	**2468101214**	**1**	**1**
1	HR	3	Michael	Williams	1357911131	1	2
1	**HR**	**4**	**Johnathon**	**Schmied**	**1212121212**	**2**	**1**
2	Der Umsatz	1	James	Schmied	1234567890	NULL	1
2	Der Umsatz	2	John	Johnson	2468101214	1	1
2	**Der Umsatz**	**3**	**Michael**	**Williams**	**1357911131**	**1**	**2**
2	Der Umsatz	4	Johnathon	Schmied	1212121212	2	1
3	Technik	1	James	Schmied	1234567890	NULL	1
3	Technik	2	John	Johnson	2468101214	1	1
3	Technik	3	Michael	Williams	1357911131	1	2
3	Technik	4	Johnathon	Schmied	1212121212	2	1

Schließlich wird jeder Ausdruck, der in der **SELECT-** Klausel verwendet wird, ausgewertet, um unsere letzte Tabelle zurückzugeben:

Abteilungen.Name	Mitarbeiter.FName
HR	James
HR	John
Der Umsatz	Richard
Technik	NULL

Selbst beitreten

Eine Tabelle kann mit sich selbst verbunden sein, wobei verschiedene Zeilen durch eine Bedingung miteinander übereinstimmen. In diesem Anwendungsfall müssen Aliase verwendet werden, um die beiden Vorkommen der Tabelle zu unterscheiden.

Im folgenden Beispiel wird für jeden Mitarbeiter in der Tabelle Employees der Beispieldatenbank ein Datensatz zurückgegeben, der den Vornamen des Mitarbeiters sowie den entsprechenden Vornamen des Vorgesetzten des Mitarbeiters enthält. Da Manager auch Angestellte sind, ist der Tisch mit sich selbst verbunden:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
    Employees e
JOIN
    Employees m
    ON e.ManagerId = m.Id
```

Diese Abfrage gibt die folgenden Daten zurück:

Mitarbeiter	Manager
John	James
Michael	James
Johnathon	John

Wie funktioniert das?

Die Originaltabelle enthält diese Datensätze:

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellung
1	James	Schmied	1234567890	NULL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Schmied	1212121212	2	1	500	24-07-2016

Die erste Aktion besteht darin, ein *kartesisches* Produkt aus allen Datensätzen in den in der **FROM-** Klausel verwendeten Tabellen zu erstellen. In diesem Fall handelt es sich zweimal um die

Employees-Tabelle, daher sieht die Zwischentabelle folgendermaßen aus (ich habe alle in diesem Beispiel nicht verwendeten Felder entfernt):

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
1	James	NULL	1	James	NULL
1	James	NULL	2	John	1
1	James	NULL	3	Michael	1
1	James	NULL	4	Johnathon	2
2	John	1	1	James	NULL
2	John	1	2	John	1
2	John	1	3	Michael	1
2	John	1	4	Johnathon	2
3	Michael	1	1	James	NULL
3	Michael	1	2	John	1
3	Michael	1	3	Michael	1
3	Michael	1	4	Johnathon	2
4	Johnathon	2	1	James	NULL
4	Johnathon	2	2	John	1
4	Johnathon	2	3	Michael	1
4	Johnathon	2	4	Johnathon	2

Die nächste Aktion ist es , nur die Aufzeichnungen, die die **JOIN** Kriterien erfüllen, so dass alle Datensätze , in denen die aliased `e` Tabelle `ManagerId` die aliased gleich `m` Tabelle `Id` :

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	John	1	1	James	NULL
3	Michael	1	1	James	NULL
4	Johnathon	2	2	John	1

Dann wird jeder in der **SELECT-** Klausel verwendete Ausdruck ausgewertet, um diese Tabelle

zurückzugeben:

e.FName	m.FName
John	James
Michael	James
Johnathon	John

Schließlich werden die Spaltennamen `e.FName` und `m.FName` durch ihre Alias-Spaltennamen ersetzt, die dem AS- Operator zugeordnet sind:

Mitarbeiter	Manager
John	James
Michael	James
Johnathon	John

CROSS JOIN

Cross Joining führt ein kartesisches Produkt der beiden Elemente aus. Ein kartesisches Produkt bedeutet, dass jede Zeile einer Tabelle mit jeder Zeile der zweiten Tabelle im Join kombiniert wird. Wenn `TABLEA` beispielsweise 20 Zeilen und `TABLEB` 20 Zeilen hat, wäre das Ergebnis `20*20 = 400` Ausgabezeilen.

Beispieldatenbank verwenden

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

Welche gibt zurück:

d.Name	e.FName
HR	James
HR	John
HR	Michael
HR	Johnathon
Der Umsatz	James
Der Umsatz	John

d.Name	e.FName
Der Umsatz	Michael
Der Umsatz	Johnathon
Technik	James
Technik	John
Technik	Michael
Technik	Johnathon

Es wird empfohlen, einen expliziten CROSS JOIN zu schreiben, wenn Sie einen kartesischen Join durchführen möchten, um hervorzuheben, dass dies der Fall ist.

An einer Unterabfrage teilnehmen

Das Verknüpfen einer Unterabfrage wird häufig verwendet, wenn Sie aggregierte Daten aus einer untergeordneten / Detailtabelle abrufen und diese zusammen mit Datensätzen aus der übergeordneten / Header-Tabelle anzeigen möchten. Sie möchten beispielsweise eine Anzahl von untergeordneten Datensätzen, einen Durchschnitt einer numerischen Spalte in untergeordneten Datensätzen oder die obere oder untere Zeile basierend auf einem Datums- oder numerischen Feld abrufen. In diesem Beispiel werden Aliase verwendet, die die Lesbarkeit von Abfragen erleichtern, wenn mehrere Tabellen beteiligt sind. So sieht eine recht typische Unterabfrage aus. In diesem Fall rufen wir alle Zeilen aus den Bestellungen der übergeordneten Tabelle und nur die erste Zeile für jeden übergeordneten Datensatz der untergeordneten Tabelle PurchaseOrderLineItems ab.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
       FROM PurchaseOrderLineItems l
       GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

CROSS APPLY & LATERAL JOIN

Eine sehr interessante Art von JOIN ist der LATERAL JOIN (neu in PostgreSQL 9.3+).
Dies wird in SQL-Server & Oracle auch als CROSS APPLY / OUTER APPLY bezeichnet.

Die Grundidee ist, dass eine Tabellenwertfunktion (oder Inline-Unterabfrage) für jede Zeile angewendet wird, der Sie beitreten.

Dadurch ist es beispielsweise möglich, nur den ersten passenden Eintrag in einer anderen Tabelle zu verknüpfen.

Der Unterschied zwischen einem normalen und einem seitlichen Join besteht darin, dass Sie eine Spalte verwenden können, die Sie zuvor **in der Unterabfrage** "CROSS APPLY" hinzugefügt haben.

Syntax:

PostgreSQL 9.3 und höher

> links | richtig | inneres **VERBINDEN LATERAL**

SQL Server:

> CROSS | Äußere **Anwendung**

`INNER JOIN LATERAL` ist das gleiche wie `CROSS APPLY`
und `LEFT JOIN LATERAL` ist das Gleiche wie `OUTER APPLY`

Verwendungsbeispiel (PostgreSQL 9.3+):

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989


LEFT JOIN LATERAL
(
    SELECT
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
   LIMIT 1
) AS FirstOE
```

Und für SQL-Server

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
```

```
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = INNER JOIN
OUTER APPLY    -- = LEFT JOIN
(
    SELECT TOP 1
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

FULL JOIN

Ein weniger bekannter JOIN-Typ ist der FULL JOIN.
(Hinweis: FULL JOIN wird ab 2016 nicht von MySQL unterstützt.)

Ein FULL OUTER JOIN gibt alle Zeilen der linken Tabelle und alle Zeilen der rechten Tabelle zurück.

Wenn in der linken Tabelle Zeilen vorhanden sind, für die in der rechten Tabelle keine Übereinstimmungen vorhanden sind, oder wenn in der rechten Tabelle Zeilen vorhanden sind, die in der linken Tabelle keine Übereinstimmungen aufweisen, werden diese Zeilen ebenfalls aufgelistet.

Beispiel 1 :

```
SELECT * FROM Table1

FULL JOIN Table2
     ON 1 = 2
```

Beispiel 2

```
SELECT
     COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
FROM T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
      ON tYear.Year = T_Budget.Year
```

Wenn Sie Soft-Deletes verwenden, müssen Sie den Soft-Delete-Status erneut in der WHERE-Klausel überprüfen (da FULL JOIN sich wie eine UNION verhält).
Es ist leicht, diese kleine Tatsache zu übersehen, da Sie AP_SoftDeleteStatus = 1 in die Join-Klausel einfügen.

Wenn Sie einen FULL JOIN ausführen, müssen Sie normalerweise NULL in der WHERE-Klausel zulassen. Wenn Sie vergessen, NULL für einen Wert zuzulassen, hat dies die gleichen Auswirkungen wie bei einem INNER-Join. Dies ist etwas, was Sie nicht möchten, wenn Sie einen FULL JOIN ausführen.

Beispiel:

```
SELECT
     T_AccountPlan.AP_UID
    ,T_AccountPlan.AP_Code
    ,T_AccountPlan.AP_Lang_EN
    ,T_BudgetPositions.BUP_Budget
    ,T_BudgetPositions.BUP_UID
    ,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
    ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
    AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS
NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

Rekursive JOINs

Rekursive Joins werden häufig verwendet, um Eltern-Kind-Daten zu erhalten. In SQL werden sie mit rekursiven allgemeinen Tabellenausdrücken implementiert, zum Beispiel:

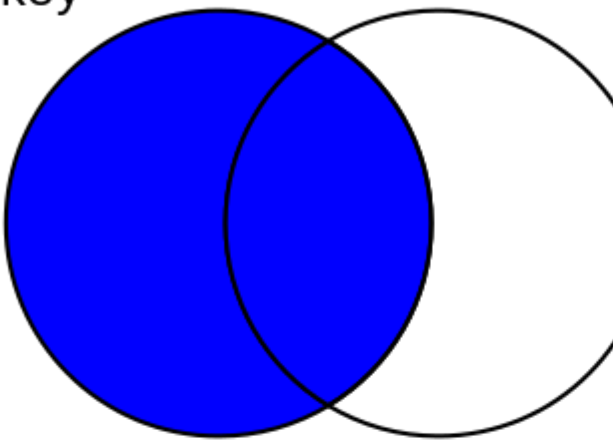
```
WITH RECURSIVE MyDescendants AS (
    SELECT Name
    FROM People
    WHERE Name = 'John Doe'

    UNION ALL

    SELECT People.Name
    FROM People
    JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

Unterschiede zwischen inneren und äußeren Verbindungen

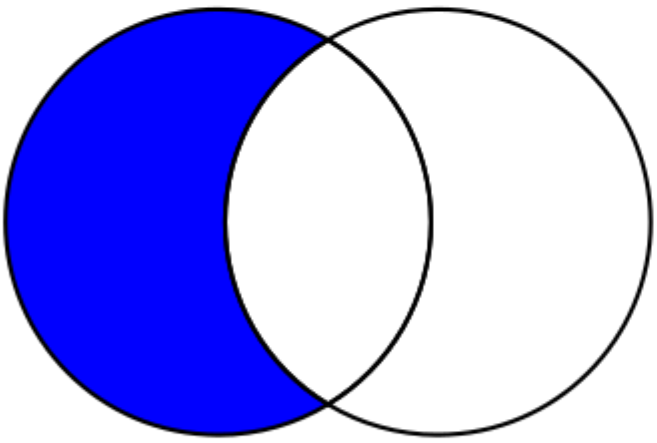
SQL bietet verschiedene Join-Typen, um anzugeben, ob (nicht) übereinstimmende Zeilen im Ergebnis enthalten sind: `INNER JOIN` , `LEFT OUTER JOIN` , `RIGHT OUTER JOIN` und `FULL OUTER JOIN` (die Schlüsselwörter `INNER` und `OUTER` sind optional). Die folgende Abbildung unterstreicht die

Unterschiede zwischen diesen Arten von Joins: Der blaue Bereich stellt die Ergebnisse dar, die vom Join zurückgegeben werden, und der weiße Bereich repräsentiert die Ergebnisse, die der Join nicht zurückgibt.

```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
```



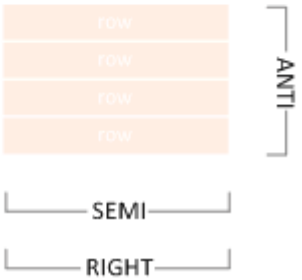
```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
WHERE B.key IS NULL
```



Seien Sie vorsichtig, wenn Sie NOT IN in einer NULL-fähigen Spalte verwenden! Weitere Details hier .

Right Anti Semi Join

Enthält rechte Zeilen, die **nicht** mit den linken Zeilen übereinstimmen.




```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);

Y
-------
Tim
Vincent
```

Wie Sie sehen, gibt es keine dedizierte NOT-IN-Syntax für einen Links-gegen-Rechts-Anti-Semi-Join - wir erzielen den Effekt einfach durch das Wechseln der Tabellenpositionen im SQL-Text.

Cross Join

Ein kartesisches Produkt mit allen rechten Reihen.

```
SELECT * FROM A CROSS JOIN B;

X      Y
```

```
-----  -------
Amy    Lisa
John   Lisa
Lisa   Lisa
Marco  Lisa
Phil   Lisa
Amy    Marco
John   Marco
Lisa   Marco
Marco  Marco
Phil   Marco
Amy    Phil
John   Phil
Lisa   Phil
Marco  Phil
Phil   Phil
Amy    Tim
John   Tim
Lisa   Tim
Marco  Tim
Phil   Tim
Amy    Vincent
John   Vincent
Lisa   Vincent
Marco  Vincent
Phil   Vincent
```

Cross Join ist gleichbedeutend mit einem inneren Join mit Join-Bedingung, die immer übereinstimmt. Daher hätte die folgende Abfrage dasselbe Ergebnis zurückgegeben:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Self-Join

Dies bezeichnet einfach eine Tabelle, die sich mit sich selbst verbindet. Ein Self-Join kann jeder der oben diskutierten Join-Typen sein. Dies ist zum Beispiel eine innere Selbstverbindung:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);

X     X
----  -----
Amy   John
Amy   Lisa
Amy   Marco
John  Marco
Lisa  Marco
Phil  Marco
Amy   Phil
```

BEITRETEN online lesen: https://riptutorial.com/de/sql/topic/261/beitreten

Kapitel 10: Bemerkungen

Examples

Einzeilige Kommentare

Einzeiligen Kommentaren wird mit `--` vorangestellt und gehen bis zum Zeilenende:

```
SELECT *
FROM Employees -- this is a comment
WHERE FName = 'John'
```

Mehrzeilige Kommentare

Mehrzeilige Code-Kommentare werden in `/* ... */` :

```
/* This query
   returns all employees */
SELECT *
FROM Employees
```

Es ist auch möglich, einen solchen Kommentar in die Mitte einer Zeile einzufügen:

```
SELECT /* all columns: */ *
FROM Employees
```

Bemerkungen online lesen: https://riptutorial.com/de/sql/topic/1597/bemerkungen

Kapitel 11: Bereinigen Sie Code in SQL

Einführung

So schreiben Sie gute, lesbare SQL-Abfragen und Beispiele für bewährte Verfahren.

Examples

Formatierung und Schreibweise von Schlüsselwörtern und Namen

Tabellen- / Spaltennamen

Zwei gängige Methoden zum Formatieren von Tabellen- / Spaltennamen sind `CamelCase` und `snake_case` :

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;
```

```
SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

Namen sollten beschreiben, was in ihrem Objekt gespeichert ist. Dies bedeutet, dass die Spaltennamen normalerweise singulär sein sollten. Ob Tabellennamen Singular oder Plural verwenden sollen, ist eine viel diskutierte Frage. In der Praxis werden jedoch häufig Tabellennamen verwendet.

Das Hinzufügen von Präfixen oder Suffixen wie `tbl` oder `col` verringert die Lesbarkeit. Vermeiden Sie sie. Sie werden jedoch manchmal verwendet, um Konflikte mit SQL-Schlüsselwörtern zu vermeiden, und werden häufig mit Triggern und Indizes (deren Namen in Abfragen normalerweise nicht erwähnt werden) verwendet.

Schlüsselwörter

SQL-Schlüsselwörter unterscheiden nicht zwischen Groß- und Kleinschreibung. Es ist jedoch üblich, sie in Großbuchstaben zu schreiben.

WÄHLEN *

`SELECT *` gibt alle Spalten in der Reihenfolge zurück, in der sie in der Tabelle definiert sind.

Bei Verwendung von `SELECT *` können sich die von einer Abfrage zurückgegebenen Daten ändern,

wenn sich die Tabellendefinition ändert. Dies erhöht das Risiko, dass verschiedene Versionen Ihrer Anwendung oder Ihrer Datenbank nicht miteinander kompatibel sind.

Wenn Sie mehr Spalten als nötig lesen, kann dies die Anzahl der Festplatten- und Netzwerk-E / A erhöhen.

Daher sollten Sie immer explizit die Spalten angeben, die Sie tatsächlich abrufen möchten:

```
--SELECT *                                  don't
  SELECT ID, FName, LName, PhoneNumber  -- do
  FROM Emplopees;
```

(Bei interaktiven Abfragen gelten diese Überlegungen nicht.)

`SELECT *` schadet jedoch nicht in der Unterabfrage eines EXISTS-Operators, da EXISTS die tatsächlichen Daten trotzdem ignoriert (es wird nur geprüft, ob mindestens eine Zeile gefunden wurde). Aus demselben Grund ist es nicht sinnvoll, eine bestimmte Spalte für EXISTS aufzulisten. Daher ist `SELECT *` eigentlich sinnvoller:

```
-- list departments where nobody was hired recently
SELECT ID,
       Name
FROM Departments
WHERE NOT EXISTS (SELECT *
                  FROM Employees
                  WHERE DepartmentID = Departments.ID
                    AND HireDate >= '2015-01-01');
```

Einrücken

Es gibt keinen allgemein akzeptierten Standard. Alle sind sich einig, dass es schlecht ist, alles in eine einzige Zeile zu quetschen:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID =
e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

Setzen Sie mindestens jede Klausel in eine neue Zeile und teilen Sie die Zeilen auf, wenn sie sonst zu lang werden würden:

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM Departments AS d
JOIN Employees AS e ON d.ID = e.DepartmentID
WHERE d.Name != 'HR'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

Manchmal wird alles nach dem SQL-Schlüsselwort, das eine Klausel einführt, in dieselbe Spalte eingerückt:

```
SELECT   d.Name,
         COUNT(*) AS Employees
FROM     Departments AS d
JOIN     Employees AS e ON d.ID = e.DepartmentID
WHERE    d.Name != 'HR'
HAVING   COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

(Dies ist auch möglich, wenn Sie die SQL-Schlüsselwörter richtig ausrichten.)

Ein anderer verbreiteter Stil besteht darin, wichtige Schlüsselwörter in die eigenen Zeilen zu setzen:

```
SELECT
    d.Name,
    COUNT(*) AS Employees
FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID
WHERE
    d.Name != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

Durch das vertikale Ausrichten mehrerer ähnlicher Ausdrücke wird die Lesbarkeit verbessert:

```
SELECT Model,
       EmployeeID
FROM Cars
WHERE CustomerID = 42
  AND Status     = 'READY';
```

Die Verwendung mehrerer Zeilen erschwert das Einbetten von SQL-Befehlen in andere Programmiersprachen. Viele Sprachen haben jedoch einen Mechanismus für mehrzeilige Zeichenfolgen, z. B. `@"..."` in C #, `"""..."""` in Python oder `R"(...)"` in C ++.

Schließt sich an

Explizite Joins sollten immer verwendet werden. implizite Joins haben mehrere Probleme:

- Die Join-Bedingung befindet sich irgendwo in der WHERE-Klausel, gemischt mit allen anderen Filterbedingungen. Dies macht es schwieriger zu erkennen, welche Tabellen wie miteinander verbunden werden.
- Aus diesen Gründen besteht ein höheres Fehlerrisiko, und es ist wahrscheinlicher, dass sie später gefunden werden.
- In Standard-SQL sind explizite Joins die einzige Möglichkeit, äußere Joins zu verwenden :

```
SELECT d.Name,
       e.Fname || e.LName AS EmpName
FROM      Departments AS d
LEFT JOIN Employees   AS e ON d.ID = e.DepartmentID;
```

- Explizite Joins erlauben die Verwendung der USING-Klausel:

```
SELECT RecipeID,
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM      Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(Dies erfordert, dass beide Tabellen denselben Spaltennamen verwenden.
USING entfernt automatisch die doppelte Spalte aus dem Ergebnis, z. B. gibt der Join in dieser Abfrage eine einzelne `RecipeID` Spalte zurück.)

Bereinigen Sie Code in SQL online lesen: https://riptutorial.com/de/sql/topic/9843/bereinigen-sie-code-in-sql

Kapitel 12: Cascading Delete

Examples

ON DELETE CASCADE

Angenommen, Sie haben eine Anwendung, die Räume verwaltet.
Nehmen Sie weiter an, dass Ihre Anwendung pro Mandant (Mandant) ausgeführt wird.
Sie haben mehrere Kunden.
Ihre Datenbank enthält also eine Tabelle für Kunden und eine für Räume.

Nun hat jeder Kunde N Räume.

Dies bedeutet, dass Sie einen Fremdschlüssel in Ihrer Zimmertabelle haben, der auf die Client-Tabelle verweist.

```
ALTER TABLE dbo.T_Room  WITH CHECK ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

Vorausgesetzt, ein Client wechselt zu einer anderen Software, müssen Sie seine Daten in Ihrer Software löschen. Aber wenn du es tust

```
DELETE FROM T_Client WHERE CLI_ID = x
```

Sie erhalten dann eine Verletzung des Fremdschlüssels, da Sie den Client nicht löschen können, wenn er noch über Räume verfügt.

Jetzt haben Sie Code in Ihre Anwendung geschrieben, der die Räume des Kunden löscht, bevor der Client gelöscht wird. Nehmen Sie weiterhin an, dass in der Datenbank in Zukunft viele weitere Fremdschlüsselabhängigkeiten hinzugefügt werden, da die Funktionalität Ihrer Anwendung erweitert wird. Schrecklich. Für jede Änderung in Ihrer Datenbank müssen Sie den Code Ihrer Anwendung in N Stellen anpassen. Möglicherweise müssen Sie den Code auch in anderen Anwendungen anpassen (z. B. Schnittstellen zu anderen Systemen).

Es gibt eine bessere Lösung als im Code.
Sie können `ON DELETE CASCADE` einfach zu Ihrem Fremdschlüssel hinzufügen.

```
ALTER TABLE dbo.T_Room  -- WITH CHECK -- SQL-Server can specify WITH CHECK/WITH NOCHECK
ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

Jetzt kannst du sagen

```
DELETE FROM T_Client WHERE CLI_ID = x
```

und die Räume werden automatisch gelöscht, wenn der Client gelöscht wird.
Problem gelöst - ohne Änderung des Anwendungscodes.

Ein Wort zur Vorsicht: In Microsoft SQL-Server funktioniert dies nicht, wenn Sie eine Tabelle haben, die auf sich selbst verweist. Wenn Sie also versuchen, eine Löschkaskade in einer rekursiven Baumstruktur zu definieren, gehen Sie wie folgt vor:

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation]  WITH CHECK ADD  CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
GO

IF  EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

Es funktioniert nicht, da der Microsoft-SQL-Server es nicht zulässt, einen Fremdschlüssel mit `ON DELETE CASCADE` in einer rekursiven Baumstruktur festzulegen. Ein Grund dafür ist, dass der Baum möglicherweise zyklisch ist und dies möglicherweise zu einem Deadlock führt.

PostgreSQL dagegen kann dies tun;
Voraussetzung ist, dass der Baum nicht zyklisch ist.
Wenn der Baum zyklisch ist, wird ein Laufzeitfehler angezeigt.
In diesem Fall müssen Sie die Löschfunktion nur selbst implementieren.

Ein Wort der Warnung:
Das bedeutet, dass Sie die Client-Tabelle nicht mehr einfach löschen und erneut einfügen können. Wenn Sie dies tun, werden alle Einträge in "T_Room" gelöscht. ...

Cascading Delete online lesen: https://riptutorial.com/de/sql/topic/3518/cascading-delete

Kapitel 13: Datenbank erstellen

Syntax

- CREATE DATABASE Datenbankname;

Examples

Datenbank erstellen

Eine Datenbank wird mit dem folgenden SQL-Befehl erstellt:

```
CREATE DATABASE myDatabase;
```

Dadurch würde eine leere Datenbank namens myDatabase erstellt, in der Sie Tabellen erstellen können.

Datenbank erstellen online lesen: https://riptutorial.com/de/sql/topic/2744/datenbank-erstellen

Kapitel 14: Datentypen

Examples

DECIMAL und NUMERIC

Feste Genauigkeit und Dezimalzahlen. `DECIMAL` und `NUMERIC` sind funktional gleichwertig.

Syntax:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Beispiele:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

FLOAT und REAL

Ungefähre-Anzahl-Datentypen zur Verwendung mit numerischen Gleitkommadaten.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Ganzzahlen

Datentypen mit genauen Zahlen, die Ganzzahldaten verwenden.

Datentyp	Angebot	Lager
Bigint	-2 ^ 63 (-9.223.372.036.854.775,808) bis 2 ^ 63-1 (9.223.372.036.854.775.807)	8 Bytes
int	-2 ^ 31 (-2.147.483.648) bis 2 ^ 31-1 (2.147.483.647)	4 Bytes
smallint	-2 ^ 15 (-32.768) bis 2 ^ 15-1 (32.767)	2 Bytes
Winzige	0 bis 255	1 Byte

Geld und Kleinigkeiten

Datentypen, die Währungs- oder Währungswerte darstellen.

Datentyp	Angebot	Lager
Geld	-922.337.203,685,477,5808 bis 922,337,203,685,477,5807	8 Bytes
kleines geld	-214,748,3648 bis 214,748,3647	4 Bytes

BINARY und VARBINARY

Binäre Datentypen mit fester oder variabler Länge.

Syntax:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

`n_bytes` können eine beliebige Anzahl von 1 bis 8000 Bytes sein. `max` gibt an, dass der maximale Speicherplatz 2 ^ 31-1 beträgt.

Beispiele:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x0000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

CHAR und VARCHAR

String-Datentypen mit fester oder variabler Länge.

Syntax:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Beispiele:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC       ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10))  -- 'ABCDEFGHIJ' (truncated to 10
characters)
```

NCHAR und NVARCHAR

UNICODE-String-Datentypen mit fester oder variabler Länge.

Syntax:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Verwenden Sie `MAX` für sehr lange Zeichenfolgen mit mehr als 8000 Zeichen.

EINDEUTIGE KENNUNG

Eine 16-Byte-GUID / UUID.

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string,   -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```

Datentypen online lesen: https://riptutorial.com/de/sql/topic/1166/datentypen

Kapitel 15: DROP oder DELETE Database

Syntax

- MSSQL-Syntax:
- DROP DATABASE [IF EXISTS] {Datenbankname | database_snapshot_name} [, ... n] [;]
- MySQL-Syntax:
- DROP {DATABASE | SCHEMA} [IF EXISTS] Datenbankname

Bemerkungen

`DROP DATABASE` wird zum Löschen einer Datenbank aus SQL verwendet. Erstellen Sie eine Sicherungskopie Ihrer Datenbank, bevor Sie sie löschen, um einen versehentlichen Verlust von Informationen zu verhindern.

Examples

DROP-Datenbank

Das Löschen der Datenbank ist eine einfache einzeilige Anweisung. Die Datenbank löschen löscht die Datenbank. Stellen Sie daher immer sicher, dass die Datenbank gesichert wird.

Nachfolgend finden Sie den Befehl zum Löschen der Mitarbeiterdatenbank

```
DROP DATABASE [dbo].[Employees]
```

DROP oder DELETE Database online lesen: https://riptutorial.com/de/sql/topic/3974/drop-oder-delete-database

Kapitel 16: DROP-Tabelle

Bemerkungen

DROP TABLE entfernt die Tabellendefinition zusammen mit den Zeilen, Indizes, Berechtigungen und Auslösern aus dem Schema.

Examples

Einfacher Tropfen

```
Drop Table MyTable;
```

Vor dem Ablegen auf Existenz prüfen

MySQL 3.19

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL 8.x

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server 2005

```
If Exists(Select * From Information_Schema.Tables
          Where Table_Schema = 'dbo'
            And Table_Name = 'MyTable')
  Drop Table dbo.MyTable
```

SQLite 3.0

```
DROP TABLE IF EXISTS MyTable;
```

DROP-Tabelle online lesen: https://riptutorial.com/de/sql/topic/1832/drop-tabelle

Kapitel 17: Duplikate in einem Spalten-Subset mit Detail suchen

Bemerkungen

- Um Zeilen ohne doppelte Einträge auszuwählen, ändern Sie die WHERE-Klausel in "RowCnt = 1".
- Um eine Zeile aus jedem Satz auszuwählen, verwenden Sie Rank () anstelle von Sum () und ändern Sie die äußere WHERE-Klausel, um Zeilen mit Rank () = 1 auszuwählen

Examples

Studenten mit demselben Namen und Geburtsdatum

```
WITH CTE (StudentId, Fname, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By
FirstName, LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

In diesem Beispiel werden ein gemeinsamer Tabellenausdruck und eine Fensterfunktion verwendet, um alle doppelten Zeilen (auf einer Teilmenge von Spalten) nebeneinander anzuzeigen.

Duplikate in einem Spalten-Subset mit Detail suchen online lesen:
https://riptutorial.com/de/sql/topic/1585/duplikate-in-einem-spalten-subset-mit-detail-suchen

Kapitel 18: EINFÜGEN

Syntax

- INSERT INTO Tabellenname (Spalte1, Spalte2, Spalte3, ...) VALUES (Wert1, Wert2, Wert3, ...);
- INSERT INTO Tabellenname (Spalte1, Spalte2 ...) SELECT Wert1, Wert2 ... aus other_table

Examples

Neue Zeile einfügen

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

Diese Anweisung fügt eine neue Zeile in die `Customers` Tabelle ein. Beachten Sie, dass für die `Id` Spalte kein Wert angegeben wurde, da dieser automatisch hinzugefügt wird. Alle anderen Spaltenwerte müssen jedoch angegeben werden.

Nur angegebene Spalten einfügen

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

Diese Anweisung fügt eine neue Zeile in die `Customers` Tabelle ein. Daten werden nur in die angegebenen Spalten eingefügt. Beachten Sie, dass für die Spalte " `PhoneNumber` kein Wert angegeben wurde. Beachten Sie jedoch, dass alle als `not null` gekennzeichneten Spalten enthalten sein müssen.

INSERT-Daten aus einer anderen Tabelle mit SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

In diesem Beispiel werden alle Mitarbeiter in die Customers- Tabelle eingefügt . Da die beiden Tabellen unterschiedliche Felder haben und Sie nicht alle Felder verschieben möchten, müssen Sie festlegen, welche Felder eingefügt werden sollen und welche Felder ausgewählt werden sollen. Die korrelierenden Feldnamen müssen nicht gleich benannt werden, sondern müssen denselben Datentyp haben. In diesem Beispiel wird davon ausgegangen, dass das Feld ID eine Identitätsspezifikation enthält und automatisch inkrementiert wird.

Wenn Sie zwei Tabellen haben, die exakt dieselben Feldnamen haben und nur alle Datensätze verschieben möchten, können Sie Folgendes verwenden:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Fügen Sie mehrere Zeilen gleichzeitig ein

Mit einem einzigen Einfügebefehl können mehrere Zeilen eingefügt werden:

```
INSERT INTO tbl_name (field1, field2, field3)

VALUES (1,2,3), (4,5,6), (7,8,9);
```

Für das gleichzeitige Einfügen großer Datenmengen (Bulk Insert) gibt es DBMS-spezifische Funktionen und Empfehlungen.

MySQL - LOAD DATA INFILE

MSSQL - BULK INSERT

EINFÜGEN online lesen: https://riptutorial.com/de/sql/topic/465/einfugen

Kapitel 19: EXISTS-Klausel

Examples

EXISTS-Klausel

Kundentabelle

Ich würde	Vorname	Nachname
1	Ozgur	Ozturk
2	Youssef	Medi
3	Henry	Tai

Tabelle bestellen

Ich würde	Kundennummer	Menge
1	2	123,50
2	3	14,80

Erhalten Sie alle Kunden mit mindestens einer Bestellung

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

Ergebnis

Ich würde	Vorname	Nachname
2	Youssef	Medi
3	Henry	Tai

Erhalten Sie alle Kunden ohne Bestellung

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Ergebnis

Ich würde	Vorname	Nachname
1	Ozgur	Ozturk

Zweck

`EXISTS` , `IN` und `JOIN` können manchmal für dasselbe Ergebnis verwendet werden, sie sind jedoch nicht gleich:

- `EXISTS` sollte verwendet werden, um zu überprüfen, ob ein Wert in einer anderen Tabelle vorhanden ist
- `IN` sollte für die statische Liste verwendet werden
- `JOIN` sollte verwendet werden, um Daten aus anderen (n) Tabellen abzurufen.

EXISTS-Klausel online lesen: https://riptutorial.com/de/sql/topic/7933/exists-klausel

Kapitel 20: EXPLAIN und BESCHREIBEN

Examples

BESCHREIBEN Tabellenname;

`DESCRIBE` und `EXPLAIN` sind Synonyme. `DESCRIBE` für einen Tabellennamen gibt die Definition der Spalten zurück.

```
DESCRIBE tablename;
```

Beispiel Ergebnis:

```
COLUMN_NAME     COLUMN_TYPE     IS_NULLABLE     COLUMN_KEY     COLUMN_DEFAULT    EXTRA
id              int(11)         NO              PRI            0
auto_increment
test            varchar(255)    YES                            (null)
```

Hier sehen Sie die Spaltennamen, gefolgt vom Spaltentyp. Es zeigt an, ob in der Spalte `null` zulässig ist und ob die Spalte einen Index verwendet. Der Standardwert wird auch angezeigt und wenn die Tabelle ein spezielles Verhalten wie ein `auto_increment` .

EXPLAIN Abfrage auswählen

Ein `Explain` infront einer `select` Abfrage zeigt Ihnen , wie die Abfrage ausgeführt wird. So können Sie feststellen, ob die Abfrage einen Index verwendet oder ob Sie Ihre Abfrage durch Hinzufügen eines Index optimieren könnten.

Beispielanfrage:

```
explain select * from user join data on user.test = data.fk_user;
```

Beispielergebnis:

```
id  select_type  table   type    possible_keys  key     key_len ref       rows  Extra
1   SIMPLE       user    index   test           test    5       (null)    1     Using where;
Using index
1   SIMPLE       data    ref     fk_user        fk_user 5       user.test 1     (null)
```

Bei `type` Sie, ob ein Index verwendet wurde. In der Spalte `possible_keys` Sie, ob der Ausführungsplan aus verschiedenen Indizes auswählen kann, falls keiner vorhanden ist. `key` sagt Ihnen den aktuell verwendeten Index. `key_len` zeigt Ihnen die Größe in Bytes für ein `key_len` . Je niedriger dieser Wert ist, desto mehr Indexelemente passen in dieselbe Speichergröße und können schneller verarbeitet werden. `rows` zeigt die erwartete Anzahl von Zeilen, die die Abfrage scannen muss, je niedriger, desto besser.

EXPLAIN und BESCHREIBEN online lesen: https://riptutorial.com/de/sql/topic/2928/explain-und-beschreiben

Kapitel 21: FALL

Einführung

Der CASE-Ausdruck wird zur Implementierung der If-Then-Logik verwendet.

Syntax

- CASE input_expression
  WANN vergleich1, dann Ergebnis1
  [WANN2 DICHTER2 Ergebnis ...] ...
  [ELSE resultX]
  ENDE
- FALL
  WENN Bedingung1 DANN Ergebnis1
  [WENN Bedingung2 dann Ergebnis2] ...
  [ELSE resultX]
  ENDE

Bemerkungen

Der *einfache CASE-Ausdruck* gibt das erste Ergebnis zurück, dessen `compareX` Wert dem `input_expression` .

Der *gesuchte CASE-Ausdruck* gibt das erste Ergebnis zurück, dessen `conditionX` wahr ist.

Examples

CASE in SELECT gesucht (entspricht einem booleschen Ausdruck)

Das *durchsuchte* CASE gibt Ergebnisse zurück, wenn ein *boolescher* Ausdruck TRUE ist.

(Dies unterscheidet sich vom einfachen Fall, der nur bei einer Eingabe die Gleichwertigkeit prüfen kann.)

```
SELECT Id, ItemId, Price,
  CASE WHEN Price < 10 THEN 'CHEAP'
       WHEN Price < 20 THEN 'AFFORDABLE'
       ELSE 'EXPENSIVE'
  END AS PriceRating
FROM ItemSales
```

Ich würde	Artikel Identifikationsnummer	Preis	PreisPreis
1	100	34,5	TEUER

Ich würde	Artikel Identifikationsnummer	Preis	PreisPreis
2	145	2.3	BILLIG
3	100	34,5	TEUER
4	100	34,5	TEUER
5	145	10	ERSCHWINGLICH

Mit CASE COUNT die Anzahl der Zeilen in einer Spalte mit einer Bedingung übereinstimmen.

Anwendungsfall

`CASE` kann in Verbindung mit `SUM` , um nur die Elemente zurückzugeben, die mit einer vordefinierten Bedingung übereinstimmen. (Dies ist ähnlich zu `COUNTIF` in Excel.)

Der Trick besteht darin, binäre Ergebnisse zurückzugeben, die Übereinstimmungen anzeigen, sodass die für übereinstimmende Einträge zurückgegebenen "1" für einen Zähler der Gesamtanzahl der Übereinstimmungen summiert werden können.

`ItemSales` , Sie möchten mit dieser Tabelle `ItemSales` die Gesamtzahl der als "teuer" eingestuften Artikel ermitteln:

Ich würde	Artikel Identifikationsnummer	Preis	PreisPreis
1	100	34,5	TEUER
2	145	2.3	BILLIG
3	100	34,5	TEUER
4	100	34,5	TEUER
5	145	10	ERSCHWINGLICH

Abfrage

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM ( CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
          END
        ) AS ExpensiveItemsCount
FROM ItemSales
```

Ergebnisse:

ItemsCount	ExpensiveItemsCount
5	3

Alternative:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN 1
            ELSE 0
        END
       ) AS ExpensiveItemsCount
FROM ItemSales
```

Abkürzung CASE in SELECT

Die Abkürzungsvariante von `CASE` wertet einen Ausdruck (normalerweise eine Spalte) anhand einer Reihe von Werten aus. Diese Variante ist etwas kürzer und erspart es, den ausgewerteten Ausdruck immer und immer wieder zu wiederholen. Die `ELSE` Klausel kann jedoch weiterhin verwendet werden:

```
SELECT Id, ItemId, Price,
  CASE Price WHEN 5  THEN 'CHEAP'
             WHEN 15 THEN 'AFFORDABLE'
             ELSE         'EXPENSIVE'
  END as PriceRating
FROM ItemSales
```

Ein Wort der Warnung. Es ist wichtig zu wissen, dass bei Verwendung der kurzen Variante die gesamte Anweisung bei jedem `WHEN` ausgewertet wird. Daher die folgende Aussage:

```
SELECT
    CASE ABS(CHECKSUM(NEWID())) % 4
        WHEN 0 THEN 'Dr'
        WHEN 1 THEN 'Master'
        WHEN 2 THEN 'Mr'
        WHEN 3 THEN 'Mrs'
    END
```

kann zu einem `NULL` Ergebnis führen. Das liegt daran , dass bei jeder `WHEN NEWID()` ist mit einem neuen Ergebnis wieder aufgerufen wird. Gleichwertig:

```
SELECT
    CASE
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
    END
```

Daher kann es alle `WHEN` Fälle verfehlen und als `NULL` .

CASE in einer Klausel ORDER BY

Wir können 1,2,3 verwenden, um die Art der Bestellung zu bestimmen

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
      WHEN 'MARKETING' THEN  1
      WHEN 'SALES' THEN 2
      WHEN 'RESEARCH' THEN 3
      WHEN 'INNOVATION' THEN 4
      ELSE        5
      END,
      CITY
```

ICH WÜRDE	REGION	STADT	ABTEILUNG	EMPLOYEES_NUMBER
12	Neu England	Boston	MARKETING	9
fünfzehn	Westen	San Francisco	MARKETING	12
9	Mittlerer Westen	Chicago	DER UMSATZ	8
14	Mid-Atlantic	New York	DER UMSATZ	12
5	Westen	Los Angeles	FORSCHUNG	11
10	Mid-Atlantic	Philadelphia	FORSCHUNG	13
4	Mittlerer Westen	Chicago	INNOVATION	11
2	Mittlerer Westen	Detroit	HUMANRESSOURCEN	9

CASE in UPDATE verwenden

Beispiel für Preiserhöhungen:

```
UPDATE ItemPrice
SET Price = Price *
  CASE ItemId
    WHEN 1 THEN 1.05
    WHEN 2 THEN 1.10
    WHEN 3 THEN 1.15
```

```
    ELSE 1.00
  END
```

CASE-Verwendung für NULL-Werte, die zuletzt bestellt wurden

Auf diese Weise wird die '0', die die bekannten Werte darstellt, an erster Stelle stehen, die '1', die die NULL-Werte darstellt, wird nach dem letzten sortiert:

```
SELECT ID
      ,REGION
      ,CITY
      ,DEPARTMENT
      ,EMPLOYEES_NUMBER
  FROM DEPT
  ORDER BY
  CASE WHEN REGION IS NULL THEN 1
  ELSE 0
  END,
  REGION
```

ICH WÜRDE	REGION	STADT	ABTEILUNG	EMPLOYEES_NUMBER
10	Mid-Atlantic	Philadelphia	FORSCHUNG	13
14	Mid-Atlantic	New York	DER UMSATZ	12
9	Mittlerer Westen	Chicago	DER UMSATZ	8
12	Neu England	Boston	MARKETING	9
5	Westen	Los Angeles	FORSCHUNG	11
fünfzehn	NULL	San Francisco	MARKETING	12
4	NULL	Chicago	INNOVATION	11
2	NULL	Detroit	HUMANRESSOURCEN	9

CASE in ORDER BY-Klausel zum Sortieren von Datensätzen nach dem niedrigsten Wert von 2 Spalten

Stellen Sie sich vor, Sie müssen Datensätze nach dem niedrigsten Wert einer der beiden Spalten sortieren. Einige Datenbanken könnten für diese Funktion eine nicht aggregierte `MIN()` oder `LEAST()` Funktion verwenden (`... ORDER BY MIN(Date1, Date2)`), aber in Standard-SQL müssen Sie

einen `CASE` Ausdruck verwenden.

Der `CASE` Ausdruck in der Abfrage sieht unten an dem `Date1` und `Date2` Spalten, überprüft , welcher Spalte den niedrigeren Wert, und sortiert die Datensätze in Abhängigkeit von diesem Wert.

Beispieldaten

Ich würde	Date1	Date2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

Abfrage

```
SELECT Id, Date1, Date2
FROM YourTable
ORDER BY CASE
           WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
           ELSE Date2
         END
```

Ergebnisse

Ich würde	Date1	Date2
1	**2017-01-01**	2017-01-31
3	2017-01-31	**2017-01-02**
2	2017-01-31	**2017-01-03**
6	**2017-01-04**	2017-01-31
5	2017-01-31	**2017-01-05**
4	**2017-01-06**	2017-01-31

Erläuterung

Wie Sie sehen, steht Zeile mit `Id = 1` erster Stelle, da `Date1` den niedrigsten Datensatz aus der gesamten Tabelle `2017-01-01` . Zeile mit `Id = 3` ist die zweite, da `Date2` `2017-01-02` entspricht und der zweitniedrigste Wert aus Tabelle ist und so weiter.

Daher haben wir Datensätze von `2017-01-01` bis `2017-01-06` aufsteigend sortiert und es ist nicht zu `Date2` in welcher Spalte `Date1` oder `Date2` diese Werte sind.

FALL online lesen: https://riptutorial.com/de/sql/topic/456/fall

Kapitel 22: Fensterfunktionen

Examples

Hinzufügen der insgesamt ausgewählten Zeilen zu jeder Zeile

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

Ich würde	Name	Ttl_Rows
1	Beispiel	5
2	foo	5
3	Bar	5
4	baz	5
5	quux	5

Anstatt zwei Abfragen zu verwenden, um einen Zähler als die Zeile abzurufen, können Sie ein Aggregat als Fensterfunktion verwenden und die vollständige Ergebnismenge als Fenster verwenden.
Dies kann als Grundlage für die weitere Berechnung verwendet werden, ohne dass zusätzliche Selbstverknüpfungen erforderlich sind.

Ein Flag setzen, wenn andere Zeilen eine gemeinsame Eigenschaft haben

Nehmen wir an, ich habe diese Daten:

Tischartikel

Ich würde	Name	Etikett
1	Beispiel	unique_tag
2	foo	einfach
42	Bar	einfach
3	baz	Hallo
51	quux	Welt

Ich möchte all diese Zeilen erhalten und wissen, ob ein Tag von anderen Zeilen verwendet wird

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

Das Ergebnis wird sein:

Ich würde	Name	Etikett	Flagge
1	Beispiel	unique_tag	falsch
2	foo	einfach	wahr
42	Bar	einfach	wahr
3	baz	Hallo	falsch
51	quux	Welt	falsch

Falls Ihre Datenbank nicht OVER und PARTITION hat, können Sie dies verwenden, um dasselbe Ergebnis zu erzielen:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM
items A
```

Laufende Summe

Angesichts dieser Daten:

Datum	Menge
2016-03-12	200
2016-03-11	-50
2016-03-14	100
2016-03-15	100
2016-03-10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

werde dir geben

Datum	Menge	Laufen
2016-03-10	-250	-250
2016-03-11	-50	-300

Datum	Menge	Laufen
2016-03-12	200	-100
2016-03-14	100	0
2016-03-15	100	-100

N die aktuellsten Zeilen über mehrere Gruppierungen abrufen

Angesichts dieser Daten

Benutzeridentifikation	Fertigstellungstermin
1	2016-07-20
1	2016-07-21
2	2016-07-20
2	2016-07-21
2	2016-07-22

```
;with CTE as
(SELECT *,
        ROW_NUMBER() OVER (PARTITION BY User_ID
                           ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FORM CTE WHERE Row_Num <= n
```

Mit n = 1 erhalten Sie die letzte Zeile pro `user_id` :

Benutzeridentifikation	Fertigstellungstermin	Row_Num
1	2016-07-21	1
2	2016-07-22	1

Suchen nach Datensätzen mit der Funktion LAG ()

Gegeben diese Beispieldaten:

ICH WÜRDE	STATUS	STATUS_TIME	STATUS_BY
1	EIN	2016-09-28-19.47.52.501398	USER_1
3	EIN	2016-09-28-19.47.52.501511	USER_2

ICH WÜRDE	STATUS	STATUS_TIME	STATUS_BY
1	DREI	2016-09-28-19.47.52.501517	USER_3
3	ZWEI	2016-09-28-19.47.52.501521	USER_2
3	DREI	2016-09-28-19.47.52.501524	USER_4

Elemente, die durch `ID` Werte identifiziert werden, müssen in der Reihenfolge von `STATUS` 'ONE' zu 'TWO' zu 'THREE' wechseln, ohne Status zu überspringen. Das Problem besteht darin, Benutzerwerte (`STATUS_BY`) zu finden, die gegen die Regel verstoßen, und sofort von "ONE" zu "DREI" wechseln.

Die analytische Funktion `LAG()` hilft, das Problem zu lösen, indem für jede Zeile der Wert in der vorhergehenden Zeile zurückgegeben wird:

```
SELECT * FROM (
 SELECT
  t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
  FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

Falls Ihre Datenbank nicht über LAG () verfügt, können Sie Folgendes verwenden, um dasselbe Ergebnis zu erzielen:

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as
prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND   B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and
id = A.id)
AND   A.status = 'THREE' AND NOT B.status = 'TWO'
```

Fensterfunktionen online lesen: https://riptutorial.com/de/sql/topic/647/fensterfunktionen

Kapitel 23: Filtern Sie die Ergebnisse mit WHERE und HAVING

Syntax

- SELECT Spaltenname
  FROM tabellenname
  WHERE Spaltenname Operatorwert
- SELECT Spaltenname, Aggregatfunktion (Spaltenname)
  FROM tabellenname
  GROUP BY Spaltenname
  HAVING Aggregate_function (Spaltenname) Operatorwert

Examples

Die WHERE-Klausel gibt nur Zeilen zurück, die ihren Kriterien entsprechen

Steam bietet Spiele unter 10 USD auf der Shop-Seite. Irgendwo tief im Herzen ihrer Systeme gibt es wahrscheinlich eine Abfrage, die ungefähr so aussieht:

```
SELECT *
FROM Items
WHERE Price < 10
```

Verwenden Sie IN, um Zeilen mit einem in einer Liste enthaltenen Wert zurückzugeben

In diesem Beispiel wird die Autotabelle aus den Beispieldatenbanken verwendet.

```
SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)
```

Diese Abfrage gibt Auto Nr. 2 zurück, das 200 kostet, und Auto Nr. 3, das 100 kostet. Beachten Sie, dass dies der Verwendung mehrerer Klauseln mit `OR` , z.

```
SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Verwenden Sie LIKE, um übereinstimmende Zeichenfolgen und Teilzeichenfolgen zu finden

Siehe vollständige Dokumentation zum LIKE-Operator .

In diesem Beispiel wird die Employees-Tabelle aus den Beispieldatenbanken verwendet.

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

Diese Abfrage gibt nur Mitarbeiter # 1 zurück, dessen Vorname genau mit 'John' übereinstimmt.

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

Durch Hinzufügen von `%` können Sie nach einer Teilzeichenfolge suchen:

- `John%` - gibt jeden Mitarbeiter zurück, dessen Name mit 'John' beginnt, gefolgt von einer beliebigen Anzahl von Zeichen
- `%John` - gibt jeden Mitarbeiter zurück, dessen Name mit 'John' endet, gefolgt von einer beliebigen Anzahl von Zeichen
- `%John%` - gibt jeden Mitarbeiter, dessen Name "John" enthält, an eine beliebige Stelle innerhalb des Werts zurück

In diesem Fall gibt die Abfrage Mitarbeiter # 2 mit dem Namen 'John' sowie Mitarbeiter # 4 mit dem Namen 'Johnathon' zurück.

WHERE-Klausel mit NULL / NOT NULL-Werten

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

Diese Anweisung gibt alle Employee- Datensätze zurück, bei denen der Wert der `ManagerId` Spalte `NULL` .

Das Ergebnis wird sein:

```
Id    FName    LName    PhoneNumber    ManagerId    DepartmentId
1     James    Smith    1234567890     NULL         1
```

```
SELECT *
FROM Employees
WHERE ManagerId IS NOT NULL
```

Diese Anweisung gibt alle Employee- Datensätze zurück, bei denen der Wert der `ManagerId` *nicht* `NULL` .

Das Ergebnis wird sein:

```
Id    FName       LName     PhoneNumber    ManagerId    DepartmentId
2     John        Johnson   2468101214     1            1
```

```
3     Michael     Williams  1357911131     1            2
4     Johnathon   Smith     1212121212     2            1
```

Anmerkung: `WHERE ManagerId = NULL` Abfrage gibt keine Ergebnisse zurück, wenn Sie die WHERE-Klausel in `WHERE ManagerId = NULL` oder `WHERE ManagerId <> NULL` .

Verwenden Sie HAVING mit Aggregatfunktionen

Im Gegensatz zur `WHERE` Klausel kann `HAVING` mit Aggregatfunktionen verwendet werden.

> Eine Aggregatfunktion ist eine Funktion, bei der die Werte mehrerer Zeilen als Eingabe für bestimmte Kriterien gruppiert werden, um einen einzelnen Wert mit signifikanterer Bedeutung oder Messung (Wikipedia) zu bilden.

Häufige Aggregatfunktionen umfassen `COUNT()` , `SUM()` , `MIN()` und `MAX()` .

In diesem Beispiel wird die Autotabelle aus den Beispieldatenbanken verwendet.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

Diese Abfrage gibt die `CustomerId` und die `Number of Cars` jedes Kunden zurück, der mehr als ein Auto hat. In diesem Fall ist der einzige Kunde, der mehr als ein Auto hat, Kunde # 1.

Die Ergebnisse werden wie folgt aussehen:

Kundennummer	Anzahl der Autos
1	2

Verwenden Sie ZWISCHEN, um Ergebnisse zu filtern

In den folgenden Beispielen werden die Beispieldatenbanken " Artikelverkauf und Kunden" verwendet.

> Hinweis: Der Operator BETWEEN *ist* inklusive.

Verwenden des BETWEEN-Operators mit Zahlen:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

Diese Abfrage gibt alle `ItemSales` Datensätze zurück, deren Menge größer oder gleich 10 und kleiner oder gleich 17 ist. Die Ergebnisse sehen folgendermaßen aus:

Ich würde	Verkaufsdatum	Artikel Identifikationsnummer	Menge	Preis
1	2013-07-01	100	10	34,5
4	2013-07-23	100	fünfzehn	34,5
5	2013-07-24	145	10	34,5

Verwenden des BETWEEN-Operators mit Datumswerten:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

Diese Abfrage gibt alle `ItemSales` Datensätze mit einem `SaleDate` , das größer oder gleich dem 11. Juli 2013 und weniger als dem 24. Mai 2013 ist.

Ich würde	Verkaufsdatum	Artikel Identifikationsnummer	Menge	Preis
3	2013-07-11	100	20	34,5
4	2013-07-23	100	fünfzehn	34,5
5	2013-07-24	145	10	34,5

> Wenn Sie datetime-Werte anstelle von Datumsangaben vergleichen, müssen Sie die datetime-Werte möglicherweise in Datumswerte konvertieren oder 24 Stunden addieren oder subtrahieren, um die richtigen Ergebnisse zu erhalten.

Verwenden des BETWEEN-Operators mit Textwerten:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L';
```

Live-Beispiel: SQL-Geige

Diese Abfrage gibt alle Kunden zurück, deren Name alphabetisch zwischen den Buchstaben 'D' und 'L' liegt. In diesem Fall werden Kunde Nr. 1 und Nr. 3 zurückgegeben. Kunde 2, dessen Name mit einem 'M' beginnt, wird nicht berücksichtigt.

Ich würde	FName	LName
1	Wilhelm	Jones
3	Richard	Davis

Gleichberechtigung

```
SELECT * FROM Employees
```

Diese Anweisung gibt alle Zeilen aus der Tabelle `Employees` .

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Wenn Sie am Ende Ihrer `SELECT` Anweisung ein `WHERE` können Sie die zurückgegebenen Zeilen auf eine Bedingung beschränken. In diesem Fall, wenn es eine genaue Übereinstimmung mit dem `=` -Zeichen gibt:

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Gibt nur die Zeilen zurück, bei denen die `DepartmentId` gleich `1` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

UND UND ODER

Sie können auch mehrere Operatoren kombinieren, um komplexere `WHERE` Bedingungen zu erstellen. Die folgenden Beispiele verwenden die `Employees` Tabelle:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

UND

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Wird zurückkehren:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
```

ODER

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Wird zurückkehren:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Verwenden Sie HAVING, um nach mehreren Bedingungen in einer Gruppe zu suchen

Bestellungstabelle

Kundennummer	Produkt ID	Menge	Preis
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

Um nach Kunden zu suchen, die beide - ProductID 2 und 3 - bestellt haben, kann HAVING verwendet werden

```
select customerId
from orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Rückgabewert:

Kundennummer
1

Die Abfrage wählt nur Datensätze mit den Produkt-IDs in Fragen aus und prüft mit der HAVING-Klausel nach Gruppen mit 2 Produkt-IDs und nicht nur einer.

Eine andere Möglichkeit wäre

```
select customerId
from orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
   and sum(case when productID = 3 then 1 else 0 end) > 0
```

Diese Abfrage wählt nur Gruppen aus, die mindestens einen Datensatz mit der Produkt-ID 2 und mindestens einen mit der Produkt-ID 3 haben.

Wo EXISTEN

`TableName` Datensätze in `TableName` , deren Datensätze in `TableName1` übereinstimmen.

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Filtern Sie die Ergebnisse mit WHERE und HAVING online lesen:
https://riptutorial.com/de/sql/topic/636/filtern-sie-die-ergebnisse-mit-where-und-having

Kapitel 24: Fremde Schlüssel

Examples

Erstellen einer Tabelle mit einem Fremdschlüssel

In diesem Beispiel haben wir eine vorhandene Tabelle, `SuperHeros` .

Diese Tabelle enthält eine Primärschlüssel- `ID` .

Wir werden eine neue Tabelle hinzufügen, um die Kräfte jedes Superhelden zu speichern:

```
CREATE TABLE HeroPowers
(
    ID int NOT NULL PRIMARY KEY,
    Name nvarchar(MAX) NOT NULL,
    HeroId int REFERENCES SuperHeros(ID)
)
```

Die Spalte `HeroId` ist ein **Fremdschlüssel** für die Tabelle `SuperHeros` .

Fremdschlüssel erklärt

Fremdschlüsseleinschränkungen stellen die Datenintegrität sicher, indem erzwingt, dass Werte in einer Tabelle mit Werten in einer anderen Tabelle übereinstimmen müssen.

Ein Beispiel, wo ein Fremdschlüssel benötigt wird, ist: In einer Universität muss ein Kurs zu einer Abteilung gehören. Code für dieses Szenario lautet:

```
CREATE TABLE Department (
    Dept_Code        CHAR (5)     PRIMARY KEY,
    Dept_Name        VARCHAR (20) UNIQUE
);
```

Fügen Sie Werte mit der folgenden Anweisung ein:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

Die folgende Tabelle enthält die Informationen zu den Fächern der Informatikbranche:

```
CREATE TABLE Programming_Courses (
    Dept_Code       CHAR(5),
    Prg_Code        CHAR(9) PRIMARY KEY,
    Prg_Name        VARCHAR (50) UNIQUE,
    FOREIGN KEY (Dept_Code) References Department(Dept_Code)
);
```

(Der Datentyp des Fremdschlüssels muss mit dem Datentyp des referenzierten Schlüssels

übereinstimmen.)

Die `Dept_Code` für die Spalte `Dept_Code` lässt Werte nur dann zu, wenn sie bereits in der referenzierten Tabelle, `Department` . Das bedeutet, wenn Sie versuchen, die folgenden Werte einzufügen:

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', 'Database Systems');
```

Die Datenbank wird einen Fehler durch einen `CS300` , da `CS300` nicht in der `Department` . Wenn Sie jedoch einen Schlüsselwert versuchen, der vorhanden ist:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```

dann erlaubt die Datenbank diese Werte.

Ein paar Tipps zur Verwendung von Fremdschlüsseln

- Ein Fremdschlüssel muss auf einen UNIQUE-Schlüssel (oder einen Primärschlüssel) in der übergeordneten Tabelle verweisen.
- Die Eingabe eines NULL-Werts in eine Fremdschlüsselspalte verursacht keinen Fehler.
- Fremdschlüsseleinschränkungen können auf Tabellen in derselben Datenbank verweisen.
- Fremdschlüsseleinschränkungen können auf eine andere Spalte in derselben Tabelle verweisen (Selbstreferenz).

Fremde Schlüssel online lesen: https://riptutorial.com/de/sql/topic/1533/fremde-schlussel

Kapitel 25: FUNKTION ERSTELLEN

Syntax

- CREATE FUNCTION Funktionsname ([Liste_der_Parameter]) RETURNS Rückgabedatentyp AS BEGIN Funktionskörper RETURN scalar_expression END

Parameter

Streit	Beschreibung
Funktionsname	der Name der Funktion
list_of_paramenters	Parameter, die die Funktion akzeptiert
return_data_type	Geben Sie diese Funktion ein. Einige SQL- Datentypen
function_body	der Code der Funktion
scalar_expression	Von Funktion zurückgegebener Skalarwert

Bemerkungen

CREATE FUNCTION erstellt eine benutzerdefinierte Funktion, die bei einer SELECT-, INSERT-, UPDATE- oder DELETE-Abfrage verwendet werden kann. Die Funktionen können erstellt werden, um eine einzelne Variable oder eine einzelne Tabelle zurückzugeben.

Examples

Erstellen Sie eine neue Funktion

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
END
```

In diesem Beispiel wird eine Funktion namens **FirstWord erstellt** , die einen varchar-Parameter akzeptiert und einen anderen varchar-Wert zurückgibt.

FUNKTION ERSTELLEN online lesen: https://riptutorial.com/de/sql/topic/2437/funktion-erstellen

Kapitel 26: Funktionen (Aggregat)

Syntax

- Funktion ([*DISTINCT*] Ausdruck) -DISTINCT ist ein optionaler Parameter
- AVG (Ausdruck [ALL | DISTINCT])
- COUNT ({[ALL | DISTINCT] Ausdruck] | *})
- GROUPING (<column_expression>)
- MAX (Ausdruck [ALL | DISTINCT])
- MIN (Ausdruck [ALL | DISTINCT])
- SUM (Ausdruck [ALL | DISTINCT])
- VAR (Ausdruck [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- VARP ([ALL | DISTINCT] Ausdruck)
  OVER ([partition_by_clause] order_by_clause
- STDEV (Ausdruck [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- STDEVP (Ausdruck [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)

Bemerkungen

In der Datenbankverwaltung ist eine Aggregatfunktion eine Funktion, bei der die Werte mehrerer Zeilen als Eingabe für bestimmte Kriterien gruppiert werden, um einen einzelnen Wert von größerer Bedeutung oder Messung zu bilden, beispielsweise eine Menge, ein Beutel oder eine Liste.

```
MIN         returns the smallest value in a given column
MAX         returns the largest value in a given column
SUM         returns the sum of the numeric values in a given column
AVG         returns the average value of a given column
COUNT       returns the total number of values in a given column
COUNT(*)    returns the number of rows in a table
GROUPING    Is a column or an expression that contains a column in a GROUP BY clause.
STDEV       returns the statistical standard deviation of all values in the specified
expression.
STDEVP      returns the statistical standard deviation for the population for all values in the
specified expression.
VAR         returns the statistical variance of all values in the specified expression. may be
followed by the OVER clause.
VARP        returns the statistical variance for the population for all values in the specified
expression.
```

> Aggregatfunktionen werden zur Berechnung einer "zurückgegebenen Spalte numerischer Daten" aus Ihrer `SELECT` Anweisung verwendet. Sie fassen im Wesentlichen die Ergebnisse einer bestimmten Spalte ausgewählter Daten zusammen. - SQLCourse2.com

Alle Aggregatfunktionen ignorieren NULL-Werte.

Examples

SUMME

`Sum` - Funktion den Wert aller Zeilen in der Gruppe zusammenzufassen. Wenn die group-by-Klausel weggelassen wird, summiert sie alle Zeilen.

```
select sum(salary) TotalSalary
from employees;
```

TotalSalary
2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DepartmentId	TotalSalary
1	2000
2	500

Bedingte Aggregation

Zahlungstabelle

Kunde	Zahlungsart	Menge
Peter	Kredit	100
Peter	Kredit	300
John	Kredit	1000
John	Lastschrift	500

```
select customer,
       sum(case when payment_type = 'credit' then amount else 0 end) as credit,
       sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
group by customer
```

Ergebnis:

Kunde	Kredit	Lastschrift
Peter	400	0
John	1000	500

```
select customer,
       sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
       sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
group by customer
```

Ergebnis:

Kunde	credit_transaction_count	debit_transaction_count
Peter	2	0
John	1	1

AVG ()

Die Aggregatfunktion AVG () gibt den Durchschnitt eines bestimmten Ausdrucks zurück, normalerweise numerische Werte in einer Spalte. Angenommen, wir haben eine Tabelle mit der jährlichen Bevölkerungsberechnung in Städten auf der ganzen Welt. Die Aufzeichnungen für New York City sehen ähnlich wie die folgenden aus:

BEISPIELTABELLE

Stadtname	Population	Jahr
New York City	8.550.405	2015
New York City	...	...
New York City	8.000.906	2005

So wählen Sie die durchschnittliche Einwohnerzahl von New York City (USA) aus einer Tabelle aus, die Ortsnamen, Bevölkerungsmessungen und Messjahre der letzten zehn Jahre enthält:

ABFRAGE

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Beachten Sie, dass das Messjahr in der Abfrage nicht vorhanden ist, da die Bevölkerung im

Zeitverlauf gemittelt wird.

ERGEBNISSE

Stadtname	avg_population
New York City	8.250.754

> Hinweis: Die Funktion AVG () konvertiert Werte in numerische Typen. Dies ist besonders wichtig, wenn Sie mit Daten arbeiten.

List Verkettung

Teilweise Verdienst dieser SO-Antwort.

Mit List Concatenation wird eine Spalte oder ein Ausdruck zusammengefasst, indem die Werte für jede Gruppe in einer einzelnen Zeichenfolge zusammengefasst werden. Ein String zur Begrenzung jedes Werts (entweder leer oder ein Komma, wenn nicht angegeben) und die Reihenfolge der Werte im Ergebnis kann angegeben werden. Obwohl es nicht Teil des SQL-Standards ist, unterstützt es jeder große Anbieter von relationalen Datenbanken auf seine eigene Weise.

MySQL

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Oracle und DB2

```
SELECT ColumnA
     , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
```

```
 ORDER BY ColumnA;
```

SQL Server

SQL Server 2016 und früher

(CTE eingeschlossen, um das DRY-Prinzip zu fördern)

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName)
SELECT t0.ColumnA
     , STUFF((
       SELECT ',' + t1.ColumnB
         FROM CTE_TableName t1
        WHERE t1.ColumnA = t0.ColumnA
        ORDER BY t1.ColumnB
          FOR XML PATH('')), 1, 1, '') AS ColumnBs
  FROM CTE_TableName t0
 GROUP BY t0.ColumnA
 ORDER BY ColumnA;
```

SQL Server 2017 und SQL Azure

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

SQLite

ohne zu bestellen:

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Bestellung erfordert eine Unterabfrage oder CTE:

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName
        ORDER BY ColumnA, ColumnB)
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM CTE_TableName
```

```
GROUP BY ColumnA
ORDER BY ColumnA;
```

Anzahl

Sie können die Anzahl der Zeilen zählen:

```
SELECT count(*) TotalRows
FROM employees;
```

Gesamtreihen
4

Oder zählen Sie die Mitarbeiter pro Abteilung:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DepartmentId	Anzahl Beschäftigte
1	3
2	1

Sie können über eine Spalte / einen Ausdruck mit dem Effekt zählen, der die `NULL` Werte nicht zählt:

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

mgr
3

(Es gibt eine Nullwertmanager-Spalte)

Sie können **DISTINCT auch** innerhalb einer anderen Funktion wie **COUNT verwenden,** um nur die **DISTINCT-** Mitglieder des Sets zu finden, auf denen die Operation ausgeführt wird.

Zum Beispiel:

```
SELECT COUNT(ContinentCode) AllCount
,      COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

Gibt verschiedene Werte zurück. Der *SingleCount zählt* nur einmal Kontinente, während der

AllCount Duplikate enthält.

ContinentCode
OC
EU
WIE
N / A
N / A
AF
AF

AllCount: 7 SingleCount: 5

Max

Ermitteln Sie den maximalen Wert der Spalte:

```
select max(age) from employee;
```

Im obigen Beispiel wird größten Wert für Spalte zurückgeben `age` der `employee` Tabelle.

Syntax:

```
SELECT MAX(column_name) FROM table_name;
```

Mindest

Finde den kleinsten Wert der Spalte:

```
 select min(age) from employee;
```

Im obigen Beispiel wird kleinsten Wert für Spalte zurückgeben `age` der `employee` Tabelle.

Syntax:

```
 SELECT MIN(column_name) FROM table_name;
```

Funktionen (Aggregat) online lesen: https://riptutorial.com/de/sql/topic/1002/funktionen--aggregat-

Kapitel 27: Funktionen (Analyse)

Einführung

Mit analytischen Funktionen bestimmen Sie Werte anhand von Wertegruppen. Mit diesem Funktionstyp können Sie beispielsweise laufende Summen, Prozentsätze oder das Spitzenergebnis innerhalb einer Gruppe ermitteln.

Syntax

1. FIRST_VALUE (scalar_expression) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
2. LAST_VALUE (scalar_expression) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
3. LAG (scalar_expression [, offset] [, default]) OVER ([partition_by_clause] order_by_clause)
4. LEAD (scalar_expression [, offset], [default]) OVER ([partition_by_clause] order_by_clause)
5. PERCENT_RANK () OVER ([partition_by_clause] order_by_clause)
6. CUME_DIST () OVER ([partition_by_clause] order_by_clause)
7. PERCENTILE_DISC (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])
8. PERCENTILE_CONT (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])

Examples

FIRST_VALUE

Mit der Funktion `FIRST_VALUE` bestimmen Sie den ersten Wert in einer geordneten Ergebnismenge, den Sie mit einem `FIRST_VALUE` identifizieren.

```
SELECT StateProvinceID, Name, TaxRate,
       FIRST_VALUE(StateProvinceID)
        OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

In diesem Beispiel wird die `FIRST_VALUE` Funktion verwendet, um die `ID` des Staates oder der Provinz mit dem niedrigsten Steuersatz zurückzugeben. Die `OVER` Klausel wird verwendet, um die Steuersätze anzuordnen, um den niedrigsten Satz zu erhalten.

StateProvinceID	Name	Steuersatz	FirstValue
74	Utah State Sales Tax	5,00	74
36	Minnesota State Umsatzsteuer	6,75	74

StateProvinceID	Name	Steuersatz	FirstValue
30	Massachusetts State Umsatzsteuer	7.00	74
1	Kanadische GST	7.00	74
57	Kanadische GST	7.00	74
63	Kanadische GST	7.00	74

LAST_VALUE

Die `LAST_VALUE` Funktion stellt den letzten Wert in einer geordneten Ergebnismenge bereit, den Sie mit einem `LAST_VALUE` angeben.

```
SELECT TerritoryID, StartDate, BusinessentityID,
       LAST_VALUE(BusinessentityID)
        OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

In diesem Beispiel wird die `LAST_VALUE` Funktion verwendet, um den letzten Wert für jedes Rowset in den geordneten Werten zurückzugeben.

TerritoryID	Anfangsdatum	BusinessentityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

LAG und LEAD

Die `LAG` Funktion liefert Daten in Zeilen vor der aktuellen Zeile in derselben Ergebnismenge. In einer `SELECT` Anweisung können Sie beispielsweise Werte in der aktuellen Zeile mit Werten in einer vorherigen Zeile vergleichen.

Sie verwenden einen Skalarausdruck, um die Werte anzugeben, die verglichen werden sollen. Der Versatzparameter ist die Anzahl der Zeilen vor der aktuellen Zeile, die im Vergleich verwendet werden. Wenn Sie die Anzahl der Zeilen nicht angeben, wird der Standardwert einer Zeile verwendet.

Der Standardparameter gibt den Wert an, der zurückgegeben werden soll, wenn der Ausdruck bei

offset einen `NULL` Wert hat. Wenn Sie keinen Wert angeben, wird der Wert `NULL` zurückgegeben.

Die `LEAD` Funktion liefert Daten in Zeilen nach der aktuellen Zeile in der Zeilengruppe. In einer `SELECT` Anweisung können Sie beispielsweise Werte in der aktuellen Zeile mit Werten in der folgenden Zeile vergleichen.

Sie geben die Werte an, die mit einem Skalarausdruck verglichen werden sollen. Der Versatzparameter ist die Anzahl der Zeilen nach der aktuellen Zeile, die im Vergleich verwendet werden.

Sie geben den Wert an, der zurückgegeben werden soll, wenn der Ausdruck bei offset einen `NULL` Wert unter Verwendung des Standardparameters hat. Wenn Sie diese Parameter nicht angeben, wird der Standardwert einer Zeile verwendet und der Wert `NULL` wird zurückgegeben.

```
SELECT BusinessEntityID, SalesYTD,
       LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
       LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

In diesem Beispiel werden die LEAD- und LAG-Funktionen verwendet, um die Verkaufswerte für jeden Mitarbeiter bis dato mit denen der oben und unten aufgeführten Mitarbeiter zu vergleichen, wobei die Datensätze basierend auf der Spalte BusinessEntityID angeordnet werden.

BusinessEntityID	SalesYTD	Lead-Wert	Verzögerungswert
274	559697.5639	3763178.1787	0,0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

PERCENT_RANK und CUME_DIST

Die `PERCENT_RANK` Funktion berechnet die Rangfolge einer Zeile relativ zum `PERCENT_RANK` . Der Prozentsatz basiert auf der Anzahl der Zeilen in der Gruppe, die einen niedrigeren Wert als die aktuelle Zeile haben.

Der erste Wert in der Ergebnismenge hat immer einen prozentualen Rang von Null. Der Wert für den höchsten Wert oder den letzten Wert in der Gruppe ist immer eins.

Die `CUME_DIST` Funktion berechnet die relative Position eines angegebenen Werts in einer Gruppe

von Werten, indem er den Prozentsatz der Werte bestimmt, die kleiner oder gleich diesem Wert sind. Dies wird als kumulative Verteilung bezeichnet.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
PERCENT_RANK() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Percent Rank",
CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Cumulative Distribution"
FROM Employee;
```

In diesem Beispiel verwenden Sie eine `ORDER` Klausel, um die Zeilen, die von der `SELECT` Anweisung basierend auf den `SELECT` der Mitarbeiter abgerufen werden, zu partitionieren oder zu gruppieren. Die Ergebnisse in jeder Gruppe werden basierend auf der Anzahl der von Mitarbeitern in Anspruch genommenen Ausfallzeiten sortiert.

BusinessEntityID	Berufsbezeichnung	SickLeaveHours	Prozentsatz	Kumulative Verteilung
267	Anwendungsspezialist	57	0	0,25
268	Anwendungsspezialist	56	0,333333333333333	0,75
269	Anwendungsspezialist	56	0,333333333333333	0,75
272	Anwendungsspezialist	55	1	1
262	Assistent des Cheif Financial Officer	48	0	1
239	Leistungsspezialist	45	0	1
252	Käufer	50	0	0,1111111111
251	Käufer	49	0,125	0,3333333333
256	Käufer	49	0,125	0,3333333333
253	Käufer	48	0,375	0,5555555555
254	Käufer	48	0,375	0,5555555555

Die `PERCENT_RANK` Funktion `PERCENT_RANK` die Einträge innerhalb jeder Gruppe ein. Für jeden Eintrag wird der Prozentsatz der Einträge in derselben Gruppe mit niedrigeren Werten zurückgegeben.

Die `CUME_DIST` Funktion ist ähnlich, abgesehen davon, dass der Prozentsatz von Werten zurückgegeben wird, die kleiner oder gleich dem aktuellen Wert sind.

PERCENTILE_DISC und PERCENTILE_CONT

Die Funktion `PERCENTILE_DISC` listet den Wert des ersten Eintrags auf, bei dem die kumulative

Verteilung höher ist als das Perzentil, das Sie mit dem Parameter `numeric_literal` .

Die Werte werden nach Rowset oder Partition gruppiert, wie von der `WITHIN GROUP` Klausel angegeben.

Die Funktion `PERCENTILE_CONT` ähnelt der Funktion `PERCENTILE_DISC` , gibt jedoch den Durchschnitt der Summe des ersten übereinstimmenden Eintrags und des nächsten Eintrags zurück.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet"
FROM Employee;
```

Um den genauen Wert aus der Zeile zu ermitteln, der dem 0,5-Perzentil entspricht oder diesen überschreitet, übergeben Sie das Perzentil als numerisches Literal in der Funktion `PERCENTILE_DISC` . Die Spalte Percentile Discreet in einer Ergebnismenge listet den Wert der Zeile auf, bei der die kumulative Verteilung höher ist als das angegebene Perzentil.

BusinessEntityID	Berufsbezeichnung	SickLeaveHours	Kumulative Verteilung	Perzentil diskret
272	Anwendungsspezialist	55	0,25	**56**
268	Anwendungsspezialist	56	0,75	**56**
269	Anwendungsspezialist	56	0,75	**56**
267	Anwendungsspezialist	57	1	**56**

Um die Berechnung auf einer Menge von Werten zu `PERCENTILE_CONT` , verwenden Sie die Funktion `PERCENTILE_CONT` . Die Spalte "Percentile Continuous" in den Ergebnissen listet den Durchschnittswert der Summe des Ergebniswerts und des nächsthöheren Übereinstimmungswerts auf.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet",
       PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Continuous"
FROM Employee;
```

BusinessEntityID	Berufsbezeichnung	SickLeaveHours	Kumulative Verteilung	Perzentil diskret	Perzentil kontinuierli
272	Anwendungsspezialist	55	0,25	56	**56**

BusinessEntityID	Berufsbezeichnung	SickLeaveHours	Kumulative Verteilung	Perzentil diskret	Perzentil kontinuierli
268	Anwendungsspezialist	56	0,75	56	**56**
269	Anwendungsspezialist	56	0,75	56	**56**
267	Anwendungsspezialist	57	1	56	**56**

Funktionen (Analyse) online lesen: https://riptutorial.com/de/sql/topic/8811/funktionen--analyse-

Kapitel 28: Funktionen (Skalar / Einzelne Reihe)

Einführung

SQL bietet mehrere integrierte Skalarfunktionen. Jede Skalarfunktion nimmt einen Wert als Eingabe und gibt einen Wert als Ausgabe für jede Zeile in einer Ergebnismenge zurück.

Sie verwenden Skalarfunktionen, wenn ein Ausdruck in einer T-SQL-Anweisung zulässig ist.

Syntax

- CAST (Ausdruck AS data_type [(Länge)])
- CONVERT (Datentyp [(Länge)], Ausdruck [, Stil])
- PARSE (string_value AS data_type [USING culture])
- DATENAME (Datumsteil, Datum)
- VERABREDUNG BEKOMMEN ( )
- DATEDIFF (Datumsteil, Startdatum, Enddatum)
- DATEADD (Datumsteil, Nummer, Datum)
- CHOOSE (Index, Wert_1, Wert_2 [, Wert_n])
- IIF (boolean_expression, true_value, false_value)
- SIGN (numerischer_Ausdruck)
- POWER (float_expression, y)

Bemerkungen

Skalar- oder Einzelzeilenfunktionen werden verwendet, um jede Datenzeile in der Ergebnismenge zu bearbeiten, im Gegensatz zu Aggregatfunktionen, die die gesamte Ergebnismenge bearbeiten.

Es gibt zehn Arten von Skalarfunktionen.

1. Konfigurationsfunktionen bieten Informationen zur Konfiguration der aktuellen SQL-Instanz.
2. Konvertierungsfunktionen konvertieren Daten in den richtigen Datentyp für eine bestimmte Operation. Diese Arten von Funktionen können beispielsweise Informationen umformatieren, indem sie eine Zeichenfolge in ein Datum oder eine Zahl konvertieren, um den Vergleich zweier verschiedener Typen zu ermöglichen.
3. Datums- und Uhrzeitfunktionen bearbeiten Felder, die Datums- und Uhrzeitwerte enthalten. Sie können numerische Werte, Datums- oder Zeichenfolgenwerte zurückgeben. Sie können beispielsweise eine Funktion verwenden, um den aktuellen Wochentag oder das aktuelle Jahr oder nur das Jahr vom Datum abzurufen.

Die von Datums- und Uhrzeitfunktionen zurückgegebenen Werte hängen von Datum und Uhrzeit ab, die für das Betriebssystem des Computers festgelegt sind, auf dem die SQL-Instanz ausgeführt wird.

4. Logische Funktion, die Operationen mit logischen Operatoren ausführt. Es wertet eine Reihe von Bedingungen aus und gibt ein einzelnes Ergebnis zurück.
5. Mathematische Funktionen führen mathematische Operationen oder Berechnungen an numerischen Ausdrücken durch. Diese Art von Funktion gibt einen einzelnen numerischen Wert zurück.
6. Metadatenfunktionen rufen Informationen zu einer angegebenen Datenbank ab, z. B. ihren Namen und Datenbankobjekte.
7. Sicherheitsfunktionen enthalten Informationen, die Sie zum Verwalten der Sicherheit einer Datenbank verwenden können, z. B. Informationen zu Datenbankbenutzern und -rollen.
8. Stringfunktionen führen Operationen mit Stringwerten aus und geben entweder numerische Werte oder Stringwerte zurück.

Mit Stringfunktionen können Sie beispielsweise Daten kombinieren, einen Teilstring extrahieren, Strings vergleichen oder einen String in Groß- oder Kleinbuchstaben konvertieren.

9. Systemfunktionen führen Vorgänge aus und geben Informationen zu Werten, Objekten und Einstellungen für die aktuelle SQL-Instanz zurück
10. Systemstatistikfunktionen liefern verschiedene Statistiken über die aktuelle SQL-Instanz. So können Sie beispielsweise die aktuellen Leistungsniveaus des Systems überwachen.

Examples

Charakteränderungen

Die Funktionen zur Zeichenänderung umfassen das Konvertieren von Zeichen in Groß- oder Kleinbuchstaben, das Konvertieren von Zahlen in formatierte Zahlen, das Durchführen von Zeichenmanipulationen usw.

Die `lower(char)` -Funktion konvertiert den angegebenen Zeichenparameter in Kleinbuchstaben.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

würde den Nachnamen des Kunden zurückgeben, der von "SMITH" in "Smith" geändert wurde.

Datum und Uhrzeit

In SQL verwenden Sie Datums- und Uhrzeitdatentypen zum Speichern von Kalenderinformationen. Zu diesen Datentypen gehören Uhrzeit, Datum, Smalldatetime, Datum und Uhrzeit, Datum und Uhrzeit sowie Datum und Uhrzeit. Jeder Datentyp hat ein bestimmtes Format.

Datentyp	Format
Zeit	hh: mm: ss [.nnnnnnn]
Datum	JJJJ-MM-TT

Datentyp	Format
Kleinzeit	JJJJ-MM-TT hh: mm: ss
Terminzeit	JJJJ-MM-TT hh: mm: ss [.nnn]
datetime2	JJJJ-MM-TT hh: mm: ss [.nnnnnnn]
datetimeoffset	JJJJ-MM-TThh: mm: ss [.nnnnnnn] [+/-] hh: mm

Die `DATENAME` Funktion gibt den Namen oder Wert eines bestimmten Teils des Datums zurück.

```
SELECT DATENAME (weekday,'2017-01-14') as Datename
```

Datename
Samstag

Mit der Funktion `GETDATE` können Sie das aktuelle Datum und die Uhrzeit des Computers ermitteln, auf dem die aktuelle SQL-Instanz ausgeführt wird. Diese Funktion beinhaltet nicht die Zeitzonenverschiebung.

```
SELECT GETDATE() as Systemdate
```

Systemdatum
2017-01-14 11: 11: 47.7230728

Die `DATEDIFF` Funktion gibt die Differenz zwischen zwei Datumsangaben zurück.

In der Syntax ist datepart der Parameter, der angibt, welchen Teil des Datums Sie zur Differenzberechnung verwenden möchten. Der Datumsteil kann Jahr, Monat, Woche, Tag, Stunde, Minute, Sekunde oder Millisekunde sein. Sie geben dann das Startdatum im Parameter "Startdatum" und das Enddatum im Parameter "Enddatum" an, für das Sie die Differenz ermitteln möchten.

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
```

SalesOrderID	Bearbeitungszeit
43659	7
43660	7
43661	7

SalesOrderID	Bearbeitungszeit
43662	7

Mit der Funktion `DATEADD` können Sie einem Teil eines bestimmten Datums ein Intervall hinzufügen.

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays
```

Hinzugefügt20MoreDays
2017-02-03 00: 00: 00.000

Konfigurations- und Konvertierungsfunktion

Ein Beispiel für eine Konfigurationsfunktion in SQL ist die Funktion `@@SERVERNAME` . Diese Funktion gibt den Namen des lokalen Servers an, auf dem SQL ausgeführt wird.

```
SELECT @@SERVERNAME AS 'Server'
```

Server
SQL064

In SQL erfolgen die meisten Datenkonvertierungen implizit ohne Eingreifen des Benutzers.

Um Konvertierungen durchzuführen, die nicht implizit abgeschlossen werden können, können Sie die Funktionen `CAST` oder `CONVERT` verwenden.

Die `CAST` Funktionssyntax ist einfacher als die `CONVERT` Funktionssyntax, ist jedoch in `CONVERT` Funktion eingeschränkt.

In hier verwenden wir die beiden `CAST` und `CONVERT` Funktionen der Datumsdatentyp zum konvertieren `varchar` Datentyp.

Die `CAST` Funktion verwendet immer die Standardstileinstellung. Beispielsweise werden Datum und Uhrzeit im Format JJJJ-MM-TT dargestellt.

Die `CONVERT` Funktion verwendet den von Ihnen angegebenen Datums- und `CONVERT` . In diesem Fall gibt 3 das Datumsformat TT / MM / JJ an.

```
USE AdventureWorks2012
GO
SELECT FirstName + ' ' + LastName + ' was hired on ' +
       CAST(HireDate AS varchar(20)) AS 'Cast',
       FirstName + ' ' + LastName + ' was hired on ' +
       CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
```

```
ON p.BusinessEntityID = e.BusinessEntityID
GO
```

Besetzung	Konvertieren
David Hamiltion wurde am 2003-02-04 eingestellt	David Hamiltion wurde am 02.04.03 eingestellt

Ein weiteres Beispiel für eine Konvertierungsfunktion ist die `PARSE` Funktion. Diese Funktion konvertiert eine Zeichenfolge in einen angegebenen Datentyp.

In der Syntax für die Funktion geben Sie die zu konvertierende Zeichenfolge, das `AS` Schlüsselwort und dann den erforderlichen Datentyp an. Optional können Sie auch die Kultur angeben, in der der Zeichenfolgewert formatiert werden soll. Wenn Sie dies nicht angeben, wird die Sprache für die Sitzung verwendet.

Wenn der Zeichenfolgenwert nicht in ein Zahlen-, Datums- oder Zeitformat konvertiert werden kann, führt dies zu einem Fehler. Sie müssen dann `CAST` oder `CONVERT` für die Konvertierung verwenden.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'
```

Datum auf Englisch
2012-08-13 00: 00: 00.0000000

Logische und mathematische Funktion

SQL hat zwei logische Funktionen - `CHOOSE` und `IIF` .

Die `CHOOSE` Funktion gibt ein Element aus einer Liste von Werten basierend auf seiner Position in der Liste zurück. Diese Position wird durch den Index angegeben.

In der Syntax gibt der Indexparameter das Element an und ist eine ganze Zahl oder ganze Zahl. Der Parameter val_1… val_n gibt die Liste der Werte an.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;
```

Ergebnis
Der Umsatz

In diesem Beispiel verwenden Sie die Funktion `CHOOSE` , um den zweiten Eintrag in einer Liste von Abteilungen zurückzugeben.

Die `IIF` Funktion gibt basierend auf einer bestimmten Bedingung einen von zwei Werten zurück. Wenn die Bedingung wahr ist, wird ein wahrer Wert zurückgegeben. Andernfalls wird ein falscher Wert zurückgegeben.

In der Syntax gibt der Parameter boolean_expression den boolean-Ausdruck an. Der Parameter true_value gibt den Wert an, der zurückgegeben werden soll, wenn der boolean_expression-Wert als true ausgewertet wird, und der Parameter false_value gibt den Wert an, der zurückgegeben werden soll, wenn der boolean_expression-Wert false ergibt.

```
SELECT BusinessEntityID, SalesYTD,
       IIF(SalesYTD > 200000, 'Bonus', 'No Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO
```

BusinessEntityID	SalesYTD	Bonus?
274	559697.5639	Bonus
275	3763178.1787	Bonus
285	172524.4512	Kein Bonus

In diesem Beispiel verwenden Sie die IIF-Funktion, um einen von zwei Werten zurückzugeben. Wenn der Umsatz eines Vertriebsmitarbeiters bis zum Jahresende über 200.000 liegt, hat er Anspruch auf einen Bonus. Werte unter 200.000 bedeuten, dass die Mitarbeiter keine Boni erhalten.

SQL enthält mehrere mathematische Funktionen, mit denen Sie Berechnungen zu Eingabewerten durchführen und numerische Ergebnisse zurückgeben können.

Ein Beispiel ist die `SIGN` Funktion, die einen Wert zurückgibt, der das Vorzeichen eines Ausdrucks angibt. Der Wert von -1 zeigt einen negativen Ausdruck an, der Wert von +1 einen positiven Ausdruck und 0 einen Nullwert.

```
SELECT SIGN(-20) AS 'Sign'
```

Zeichen
-1

Im Beispiel ist die Eingabe eine negative Zahl, daher wird im Ergebnisbereich das Ergebnis -1 aufgeführt.

Eine weitere mathematische Funktion ist die `POWER` Funktion. Diese Funktion liefert den Wert eines Ausdrucks, der mit einer bestimmten Potenz erzeugt wird.

In der Syntax gibt der Parameter float_expression den Ausdruck an, und der Parameter y gibt die Potenz an, um die der Ausdruck erhöht werden soll.

```
SELECT POWER(50, 3) AS Result
```

Ergebnis
125000

Funktionen (Skalar / Einzelne Reihe) online lesen:
https://riptutorial.com/de/sql/topic/6898/funktionen--skalar---einzelne-reihe-

Kapitel 29: Gespeicherte Prozeduren

Bemerkungen

Gespeicherte Prozeduren sind in der Datenbank gespeicherte SQL-Anweisungen, die in Abfragen ausgeführt oder aufgerufen werden können. Die Verwendung einer gespeicherten Prozedur ermöglicht die Kapselung komplizierter oder häufig verwendeter Logik und verbessert die Abfrageleistung durch Verwendung zwischengespeicherter Abfragepläne. Sie können jeden Wert zurückgeben, den eine Standardabfrage zurückgeben kann.

Andere Vorteile gegenüber dynamischen SQL-Ausdrücken sind auf Wikipeida aufgeführt .

Examples

Erstellen Sie eine gespeicherte Prozedur, und rufen Sie sie auf

Gespeicherte Prozeduren können über eine Datenbankverwaltungs-GUI (SQL Server-Beispiel) oder über eine SQL-Anweisung wie folgt erstellt werden:

```
-- Define a name and parameters
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- Define the query to be run
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

Aufruf der Prozedur:

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- Or
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- Or
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

Gespeicherte Prozeduren online lesen: https://riptutorial.com/de/sql/topic/1701/gespeicherte-prozeduren

Kapitel 30: GRANT und REVOKE

Syntax

- GRANT [privileg1] [, [privileg2] ...] ON [Tabelle] TO [grantee1] [, [grantee2] ...] [MIT GRANT OPTION]
- REVOKE [Privileg1] [, [Privileg2] ...] ON [Tabelle] FROM [Grantee1] [, [Grantee2] ...]

Bemerkungen

Erteilen Sie Berechtigungen für Benutzer. Wenn die `WITH GRANT OPTION` angegeben ist, erhält der Empfänger außerdem die Berechtigung, die angegebene Berechtigung zu erteilen oder zuvor erteilte Berechtigungen zu widerrufen.

Examples

Berechtigungen erteilen / entziehen

```
GRANT SELECT, UPDATE
ON Employees
TO User1, User2;
```

Erteilen Sie `User1` und `User2` Berechtigung, `SELECT` und `UPDATE` Vorgänge für die Tabelle `Employees` auszuführen.

```
REVOKE SELECT, UPDATE
ON Employees
FROM User1, User2;
```

Widerrufen Sie von `User1` und `User2` die Berechtigung zum Ausführen von `SELECT` und `UPDATE` Operationen für die Tabelle Employees.

GRANT und REVOKE online lesen: https://riptutorial.com/de/sql/topic/5574/grant-und-revoke

Kapitel 31: GRUPPIERE NACH

Einführung

Die Ergebnisse einer SELECT-Abfrage können mithilfe der `GROUP BY` Anweisung nach einer oder mehreren Spalten gruppiert werden: Alle Ergebnisse mit dem gleichen Wert in den gruppierten Spalten werden zusammengefasst. Dies erzeugt eine Tabelle mit Teilergebnissen anstelle eines Ergebnisses. GROUP BY kann in Verbindung mit Aggregationsfunktionen mit der Anweisung `HAVING` werden, um zu definieren, wie nicht gruppierte Spalten aggregiert werden.

Syntax

- GRUPPIERE NACH {
  Spaltenausdruck
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<group_by_expression> [, ... n])
  | GROUPING SETS ([, ... n])
  | () - berechnet die Gesamtsumme
  } [, ... n]

- <group_by_expression> :: =
  Spaltenausdruck
  | (Spaltenausdruck [, ... n])

- <Gruppierungssatz> :: =
  () - berechnet die Gesamtsumme
  | <grouping_set_item>
  | (<grouping_set_item> [, ... n])

- <grouping_set_item> :: =
  <group_by_expression>
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<group_by_expression> [, ... n])

Examples

USE GROUP BY COUNT die Anzahl der Zeilen für jeden eindeutigen Eintrag in einer bestimmten Spalte

Angenommen, Sie möchten für einen bestimmten Wert in einer Spalte Zählungen oder Zwischensummen generieren.

Angesichts dieser Tabelle "Westerosians":

Name	GreatHouseAllegience
Arya	Stark
Cercei	Lannister
Myrcella	Lannister
Yara	Greyjoy
Catelyn	Stark
Sansa	Stark

Ohne GROUP BY wird COUNT einfach eine Gesamtanzahl von Zeilen zurückgeben:

```
SELECT Count(*) Number_of_Westerosians
FROM Westerosians
```

kehrt zurück...

Anzahl_der_Westerosianer
6

Durch das Hinzufügen von GROUP BY können wir die Benutzer für jeden Wert in einer bestimmten Spalte ZÄHLEN, um die Anzahl der Personen in einem bestimmten Großen Haus zurückzugeben.

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

kehrt zurück...

Haus	Anzahl_der_Westerosianer
Stark	3
Greyjoy	1
Lannister	2

Es ist üblich, GROUP BY mit ORDER BY zu kombinieren, um die Ergebnisse nach der größten oder kleinsten Kategorie zu sortieren:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

```
ORDER BY Number_of_Westerosians Desc
```

kehrt zurück...

Haus	Anzahl_der_Westerosianer
Stark	3
Lannister	2
Greyjoy	1

Filtern Sie die GROUP BY-Ergebnisse mit einer HAVING-Klausel

Eine HAVING-Klausel filtert die Ergebnisse eines GROUP BY-Ausdrucks. Hinweis: Die folgenden Beispiele verwenden die Bibliotheksbeispieldatenbank .

Beispiele:

Gib alle Autoren zurück, die mehr als ein Buch geschrieben haben (Live-Beispiel).

```
SELECT
  a.Id,
  a.Name,
  COUNT(*) BooksWritten
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
GROUP BY
  a.Id,
  a.Name
HAVING COUNT(*) > 1    -- equals to HAVING BooksWritten > 1
;
```

Alle Bücher zurückgeben, die mehr als drei Autoren haben (Live-Beispiel).

```
SELECT
  b.Id,
  b.Title,
  COUNT(*) NumberOfAuthors
FROM BooksAuthors ba
  INNER JOIN Books b ON b.id = ba.bookid
GROUP BY
  b.Id,
  b.Title
HAVING COUNT(*) > 3    -- equals to HAVING NumberOfAuthors > 3
;
```

Grundlegendes GROUP BY-Beispiel

Es ist möglicherweise einfacher, wenn Sie sich GROUP OF BY als "für jeden" zur Erklärung ansehen. Die Abfrage unten:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
GROUP BY EmpID
```

sagt:

"Gib mir die Summe der MonthlySalary's **für jede** EmpID"

Wenn also Ihr Tisch so aussah:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Ergebnis:

```
+-+---+
|1|200|
+-+---+
|2|300|
+-+---+
```

Sum würde scheinbar nichts tun, da die Summe einer Zahl diese Zahl ist. Andererseits, wenn es so aussah:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|1    |300          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Ergebnis:

```
+-+---+
|1|500|
+-+---+
|2|300|
+-+---+
```

Dann wäre es, weil es zwei EmpID 1 gibt, die zusammenzufassen sind.

ROLAP-Aggregation (Data Mining)

Beschreibung

Der SQL-Standard bietet zwei zusätzliche Aggregatoperatoren. Diese verwenden den polymorphen Wert "ALL", um die Menge aller Werte anzugeben, die ein Attribut annehmen kann. Die zwei Operatoren sind:

- `with data cube` werden alle möglichen Kombinationen als die Argumentattribute der Klausel bereitgestellt.
- `with roll up` werden die Aggregate bereitgestellt, die sich aus der Reihenfolge der Attribute von links nach rechts ergeben, verglichen mit der Auflistung im Argument der Klausel.

SQL-Standardversionen, die diese Funktionen unterstützen: 1999,2003,2006,2008,2011.

Beispiele

Betrachten Sie diese Tabelle:

Essen	Marke	Gesamtmenge
Pasta	Brand1	100
Pasta	Brand2	250
Pizza	Brand2	300

Mit Würfel

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with cube
```

Essen	Marke	Gesamtmenge
Pasta	Brand1	100
Pasta	Brand2	250
Pasta	ALLES	350
Pizza	Brand2	300
Pizza	ALLES	300
ALLES	Brand1	100
ALLES	Brand2	550
ALLES	ALLES	650

Mit aufrollen

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with roll up
```

Essen	Marke	Gesamtmenge
Pasta	Brand1	100
Pasta	Brand2	250
Pizza	Brand2	300
Pasta	ALLES	350
Pizza	ALLES	300
ALLES	ALLES	650

GRUPPIERE NACH online lesen: https://riptutorial.com/de/sql/topic/627/gruppiere-nach

Kapitel 32: Indizes

Einführung

Indizes sind eine Datenstruktur, die Zeiger auf den Inhalt einer Tabelle enthält, die in einer bestimmten Reihenfolge angeordnet ist, um die Datenbank bei der Optimierung von Abfragen zu unterstützen. Sie ähneln dem Buchindex, bei dem die Seiten (Tabellenzeilen) anhand ihrer Seitennummer indiziert werden.

Es gibt verschiedene Arten von Indizes, die für eine Tabelle erstellt werden können. Wenn für die in der WHERE-Klausel, JOIN-Klausel oder ORDER BY-Klausel einer Abfrage verwendeten Spalten ein Index vorhanden ist, kann die Abfrageleistung erheblich verbessert werden.

Bemerkungen

Indizes sind eine Möglichkeit, Leseanfragen zu beschleunigen, indem die Zeilen einer Tabelle nach einer Spalte sortiert werden.

Die Auswirkungen eines Index sind für kleine Datenbanken wie im Beispiel nicht erkennbar. Wenn jedoch eine große Anzahl von Zeilen vorhanden ist, kann dies die Leistung erheblich verbessern. Anstatt jede Zeile der Tabelle zu überprüfen, kann der Server eine binäre Suche nach dem Index durchführen.

Der Kompromiss beim Erstellen eines Indexes ist die Schreibgeschwindigkeit und die Datenbankgröße. Das Speichern des Index benötigt Platz. Jedes Mal, wenn ein INSERT ausgeführt wird oder die Spalte aktualisiert wird, muss der Index aktualisiert werden. Dies ist zwar nicht so teuer wie das Durchsuchen der gesamten Tabelle in einer SELECT-Abfrage, ist aber dennoch zu beachten.

Examples

Index erstellen

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

Dadurch wird ein Index für die Spalte *EmployeeId* in der Tabelle *Cars erstellt* . Dieser Index verbessert die Geschwindigkeit von Abfragen, die den Server auffordern, nach Werten in *EmployeeId* zu sortieren oder auszuwählen, z. B. die folgenden:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

Der Index kann wie im Folgenden mehr als eine Spalte enthalten.

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

In diesem Fall wäre der Index für Abfragen hilfreich, die nach allen enthaltenen Spalten sortieren oder auswählen, wenn der Satz von Bedingungen auf dieselbe Weise angeordnet ist. Das bedeutet, dass beim Abrufen der Daten die Zeilen abgerufen werden können, die mithilfe des Index abgerufen werden sollen, anstatt die vollständige Tabelle zu durchsuchen.

Im folgenden Fall würde beispielsweise der zweite Index verwendet.

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

Wenn sich die Reihenfolge unterscheidet, hat der Index jedoch nicht die gleichen Vorteile wie im Folgenden.

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

Der Index ist nicht so hilfreich, da die Datenbank den gesamten Index über alle Werte von EmployeeId und CarID abrufen muss, um herauszufinden, welche Elemente `OwnerId = 17` .

(Der Index kann weiterhin verwendet werden; der Abfrageoptimierer kann jedoch feststellen, dass das Abrufen des Indexes und das Filtern nach `OwnerId` und das Abrufen nur der benötigten Zeilen schneller ist als das Abrufen der vollständigen Tabelle, insbesondere wenn die Tabelle groß ist.)

Gruppierte, eindeutige und sortierte Indizes

Indizes können mehrere Merkmale aufweisen, die entweder beim Erstellen oder durch Ändern vorhandener Indizes festgelegt werden können.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

Die obige SQL-Anweisung erstellt einen neuen Clustered-Index für Mitarbeiter. Clustered-Indizes sind Indizes, die die tatsächliche Struktur der Tabelle bestimmen. Die Tabelle selbst wird nach der Indexstruktur sortiert. Das heißt, es kann höchstens einen Clustered-Index für eine Tabelle geben. Wenn für die Tabelle bereits ein gruppierter Index vorhanden ist, schlägt die obige Anweisung fehl. (Tabellen ohne Clustered-Indizes werden auch als Heap-Speicher bezeichnet.)

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

Dadurch wird ein eindeutiger Index für die Spalte *E-Mail* in der Tabelle *Customers erstellt* . Dieser Index sowie die Beschleunigung von Abfragen wie ein normaler Index erzwingen auch, dass jede E-Mail-Adresse in dieser Spalte eindeutig ist. Wenn eine Zeile mit einem nicht eindeutigen *E - Mail-* Wert eingefügt oder aktualisiert wird, *schlägt* das Einfügen oder Aktualisieren standardmäßig fehl.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

Dadurch wird ein Index für Kunden erstellt, der auch eine Tabellenbeschränkung erstellt, wonach die EmployeeID eindeutig sein muss. (Dies schlägt fehl, wenn die Spalte derzeit nicht eindeutig ist - in diesem Fall, wenn Mitarbeiter eine ID gemeinsam nutzen.)

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

Dadurch wird ein Index erstellt, der in absteigender Reihenfolge sortiert ist. Standardmäßig sind Indizes (zumindest im MSSQL-Server) aufsteigend, dies kann jedoch geändert werden.

Einfügen mit einem eindeutigen Index

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

Dies schlägt fehl, wenn für die *E-Mail*- Spalte von *Kunden* ein eindeutiger Index festgelegt ist. Für diesen Fall kann jedoch ein alternatives Verhalten definiert werden:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

SAP ASE: Index löschen

Dieser Befehl löscht den Index in der Tabelle. Es funktioniert auf einem `SAP ASE` Server.

Syntax:

```
DROP INDEX [table name].[index name]
```

Beispiel:

```
DROP INDEX Cars.index_1
```

Sortierter Index

Wenn Sie einen Index verwenden, der so sortiert ist, wie Sie ihn abrufen würden, führt die `SELECT` Anweisung beim Abrufen keine zusätzliche Sortierung durch.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

Wenn Sie die Abfrage ausführen

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

Das Datenbanksystem führt keine zusätzliche Sortierung durch, da es in dieser Reihenfolge eine Indexsuche durchführen kann.

Einen Index löschen oder ihn deaktivieren und neu erstellen

```
DROP INDEX ix_cars_employee_id ON Cars;
```

Wir können den Befehl `DROP` , um unseren Index zu löschen. In diesem Beispiel werden wir `DROP` den Index *ix_cars_employee_id* auf dem Tisch *Autos* genannt.

Dadurch wird der Index vollständig gelöscht. Wenn der Index in einem Cluster zusammengefasst ist, wird das Clustering entfernt. Es kann nicht neu erstellt werden, ohne den Index neu zu erstellen, was langsam und rechenintensiv sein kann. Alternativ kann der Index deaktiviert werden:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

Dadurch kann die Tabelle die Struktur zusammen mit den Metadaten zum Index beibehalten.

Kritisch bleibt dabei die Indexstatistik erhalten, so dass die Änderung leicht ausgewertet werden kann. Bei Bedarf kann der Index später erneut erstellt werden, anstatt ihn vollständig neu zu erstellen.

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Eindeutiger Index, der NULL erlaubt

```
CREATE UNIQUE INDEX idx_license_id
   ON Person(DrivingLicenseID) WHERE DrivingLicenseID IS NOT NULL
GO
```

Dieses Schema ermöglicht eine 0..1-Beziehung - Personen können keinen oder einen Führerschein haben und jeder Führerschein kann nur einer Person gehören

Index neu erstellen

Im Laufe der Zeit können B-Tree-Indizes aufgrund der Aktualisierung / Löschung / Einfügung von Daten fragmentiert werden. In der SQLServer-Terminologie können wir interne (Indexseite, die halb leer ist) und extern (logische Seitenreihenfolge entspricht nicht der physischen Reihenfolge) haben. Die Neuerstellung des Index ist dem Löschen und Neuerstellen des Index sehr ähnlich.

Wir können einen Index mit neu erstellen

```
ALTER INDEX index_name REBUILD;
```

Standardmäßig ist der Index für die Neuerstellung ein Offline-Vorgang, durch den die Tabelle gesperrt und DML dagegen verhindert wird. In vielen RDBMS ist jedoch eine Online-Wiederherstellung möglich. Einige DB-Anbieter bieten auch Alternativen zur `REORGANIZE` von Indizes an, wie z. B. `REORGANIZE` (SQLServer) oder `COALESCE` / `SHRINK SPACE` (Oracle).

Clustered-Index

Bei Verwendung eines Clustered-Index werden die Zeilen der Tabelle nach der Spalte sortiert, auf die der Clustered-Index angewendet wird. Daher kann es nur einen Clusterindex für die Tabelle geben, da Sie die Tabelle nicht nach zwei verschiedenen Spalten sortieren können.

Im Allgemeinen ist es am besten, einen Clustered-Index zu verwenden, wenn Lesevorgänge für

große Datentabellen ausgeführt werden. Der Vorteil des Clustered-Index liegt beim Schreiben in eine Tabelle, und Daten müssen neu organisiert (sortiert) werden.

Ein Beispiel zum Erstellen eines gruppierten Index für eine Tabelle Employees in der Spalte Employee_Surname:

```
CREATE CLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Nicht gruppierter Index

Nicht gruppierte Indizes werden getrennt von der Tabelle gespeichert. Jeder Index in dieser Struktur enthält einen Zeiger auf die Zeile in der Tabelle, die er darstellt.

Diese Zeiger werden Zeilenlokatoren genannt. Die Struktur des Zeilenlokators hängt davon ab, ob die Datenseiten in einem Heap oder in einer Clustertabelle gespeichert sind. Bei einem Heap ist ein Zeilenlokator ein Zeiger auf die Zeile. Bei einer gruppierten Tabelle ist der Zeilenlokator der gruppierte Indexschlüssel.

Ein Beispiel zum Erstellen eines nicht gruppierten Index für die Tabelle Employees und die Spalte Employee_Surname:

```
CREATE NONCLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Es können mehrere nicht gruppierte Indizes für die Tabelle vorhanden sein. Die Leseoperationen sind bei nicht gruppierten Indizes im Allgemeinen langsamer als bei gruppierten Indizes, da Sie zuerst zum Index und dann zur Tabelle gehen müssen. Es gibt jedoch keine Einschränkungen bei Schreibvorgängen.

Teilweiser oder gefilterter Index

SQL Server und SQLite ermöglichen das Erstellen von Indizes, die nicht nur eine Teilmenge von Spalten enthalten, sondern auch eine Teilmenge von Zeilen.

Betrachten Sie eine konstant wachsende Anzahl von Bestellungen, wobei `order_state_id` gleich beendete (2) ist, und eine stabile Anzahl von Bestellungen, wobei `order_state_id equal` gestartet ist (1).

Wenn Ihr Unternehmen Fragen wie diese verwendet:

```
SELECT id, comment
  FROM orders
 WHERE order_state_id =  1
   AND product_id = @some_value;
```

Mit der teilweisen Indizierung können Sie den Index einschränken, einschließlich nur der noch nicht abgeschlossenen Aufträge:

```
CREATE INDEX Started_Orders
```

```
   ON orders(product_id)
WHERE order_state_id = 1;
```

Dieser Index ist kleiner als ein ungefilterter Index, wodurch Platz gespart und die Aktualisierungskosten des Index reduziert werden.

Indizes online lesen: https://riptutorial.com/de/sql/topic/344/indizes

Kapitel 33: Informationsschema

Examples

Grundlegende Informationsschemasuche

Eine der nützlichsten Abfragen für Endbenutzer großer RDBMS ist die Suche nach einem Informationsschema.

Eine solche Abfrage ermöglicht Benutzern das schnelle Auffinden von Datenbanktabellen, die interessierende Spalten enthalten, z. B. wenn versucht wird, Daten aus zwei Tabellen indirekt über eine dritte Tabelle zu verknüpfen, ohne zu wissen, welche Tabellen Schlüssel oder andere nützliche Spalten enthalten, die mit den Zieltabellen gemeinsam sind .

Bei Verwendung von T-SQL für dieses Beispiel kann das Informationsschema einer Datenbank wie folgt durchsucht werden:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

Das Ergebnis enthält eine Liste übereinstimmender Spalten, deren Tabellennamen und andere nützliche Informationen.

Informationsschema online lesen: https://riptutorial.com/de/sql/topic/3151/informationsschema

Kapitel 34: IN-Klausel

Examples

Einfache IN-Klausel

Datensätze mit **einer** der angegebenen `id`

```
select *
from products
where id in (1,8,3)
```

Die obige Abfrage ist gleich

```
select *
from products
where id = 1
   or id = 8
   or id = 3
```

IN-Klausel mit einer Unterabfrage verwenden

```
SELECT *
FROM customers
WHERE id IN (
    SELECT DISTINCT customer_id
    FROM orders
);
```

Die obigen Informationen geben Ihnen alle Kunden, die Bestellungen im System haben.

IN-Klausel online lesen: https://riptutorial.com/de/sql/topic/3169/in-klausel

Kapitel 35: Kennung

Einführung

In diesem Thema werden Bezeichner behandelt, z. B. Syntaxregeln für Namen von Tabellen, Spalten und anderen Datenbankobjekten.

Gegebenenfalls sollten die Beispiele Variationen abdecken, die von verschiedenen SQL-Implementierungen verwendet werden, oder die SQL-Implementierung des Beispiels angeben.

Examples

Nicht zitierte Bezeichner

Bezeichner ohne Anführungszeichen können Buchstaben (`a` - `z`), Ziffern (`0` - `9`) und Unterstrich (`_`) verwenden und müssen mit einem Buchstaben beginnen.

Abhängig von der SQL-Implementierung und / oder den Datenbankeinstellungen können andere Zeichen erlaubt sein, einige sogar als erstes Zeichen, z

- MS SQL: `@` , `$` , `#` und andere Unicode-Buchstaben *(Quelle)*
- MySQL: `$` *(Quelle)*
- Oracle: `$` , `#` und andere Buchstaben aus dem Datenbankzeichensatz *(Quelle)*
- PostgreSQL: `$` und andere Unicode-Buchstaben *(Quelle)*

Nicht notierte Bezeichner unterscheiden nicht zwischen Groß- und Kleinschreibung. Wie dies gehandhabt wird, hängt stark von der SQL-Implementierung ab:

- MS SQL: Die Groß- und Kleinschreibung wird durch den Datenbankzeichensatz festgelegt.
- MySQL: Die Groß- und Kleinschreibung hängt von der Datenbankeinstellung und dem zugrunde liegenden Dateisystem ab.
- Oracle: Wird in Großbuchstaben konvertiert und dann wie in Bezeichner in Anführungszeichen behandelt.
- PostgreSQL: Wird in Kleinbuchstaben konvertiert und dann als Bezeichner in Anführungszeichen behandelt.
- SQLite: Groß- und Kleinschreibung; Groß- und Kleinschreibung nur für ASCII-Zeichen.

Kennung online lesen: https://riptutorial.com/de/sql/topic/9677/kennung

Kapitel 36: Kreuz anwenden, außen anwenden

Examples

GRUNDLAGEN VON CROSS APPLY und OUTER APPLY

Anwenden wird verwendet, wenn bei der Tabellenwertfunktion der rechte Ausdruck verwendet wird.

Erstellen Sie eine Abteilungstabelle, die Informationen zu Abteilungen enthält. Erstellen Sie anschließend eine Employee-Tabelle, die Informationen zu den Mitarbeitern enthält. Bitte beachten Sie, dass jeder Mitarbeiter zu einer Abteilung gehört. Daher verfügt die Tabelle Employee über die referenzielle Integrität der Abteilungstabelle.

Die erste Abfrage wählt Daten aus der Abteilungstabelle aus und wertet CROSS APPLY aus, um die Employee-Tabelle für jeden Datensatz der Abteilungstabelle auszuwerten. Die zweite Abfrage verbindet einfach die Department-Tabelle mit der Employee-Tabelle, und alle passenden Datensätze werden erzeugt.

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
```

Wenn Sie sich die Ergebnisse ansehen, die sie produziert haben, ist es genau dieselbe Ergebnismenge. Wie unterscheidet es sich von einem JOIN und wie hilft es, effizientere Abfragen zu schreiben.

Die erste Abfrage in Skript Nr. 2 wählt Daten aus der Abteilungstabelle aus und wertet OUTER APPLY aus, um die Employee-Tabelle für jeden Datensatz der Abteilungstabelle auszuwerten. Für die Zeilen, für die in der Employee-Tabelle keine Übereinstimmung vorhanden ist, enthalten diese Zeilen NULL-Werte, wie Sie in den Zeilen 5 und 6 sehen können. Die zweite Abfrage verwendet lediglich einen LEFT-OUTER-JOIN zwischen der Department-Tabelle und der Employee-Tabelle. Wie erwartet gibt die Abfrage alle Zeilen aus der Department-Tabelle zurück. auch für die Zeilen, für die in der Employee-Tabelle keine Übereinstimmung vorhanden ist.

```
SELECT *
FROM Department D
```

```
OUTER APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
GO
```

Obwohl die beiden obigen Abfragen dieselbe Information zurückgeben, ist der Ausführungsplan etwas anders. Aber aus Kostengründen wird es keinen großen Unterschied geben.

Jetzt kommt die Zeit, um zu sehen, wo der APPLY-Operator wirklich benötigt wird. In Skript Nr. 3 erstelle ich eine Tabellenwertfunktion, die DepartmentID als Parameter akzeptiert und alle Mitarbeiter zurückgibt, die dieser Abteilung angehören. Die nächste Abfrage wählt Daten aus der Abteilungstabelle aus und verwendet CROSS APPLY, um die von uns erstellte Funktion zu verbinden. Sie übergibt die DepartmentID für jede Zeile aus dem äußeren Tabellenausdruck (in unserem Fall Department-Tabelle) und wertet die Funktion für jede Zeile ähnlich einer korrelierten Unterabfrage aus. Die nächste Abfrage verwendet OUTER APPLY anstelle von CROSS APPLY. Daher gibt OUTER APPLY im Gegensatz zu CROSS APPLY, bei dem nur korrelierte Daten zurückgegeben wurden, auch nicht korrelierte Daten zurück, wodurch NULL-Werte in die fehlenden Spalten eingefügt werden.

```
CREATE FUNCTION dbo.fn_GetAllEmployeeOfADepartment (@DeptID AS int)
RETURNS TABLE
AS
  RETURN
  (
  SELECT
    *
  FROM Employee E
  WHERE E.DepartmentID = @DeptID
  )
GO
SELECT
  *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
  *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

Wenn Sie sich also fragen, können wir anstelle der obigen Abfragen einen einfachen Join verwenden? Die Antwort lautet NEIN. Wenn Sie in den obigen Abfragen CROSS / OUTER APPLY durch INNER JOIN / LEFT OUTER JOIN ersetzen, die ON-Klausel angeben (etwas wie 1 = 1) und die Abfrage ausführen, erhalten Sie "Die mehrteilige Kennung". D.DepartmentID "konnte nicht gebunden werden." Error. Dies liegt daran, dass sich bei JOINs der Ausführungskontext der äußeren Abfrage vom Ausführungskontext der Funktion (oder einer abgeleiteten Tabelle)

unterscheidet und Sie einen Wert / eine Variable nicht von der äußeren Abfrage als Parameter an die Funktion binden können. Daher ist der APPLY-Operator für solche Abfragen erforderlich.

Kreuz anwenden, außen anwenden online lesen: https://riptutorial.com/de/sql/topic/2516/kreuz-anwenden--au-en-anwenden

Kapitel 37: KÜRZEN

Einführung

Die TRUNCATE-Anweisung löscht alle Daten aus einer Tabelle. Dies ähnelt DELETE ohne Filter, hat jedoch abhängig von der Datenbanksoftware bestimmte Einschränkungen und Optimierungen.

Syntax

- TRUNCATE TABLE Tabellenname;

Bemerkungen

TRUNCATE ist ein DDL-Befehl (Data Definition Language), und als solcher gibt es erhebliche Unterschiede zwischen DELETE (ein Befehl zur Datenmanipulationssprache, DML). Obwohl TRUNCATE ein Mittel zum schnellen Entfernen großer Datensätze aus einer Datenbank sein kann, sollten diese Unterschiede verstanden werden, um zu entscheiden, ob die Verwendung eines TRUNCATE-Befehls in Ihrer speziellen Situation geeignet ist.

- TRUNCATE ist eine Datenseitenoperation. Daher werden DML-Auslöser (ON DELETE), die der Tabelle zugeordnet sind, nicht ausgelöst, wenn Sie eine TRUNCATE-Operation ausführen. Dies spart zwar sehr viel Zeit für umfangreiche Löschvorgänge, Sie müssen jedoch möglicherweise die zugehörigen Daten manuell löschen.
- TRUNCATE gibt den von den gelöschten Zeilen belegten Speicherplatz frei, DELETE gibt Speicherplatz frei
- Wenn die zu schneidende Tabelle Identitätsspalten (MS SQL Server) verwendet, wird der Startwert durch den Befehl TRUNCATE zurückgesetzt. Dies kann zu Problemen der referentiellen Integrität führen
- Abhängig von den vorhandenen Sicherheitsrollen und der verwendeten SQL-Variante verfügen Sie möglicherweise nicht über die erforderlichen Berechtigungen, um einen TRUNCATE-Befehl auszuführen

Examples

Alle Zeilen aus der Employee-Tabelle entfernen

```
TRUNCATE TABLE Employee;
```

Die Verwendung einer abgeschnittenen Tabelle ist häufig besser als die Verwendung von DELETE TABLE, da alle Indizes und Trigger ignoriert und einfach alles entfernt wird.

Tabelle löschen ist eine zeilenbasierte Operation. Das bedeutet, dass jede Zeile gelöscht wird. Tabelle abschneiden ist eine Datenseitenoperation, bei der die gesamte Datenseite neu zugewiesen wird. Wenn Sie eine Tabelle mit einer Million Zeilen haben, wird das Abschneiden der

Tabelle viel schneller erfolgen, als wenn Sie eine Anweisung zum Löschen von Tabellen verwenden würden.

Obwohl wir bestimmte Zeilen mit DELETE löschen können, können wir bestimmte Zeilen nicht TRUNCATE. Wir können nur alle Datensätze gleichzeitig TRUNCATE. Wenn Sie Alle Zeilen löschen und dann einen neuen Datensatz einfügen, wird der Wert des automatisch hinzugefügten Primärschlüssels weiterhin aus dem zuvor eingefügten Wert hinzugefügt. Dabei wird, wie bei Abschneiden, auch der automatisch inkrementelle Primärschlüsselwert zurückgesetzt und beginnt bei 1.

Beachten Sie, dass beim Abschneiden der Tabelle **keine Fremdschlüssel vorhanden sein müssen** . Andernfalls wird ein Fehler angezeigt.

KÜRZEN online lesen: https://riptutorial.com/de/sql/topic/1466/kurzen

Kapitel 38: LIKE Operator

Syntax

- Platzhalter **mit%:** SELECT * FROM [Tabelle] WHERE [Spaltenname] Wie '% Value%'

  Platzhalter **mit _:** SELECT * FROM [Tabelle] WHERE [Spaltenname] Wie 'V_n%'

  Platzhalter mit [charlist]: SELECT * FROM [Tabelle] WHERE [Spaltenname] Wie 'V [abc] n%'

Bemerkungen

Die LIKE-Bedingung in der WHERE-Klausel wird verwendet, um nach Spaltenwerten zu suchen, die dem angegebenen Muster entsprechen. Muster werden gebildet, indem zwei Platzhalterzeichen verwendet werden

- % (Prozentzeichen) - Wird für die Darstellung von null oder mehr Zeichen verwendet
- _ (Unterstrich) - Wird zur Darstellung eines einzelnen Zeichens verwendet

Examples

Übereinstimmung mit offenem Muster

Der an den Anfang oder das Ende (oder beide) einer Zeichenfolge angehängte Platzhalter `%` ermöglicht die Übereinstimmung von 0 oder mehr Zeichen vor oder nach dem Ende des Musters.

Bei Verwendung von '%' in der Mitte können 0 oder mehr Zeichen zwischen den beiden Teilen des Musters übereinstimmen.

Wir werden diese Mitarbeiter-Tabelle verwenden:

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellungsc
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	Schmied	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

Die folgenden Anweisungen stimmen für alle Datensätze mit FName überein, **die die**

Zeichenfolge "Ein" aus der Employees Table enthalten.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellungsd
3	R **auf** ny	Schmied	2462544026	2	1	600	06-08-2015
4	J **ein**	Sanchez	2454124602	1	1	400	23-03-2005

Die folgende Anweisung stimmt mit allen Datensätzen überein, deren Telefonnummer **mit der** Zeichenfolge '246' von Employees **beginnt** .

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellungsd
1	John	Johnson	**246** 8101214	1	1	400	23-03-2005
3	Ronny	Schmied	**246** 2544026	2	1	600	06-08-2015
5	Hilde	Knag	**246** 8021911	2	1	800	01-01-2000

Die folgende Anweisung stimmt mit allen Datensätzen überein, deren Telefonnummer **mit der** Zeichenfolge '11' von Employees **endet** .

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellungsc
2	Sophie	Amudsen	24791002 **11**	1	1	400	11-01-2010
5	Hilde	Knag	24680219 **11**	2	1	800	01-01-2000

Alle Datensätze, bei denen das dritte **Zeichen** "Fname" von Angestellten "n" ist.

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

(Zwei Unterstriche werden vor 'n' verwendet, um die ersten 2 Zeichen zu überspringen.)

Ich würde	FName	LName	Telefonnummer	ManagerId	DepartmentId	Gehalt	Anstellungsd
3	Ronny	Schmied	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Einzelzeichenübereinstimmung

Um die Auswahl einer Anweisung für eine strukturierte Abfragesprache (SQL-SELECT) zu erweitern, können Platzhalterzeichen, das Prozentzeichen (%) und der Unterstrich (_) verwendet werden.

Das Zeichen _ (Unterstrich) kann als Platzhalter für ein einzelnes Zeichen in einer Musterübereinstimmung verwendet werden.

Finden Sie alle Mitarbeiter, deren Fname mit 'j' beginnen und mit 'n' enden und in Fname genau 3 Zeichen haben.

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

_ (Unterstrich) Zeichen können auch mehrmals als Platzhalter für Muster verwendet werden.

Zum Beispiel würde dieses Muster mit "jon", "jan", "jen" usw. übereinstimmen.

Diese Namen werden nicht "jn", "john", "jordan", "justin", "jason", "julian", "jillian", "joann" angezeigt, da in unserer Abfrage ein Unterstrich verwendet wird, der genau überspringen kann Ein Zeichen, das Ergebnis muss aus 3 Zeichen bestehen.

Dieses Muster würde beispielsweise "LaSt", "LoSt", "HaLt" usw. entsprechen.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Übereinstimmung nach Bereich oder Satz

Stimmen Sie jedes einzelne Zeichen innerhalb des angegebenen Bereichs ab (zB: `[af]`) oder setzen Sie (zB: `[abcdef]`)

Dieses Bereichsmuster würde mit "gary" übereinstimmen, aber nicht mit "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

Dieses Set Pattern würde "Mary" entsprechen, aber nicht "Gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[lmnop]ary'
```

Der Bereich oder das Set kann auch negiert werden, indem das ^ -Zeichen vor dem Range oder Set angehängt wird:

Dieses Bereichsmuster würde *nicht* mit "gary" übereinstimmen, sondern mit "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

Dieses Set Pattern würde *nicht* mit "Mary" übereinstimmen, sondern mit "Gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[^lmnop]ary'
```

JEDER gegen ALLE

Übereinstimmung mit:
Mindestens eine Zeichenfolge muss übereinstimmen. In diesem Beispiel muss der Produkttyp entweder "Elektronik", "Bücher" oder "Video" sein.

```
SELECT *
FROM   purchase_table
WHERE  product_type LIKE ANY ('electronics', 'books', 'video');
```

Passen Sie alle an (muss alle Anforderungen erfüllen).
In diesem Beispiel müssen sowohl 'Großbritannien' *als auch* 'London' *und* 'Oststraße' (einschließlich Variationen) übereinstimmen.

```
SELECT *
FROM   customer_table
WHERE  full_address LIKE ALL ('%united kingdom%', '%london%', '%eastern road%');
```

Negative Auswahl:
Verwenden Sie ALL, um alle Elemente auszuschließen.
Dieses Beispiel liefert alle Ergebnisse, bei denen der Produkttyp nicht "Elektronik" und nicht "Bücher" und nicht "Video" ist.

```
SELECT *
FROM   customer_table
WHERE  product_type NOT LIKE ALL ('electronics', 'books', 'video');
```

Suchen Sie nach einer Reihe von Zeichen

Die folgende Anweisung stimmt mit allen Datensätzen überein, die FName haben, die mit einem Buchstaben von A bis F aus der Employees Table beginnen.

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

ESCAPE-Anweisung in der LIKE-Abfrage

Wenn Sie eine Textsuche als `LIKE` -Query implementieren, machen Sie das normalerweise so:

```
SELECT *
```

```
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

(Abgesehen davon, dass Sie `LIKE` nicht zwingend verwenden sollten, wenn Sie die Volltextsuche verwenden können), führt dies zu einem Problem, wenn jemand Text wie "50%" oder "a_b" eingibt.

Also (anstatt zur Volltextsuche zu wechseln), können Sie dieses Problem mit der `LIKE` -escape-Anweisung lösen:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

Das bedeutet, dass `\` jetzt als ESCAPE-Zeichen behandelt wird. Das bedeutet, Sie können jetzt jedem Zeichen in der Zeichenfolge, die Sie suchen, ein `\` voranstellen, und die Ergebnisse werden korrekt angezeigt, auch wenn der Benutzer ein Sonderzeichen wie `%` oder _ eingibt.

z.B

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
     newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

Hinweis: Der obige Algorithmus dient nur zu Demonstrationszwecken. Es funktioniert nicht in Fällen, in denen ein Graphem aus mehreren Zeichen besteht (utf-8). zB `string stringToSearch = "Les Mise\u0301rables";` Sie müssen dies für jedes Graphem tun, nicht für jedes Zeichen. Sie sollten den obigen Algorithmus nicht verwenden, wenn Sie mit asiatischen / ostasiatischen / südasiatischen Sprachen arbeiten. Oder, wenn Sie möchten, dass korrekter Code beginnt, sollten Sie dies für jeden graphemeCluster tun.

Siehe auch ReverseString, eine C # Interviewfrage

Platzhalterzeichen

Platzhalterzeichen werden mit dem SQL-LIKE-Operator verwendet. SQL-Platzhalter werden zum Suchen nach Daten in einer Tabelle verwendet.

Platzhalter in SQL sind:%, _, [charlist], [^ charlist]

% - Ein Ersatz für null oder mehr Zeichen

```
   Eg:  //selects all customers with a City starting with "Lo"
        SELECT * FROM Customers
        WHERE City LIKE 'Lo%';

       //selects all customers with a City containing the pattern "es"
```

```
      SELECT * FROM Customers
       WHERE City LIKE '%es%';
```

_ - Ein Ersatz für ein einzelnes Zeichen

```
Eg://selects all customers with a City starting with any character, followed by "erlin"
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

[charlist] - Legt fest, welche Zeichen übereinstimmen, und **legt die Anzahl** der Zeichen fest

```
Eg://selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[adl]%';

//selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^ charlist] - **Stimmt** nur mit einem Zeichen **überein,** das NICHT in den Klammern angegeben ist

```
Eg://selects all customers with a City starting with a character that is not "a", "p", or "l"
SELECT * FROM Customers
WHERE City LIKE '[^apl]%';

or

SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```

LIKE Operator online lesen: https://riptutorial.com/de/sql/topic/860/like-operator

Kapitel 39: LÖSCHEN

Einführung

Die DELETE-Anweisung wird zum Löschen von Datensätzen aus einer Tabelle verwendet.

Syntax

1. DELETE FROM *TableName* [WHERE- *Bedingung*] [LIMIT- *Anzahl*]

Examples

LÖSCHEN Sie bestimmte Zeilen mit WHERE

Dadurch werden alle Zeilen gelöscht, die den `WHERE` Kriterien entsprechen.

```
DELETE FROM Employees
WHERE FName = 'John'
```

LÖSCHEN Sie alle Zeilen

Wenn Sie eine `WHERE` Klausel weglassen, werden alle Zeilen aus einer Tabelle gelöscht.

```
DELETE FROM Employees
```

In der TRUNCATE- Dokumentation finden Sie Details dazu, wie die Leistung von TRUNCATE verbessert werden kann, da Trigger, Indizes und Protokolle zum Löschen der Daten ignoriert werden.

TRUNCATE-Klausel

Verwenden Sie dies, um die Tabelle in den Zustand zurückzusetzen, in dem sie erstellt wurde. Dadurch werden alle Zeilen gelöscht und Werte wie das automatische Inkrementieren zurückgesetzt. Es protokolliert auch nicht jede einzelne Zeilenlöschung.

```
TRUNCATE TABLE Employees
```

LÖSCHEN Sie bestimmte Zeilen basierend auf Vergleichen mit anderen Tabellen

Es ist möglich, Daten aus einer Tabelle zu `DELETE` wenn sie mit bestimmten Daten in anderen Tabellen übereinstimmen (oder nicht übereinstimmen).

Nehmen wir an, wir wollen Daten von Source `DELETE` sobald sie in Target geladen sind.

```
DELETE FROM Source
WHERE  EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Die meisten gängigen RDBMS-Implementierungen (z. B. MySQL, Oracle, PostgresSQL, Teradata) ermöglichen das Verknüpfen von Tabellen während `DELETE` was einen komplexeren Vergleich in einer kompakten Syntax ermöglicht.

Nehmen wir an, das Aggregat wird einmal täglich aus Target erstellt und enthält nicht dieselbe ID, sondern dasselbe Datum. Nehmen wir an, dass wir Daten von Quelle gelöscht werden soll *erst* nach dem Aggregat für den Tag aufgefüllt wird.

Auf MySQL, Oracle und Teradata kann dies mit folgendem geschehen:

```
DELETE FROM Source
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

In PostgreSQL verwenden Sie:

```
DELETE FROM Source
USING  TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Dies führt im Wesentlichen zu INNER JOINs zwischen Quelle, Ziel und Aggregat. Die Löschung wird für Source ausgeführt, wenn die gleichen IDs im Ziel-UND-Datum vorhanden sind, das für diese IDs im Target vorhanden ist. Diese IDs sind auch in Aggregat vorhanden.

Dieselbe Abfrage kann auch (auf MySQL, Oracle, Teradata) geschrieben werden als:

```
DELETE Source
FROM   Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Explizite Joins können in `Delete` Anweisungen in einigen RDBMS-Implementierungen (z. B. Oracle, MySQL) erwähnt werden, werden jedoch nicht auf allen Plattformen unterstützt (z. B. unterstützt Teradata diese nicht).

Vergleiche können entworfen werden, um nicht übereinstimmende Szenarien mit allen `NOT EXISTS` (`NOT EXISTS`)

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

LÖSCHEN online lesen: https://riptutorial.com/de/sql/topic/1105/loschen

Kapitel 40: Löst aus

Examples

TRIGGER ERSTELLEN

In diesem Beispiel wird ein Trigger erstellt, der einen Datensatz in eine zweite Tabelle (MyAudit) einfügt, nachdem ein Datensatz in die Tabelle eingefügt wurde, für die der Trigger definiert ist (MyTable). Hier ist die "eingefügte" Tabelle eine spezielle Tabelle, die von Microsoft SQL Server verwendet wird, um betroffene Zeilen während INSERT- und UPDATE-Anweisungen zu speichern. Es gibt auch eine spezielle "gelöschte" Tabelle, die dieselbe Funktion für DELETE-Anweisungen ausführt.

```
CREATE TRIGGER MyTrigger
    ON MyTable
    AFTER INSERT

AS

BEGIN
    -- insert audit record to MyAudit table
    INSERT INTO MyAudit(MyTableId, User)
    (SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Verwenden Sie Auslöser, um einen "Papierkorb" für gelöschte Elemente zu verwalten

```
CREATE TRIGGER BooksDeleteTrigger
    ON MyBooksDB.Books
    AFTER DELETE
AS
  INSERT INTO BooksRecycleBin
    SELECT *
    FROM deleted;
GO
```

Löst aus online lesen: https://riptutorial.com/de/sql/topic/1432/lost-aus

Kapitel 41: Materialisierte Ansichten

Einführung

Eine materialisierte Ansicht ist eine Ansicht, deren Ergebnisse physisch gespeichert werden und regelmäßig aktualisiert werden müssen, um aktuell zu bleiben. Sie sind daher nützlich, um die Ergebnisse komplexer, langwieriger Abfragen zu speichern, wenn keine Echtzeitergebnisse erforderlich sind. Materialisierte Ansichten können in Oracle und PostgreSQL erstellt werden. Andere Datenbanksysteme bieten ähnliche Funktionen, wie z. B. die indizierten Ansichten von SQL Server oder die materialisierten Abfragetabellen von DB2.

Examples

PostgreSQL-Beispiel

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
 number
--------
      1
(1 row)

INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
 number
--------
      1
(1 row)

REFRESH MATERIALIZED VIEW myview;

SELECT * FROM myview;
 number
--------
      1
      2
(2 rows)
```

Materialisierte Ansichten online lesen: https://riptutorial.com/de/sql/topic/8367/materialisierte-ansichten

Kapitel 42: NULL

Einführung

`NULL` in SQL sowie Programmieren im Allgemeinen bedeutet wörtlich "nichts". In SQL ist es einfacher zu verstehen als "das Fehlen jeglicher Werte".

Es ist wichtig, es von scheinbar leeren Werten zu unterscheiden, wie z. B. der leere Zeichenfolge `''` oder der Zahl `0` , von denen keiner tatsächlich `NULL` .

Es ist auch wichtig, darauf zu achten, dass `NULL` in Anführungszeichen eingeschlossen wird, wie `'NULL'` ist in Spalten zulässig, die Text akzeptieren, aber nicht `NULL` und Fehler und falsche Datensätze verursachen kann.

Examples

Filtern nach NULL in Abfragen

Die Syntax für das Filtern nach `NULL` (dh das Fehlen eines Werts) in `WHERE` Blöcken unterscheidet sich geringfügig von der Filterung nach bestimmten Werten.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

Beachten Sie, dass die Verwendung von Gleichheitsoperatoren `= NULL` oder `<> NULL` (oder `!= NULL`) immer den Wahrheitswert von `UNKNOWN` liefert, da `NULL` nicht gleich irgendetwas ist, auch nicht für sich selbst. `UNKNOWN` wird von `WHERE` .

`WHERE` filtert alle Zeilen, bei denen die Bedingung `FALSE` oder `UKNOWN` und behält nur die Zeilen bei, bei denen die Bedingung `TRUE` .

Nullable-Spalten in Tabellen

Beim Erstellen von Tabellen ist es möglich, eine Spalte als null- oder nicht-nullfähig zu deklarieren.

```
CREATE TABLE MyTable
(
    MyCol1 INT NOT NULL, -- non-nullable
    MyCol2 INT NULL      -- nullable
) ;
```

Standardmäßig ist jede Spalte (außer der in der Primärschlüsseleinschränkung) nullwertfähig, sofern nicht ausdrücklich die `NOT NULL` Einschränkung festgelegt wird.

Der Versuch, einer nicht-nullfähigen Spalte `NULL` zuzuweisen, führt zu einem Fehler.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ;  -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
        -- cannot insert
        -- the value NULL into column 'MyCol1', table 'MyTable';
        -- column does not allow nulls. INSERT fails.
```

Felder auf NULL aktualisieren

Das Festlegen eines Felds auf `NULL` funktioniert genauso wie bei jedem anderen Wert:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Zeilen mit NULL-Feldern einfügen

Einfügen eines Mitarbeiters ohne Telefonnummer und ohne Manager in die Beispieltabelle Employees :

```
INSERT INTO Employees
    (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

NULL online lesen: https://riptutorial.com/de/sql/topic/3421/null

Kapitel 43: Primärschlüssel

Syntax

- MySQL: CREATE TABLE-Mitarbeiter (Id int NOT NULL, PRIMARY KEY (Id), ...);
- Andere: CREATE TABLE-Mitarbeiter (Id int NOT NULL PRIMARY KEY, ...);

Examples

Erstellen eines Primärschlüssels

```
CREATE TABLE Employees (
    Id int NOT NULL,
    PRIMARY KEY (Id),
    ...
);
```

Dadurch wird die Employees-Tabelle mit 'Id' als Primärschlüssel erstellt. Mit dem Primärschlüssel können die Zeilen einer Tabelle eindeutig identifiziert werden. Pro Tabelle ist nur ein Primärschlüssel zulässig.

Ein Schlüssel kann auch aus einem oder mehreren Feldern bestehen, dem sogenannten zusammengesetzten Schlüssel mit der folgenden Syntax:

```
CREATE TABLE EMPLOYEE (
    e1_id INT,
    e2_id INT,
    PRIMARY KEY (e1_id, e2_id)
)
```

Auto Increment verwenden

In vielen Datenbanken kann der Primärschlüsselwert automatisch erhöht werden, wenn ein neuer Schlüssel hinzugefügt wird. Dies stellt sicher, dass jeder Schlüssel anders ist.

MySQL

```
CREATE TABLE Employees (
    Id int NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (Id)
);
```

PostgreSQL

```
CREATE TABLE Employees (
    Id SERIAL PRIMARY KEY
);
```

SQL Server

```
CREATE TABLE Employees (
    Id int NOT NULL IDENTITY,
    PRIMARY KEY (Id)
);
```

SQLite

```
CREATE TABLE Employees (
    Id INTEGER PRIMARY KEY
);
```

Primärschlüssel online lesen: https://riptutorial.com/de/sql/topic/505/primarschlussel

Kapitel 44: Reihenfolge der Ausführung

Examples

Logische Reihenfolge der Abfrageverarbeitung in SQL

```
/*(8)*/  SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/  FROM
/*(3)*/         JOIN
/*(2)*/        ON
/*(4)*/  WHERE
/*(5)*/  GROUP BY
/*(6)*/  WITH {CUBE | ROLLUP}
/*(7)*/  HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

Die Reihenfolge, in der eine Abfrage verarbeitet wird, und eine Beschreibung jedes Abschnitts.

VT steht für 'Virtual Table' und zeigt, wie verschiedene Daten bei der Verarbeitung der Abfrage erzeugt werden

1. FROM: Ein kartesisches Produkt (Cross Join) wird zwischen den ersten beiden Tabellen in der FROM-Klausel ausgeführt, und als Ergebnis wird die virtuelle Tabelle VT1 generiert.

2. ON: Der ON-Filter wird auf VT1 angewendet. Nur Zeilen, für die der Wert TRUE ist, werden in VT2 eingefügt.

3. OUTER (join): Wenn ein OUTER JOIN angegeben ist (im Gegensatz zu einem CROSS JOIN oder einem INNER JOIN), werden Zeilen aus der konservierten Tabelle oder den Tabellen, für die keine Übereinstimmung gefunden wurde, zu den Zeilen aus VT2 als äußere Zeilen hinzugefügt VT3. Wenn in der FROM-Klausel mehr als zwei Tabellen angezeigt werden, werden die Schritte 1 bis 3 wiederholt zwischen dem Ergebnis des letzten Joins und der nächsten Tabelle in der FROM-Klausel angewendet, bis alle Tabellen verarbeitet sind.

4. WHERE: Der WHERE-Filter wird auf VT3 angewendet. Nur Zeilen, für die der Wert TRUE ist, werden in VT4 eingefügt.

5. GROUP BY: Die Zeilen von VT4 sind in Gruppen angeordnet, basierend auf der in der GROUP BY-Klausel angegebenen Spaltenliste. VT5 wird generiert.

6. CUBE | ROLLUP: Supergruppen (Gruppen von Gruppen) werden zu den Zeilen von VT5 hinzugefügt, wodurch VT6 generiert wird.

7. HAVING: Der HAVING-Filter wird auf VT6 angewendet. Nur Gruppen, für die der Wert TRUE ist, werden in VT7 eingefügt.

8. SELECT: Die SELECT-Liste wird verarbeitet und generiert VT8.

9. DISTINCT: Doppelte Zeilen werden aus VT8 entfernt. VT9 wird generiert.
10. ORDER BY: Die Zeilen von VT9 werden nach der in der ORDER BY-Klausel angegebenen Spaltenliste sortiert. Ein Cursor wird generiert (VC10).
11. TOP: Die angegebene Anzahl oder der Prozentsatz der Zeilen wird am Anfang von VC10 ausgewählt. Die Tabelle VT11 wird generiert und an den Anrufer zurückgegeben. LIMIT hat in einigen SQL-Dialekten wie Postgres und Netezza die gleiche Funktionalität wie TOP.

Reihenfolge der Ausführung online lesen: https://riptutorial.com/de/sql/topic/3671/reihenfolge-der-ausfuhrung

Kapitel 45: Relationale Algebra

Examples

Überblick

Relational Algebra ist keine ausgewachsene *SQL*- Sprache, sondern ein Weg, um theoretisches Verständnis der relationalen Verarbeitung zu erlangen. Daher sollte es nicht auf physische Entitäten wie Tabellen, Datensätze und Felder verweisen. Es sollte auf abstrakte Konstrukte wie Beziehungen, Tupel und Attribute Bezug nehmen. Wenn ich das sage, werde ich die akademischen Begriffe in diesem Dokument nicht verwenden und werde mich an die bekannteren Laienbegriffe halten - Tabellen, Aufzeichnungen und Felder.

Bevor wir loslegen, einige Regeln der relationalen Algebra:

- Die in der relationalen Algebra verwendeten Operatoren arbeiten mit ganzen Tabellen und nicht mit einzelnen Datensätzen.
- Das Ergebnis eines relationalen Ausdrucks ist immer eine Tabelle (dies wird *Schließungseigenschaft genannt*).

In diesem Dokument werde ich auf die folgenden zwei Tabellen Bezug nehmen:

Departments

ID	Dept
1	Production
2	Quality Control

People

ID	PersonName	StartYear	ManagerID	DepartmentID
1	Darren	2005		1
2	David	2006	1	1
3	Burt	2006	1	1
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

WÄHLEN

Der **select-** Operator gibt eine Teilmenge der Haupttabelle zurück.
Wählen Sie *<table>* ***wo*** *<Bedingung>*

Untersuchen Sie zum Beispiel den Ausdruck:

Wählen Sie Personen aus, bei **denen** DepartmentID = 2 ist

Dies kann geschrieben werden als:

$$\sigma_{DepartmentID = 2}(People)$$

Dies führt zu einer Tabelle, deren Datensätze alle Datensätze der *People*- Tabelle umfassen,

deren *Abteilungs*- ID gleich 2 ist:

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

Bedingungen können auch verbunden werden, um den Ausdruck weiter einzuschränken:

Wählen **Sie** Personen, bei **denen** StartYear> 2005 **und** DepartmentID = 2 sind

ergibt die folgende Tabelle:

ID	PersonName	StartYear	ManagerID	DepartmentID
5	Fred	2008	4	2

PROJEKT

Der **Projektbediener** gibt verschiedene Feldwerte aus einer Tabelle zurück.
Projekt <*Tabelle*> ***über*** <*Feldliste*>

Untersuchen Sie beispielsweise den folgenden Ausdruck:
Projekt Menschen **über** StartYear

Dies kann geschrieben werden als:

$$\Pi_{\text{StartYear}}^{\text{(People)}}$$

Dies führt zu einer Tabelle mit den unterschiedlichen Werten, die im Feld *StartYear* der Tabelle *People enthalten sind* .

StartYear
2005
2006
2004
2008

Doppelte Werte werden aus der resultierenden Tabelle entfernt, da die *Schließungseigenschaft* eine relationale Tabelle erstellt: Alle Datensätze in einer relationalen Tabelle müssen eindeutig sein.

Wenn die *Feldliste* mehr als ein einzelnes Feld umfasst dann die resultierende Tabelle ist eine deutliche Version dieser Felder.
Projekt People **over** StartYear, wird DepartmentID zurückgegeben:

StartYear	DepartmentID
2005	1
2006	1
2004	2
2008	2
2005	2

Ein Datensatz wird aufgrund der Duplizierung von 2006 *StartYear* und 1 *DepartmentID entfernt* .

GEBEN

Relationale Ausdrücke können durch die Benennung der einzelnen Ausdrücke unter Verwendung der **Angabe** Schlüsselwort miteinander verkettet werden, oder durch Einbetten eines Ausdrucks innerhalb der anderen.

<relationaler Algebra-Ausdruck> ***mit*** *<Aliasname>*

Betrachten Sie zum Beispiel die folgenden Ausdrücke:
Wählen Sie Personen aus, bei **denen** DepartmentID = 2 ist und **geben Sie** A
Projekt A **über** PersonName **, das** B gibt

Dies führt zu Tabelle B unten, wobei Tabelle A das Ergebnis des ersten Ausdrucks ist.

A

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

B

PersonName
Sarah
Fred
Joanne

Der erste Ausdruck wird ausgewertet und die resultierende Tabelle erhält den Alias A. Diese Tabelle wird dann innerhalb des zweiten Ausdrucks verwendet, um der finalen Tabelle den Alias von B zu geben.

Eine andere Möglichkeit, diesen Ausdruck zu schreiben, besteht darin, den Tabellenaliasnamen im zweiten Ausdruck durch den gesamten Text des ersten in Klammern gesetzten Ausdrucks zu ersetzen:
Projekt (Personen **auswählen, bei denen** DepartmentID = 2 ist) **über** PersonName **mit** B

Dies wird als *verschachtelter Ausdruck bezeichnet* .

NATÜRLICHER JOIN

Bei einem natürlichen Join werden zwei Tabellen zusammengefügt, wobei ein gemeinsames Feld zwischen den Tabellen verwendet wird.

Join *<Tabelle 1>* ***und*** *<Tabelle 2>* ***wobei*** *<Feld 1> = <Feld 2>*
Angenommen, <Feld 1> befindet sich in <Tabelle 1> und <Feld 2> befindet sich in <Tabelle 2>.

Beispielsweise wird der folgende Join-Ausdruck *Personen* und *Abteilungen* basierend auf den *Abteilungs- ID-* und *ID-* Spalten in den jeweiligen Tabellen hinzufügen:
Treten Sie Personen **und** Abteilungen **bei, für die** DepartmentID = ID gilt

ID	PersonName	StartYear	ManagerID	DepartmentID	Dept
1	Darren	2005		1	Production
2	David	2006	1	1	Production
3	Burt	2006	1	1	Production
4	Sarah	2004		2	Quality Control
5	Fred	2008	4	2	Quality Control
6	Joanne	2005	4	2	Quality Control

Beachten Sie, dass nur *DepartmentID* aus der *People-* Tabelle und keine *ID* aus der *Department-* Tabelle angezeigt wird. Es muss nur eines der Felder angezeigt werden, das verglichen wird. Dies ist im Allgemeinen der Feldname aus der ersten Tabelle der Verknüpfungsoperation.

Obwohl in diesem Beispiel nicht gezeigt, ist es möglich, dass das Verknüpfen von Tabellen zu zwei Feldern mit derselben Überschrift führt. Zum Beispiel, wenn ich die Überschrift *Namen* verwendet hatte den *Personnamen* und *Dept* Felder zu identifizieren (dh die Person Namen und den Namen der Abteilung zu identifizieren). In dieser Situation verwenden wir den Tabellennamen, um die Feldnamen anhand der Punktnotation zu qualifizieren: *People.Name* und *Departments.Name*

join kombiniert mit **select** und **project** kann zusammen verwendet werden, um Informationen abzurufen:

beitreten** Personen **und** Abteilungen, bei **denen** DepartmentID = ID A **ergibt
*Wählen **Sie** A **mit** StartYear = 2005 **und** Dept = 'Production' aus, **wobei** B angezeigt wird*
***Projekt** B **über** PersonName **, das** C gibt*

oder als kombinierter Ausdruck:

Projekt (wählen (Join** Personen **und** Abteilungen* , in ***denen** DepartmentID = ID)* , in ***dem** startyear = 2005 **und** Dept = ‚Produktion') **über** Person C **geben

Dies führt zu dieser Tabelle:

PersonName
Darren

ALIAS

TEILEN

UNION

ÜBERSCHNEIDUNG

UNTERSCHIED

UPDATE (: =)

MAL

Relationale Algebra online lesen: https://riptutorial.com/de/sql/topic/7311/relationale-algebra

Kapitel 46: Sequenz

Examples

Sequenz erstellen

```
CREATE SEQUENCE orders_seq
START WITH     1000
INCREMENT BY   1;
```

Erzeugt eine Sequenz mit einem Startwert von 1000, die um 1 erhöht wird.

Sequenzen verwenden

Ein Verweis auf *seq_name* .NEXTVAL wird verwendet, um den nächsten Wert in einer Sequenz abzurufen. Eine einzelne Anweisung kann nur einen einzelnen Sequenzwert generieren. Wenn in einer Anweisung mehrere Verweise auf NEXTVAL vorhanden sind, wird dieselbe generierte Nummer verwendet.

NEXTVAL kann für INSERTS verwendet werden

```
INSERT INTO Orders (Order_UID, Customer)
        VALUES (orders_seq.NEXTVAL, 1032);
```

Es kann für UPDATES verwendet werden

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

Es kann auch für SELEKTE verwendet werden

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Sequenz online lesen: https://riptutorial.com/de/sql/topic/1586/sequenz

Kapitel 47: SKIP TAKE (Paginierung)

Examples

Einige Zeilen aus dem Ergebnis überspringen

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- skips 20 for take use very large number that is more than rows in table
```

Orakel:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL:

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

Begrenzung der Anzahl der Ergebnisse

ISO / ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Orakel:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

SQL Server:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

Überspringen und einige Ergebnisse aufnehmen (Paginierung)

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Orakel; SQL Server:

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

SKIP TAKE (Paginierung) online lesen: https://riptutorial.com/de/sql/topic/2927/skip-take--paginierung-

Kapitel 48: SORTIEREN NACH

Examples

Verwenden Sie ORDER BY mit TOP, um die ersten x Zeilen basierend auf dem Wert einer Spalte zurückzugeben

In diesem Beispiel können wir nicht nur verwenden GROUP BY bestimmt die *Art* der zurückgegebenen Zeilen, sondern auch , welche Zeilen zurückgegeben werden , da wir TOP verwenden die Ergebnismenge zu begrenzen.

Nehmen wir an, wir möchten die Top 5 der besten Nutzer von einer ungenannten beliebten Q & A-Site zurückgeben.

Ohne ORDER BY

Diese Abfrage gibt die obersten 5 Zeilen nach Standard sortiert zurück. In diesem Fall handelt es sich um "Id", die erste Spalte in der Tabelle (obwohl dies keine in den Ergebnissen angezeigte Spalte ist).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

kehrt zurück...

Anzeigename	Ruf
Gemeinschaft	1
Geoff Dalgas	12567
Jarrod Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

Mit ORDER BY

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
ORDER BY Reputation desc
```

kehrt zurück...

Anzeigename	Ruf
JonSkeet	**865023**
Darin Dimitrov	**661741**
BalusC	**650237**
Hans Passant	**625870**
Marc Gravell	**601636**

Bemerkungen

Einige SQL-Versionen (wie MySQL) verwenden eine `LIMIT` Klausel am Ende einer `SELECT` anstelle von `TOP` am Anfang. Beispiel:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Sortierung nach mehreren Spalten

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY JoinDate, Reputation
```

Anzeigename	Beitrittsdatum	Ruf
Gemeinschaft	**2008-09-15**	**1**
Jeff Atwood	**2008-09-16**	**25784**
Joel Spolsky	2008-09-16	**37628**
Jarrod Dixon	**2008-10-03**	**11739**
Geoff Dalgas	2008-10-03	**12567**

Sortierung nach Spaltennummer (statt Name)

Sie können die Nummer einer Spalte verwenden (wobei die Spalte ganz links "1" ist), um anzugeben, auf welcher Spalte die Sortierung basieren soll, anstatt die Spalte anhand ihres Namens zu beschreiben.

Pro: Wenn Sie der Meinung sind, dass Sie die Spaltennamen möglicherweise später ändern, wird der Code dadurch nicht beschädigt.

Con: Dies verringert im Allgemeinen die Lesbarkeit der Abfrage (Es ist sofort klar, was 'ORDER BY Reputation' bedeutet, während 'ORDER BY 14' etwas zählt, wahrscheinlich mit einem Finger auf dem Bildschirm.)

Diese Abfrage sortiert das Ergebnis anhand der Informationen in relativer Spaltenposition `3` aus der SELECT-Anweisung anstelle des Spaltennamens `Reputation` .

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY 3
```

Anzeigename	Beitrittsdatum	Ruf
Gemeinschaft	2008-09-15	**1**
Jarrod Dixon	2008-10-03	**11739**
Geoff Dalgas	2008-10-03	**12567**
Joel Spolsky	2008-09-16	**25784**
Jeff Atwood	2008-09-16	**37628**

Auftrag von Alias

Aufgrund der Reihenfolge der logischen Abfrageverarbeitung können Aliasnamen in der Reihenfolge nach verwendet werden.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

Und kann die relative Reihenfolge der Spalten in der select-Anweisung verwenden. Beachten Sie dasselbe Beispiel wie oben, und verwenden Sie anstelle des Alias die relative Reihenfolge wie für den Anzeigenamen 1, für Jd 2 und so weiter

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3
```

Angepasste Sortierreihenfolge

Um diese Tabelle `Employee` nach Abteilung zu sortieren, verwenden Sie `ORDER BY Department` . Wenn Sie jedoch eine andere Sortierreihenfolge wünschen, die nicht alphabetisch ist, müssen Sie die `Department` in verschiedene Werte einordnen, die korrekt sortiert werden. Dies kann mit einem CASE-Ausdruck erfolgen:

Name	Abteilung
Hasan	ES
Yusuf	HR
Hillary	HR
Joe	ES
Fröhlich	HR
Ken	Buchhalter

```
SELECT *
FROM Employee
ORDER BY CASE Department
         WHEN 'HR'         THEN 1
         WHEN 'Accountant' THEN 2
         ELSE                   3
         END;
```

Name	Abteilung
Yusuf	**HR**
Hillary	**HR**
Fröhlich	**HR**
Ken	**Buchhalter**
Hasan	**ES**
Joe	**ES**

SORTIEREN NACH online lesen: https://riptutorial.com/de/sql/topic/620/sortieren-nach

Kapitel 49: SQL CURSOR

Examples

Beispiel für einen Cursor, der alle Zeilen nach Index für jede Datenbank abfragt

Hier wird ein Cursor zum Durchlaufen aller Datenbanken verwendet.
Darüber hinaus wird ein Cursor aus dynamischem SQL verwendet, um jede Datenbank abzufragen, die vom ersten Cursor zurückgegeben wird.
Dies soll den Verbindungsbereich eines Cursors demonstrieren.

```
DECLARE @db_name nvarchar(255)
DECLARE @sql nvarchar(MAX)

DECLARE @schema nvarchar(255)
DECLARE @table nvarchar(255)
DECLARE @column nvarchar(255)




DECLARE db_cursor CURSOR FOR
SELECT name FROM sys.databases


OPEN db_cursor
FETCH NEXT FROM db_cursor INTO @db_name

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @sql = 'SELECT * FROM ' + QUOTENAME(@db_name) + '.information_schema.columns'
    PRINT ''
    PRINT ''
    PRINT ''
    PRINT @sql
    -- EXECUTE(@sql)



    -- For each database

    DECLARE @sqlstatement nvarchar(4000)
    --move declare cursor into sql to be executed
    SET @sqlstatement = 'DECLARE  columns_cursor CURSOR FOR SELECT TABLE_SCHEMA, TABLE_NAME,
COLUMN_NAME FROM ' + QUOTENAME(@db_name) + '.information_schema.columns ORDER BY TABLE_SCHEMA,
TABLE_NAME, ORDINAL_POSITION'



    EXEC sp_executesql @sqlstatement


    OPEN columns_cursor
    FETCH NEXT FROM columns_cursor
```

```
    INTO @schema, @table, @column

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @schema + '.' + @table + '.' + @column
        --EXEC asp_DoSomethingStoredProc @UserId

    FETCH NEXT FROM columns_cursor --have to fetch again within loop
    INTO @schema, @table, @column

    END
    CLOSE columns_cursor
    DEALLOCATE columns_cursor

    -- End for each database




    FETCH NEXT FROM db_cursor INTO @db_name
END

CLOSE db_cursor
DEALLOCATE db_cursor
```

SQL CURSOR online lesen: https://riptutorial.com/de/sql/topic/8895/sql-cursor

Kapitel 50: SQL Group By vs. Distinct

Examples

Unterschied zwischen GROUP BY und DISTINCT

`GROUP BY` wird in Kombination mit Aggregationsfunktionen verwendet. Betrachten Sie die folgende Tabelle:

Auftragsnummer	Benutzeridentifikation	storeName	Bestellwert	Auftragsdatum
1	43	Speichern Sie A	25	20-03-2016
2	57	Speicher B	50	22-03-2016
3	43	Speichern Sie A	30	25-03-2016
4	82	Speichern Sie C	10	26-03-2016
5	21	Speichern Sie A	45	29-03-2016

Die folgende Abfrage verwendet `GROUP BY` , um aggregierte Berechnungen durchzuführen.

```
SELECT
    storeName,
    COUNT(*) AS total_nr_orders,
    COUNT(DISTINCT userId) AS nr_unique_customers,
    AVG(orderValue) AS average_order_value,
    MIN(orderDate) AS first_order,
    MAX(orderDate) AS lastOrder
FROM
    orders
GROUP BY
    storeName;
```

und gibt die folgenden Informationen zurück

storeName	total_nr_orders	nr_unique_customers	average_order_value	erste Bestellung	letzte Beste
Speichern Sie A	3	2	33.3	20-03-2016	29-03-2016
Speicher B	1	1	50	22-03-	22-03

storeName	total_nr_orders	nr_unique_customers	average_order_value	erste Bestellung	letzte Beste
				2016	2016
Speichern Sie C	1	1	10	26-03-2016	26-03-2016

`DISTINCT` wird verwendet, um eine eindeutige Kombination verschiedener Werte für die angegebenen Spalten aufzulisten.

```
SELECT DISTINCT
    storeName,
    userId
FROM
    orders;
```

storeName	Benutzeridentifikation
Speichern Sie A	43
Speicher B	57
Speichern Sie C	82
Speichern Sie A	21

SQL Group By vs. Distinct online lesen: https://riptutorial.com/de/sql/topic/2499/sql-group-by-vs--distinct

Kapitel 51: SQL-Injektion

Einführung

SQL-Injection ist ein Versuch, auf die Datenbanktabellen einer Website zuzugreifen, indem SQL in ein Formularfeld eingefügt wird. Wenn ein Webserver keinen Schutz vor SQL-Injection-Angriffen bietet, kann ein Hacker die Datenbank dazu bringen, den zusätzlichen SQL-Code auszuführen. Durch Ausführen ihres eigenen SQL-Codes können Hacker den Zugriff auf ihr Konto verbessern, die privaten Informationen einer anderen Person anzeigen oder andere Änderungen an der Datenbank vornehmen.

Examples

Beispiel einer SQL-Injektion

Angenommen, der Aufruf des Login-Handlers Ihrer Webanwendung sieht folgendermaßen aus:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Jetzt lesen Sie in login.ashx diese Werte:

```
strUserName = getHttpsRequestParameterString("username");
strPassword = getHttpsRequestParameterString("password");
```

und fragen Sie Ihre Datenbank ab, um festzustellen, ob ein Benutzer mit diesem Kennwort vorhanden ist.

Sie konstruieren also eine SQL-Abfragezeichenfolge:

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '"+
strPassword +"'";
```

Dies funktioniert, wenn der Benutzername und das Passwort kein Angebot enthalten.

Wenn jedoch einer der Parameter ein Zitat enthält, sieht die an die Datenbank gesendete SQL folgendermaßen aus:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

Dies führt zu einem Syntaxfehler, da das Anführungszeichen nach `d` in `d'Alambert` die SQL-Zeichenfolge beendet.

Sie können dies korrigieren, indem Sie Anführungszeichen in Benutzernamen und Kennwort verwenden, z.

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

Es ist jedoch sinnvoller, Parameter zu verwenden:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

Wenn Sie keine Parameter verwenden und vergessen, das Anführungszeichen in einem der Werte zu ersetzen, kann ein böswilliger Benutzer (auch als Hacker bezeichnet) SQL-Befehle in Ihrer Datenbank ausführen.

Wenn ein Angreifer beispielsweise böse ist, setzt er / sie das Kennwort auf

```
lol'; DROP DATABASE master; --
```

und dann sieht die SQL so aus:

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --'";
```

Leider ist dies gültiges SQL, und die DB führt das aus!

Diese Art von Exploit wird als SQL-Injection bezeichnet.

Es gibt viele andere Möglichkeiten, die ein böswilliger Benutzer tun könnte, beispielsweise die E-Mail-Adresse jedes Benutzers zu stehlen, das Passwort von jedem zu stehlen, Kreditkartennummern zu stehlen, Daten in Ihrer Datenbank zu stehlen usw.

Aus diesem Grund müssen Sie immer Ihre Zeichenfolgen entziehen.
Die Tatsache, dass Sie dies früher oder später vergessen werden, ist genau der Grund, warum Sie Parameter verwenden sollten. Denn wenn Sie Parameter verwenden, wird Ihr Programmiersprachen-Framework alle erforderlichen Fluchtwege für Sie ausführen.

einfache Injektionsprobe

Wenn die SQL-Anweisung folgendermaßen aufgebaut ist:

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password ='" + pw + "'";
db.execute(SQL);
```

Ein Hacker könnte dann Ihre Daten abrufen, indem er ein Kennwort wie `pw' or '1'='1` . Die resultierende SQL-Anweisung lautet:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

Dieser wird die Passwortprüfung für alle Zeilen in der Tabelle `Users` durchlaufen, da `'1'='1'` immer

true ist.

Um dies zu verhindern, verwenden Sie SQL-Parameter:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

SQL-Injektion online lesen: https://riptutorial.com/de/sql/topic/3517/sql-injektion

Kapitel 52: String-Funktionen

Einführung

Stringfunktionen führen Operationen mit Stringwerten aus und geben entweder numerische Werte oder Stringwerte zurück.

Mit Stringfunktionen können Sie beispielsweise Daten kombinieren, einen Teilstring extrahieren, Strings vergleichen oder einen String in Groß- oder Kleinbuchstaben konvertieren.

Syntax

- CONCAT (string_value1, string_value2 [, string_valueN])
- LTRIM (Zeichenausdruck)
- RTRIM (Zeichenausdruck)
- SUBSTRING (Ausdruck, Start, Länge)
- ASCII (Zeichenausdruck)
- REPLICATE (string_expression, integer_expression)
- REVERSE (string_expression)
- UPPER (Zeichenausdruck)
- TRIM ([Zeichen FROM] Zeichenfolge)
- STRING_SPLIT (Zeichenfolge, Trennzeichen)
- STUFF (character_expression, start, length, replaceWith_expression)
- REPLACE (string_expression, string_pattern, string_replacement)

Bemerkungen

String-Funktionsreferenz für Transact-SQL / Microsoft

String-Funktionsreferenz für MySQL

String-Funktionsreferenz für PostgreSQL

Examples

Leere Räume abschneiden

Trimmen wird verwendet, um den Schreibbereich am Anfang oder Ende der Auswahl zu entfernen

In MSSQL gibt es kein einzelnes `TRIM()`

```
SELECT LTRIM('  Hello  ') --returns 'Hello  '
SELECT RTRIM('  Hello  ') --returns '  Hello'
SELECT LTRIM(RTRIM('  Hello  ')) --returns 'Hello'
```

MySql und Oracle

```
SELECT TRIM('  Hello  ') --returns 'Hello'
```

Verketten

In SQL (Standard ANSI / ISO) ist der Operator für die String-Verkettung `||` . Diese Syntax wird von allen wichtigen Datenbanken mit Ausnahme von SQL Server unterstützt:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

Viele Datenbanken unterstützen eine `CONCAT` Funktion zum Verknüpfen von Zeichenfolgen:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Einige Datenbanken unterstützen die Verwendung von `CONCAT` zum Verknüpfen von mehr als zwei Zeichenfolgen (Oracle nicht):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

In einigen Datenbanken müssen Nicht-String-Typen umgewandelt oder konvertiert werden:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Einige Datenbanken (z. B. Oracle) führen implizit verlustfreie Konvertierungen durch. Zum Beispiel kann ein `CONCAT` auf einem `CLOB` und `NCLOB` ergibt eine `NCLOB` . Ein `CONCAT` auf einer Zahl und einem `varchar2` führt zu einem `varchar2` usw.

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Einige Datenbanken können den Nicht-Standard-Operator `+` (in den meisten `+` funktioniert `+` nur für Zahlen):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

Auf SQL Server <2012, wo `CONCAT` nicht unterstützt wird, ist `+` die einzige Möglichkeit, Zeichenfolgen zu `CONCAT` .

Groß- und Kleinschreibung

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

Unterstring

Die Syntax lautet: `SUBSTRING ( string_expression, start, length )` . Beachten Sie, dass SQL-

Zeichenfolgen 1-indiziert sind.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

Dies wird häufig in Verbindung mit der `LEN()` Funktion verwendet, um die letzten `n` Zeichen einer Zeichenfolge unbekannter Länge abzurufen.

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --returns 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --returns 'Baz'
```

Teilt

Teilt einen Zeichenfolgenausdruck mit einem Trennzeichen. Beachten Sie, dass `STRING_SPLIT()` eine Tabellenwertfunktion ist.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Ergebnis:

```
value
-----
Lorem
ipsum
dolor
sit
amet.
```

Zeug

Füllen Sie einen String in einen anderen und ersetzen Sie an einer bestimmten Position 0 oder mehr Zeichen.

Hinweis: `start` Position 1-indiziert (Sie die Indizierung bei 1 beginnen, nicht 0).

Syntax:

```
STUFF ( character_expression , start , length , replaceWith_expression )
```

Beispiel:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

Länge

SQL Server

Das LEN zählt das nachgestellte Leerzeichen nicht.

```
SELECT LEN('Hello') -- returns 5

SELECT LEN('Hello '); -- returns 5
```

Die DATALENGTH zählt den nachgestellten Raum.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

Es sei jedoch darauf hingewiesen, dass DATALENGTH die Länge der zugrunde liegenden Bytendarstellung der Zeichenfolge zurückgibt, die ua vom Zeichensatz abhängt, der zum Speichern der Zeichenfolge verwendet wird.

```
DECLARE @str varchar(100) = 'Hello ' --varchar is usually an ASCII string, occupying 1 byte
per char
SELECT DATALENGTH(@str) -- returns 6

DECLARE @nstr nvarchar(100) = 'Hello ' --nvarchar is a unicode string, occupying 2 bytes per
char
SELECT DATALENGTH(@nstr) -- returns 12
```

Orakel

Syntax: Länge (Zeichen)

Beispiele:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

Siehe auch: Länge B, Länge C, Länge2, Länge4

Ersetzen

Syntax:

`REPLACE(` String zu suchen , String zu suchen und ersetzen , String in die ursprüngliche Zeichenfolge zu platzieren `)`

Beispiel:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

LINKS RECHTS

Syntax ist:
LEFT (Zeichenfolgenausdruck, Ganzzahl)
RECHTS (Zeichenfolgenausdruck, Ganzzahl)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL hat keine LINKS- und RECHTS-Funktionen. Sie können mit SUBSTR und LENGTH emuliert werden.
SUBSTR (Zeichenfolgenausdruck, 1, Ganzzahl)
SUBSTR (Zeichenfolgenausdruck, Länge (Zeichenfolgenausdruck) -Zahl + 1, Ganzzahl)

```
SELECT SUBSTR('Hello',1,2)  --return He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --return lo
```

UMKEHREN

Syntax ist: REVERSE (Zeichenfolgenausdruck)

```
SELECT REVERSE('Hello') --returns olleH
```

ERSETZEN

Die `REPLICATE` Funktion verkettet einen String mit einer angegebenen Anzahl von Malen.

Syntax ist: REPLICATE (Zeichenfolgenausdruck, Ganzzahl)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

REGEXP

MySQL 3.19

Überprüft, ob eine Zeichenfolge mit einem regulären Ausdruck übereinstimmt (definiert durch eine andere Zeichenfolge).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True

SELECT 'beam' REGEXP '[a-f]' -- returns False
```

Funktion in SQL-Abfrage ersetzen und aktualisieren

Mit der Replace-Funktion in SQL wird der Inhalt einer Zeichenfolge aktualisiert. Der Funktionsaufruf ist REPLACE () für MySQL, Oracle und SQL Server.

Die Syntax der Replace-Funktion lautet:

```
REPLACE (str, find, repl)
```

Das folgende Beispiel ersetzt Vorkommen von `South` durch `Southern` in der Tabelle "Mitarbeiter":

Vorname	Adresse
James	South New York
John	South Boston
Michael	South San Diego

Aussage auswählen:

Wenn wir die folgende Replace-Funktion anwenden:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Ergebnis:

Vorname	Adresse
James	Südliches New York
John	Südliches Boston
Michael	Südliches San Diego

Update-Anweisung:

Wir können eine Ersetzungsfunktion verwenden, um permanente Änderungen in unserer Tabelle vorzunehmen.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

Ein üblicherer Ansatz besteht darin, dies in Verbindung mit einer WHERE-Klausel wie folgt zu verwenden:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

PARSENAME

DATENBANK : SQL Server

Die Funktion **PARSENAME** gibt den spezifischen Teil der angegebenen Zeichenfolge (Objektname) zurück. Objektname kann Zeichenfolge wie Objektname, Besitzername, Datenbankname und Servername enthalten.

Weitere Details `MSDN: PARSENAME`

Syntax

```
PARSENAME('NameOfStringToParse',PartIndex)
```

Beispiel

Um den Objektnamen zu erhalten, verwenden Sie den Teilindex `1`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1)  // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1)     // returns `Student`
```

Um den Schemanamen zu erhalten, verwenden Sie den Teilindex `2`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2)  // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2)     // returns `school`
```

Um den Datenbanknamen zu erhalten, verwenden Sie den Teilindex `3`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3)    // returns `SchoolDatabase`
```

Um den Servernamen zu erhalten, verwenden Sie den Teilindex `4`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4)  // returns `ServerName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4)     // returns `[1012-1111]`
```

PARSENAME gibt zurück, dass null angegeben wurde. Ein Teil ist in der angegebenen Objektnamenzeichenfolge nicht vorhanden

INSTR

Liefert den Index des ersten Vorkommens eines Teilstrings

Syntax: INSTR (Zeichenfolge, Teilzeichenfolge)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

String-Funktionen online lesen: https://riptutorial.com/de/sql/topic/1120/string-funktionen

Kapitel 53: Synonyme

Examples

Synonym erstellen

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Synonyme online lesen: https://riptutorial.com/de/sql/topic/2518/synonyme

Kapitel 54: TABELLE ERSTELLEN

Einführung

Die CREATE TABLE-Anweisung wird verwendet, um eine neue Tabelle in der Datenbank zu erstellen. Eine Tabellendefinition besteht aus einer Liste von Spalten, ihren Typen und beliebigen Integritätsbedingungen.

Syntax

- CREATE TABLE tabellenname ([spaltename1] [datentyp1] [, [spaltenname2] [datentyp2] ...])

Parameter

Parameter	Einzelheiten
Tabellenname	Der Name der Tabelle
Säulen	Enthält eine Aufzählung aller Spalten der Tabelle. Weitere Informationen finden Sie unter **Erstellen einer neuen Tabelle** .

Bemerkungen

Tabellennamen müssen eindeutig sein.

Examples

Erstellen Sie eine neue Tabelle

Mit können Sie eine einfache `Employees` Tabelle mit einer ID sowie den Vor- und Nachnamen des Mitarbeiters sowie dessen Telefonnummer erstellen

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

Dieses Beispiel ist spezifisch für Transact-SQL

`CREATE TABLE` erstellt eine neue Tabelle in der Datenbank, gefolgt vom Tabellennamen `Employees`

Darauf folgt die Liste der Spaltennamen und ihrer Eigenschaften, z. B. der ID

```
Id int identity(1,1) not null
```

Wert	Bedeutung
`Id`	der Name der Spalte
`int`	ist der Datentyp.
`identity(1,1)`	gibt an, dass die Spalte automatisch generierte Werte hat, die bei 1 beginnen und für jede neue Zeile um 1 erhöht werden.
`primary key`	gibt an, dass alle Werte in dieser Spalte eindeutige Werte haben
`not null`	gibt an, dass diese Spalte keine Nullwerte enthalten darf

Tabelle erstellen aus Auswählen

Möglicherweise möchten Sie ein Duplikat einer Tabelle erstellen:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

Sie können alle anderen Funktionen einer SELECT-Anweisung verwenden, um die Daten zu ändern, bevor Sie sie an die neue Tabelle übergeben. Die Spalten der neuen Tabelle werden automatisch entsprechend den ausgewählten Zeilen erstellt.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName," ",LName) AS FullName FROM Employees
WHERE Id > 10;
```

Eine Tabelle duplizieren

Um eine Tabelle zu duplizieren, gehen Sie einfach wie folgt vor:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

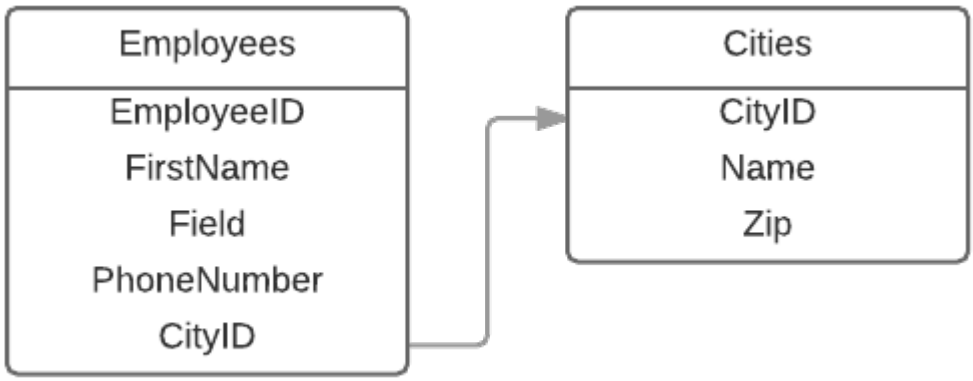
CREATE TABLE Mit dem FOREIGN KEY

Nachfolgend finden Sie die Tabelle `Employees` mit Hinweis auf die Tabelle `Cities` .

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);

CREATE TABLE Employees(
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
```

```
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

Hier finden Sie ein Datenbankdiagramm.



Die Spalte `CityID` der Tabelle `Employees` verweist auf die Spalte `CityID` der Tabelle `Cities` . Nachfolgend finden Sie die Syntax, um dies zu erreichen.

```
CityID INT FOREIGN KEY REFERENCES Cities(CityID)
```

Wert	Bedeutung
`CityID`	Name der Spalte
`int`	Typ der Spalte
`FOREIGN KEY`	Macht den Fremdschlüssel *(optional)*
`REFERENCES Cities(CityID)`	Macht die Referenz in der Tabelle `Cities` Spalte `CityID`

Wichtig: Sie konnten keinen Verweis auf eine Tabelle erstellen, die nicht in der Datenbank vorhanden ist. Seien Sie Quelle, um zuerst die Tabelle `Cities` und dann die Tabelle `Employees` . Wenn Sie es umgekehrt machen, wird ein Fehler ausgegeben.

Erstellen Sie eine temporäre oder speicherinterne Tabelle

PostgreSQL und SQLite

So erstellen Sie eine temporäre Tabelle für die Sitzung:

```
CREATE TEMP TABLE MyTable(...);
```

SQL Server

So erstellen Sie eine temporäre Tabelle für die Sitzung:

```
CREATE TABLE #TempPhysical(...);
```

So erstellen Sie eine temporäre Tabelle, die für alle sichtbar ist:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

So erstellen Sie eine In-Memory-Tabelle:

```
DECLARE @TempMemory TABLE(...);
```

TABELLE ERSTELLEN online lesen: https://riptutorial.com/de/sql/topic/348/tabelle-erstellen

Kapitel 55: Tisch Design

Bemerkungen

The Open University (1999) Relationale Datenbanksysteme: Block 2 Relationale Theorie, Milton Keynes, The Open University.

Examples

Eigenschaften eines gut gestalteten Tisches.

Eine echte relationale Datenbank muss nicht nur Daten in ein paar Tabellen werfen, sondern auch einige SQL-Anweisungen schreiben, um diese Daten abzurufen.
Im besten Fall verlangsamt eine schlecht entworfene Tabellenstruktur die Ausführung von Abfragen und kann dazu führen, dass die Datenbank nicht wie vorgesehen funktioniert.

Eine Datenbanktabelle sollte nicht nur als eine andere Tabelle betrachtet werden. es muss einer Reihe von Regeln folgen, um als wirklich relational zu gelten. Akademisch wird es als "Relation" bezeichnet, um die Unterscheidung zu treffen.

Die fünf Regeln einer relationalen Tabelle sind:

1. Jeder Wert ist *atomar* . Der Wert in jedem Feld in jeder Zeile muss ein einzelner Wert sein.
2. Jedes Feld enthält Werte, die denselben Datentyp haben.
3. Jede Feldüberschrift hat einen eindeutigen Namen.
4. Jede Zeile in der Tabelle muss mindestens einen Wert haben, der sie unter den anderen Datensätzen in der Tabelle eindeutig macht.
5. Die Reihenfolge der Zeilen und Spalten hat keine Bedeutung.

Eine Tabelle, die den fünf Regeln entspricht:

Ich würde	Name	DOB	Manager
1	Fred	11/02/1971	3
2	Fred	11/02/1971	3
3	Verklagen	08/07/1975	2

- Regel 1: Jeder Wert ist atomar. `Id` , `Name` , `DOB` und `Manager` enthalten nur einen einzelnen Wert.
- Regel 2: `Id` enthält nur Ganzzahlen, `Name` enthält Text (wir können hinzufügen, dass der Text aus vier Zeichen oder weniger besteht), `DOB` enthält Datumsangaben eines gültigen Typs und `Manager` enthält Ganzzahlen (wir könnten hinzufügen, dass dies einem Primärschlüsselfeld in einem Manager entspricht Tabelle).

- Regel 3: `Id` , `Name` , `DOB` und `Manager` sind eindeutige Überschriften in der Tabelle.
- Regel 4: Durch das Einfügen des Felds " `Id` wird sichergestellt, dass sich jeder Datensatz von allen anderen Datensätzen in der Tabelle unterscheidet.

Ein schlecht entworfener Tisch:

Ich würde	Name	DOB	Name
1	Fred	11/02/1971	3
1	Fred	11/02/1971	3
3	Verklagen	Freitag, der 18. Juli 1975	2, 1

- Regel 1: Das zweite Namensfeld enthält zwei Werte - 2 und 1.
- Regel 2: Das DOB-Feld enthält Datumsangaben und Text.
- Regel 3: Es gibt zwei Felder, die "Name" genannt werden.
- Regel 4: Der erste und der zweite Datensatz sind genau gleich.
- Regel 5: Diese Regel ist nicht gebrochen.

Tisch Design online lesen: https://riptutorial.com/de/sql/topic/2515/tisch-design

Kapitel 56: Transaktionen

Bemerkungen

Eine Transaktion ist eine logische Arbeitseinheit, die einen oder mehrere Schritte enthält, von denen jeder erfolgreich abgeschlossen werden muss, damit die Transaktion an die Datenbank übergeben werden kann. Wenn Fehler auftreten, werden alle Datenänderungen gelöscht und die Datenbank wird zu Beginn der Transaktion in ihren ursprünglichen Zustand zurückgesetzt.

Examples

Einfache Transaktion

```
BEGIN TRANSACTION
    INSERT INTO DeletedEmployees(EmployeeID, DateDeleted, User)
        (SELECT 123, GetDate(), CURRENT_USER);
    DELETE FROM Employees WHERE EmployeeID = 123;
COMMIT TRANSACTION
```

Rollback-Transaktion

Wenn in Ihrem Transaktionscode ein Fehler auftritt und Sie ihn rückgängig machen möchten, können Sie Ihre Transaktion rückgängig machen:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
   COMMIT TRANSACTION
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
END CATCH
```

Transaktionen online lesen: https://riptutorial.com/de/sql/topic/2424/transaktionen

Kapitel 57: UND-ODER-Operatoren

Syntax

1. SELECT * FROM-Tabelle WHERE (Bedingung1) AND (Bedingung2);
2. SELECT * FROM Tabelle WHERE (Bedingung1) ODER (Bedingung2);

Examples

UND ODER Beispiel

Einen Tisch haben

Name	Alter	Stadt
Bob	10	Paris
Matte	20	Berlin
Maria	24	Prag

```
select Name from table where Age>10 AND City='Prague'
```

Gibt

Name
Maria

```
select Name from table where Age=10 OR City='Prague'
```

Gibt

Name
Bob
Maria

UND-ODER-Operatoren online lesen: https://riptutorial.com/de/sql/topic/1386/und-oder-operatoren

Kapitel 58: UNION / UNION ALL

Einführung

Das UNION- Schlüsselwort in SQL wird verwendet, um die ***SELECT-*** Anweisungsergebnisse ohne Duplikat zu kombinieren. Um UNION zu verwenden und Ergebnisse zu kombinieren, sollten beide SELECT-Anweisungen dieselbe Spaltenanzahl mit demselben Datentyp in derselben Reihenfolge aufweisen. Die Spaltenlänge kann jedoch unterschiedlich sein.

Syntax

- SELECT Spalte_1 [, Spalte_2] FROM Tabelle_1 [, Tabelle_2] [WHERE-Bedingung]
  UNION | UNION ALL
  SELECT Spalte_1 [, Spalte_2] FROM Tabelle_1 [, Tabelle_2] [WHERE-Bedingung]

Bemerkungen

`UNION` und `UNION ALL` Klauseln kombinieren die Ergebnismenge von zwei oder mehr identisch aufgebauten SELECT-Anweisungen in einem einzigen Ergebnis / einer einzigen Tabelle.

Sowohl die Spaltenanzahl als auch die Spaltentypen für jede Abfrage müssen übereinstimmen, damit `UNION` / `UNION ALL` funktionieren kann.

Der Unterschied zwischen einer `UNION` und einer `UNION ALL` Abfrage besteht darin, dass mit der `UNION` Klausel doppelte Zeilen im Ergebnis entfernt werden, in denen die `UNION ALL` nicht `UNION ALL` .

Durch das eindeutige Entfernen von Datensätzen können Abfragen erheblich verlangsamt werden, selbst wenn keine separaten Zeilen entfernt werden müssen, wenn Sie wissen, dass es keine Duplikate gibt (oder egal,), die für eine optimierte Abfrage immer auf `UNION ALL` .

Examples

Grundlegende UNION ALL-Abfrage

```
CREATE TABLE HR_EMPLOYEES
(
    PersonID int,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    Position VARCHAR(30)
);

CREATE TABLE FINANCE_EMPLOYEES
(
    PersonID INT,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
```

```
    Position VARCHAR(30)
);
```

Angenommen, wir möchten die Namen aller `managers` aus unseren Abteilungen extrahieren.

Mit einer `UNION` wir alle Mitarbeiter sowohl aus der Personalabteilung als auch aus der Finanzabteilung beziehen, die die `position` eines `manager` `position`

```
SELECT
    FirstName, LastName
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Die `UNION` Anweisung entfernt doppelte Zeilen aus den Abfrageergebnissen. Da es in beiden Abteilungen möglich ist, Personen mit demselben Namen und derselben Position zu haben, verwenden wir `UNION ALL` , um Doppeleinträge nicht zu entfernen.

Wenn Sie für jede Ausgabespalte einen Alias verwenden möchten, können Sie sie wie folgt in die erste select-Anweisung einfügen:

```
SELECT
    FirstName as 'First Name', LastName as 'Last Name'
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Einfache Erklärung und Beispiel

In einfachen Worten:

- `UNION` verbindet 2 Ergebnissätze und entfernt Duplikate aus dem Ergebnissatz
- `UNION ALL` verbindet 2 Ergebnissätze, ohne zu versuchen, Duplikate zu entfernen

  Ein Fehler, den viele Leute machen, ist die Verwendung einer `UNION` wenn die Duplikate nicht entfernt werden müssen. Die zusätzlichen Leistungskosten bei großen Ergebnismengen können sehr hoch sein.

Wenn Sie `UNION` brauchen

Angenommen, Sie müssen eine Tabelle nach zwei verschiedenen Attributen filtern, und Sie haben für jede Spalte separate, nicht gruppierte Indizes erstellt. Mit einer `UNION` können Sie beide Indizes nutzen und trotzdem Duplikate vermeiden.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

Dies vereinfacht die Leistungsoptimierung, da nur einfache Indizes erforderlich sind, um diese Abfragen optimal auszuführen. Sie können sogar mit etwas weniger nicht gruppierten Indizes auskommen, was die allgemeine Schreibleistung auch gegenüber der Quelltabelle verbessert.

Wenn Sie `UNION ALL` brauchen

Angenommen, Sie müssen immer noch eine Tabelle nach zwei Attributen filtern, aber Sie müssen keine doppelten Datensätze filtern (entweder, weil dies unwichtig ist oder Ihre Daten während der Vereinigung aufgrund Ihres Datenmodellentwurfs keine Duplikate erzeugen würden).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

Dies ist besonders nützlich, wenn Sie Ansichten erstellen, die Daten verknüpfen, die physisch über mehrere Tabellen hinweg partitioniert werden sollen (möglicherweise aus Leistungsgründen, aber dennoch Datensätze aufrollen). Da die Daten bereits aufgeteilt sind, fügt das Entfernen von Duplikaten durch die Datenbank-Engine keinen Wert hinzu und erhöht lediglich die Verarbeitungszeit für die Abfragen.

UNION / UNION ALL online lesen: https://riptutorial.com/de/sql/topic/349/union---union-all

Kapitel 59: Unterabfragen

Bemerkungen

Unterabfragen können in verschiedenen Klauseln einer äußeren Abfrage oder in der Setoperation erscheinen.

Sie müssen in Klammern `()` . Wenn das Ergebnis der Unterabfrage mit etwas anderem verglichen wird, muss die Anzahl der Spalten übereinstimmen. Tabellenaliasnamen sind für Unterabfragen in der FROM-Klausel erforderlich, um die temporäre Tabelle zu benennen.

Examples

Unterabfrage in WHERE-Klausel

Verwenden Sie eine Unterabfrage, um die Ergebnismenge zu filtern. Zum Beispiel werden alle Mitarbeiter mit einem Gehalt entlohnt, das dem bestbezahlten Mitarbeiter entspricht.

```
SELECT *
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Unterabfrage in FROM-Klausel

Eine Unterabfrage in einer `FROM` Klausel verhält sich ähnlich wie eine temporäre Tabelle, die während der Ausführung einer Abfrage generiert wird und danach verloren geht.

```
SELECT Managers.Id, Employees.Salary
FROM (
  SELECT Id
  FROM Employees
  WHERE ManagerId IS NULL
) AS Managers
JOIN Employees ON Managers.Id = Employees.Id
```

Unterabfrage in SELECT-Klausel

```
SELECT
  Id,
  FName,
  LName,
  (SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id) AS NumberOfCars
FROM Customers
```

Unterabfragen in FROM-Klausel

Sie können Unterabfragen verwenden, um eine temporäre Tabelle zu definieren und in der

FROM-Klausel einer "äußeren" Abfrage zu verwenden.

```
SELECT * FROM (SELECT city, temp_hi - temp_lo AS temp_var FROM weather) AS w
WHERE temp_var > 20;
```

Oben werden Städte aus der Wettertabelle gefunden, deren tägliche Temperaturschwankung größer als 20 ist. Das Ergebnis ist:

Stadt	temp_var
ST LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

.

Unterabfragen in der WHERE-Klausel

Im folgenden Beispiel werden Städte (aus dem Beispiel für Städte) gefunden, deren Bevölkerung unter der Durchschnittstemperatur liegt (über eine Unterabfrage ermittelt):

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000)  FROM cities);
```

Hier: Mit der Unterabfrage (SELECT avg (pop2000) FROM cities) werden Bedingungen in der WHERE-Klausel angegeben. Das Ergebnis ist:

Name	pop2000
San Francisco	776733
ST LOUIS	348189
Kansas City	146866

Unterabfragen in SELECT-Klausel

Unterabfragen können auch im `SELECT` Teil der äußeren Abfrage verwendet werden. Die folgende Abfrage zeigt alle Wettertabellenspalten mit den entsprechenden Bundesländern aus der Städtetabelle .

```
SELECT w.*,  (SELECT c.state FROM cities AS c WHERE c.name = w.city ) AS state
FROM weather AS w;
```

Filtern Sie die Abfrageergebnisse mithilfe der Abfrage in einer anderen Tabelle

Diese Abfrage wählt alle Mitarbeiter aus, die sich nicht in der Supervisors-Tabelle befinden.

```
SELECT *
FROM Employees
WHERE EmployeeID not in (SELECT EmployeeID
                            FROM Supervisors)
```

Die gleichen Ergebnisse können mit einem LEFT JOIN erzielt werden.

```
SELECT *
FROM Employees AS e
LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID is NULL
```

Korrelierte Unterabfragen

Korrelierte (auch als synchronisierte oder koordinierte) Unterabfragen bezeichnet verschachtelte Abfragen, die auf die aktuelle Zeile ihrer äußeren Abfrage verweisen:

```
SELECT EmployeeId
    FROM Employee AS eOuter
    WHERE Salary > (
       SELECT AVG(Salary)
       FROM Employee eInner
       WHERE eInner.DepartmentId = eOuter.DepartmentId
    )
```

Die Unterabfrage `SELECT AVG(Salary)` ... ist *korreliert,* da sie von ihrer äußeren Abfrage auf die `Employee eOuter` verweist.

Unterabfragen online lesen: https://riptutorial.com/de/sql/topic/1606/unterabfragen

Kapitel 60: VERSCHMELZEN

Einführung

MERGE (oft auch UPSERT für "Update or Insert" genannt) ermöglicht das Einfügen neuer Zeilen oder, falls bereits eine Zeile vorhanden ist, die vorhandene Zeile zu aktualisieren. Es geht darum, den gesamten Satz von Operationen atomar auszuführen (um sicherzustellen, dass die Daten konsistent bleiben) und um den Kommunikationsaufwand für mehrere SQL-Anweisungen in einem Client / Server-System zu verhindern.

Examples

MERGE, um das Ziel mit der Quelle zu verbinden

```
MERGE INTO targetTable t
    USING sourceTable s
        ON t.PKID = s.PKID
    WHEN MATCHED AND NOT EXISTS (
            SELECT s.ColumnA, s.ColumnB, s.ColumnC
            INTERSECT
            SELECT t.ColumnA, t.ColumnB, s.ColumnC
            )
        THEN UPDATE SET
            t.ColumnA = s.ColumnA
            ,t.ColumnB = s.ColumnB
            ,t.ColumnC = s.ColumnC
    WHEN NOT MATCHED BY TARGET
        THEN INSERT (PKID, ColumnA, ColumnB, ColumnC)
        VALUES (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
    WHEN NOT MATCHED BY SOURCE
        THEN DELETE
    ;
```

Hinweis: Der Abschnitt `AND NOT EXISTS` verhindert die Aktualisierung von Datensätzen, die nicht geändert wurden. Durch die Verwendung des `INTERSECT` Konstrukts können nullfähige Spalten ohne besondere Behandlung verglichen werden.

MySQL: Benutzer nach Namen zählen

Angenommen, wir möchten wissen, wie viele Benutzer denselben Namen haben. Lassen Sie uns Tabelle erstellen `users` wie folgt:

```
create table users(
    id int primary key auto_increment,
    name varchar(8),
    count int,
    unique key name(name)
);
```

Wir haben gerade einen neuen Benutzer namens Joe entdeckt und möchten ihn gerne berücksichtigen. Um dies zu erreichen, müssen wir feststellen, ob eine Zeile mit seinem Namen vorhanden ist. Wenn ja, aktualisieren Sie sie, um die Anzahl zu erhöhen. Wenn es jedoch keine Zeile gibt, sollten wir sie erstellen.

MySQL verwendet die folgende Syntax: insert… bei Duplikatschlüssel-Update… . In diesem Fall:

```
insert into users(name, count)
       values ('Joe', 1)
       on duplicate key update count=count+1;
```

PostgreSQL: Benutzer nach Namen zählen

Angenommen, wir möchten wissen, wie viele Benutzer denselben Namen haben. Lassen Sie uns Tabelle erstellen `users` wie folgt:

```
create table users(
    id serial,
    name varchar(8) unique,
    count int
);
```

Wir haben gerade einen neuen Benutzer namens Joe entdeckt und möchten ihn gerne berücksichtigen. Um dies zu erreichen, müssen wir feststellen, ob eine Zeile mit seinem Namen vorhanden ist. Wenn ja, aktualisieren Sie sie, um die Anzahl zu erhöhen. Wenn es jedoch keine Zeile gibt, sollten wir sie erstellen.

PostgreSQL verwendet die folgende Syntax: insert ... in Konflikt ... Aktualisierung In diesem Fall:

```
insert into users(name, count)
    values('Joe', 1)
    on conflict (name) do update set count = users.count + 1;
```

VERSCHMELZEN online lesen: https://riptutorial.com/de/sql/topic/1470/verschmelzen

Kapitel 61: VERSUCHEN / FANGEN

Bemerkungen

TRY / CATCH ist ein für T SQL von MS SQL Server spezifisches Sprachkonstrukt.

Es ermöglicht die Fehlerbehandlung innerhalb von T-SQL, ähnlich wie in .NET-Code.

Examples

Transaktion in einem TRY / CATCH

Dadurch werden beide Inserts aufgrund einer ungültigen Datumszeit zurückgesetzt:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Dadurch werden beide Einfügungen festgelegt:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

VERSUCHEN / FANGEN online lesen: https://riptutorial.com/de/sql/topic/4420/versuchen---fangen

Kapitel 62: WÄHLEN

Einführung

Die SELECT-Anweisung steht im Mittelpunkt der meisten SQL-Abfragen. Sie definiert, welche Ergebnismenge von der Abfrage zurückgegeben werden soll, und wird fast immer in Verbindung mit der FROM-Klausel verwendet, die definiert, welche Teile der Datenbank abgefragt werden sollen.

Syntax

- SELECT [DISTINCT] [Spalte1] [, [Spalte2] ...]
  FROM [Tabelle]
  [WO Bedingung]
  [GROUP BY [Spalte1] [, [Spalte2] ...]

  [HABEN [Spalte1] [, [Spalte2] ...]

  [BESTELLEN VON ASC | DESC]

Bemerkungen

SELECT bestimmt, welche Spaltendaten in welcher Reihenfolge FROM in einer bestimmten Tabelle zurückgegeben werden sollen (vorausgesetzt, sie entsprechen den anderen Anforderungen Ihrer Abfrage, und zwar wo und mit Filtern und Verknüpfungen).

```
SELECT Name, SerialNumber
FROM ArmyInfo
```

gibt nur Ergebnisse aus den Spalten `Name` und `Serial Number` , nicht jedoch aus der Spalte `Rank`

```
SELECT *
FROM ArmyInfo
```

zeigt an, dass **alle** Spalten zurückgegeben werden. Beachten Sie jedoch, dass es üblich ist, `SELECT *` da Sie buchstäblich alle Spalten einer Tabelle zurückgeben.

Examples

Verwenden Sie das Platzhalterzeichen, um alle Spalten in einer Abfrage auszuwählen.

Betrachten Sie eine Datenbank mit den folgenden zwei Tabellen.

Mitarbeiter-Tabelle:

Ich würde	FName	LName	DeptId
1	James	Schmied	3
2	John	Johnson	4

Abteilungen Tabelle:

Ich würde	Name
1	Der Umsatz
2	Marketing
3	Finanzen
4	ES

Einfache select-Anweisung

`*` ist das **Platzhalterzeichen** , mit dem alle verfügbaren Spalten in einer Tabelle ausgewählt werden.

Bei der Verwendung als Ersatz für explizite Spaltennamen werden alle Spalten in allen Tabellen zurückgegeben, für die eine Abfrage `FROM` . Dieser Effekt gilt für **alle Tabellen, auf die** die Abfrage über ihre `JOIN` Klauseln zugreift.

Betrachten Sie die folgende Abfrage:

```
SELECT * FROM Employees
```

Es werden alle Felder aller Zeilen der `Employees` Tabelle zurückgegeben:

Ich würde	FName	LName	DeptId
1	James	Schmied	3
2	John	Johnson	4

Punktnotation

Um alle Werte aus einer bestimmten Tabelle auszuwählen, kann das Platzhalterzeichen mit *Punktnotation* auf die Tabelle angewendet *werden* .

Betrachten Sie die folgende Abfrage:

```
SELECT
    Employees.*,
    Departments.Name
FROM
    Employees
JOIN
    Departments
    ON Departments.Id = Employees.DeptId
```

Dadurch wird ein Datensatz mit allen Feldern in der `Employee` Tabelle zurückgegeben, gefolgt von dem Feld `Name` in der `Departments` Tabelle:

Ich würde	FName	LName	DeptId	Name
1	James	Schmied	3	Finanzen
2	John	Johnson	4	ES

Warnungen vor dem Gebrauch

Im Allgemeinen wird empfohlen, die Verwendung von `*` im Produktionscode möglichst zu vermeiden, da dies eine Reihe potenzieller Probleme verursachen kann, darunter:

1. Überschüssiges E / A, Netzwerklast, Speicherauslastung usw., da die Datenbank-Engine nicht benötigte Daten liest und an den Front-End-Code überträgt. Dies ist insbesondere dann von Belang, wenn große Felder vorhanden sind, beispielsweise zum Speichern langer Notizen oder angehängter Dateien.
2. Weitere übermäßige E / A-Belastung, wenn die Datenbank interne Ergebnisse als Teil der Verarbeitung für eine komplexere Abfrage als `SELECT <columns> FROM <table>` auf die Festplatte `SELECT <columns> FROM <table>` .
3. Zusätzliche Verarbeitung (und / oder sogar mehr E / A), wenn einige der nicht benötigten Spalten folgende sind:
    - berechnete Spalten in Datenbanken, die sie unterstützen
    - Im Falle der Auswahl aus einer Ansicht Spalten aus einer Tabelle / Ansicht, die der Abfrageoptimierer ansonsten optimieren könnte
4. Das Potenzial für unerwartete Fehler, wenn zu Tabellen und Ansichten später Spalten hinzugefügt werden, die mehrdeutige Spaltennamen ergeben. Zum Beispiel `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` - Wenn der Spalte " `displayname` eine Spalte mit dem Namen `displayname` hinzugefügt wird, damit Benutzer ihren Bestellungen aussagekräftige Namen für die spätere Bezugnahme geben können, wird der Spaltenname `displayname` zweimal in der Ausgabe, so dass die `ORDER BY` Klausel mehrdeutig ist, was zu Fehlern führen kann ("mehrdeutiger Spaltenname" in den letzten MS SQL Server-Versionen), und wenn dies nicht der Fall ist, zeigt der Anwendungscode möglicherweise den Auftragsnamen an, in dem sich der Personenname befindet bestimmt, weil die neue Spalte der erste Name dieses Namens ist und so weiter.

Wann können Sie `*` , um die obige Warnung zu beachten?

In Produktionscode am besten vermieden, ist die Verwendung von `*` als Abkürzung für manuelle

Abfragen der Datenbank für Untersuchungen oder Prototypen geeignet.

Designentscheidungen in Ihrer Anwendung machen es manchmal unvermeidlich (bevorzugen `tablealias.*` in solchen Fällen `tablealias.*` nur `*` wenn möglich).

Wenn Sie `EXISTS` , z. B. `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` , werden keine Daten von B zurückgegeben. Daher ist ein Join nicht `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` Engine weiß, dass keine Werte von B zurückgegeben werden sollen, daher wird für die Verwendung von `*` kein Performance-Treffer erzielt. Ebenso ist `COUNT(*)` in Ordnung, da es auch keine der Spalten zurückgibt. Es müssen also nur diejenigen gelesen und verarbeitet werden, die für Filterzwecke verwendet werden.

Auswahl mit Bedingung

Die Grundsyntax der Klausel SELECT mit WHERE lautet:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

Die *[Bedingung]* kann ein beliebiger SQL-Ausdruck sein, der mit Vergleichs- oder logischen Operatoren wie>, <, =, <>,> =, <=, LIKE, NOT, IN, BETWEEN usw. angegeben wird

Die folgende Anweisung gibt alle Spalten aus der Tabelle 'Cars' zurück, in der die Statusspalte 'READY' lautet:

```
SELECT * FROM Cars WHERE status = 'READY'
```

Weitere Beispiele finden Sie unter WO und HAVING .

Wählen Sie einzelne Spalten aus

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
FROM Customers
```

Diese Anweisung gibt die Spalten `PhoneNumber` , `Email` und `PreferredContact` aus allen Zeilen der `Customers` Tabelle zurück. Die Spalten werden auch in der Reihenfolge zurückgegeben, in der sie in der `SELECT` Klausel erscheinen.

Das Ergebnis wird sein:

Telefonnummer	Email	PreferredContact
3347927472	william.jones@example.com	TELEFON
2137921892	dmiller@example.net	EMAIL

Telefonnummer	Email	PreferredContact
NULL	richard0123@example.com	EMAIL

Wenn mehrere Tabellen zusammengefügt werden, können Sie Spalten aus bestimmten Tabellen auswählen, indem Sie den Tabellennamen vor dem Spaltennamen `[table_name].[column_name]` : `[table_name].[column_name]`

```
SELECT
    Customers.PhoneNumber,
    Customers.Email,
    Customers.PreferredContact,
    Orders.Id AS OrderId
FROM
    Customers
LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

* `AS OrderId` bedeutet, dass das `Id` Feld der `Orders` Tabelle als Spalte mit dem Namen `OrderId` . Weitere Informationen finden Sie unter Auswählen mit Spaltenalias .

Um zu vermeiden, lange Tabellennamen zu verwenden, können Sie Tabellen-Aliase verwenden. Dadurch wird das Schreiben von langen Tabellennamen für jedes Feld, das Sie in den Joins auswählen, verringert. Wenn Sie eine Selbstverknüpfung durchführen (eine Verknüpfung zwischen zwei Instanzen *derselben* Tabelle), müssen Sie Tabellenaliasnamen verwenden, um Ihre Tabellen zu unterscheiden. Wir können einen Tabellenalias wie `Customers c` oder `Customers AS c` schreiben. Hier fungiert `c` als Alias für `Customers` und wir können `Email` wie `c.Email` auswählen: `c.Email` .

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact,
    o.Id AS OrderId
FROM
    Customers c
LEFT JOIN
    Orders o ON o.CustomerId = c.Id
```

SELECT Verwenden Sie Spaltenaliasnamen

Spaltenaliase werden hauptsächlich verwendet, um Code zu verkürzen und Spaltennamen lesbarer zu machen.

Code wird kürzer, da lange Tabellennamen und unnötige Identifizierung von Spalten *(z. B. zwei IDs in der Tabelle, aber nur eine in der Anweisung verwendet wird)* vermieden werden können. Zusammen mit Tabellenaliasnamen können Sie in der Datenbankstruktur längere beschreibende Namen verwenden, während die Abfragen für diese Struktur kurz gehalten werden.

Darüber hinaus sind sie manchmal *erforderlich* , zum Beispiel in Ansichten, um berechnete Ausgaben zu benennen.

Alle Versionen von SQL

Aliase können in allen SQL-Versionen mit Anführungszeichen (`"`) erstellt werden.

```
SELECT
    FName AS "First Name",
    MName AS "Middle Name",
    LName AS "Last Name"
FROM Employees
```

Verschiedene SQL-Versionen

Sie können einfache Anführungszeichen (`'`), doppelte Anführungszeichen (`"`) und eckige Klammern (`[]`) verwenden, um einen Alias in Microsoft SQL Server zu erstellen.

```
SELECT
    FName AS "First Name",
    MName AS 'Middle Name',
    LName AS [Last Name]
FROM Employees
```

Beides führt zu:

Vorname	Zweiter Vorname	Nachname
James	John	Schmied
John	James	Johnson
Michael	Marcus	Williams

Diese Anweisung `LName` Spalten `FName` und `LName` mit einem angegebenen Namen (einem Alias) zurück. Dies wird erreicht, indem der `AS` Operator gefolgt von dem Alias verwendet wird oder indem der Alias direkt nach dem Spaltennamen geschrieben wird. Dies bedeutet, dass die folgende Abfrage dasselbe Ergebnis wie oben ergibt.

```
SELECT
    FName "First Name",
    MName "Middle Name",
    LName "Last Name"
FROM Employees
```

Vorname	Zweiter Vorname	Nachname
James	John	Schmied
John	James	Johnson
Michael	Marcus	Williams

Die explizite Version (dh die Verwendung des `AS` Operators) ist jedoch besser lesbar.

Wenn der Alias ein einzelnes Wort hat, das kein reserviertes Wort ist, können wir es ohne einfache Anführungszeichen, doppelte Anführungszeichen oder Klammern schreiben:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

Vorname	Nachname
James	Schmied
John	Johnson
Michael	Williams

Eine weitere Variante, die unter anderem in MS SQL Server verfügbar ist, ist `<alias> = <column-or-calculation>` , zum Beispiel:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1    = FullStreetAddress,
       Addr2    = TownName
FROM CustomerDetails
```

was äquivalent ist zu:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress          As Addr1,
       TownName                   As Addr2
FROM CustomerDetails
```

Beides führt zu:

Vollständiger Name	Addr1	Addr2
James Smith	123 AnyStreet	Townville
John Johnson	668 MyRoad	Irgendeine Stadt
Michael Williams	999 High End Dr	Williamsburgh

Einige finden using `=` anstelle von `As` einfacher zu lesen, obwohl viele dieses Format empfehlen, vor allem, weil es nicht standardmäßig ist und daher nicht von allen Datenbanken unterstützt wird. Es kann zu Verwechslungen mit anderen Verwendungen des Zeichens `=` .

Alle Versionen von SQL

Auch, wenn Sie reservierte Wörter verwenden *müssen,* können Sie Klammern oder Anführungszeichen zu entkommen:

```
SELECT
    FName as "SELECT",
    MName as "FROM",
    LName as "WHERE"
FROM Employees
```

Verschiedene SQL-Versionen

Ebenso können Sie Schlüsselwörter in MSSQL mit allen unterschiedlichen Ansätzen schützen:

```
SELECT
    FName AS "SELECT",
    MName AS 'FROM',
    LName AS [WHERE]
FROM Employees
```

WÄHLEN	VON	WOHER
James	John	Schmied
John	James	Johnson
Michael	Marcus	Williams

Ein Spaltenalias kann auch eine der abschließenden Klauseln derselben Abfrage verwenden, beispielsweise eine `ORDER BY` :

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
    Employees
ORDER BY
    LastName DESC
```

Sie dürfen jedoch *nicht* verwenden

```
SELECT
    FName AS SELECT,
    LName AS FROM
FROM
    Employees
ORDER BY
    LastName DESC
```

So erstellen Sie einen Alias aus diesen reservierten Wörtern (`SELECT` und `FROM`).

Dies führt zu zahlreichen Fehlern bei der Ausführung.

Auswahl mit sortierten Ergebnissen

```
SELECT * FROM Employees ORDER BY LName
```

Diese Anweisung gibt alle Spalten aus der Tabelle `Employees` .

Ich würde	FName	LName	Telefonnummer
2	John	Johnson	2468101214
1	James	Schmied	1234567890
3	Michael	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

Oder

```
 SELECT * FROM Employees ORDER BY LName ASC
```

Diese Anweisung ändert die Sortierrichtung.

Man kann auch mehrere Sortierspalten angeben. Zum Beispiel:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

In diesem Beispiel werden die Ergebnisse zuerst nach `LName` und dann für Datensätze mit demselben `LName` nach `FName` . Dadurch erhalten Sie ein ähnliches Ergebnis wie in einem Telefonbuch.

Um die erneute Eingabe des Spaltennamens in der `ORDER BY` Klausel zu speichern, können Sie stattdessen die Spaltennummer verwenden. Beachten Sie, dass die Spaltennummern mit 1 beginnen.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

Sie können auch eine `CASE` Anweisung in die `ORDER BY` Klausel einbetten.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones` THEN 0
ELSE 1 END ASC
```

Dadurch werden Ihre Ergebnisse so sortiert, dass alle Datensätze mit dem `LName` von "Jones" oben `LName` .

Wählen Sie Spalten aus, die nach reservierten Schlüsselwörtern benannt sind

Wenn ein Spaltenname mit einem reservierten Schlüsselwort übereinstimmt, müssen Sie für

Standard-SQL-Anführungszeichen doppelte Anführungszeichen verwenden:

```
SELECT
    "ORDER",
    ID
FROM ORDERS
```

Beachten Sie, dass der Spaltenname zwischen Groß- und Kleinschreibung unterscheidet.

Einige Datenbankverwaltungssysteme verfügen über proprietäre Möglichkeiten, Namen anzugeben. Beispielsweise verwendet SQL Server für diesen Zweck eckige Klammern:

```
SELECT
    [Order],
    ID
FROM ORDERS
```

Während MySQL (und MariaDB) standardmäßig Backticks verwenden:

```
SELECT
    `Order`,
    id
FROM orders
```

Festlegen der angegebenen Anzahl von Datensätzen

Der SQL 2008-Standard definiert die Klausel `FETCH FIRST` , um die Anzahl der zurückgegebenen Datensätze zu begrenzen.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

Dieser Standard wird nur in neueren Versionen einiger RDMS unterstützt. In anderen Systemen wird herstellerspezifische Nicht-Standardsyntax bereitgestellt. Progress OpenEdge 11.x unterstützt auch die Syntax `FETCH FIRST <n> ROWS ONLY` .

Darüber hinaus `OFFSET <m> ROWS` vor `FETCH FIRST <n> ROWS ONLY` Überspringen von Zeilen, bevor Zeilen `OFFSET <m> ROWS` .

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

Die folgende Abfrage wird in SQL Server und MS Access unterstützt:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
```

```
ORDER BY UnitPrice DESC
```

Um dies in MySQL oder PostgreSQL zu tun, muss das Schlüsselwort `LIMIT` verwendet werden:

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

In Oracle kann das gleiche mit `ROWNUM` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Ergebnisse : **10** Datensätze.

```
Id    ProductName               UnitPrice             Package
38    Côte de Blaye             263.50                12 - 75 cl bottles
29    Thüringer Rostbratwurst   123.79                50 bags x 30 sausgs.
9    Mishi Kobe Niku            97.00                 18 - 500 g pkgs.
20    Sir Rodney's Marmalade    81.00                 30 gift boxes
18    Carnarvon Tigers          62.50                 16 kg pkg.
59    Raclette Courdavault      55.00                 5 kg pkg.
51    Manjimup Dried Apples     53.00                 50 - 300 g pkgs.
62    Tarte au sucre            49.30                 48 pies
43    Ipoh Coffee               46.00                 16 - 500 g tins
28    Rössle Sauerkraut         45.60                 25 - 825 g cans
```

Verkäufer Nuances:

Es ist wichtig , dass die beachten `TOP` in Microsoft SQL nach der arbeitet `WHERE` - Klausel und die angegebene Anzahl der Ergebnisse zurück , wenn sie überall in der Tabelle existieren, während `ROWNUM` im Rahmen der Arbeiten `WHERE` - Klausel so , wenn andere Bedingungen existieren nicht in dem Wenn Sie eine bestimmte Anzahl von Zeilen am Anfang der Tabelle angeben, erhalten Sie null Ergebnisse, wenn andere gefunden werden können.

Auswahl mit Tabellenalias

```
SELECT e.Fname, e.LName
FROM Employees e
```

Die Employees-Tabelle erhält den Alias 'e' direkt nach dem Tabellennamen. Dies hilft, Mehrdeutigkeiten in Szenarien zu beseitigen, in denen mehrere Tabellen denselben Feldnamen haben und Sie müssen angeben, aus welcher Tabelle Sie Daten zurückgeben möchten.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
    JOIN Managers m ON e.ManagerId = m.Id
```

Wenn Sie einen Alias definiert haben, können Sie den kanonischen Tabellennamen nicht mehr verwenden. dh

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

würde einen Fehler werfen.

Es ist erwähnenswert, dass Tabellenaliasnamen - formell 'Bereichsvariablen' - in die SQL-Sprache eingeführt wurden, um das Problem der durch `INNER JOIN` verursachten doppelten Spalten zu lösen. Der SQL-Standard von 1992 korrigierte diesen früheren Konstruktionsfehler durch Einführung von `NATURAL JOIN` (implementiert in mySQL, PostgreSQL und Oracle, aber noch nicht in SQL Server), dessen Ergebnis niemals doppelte Spaltennamen hat. Das obige Beispiel ist `ManagerId` interessant, als die Tabellen in Spalten mit unterschiedlichen Namen (`Id` und `ManagerId`) zusammengefügt werden, jedoch nicht in den Spalten mit demselben Namen (`LName` , `FName`), die die Spalten umbenennen müssen *vor* dem join:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
     NATURAL JOIN
     ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
       FROM Managers ) m;
```

Beachten Sie, dass zwar eine Alias- / Bereichsvariable für die dervied-Tabelle deklariert werden muss (andernfalls löst SQL einen Fehler aus), es ist jedoch niemals sinnvoll, sie in der Abfrage zu verwenden.

Zeilen aus mehreren Tabellen auswählen

```
SELECT *
FROM
    table1,
    table2
```

```
SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1,
    table2
```

Dieses Produkt wird in SQL als Kreuzprodukt bezeichnet. Es ist dasselbe wie ein Kreuzprodukt in Sets

Diese Anweisungen geben die ausgewählten Spalten aus mehreren Tabellen in einer Abfrage zurück.

Es gibt keine bestimmte Beziehung zwischen den von jeder Tabelle zurückgegebenen Spalten.

Durchschnittlich

Die Aggregatfunktion `AVG()` gibt den Durchschnitt der ausgewählten Werte zurück.

```
SELECT AVG(Salary) FROM Employees
```

Aggregatfunktionen können auch mit der where-Klausel kombiniert werden.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Aggregatfunktionen können auch mit der Group-by-Klausel kombiniert werden.

Wenn ein Mitarbeiter nach mehreren Abteilungen kategorisiert ist und wir für jede Abteilung einen Durchschnittslohn suchen, können wir die folgende Abfrage verwenden.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Minimum

Die Aggregatfunktion `MIN()` gibt das Minimum der ausgewählten Werte zurück.

```
SELECT MIN(Salary) FROM Employees
```

Maximal

Die Aggregatfunktion `MAX()` gibt das Maximum der ausgewählten Werte zurück.

```
SELECT MAX(Salary) FROM Employees
```

Anzahl

Die Aggregatfunktion `COUNT()` gibt die Anzahl der ausgewählten Werte zurück.

```
SELECT Count(*) FROM Employees
```

Sie kann auch mit "where" -Bedingungen kombiniert werden, um die Anzahl der Zeilen zu ermitteln, die bestimmte Bedingungen erfüllen.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

Es können auch bestimmte Spalten angegeben werden, um die Anzahl der Werte in der Spalte abzurufen. Beachten Sie, dass `NULL` Werte nicht gezählt werden.

```
Select Count(ManagerId) from Employees
```

Die Zählung kann auch mit dem eindeutigen Schlüsselwort für eine eindeutige Zählung kombiniert werden.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Summe

Die Aggregatfunktion `SUM()` gibt die Summe der für alle Zeilen ausgewählten Werte zurück.

```
SELECT SUM(Salary) FROM Employees
```

Auswahl mit null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

Die Auswahl mit Nullen hat eine andere Syntax. Verwenden Sie nicht `=` , verwenden Sie stattdessen `IS NULL` oder `IS NOT NULL` .

Auswahl mit CASE

Wenn für die Ergebnisse eine sofortige Anwendung der Logik erforderlich ist, können Sie die CASE-Anweisung verwenden, um sie zu implementieren.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

kann auch verkettet werden

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         WHEN Col1 > 50 AND Col1 <100 THEN 'between'
         ELSE 'over'
    END threshold
FROM TableName
```

man kann auch `CASE` in einer anderen `CASE` Anweisung haben

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         ELSE
            CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
            ELSE 'over' END
    END threshold
FROM TableName
```

Auswählen ohne die Tabelle zu sperren

Wenn Tabellen meistens (oder nur) für Lesevorgänge verwendet werden, hilft die Indexierung nicht mehr und jedes bisschen zählt, man kann Selects ohne LOCK verwenden, um die Leistung zu verbessern.

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Orakel

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

wobei `UR` für "Uncommitted Read" steht.

Bei Verwendung in einer Tabelle mit laufenden Datensatzänderungen können unvorhersehbare Ergebnisse auftreten.

Wähle eindeutig (nur eindeutige Werte)

```
SELECT DISTINCT ContinentCode
FROM Countries;
```

Diese Abfrage gibt alle `DISTINCT` Werte (unique, different) aus der `ContinentCode` Spalte aus der `Countries` Tabelle zurück

ContinentCode
OC
EU
WIE
N / A

ContinentCode
AF

SQLFiddle-Demo

Wählen Sie mit der Bedingung mehrerer Werte aus der Spalte

```
SELECT * FROM Cars  WHERE status IN ( 'Waiting', 'Working' )
```

Dies ist semantisch äquivalent zu

```
SELECT * FROM Cars  WHERE ( status = 'Waiting' OR status = 'Working' )
```

dh `value IN ( <value list> )` ist eine Abkürzung für Disjunktion (logisches `OR`).

Sammelt das Ergebnis für Zeilengruppen

Zeilen basierend auf einem bestimmten Spaltenwert zählen:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Durchschnittliches Einkommen nach Abteilungen erzielen:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

Wichtig ist, nur die in der `GROUP BY` Klausel angegebenen Spalten auszuwählen oder mit Aggregatfunktionen zu verwenden .

Die `WHERE` Klausel kann auch mit `GROUP BY` , aber `WHERE` filtert Datensätze heraus, *bevor* eine Gruppierung vorgenommen wird:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

Wenn Sie die Ergebnisse nach der Gruppierung filtern müssen, um z. B. nur Abteilungen `HAVING` , deren durchschnittliches Einkommen mehr als 1000 beträgt, müssen Sie die `HAVING` Klausel verwenden:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
```

```
GROUP BY department
HAVING avg(income) > 1000;
```

Auswahl mit mehr als 1 Bedingung.

Das `AND` Schlüsselwort wird verwendet, um der Abfrage weitere Bedingungen hinzuzufügen.

Name	Alter	Geschlecht
Sam	18	M
John	21	M
Bob	22	M
Maria	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

Dies wird zurückkehren:

Name
John
Bob

Verwenden Sie das `OR` Schlüsselwort

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

Dies wird zurückkehren:

Name
Sam
John
Bob

Diese Schlüsselwörter können kombiniert werden, um komplexere Kriterienkombinationen zu ermöglichen:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
   OR (gender = 'F' AND age > 20);
```

Dies wird zurückkehren:

Name
Sam
Maria

WÄHLEN online lesen: https://riptutorial.com/de/sql/topic/222/wahlen

Kapitel 63: XML

Examples

Abfrage vom XML-Datentyp

```
DECLARE @xmlIN XML = '<TableData>
<aaa Main="First">
  <row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value',  'VARCHAR(25)') [Value]
FROM    @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

Ergebnisse

```
Header    name    Value
First      a        1
First      b        2
First      c        3
Second     a        3
Second     b        4
Second     c        5
Third      a        10
Third      b        20
Third      c        30
```

XML online lesen: https://riptutorial.com/de/sql/topic/4421/xml

Kapitel 64: Zeilennummer

Syntax

- ZEILENNUMMER ( )
- OVER ([PARTITION BY value_expression, ... [n]] order_by_clause)

Examples

Zeilennummern ohne Partitionen

Fügen Sie eine Zeilennummer gemäß der angegebenen Reihenfolge ein.

```
SELECT
  ROW_NUMBER() OVER(ORDER BY Fname ASC) AS RowNumber,
  Fname,
  LName
FROM Employees
```

Zeilennummern mit Partitionen

Verwendet ein Partitionskriterium, um die Zeilennummerierung danach zu gruppieren.

```
SELECT
  ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC) AS RowNumber,
  DepartmentId, Fname, LName
FROM Employees
```

Alle bis auf den letzten Datensatz löschen (1 bis viele Tabelle)

```
WITH cte AS (
  SELECT ProjectID,
         ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn
  FROM ProjectNotes
)
DELETE FROM cte WHERE rn > 1;
```

Zeilennummer online lesen: https://riptutorial.com/de/sql/topic/1977/zeilennummer

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit SQL	Arjan Einbu, brichins, Burkhard, cale_b, CL., Community, Devmati Wadikar, Epodax, geeksal, H. Pauwelyn, Hari, Joey, JohnLBevan, Jon Ericson, Lankymart, Laurel, Mureinik, Nathan, omini data, PeterRing, Phrancis, Prateek, RamenChef, Ray, Simone Carletti, SZenC, t1gor, ypercube
2	AKTUALISIEREN	Akshay Anand, CL., Daniel Vérité, Dariusz, Dipesh Poudel, FlyingPiMonster, Gidil, H. Pauwelyn, Jon Chan, KIRAN KUMAR MATAM, Matas Vaitkevicius, Matt, Phrancis, Sanjay Bharwani, sunkuet02, Tot Zam, TriskalJM, vmaroli, WesleyJohnson
3	Allgemeine Tabellenausdrücke	CL., Daniel, dd4711, fuzzy_logic, Gidil, Luis Lema, ninesided, Peter K, Phrancis, Sibeesh Venu
4	ALTER TABELLE	Aidan, blackbishop, bluefeet, CL., Florin Ghita, Francis Lord, guiguiblitz, Joe W, KIRAN KUMAR MATAM, Lexi, mithra chintha , Ozair Kafray, Simon Foster, Siva Rama Krishna
5	Ansichten	Amir978, CL., Florin Ghita
6	Ausführungsblöcke	Phrancis
7	AUSSER	LCIII
8	Beispieldatenbanken und Tabellen	Abhilash R Vankayala, Arulkumar, Athafoud, bignose, Bostjan, Brad Larson, Christian, CL., Dariusz, Dr. J. Testington, enrico.bacis, Florin Ghita, FlyingPiMonster, forsvarir, Franck Dernoncourt, hairboat, JavaHopper, Jaydles, Jon Ericson, Magisch, Matt, Mureinik, Mzzzzzz, Prateek, rdans, Shiva, tinlyx, Tot Zam, WesleyJohnson
9	BEITRETEN	A_Arnold, Akshay Anand, Andy G, bignose, Branko Dimitrijevic, Casper Spruit, CL., Daniel Langemann, Darren Bartrup-Cook, Dipesh Poudel, enrico.bacis, Florin Ghita, forsvarir, Franck Dernoncourt, hairboat, Hari K M, HK1, HLGEM, inquisitive_mind, John C, John Odom, John Slegers, Mark Iannucci, Marvin, Mureinik, Phrancis, raholling, Raidri, Saroj Sasmal, Stefan Steiger, sunkuet02, Tot Zam, xenodevil, ypercube, Paxул Маквана
10	Bemerkungen	CL., Phrancis

11	Bereinigen Sie Code in SQL	CL., Stivan
12	Cascading Delete	Stefan Steiger
13	Datenbank erstellen	Emil Rowland
14	Datentypen	bluefeet, Jared Hooper, John Odom, Jon Chan, JonMark Perry, Phrancis
15	DROP oder DELETE Database	Abhilash R Vankayala, John Odom
16	DROP-Tabelle	CL., Joel, KIRAN KUMAR MATAM, Stu
17	Duplikate in einem Spalten-Subset mit Detail suchen	Darrel Lee, mnoronha
18	EINFÜGEN	Ameya Deshpande, CL., Daniel Langemann, Dipesh Poudel, inquisitive_mind, KIRAN KUMAR MATAM, rajarshig, Tot Zam, zplizzi
19	EXISTS-Klausel	Blag, Özgür Öztürk
20	EXPLAIN und BESCHREIBEN	Simulant
21	FALL	ɐləx, Christos, CL., Dariusz, Fenton, Infinity, Jaydles, Matt, MotKohn, Mureinik, Peter Lang, Stanislovas Kalašnikovas
22	Fensterfunktionen	Arkh, beercohol, bhs, Gidil, Jerry Jeremiah, Mureinik, mustaccio
23	Filtern Sie die Ergebnisse mit WHERE und HAVING	Arulkumar, Bostjan, CL., Community, Franck Dernoncourt, H. Pauwelyn, Jon Chan, Jon Ericson, juergen d, Matas Vaitkevicius , Mureinik, Phrancis, Tot Zam
24	Fremde Schlüssel	CL., Harjot, Yehuda Shapira
25	FUNKTION ERSTELLEN	John Odom, Ricardo Pontual
26	Funktionen (Aggregat)	ashja99, CL., Florin Ghita, Ian Kenney, Imran Ali Khan, Jon Chan, juergen d, KIRAN KUMAR MATAM, Mark Stewart, Maverick, Nathan, omini data, Peter K, Reboot, Tot Zam, William Ledbetter, winseybash, Алексей Неудачин
27	Funktionen (Analyse)	CL., omini data

28	Funktionen (Skalar / Einzelne Reihe)	CL., Kewin Björk Nielsen, Mark Stewart
29	Gespeicherte Prozeduren	brichins, John Odom, Lamak, Ryan
30	GRANT und REVOKE	RamenChef, user2314737
31	GRUPPIERE NACH	3N1GM4, Abe Miessler, Bostjan, Devmati Wadikar, Filipe Manuel, Frank, Gidil, Jaydles, juergen d, Nathaniel Ford, Peter Gordon, Simone - Ali One, WesleyJohnson, Zahiro Mor, Zoyd
32	Indizes	a1ex07, Almir Vuk, carlosb, CL., David Manheim, FlyingPiMonster, forsvarir, Franck Dernoncourt, Horaciux, Jenism, KIRAN KUMAR MATAM, mauris, Parado, Paulo Freitas, Ryan
33	Informationsschema	Hack-R
34	IN-Klausel	CL., juergen d, walid, Zaga
35	Kennung	Andreas, CL.
36	Kreuz anwenden, außen anwenden	Karthikeyan, RamenChef
37	KÜRZEN	Abhilash R Vankayala, CL., Cristian Abelleira, DaImTo, Hynek Bernard, inquisitive_mind, KIRAN KUMAR MATAM, Paul Bambury, ss005
38	LIKE Operator	Abhilash R Vankayala, Aidan, ashja99, Bart Schuijt, CL., Cristian Abelleira, guiguiblitz, Harish Gyanani, hellyale, Jenism, Lohitha Palagiri, Mark Perera, Mr. Developer, Ojen, Phrancis, RamenChef, Redithion, Stefan Steiger, Tot Zam, Vikrant, vmaroli
39	LÖSCHEN	Batsu, Chip, CL., Dylan Vander Berg, fredden, Joel, KIRAN KUMAR MATAM, Phrancis, Umesh, xenodevil, Zoyd
40	Löst aus	Daryl, IncrediApp
41	Materialisierte Ansichten	dmfay
42	NULL	Bart Schuijt, CL., dd4711, Devmati Wadikar, Phrancis, Saroj Sasmal, StanislavL, walid, ypercube
43	Primärschlüssel	Andrea Montanari, CL., FlyingPiMonster, KjetilNordin

44	Reihenfolge der Ausführung	a1ex07, Gallus, Ryan Rockey, ypercube
45	Relationale Algebra	CL., Darren Bartrup-Cook, Martin Smith
46	Sequenz	John Smith
47	SKIP TAKE (Paginierung)	CL., Karl Blacquiere, Matas Vaitkevicius, RamenChef
48	SORTIEREN NACH	Andi Mohr, CL., Cristian Abelleira, Jaydles, mithra chintha, nazark, Özgür Öztürk, Parado, Phrancis, Wolfgang
49	SQL CURSOR	Stefan Steiger
50	SQL Group By vs. Distinct	carlosb
51	SQL-Injektion	120196, CL., Clomp, Community, Epodax, Knickerless-Noggins, Stefan Steiger
52	String-Funktionen	ɐlǝx, Allan S. Hansen, Arthur D, Arulkumar, Batsu, Chris, CL., Damon Smithies, Franck Dernoncourt, Golden Gate, hatchet, Imran Ali Khan, IncrediApp, Jaydip Jadhav, Jones Joseph, Kewin Björk Nielsen, Leigh Riffel, Matas Vaitkevicius, Mateusz Piotrowski, Neria Nachum, Phrancis, RamenChef, Robert Columbia, vmaroli, ypercube
53	Synonyme	Daryl
54	TABELLE ERSTELLEN	Aidan, alex9311, Almir Vuk, Ares, CL., drunken_monkey, Dylan Vander Berg, Franck Dernoncourt, H. Pauwelyn, Jojodmo, KIRAN KUMAR MATAM, Matas Vaitkevicius, Prateek
55	Tisch Design	Darren Bartrup-Cook
56	Transaktionen	Amir Pourmand, CL., Daryl, John Odom
57	UND-ODER-Operatoren	guiguiblitz
58	UNION / UNION ALL	Andrea, Athafoud, Daniel Langemann, Jason W, Jim, Joe Taras, KIRAN KUMAR MATAM, Lankymart, Mihai-Daniel Virna, sunkuet02
59	Unterabfragen	CL., dasblinkenlight, KIRAN KUMAR MATAM, Nunie123, Phrancis, RamenChef, tinlyx
60	VERSCHMELZEN	Abhilash R Vankayala, CL., Kyle Hale, SQLFox, Zoyd

61	VERSUCHEN / FANGEN	Uberzen1
62	WÄHLEN	Abhilash R Vankayala, aholmes, Alok Singh, Amnon, Andrii Abramov, apomene, Arpit Solanki, Arulkumar, AstraSerg, Brent Oliver, Charlie West, Chris, Christian Sagmüller, Christos, CL., controller, dariru, Daryl, David Pine, David Spillett, day_dreamer, Dean Parker, DeepSpace, Dipesh Poudel, Dror, Durgpal Singh, Epodax, Eric VB, FH-Inway, Florin Ghita, FlyingPiMonster, Franck Dernoncourt, geeksal, George Bailey, Hari K M, HoangHieu, iliketocode, Imran Ali Khan, Inca, Jared Hooper, Jaydles, John Odom, John Slegers, Jojodmo, JonH, Kapep, KartikKannapur, Lankymart, Mark Iannucci, Mark Perera, Mark Wojciechowicz, Matas Vaitkevicius, Matt, Matt S, Mattew Whitt, Matthew Moisen, MegaTom, Mihai-Daniel Virna, Mureinik, mustaccio, mxmissile, Oded, Ojen, onedaywhen, Paul Bambury, penderi, Peter Gordon, Prateek, Praveen Tiwari, Přemysl Šťastný, Preuk, Racil Hilan, Robert Columbia, Ronnie Wang, Ryan, Saroj Sasmal, Shiva, SommerEngineering, sqluser, stark, sunkuet02, ThisIsImpossible, Timothy, user1336087, user1605665, waqasahmed, wintersolider, WMios, xQbert, Yury Fedorov, Zahiro Mor, zedfoxus
63	XML	Steven
64	Zeilennummer	CL., Phrancis, user1221533