Kapitel 2
Die ersten Schritte mit SQLScript

In diesem Kapitel vermittle ich Ihnen die sprachlichen Grundlagen von SQLScript. Hierbei geht es zunächst um einige wichtige formale Aspekte der Programmiersprache. Danach lernen Sie das Anlegen und Aufrufen von Prozeduren und benutzerdefinierten Funktionen kennen.

In diesem Kapitel geht es mit *SQLScript* richtig los. Zunächst wird die Frage geklärt, was SQLScript eigentlich für eine Sprache ist. Danach geht es in Abschnitt 2.2, »Grundlegende Sprachelemente«, weiter mit dem formalen Teil der Sprache SQLScript. Die Grundlage jeder Programmiersprache bilden die *lexikalischen Elemente*. So werden die kleinsten Spracheinheiten bezeichnet, aus denen sich ein Programm zusammensetzt. Dazu gehören z. B. Kommentare, Bezeichner, Anweisungen, Ausdrücke und Literale. Diese lexikalischen Elemente müssen gewissen formalen Kriterien entsprechen, damit sie vom System korrekt erkannt werden.

Dann geht es um zwei unterschiedliche Konzepte von »nichts«. Der Wert `NULL` repräsentiert in Datenbankfeldern die Abwesenheit eines konkreten Werts. Damit verhält er sich anders, als man es naiv erwarten könnte. Gerade für ABAP-Entwickler ist das eine beliebte Stolperfalle, da sie bislang `NULL` (fast) nicht beachten mussten. Ein anderes erklärungsbedürftiges Konzept ist eine Tabelle namens `DUMMY`, die mit nur genau einer Spalte gleichen Namens genau einen Datensatz enthält. Trotzdem ist sie in manchen Fällen extrem nützlich.

In Abschnitt 2.3, »Modularisierung und logische Container«, geht es um *logische Container*. Diese bilden einen Rahmen für SQLScript-Anweisungen. Dazu gehören z. B. Prozeduren, Funktionen und anonyme Blöcke.

Das Kapitel wird in Abschnitt 2.4, »Programmbeispiel«, mit einem kompletten Beispiel abgerundet, an dem Sie die gelernten Punkte noch einmal nachvollziehen können. Die Beispiele in diesem Kapitel zeigen teilweise SQLScript-Anweisungen, die erst in späteren Kapiteln im Detail erklärt werden. Wenn Sie aber Erfahrungen mit anderen Programmiersprachen haben, können Sie diese leicht nachvollziehen. Der für das Beispiel relevante Aspekt ist jeweils fett hervorgehoben.

2.1 SQL vs. SQLScript

SQL ist eine Datenbanksprache zur Definition von Datenstrukturen in relationalen Datenbanken sowie zum Bearbeiten (Einfügen, Verändern, Löschen) und Abfragen von darauf basierenden Datenbeständen. (Wikipedia, https://de.wikipedia.org/wiki/SQL, [09.04.2020])

Anwendungen nutzen die Sprache *SQL*, um mit der Datenbank zu kommunizieren. Dabei werden immer einzelne Anweisungen nacheinander an die Datenbank gesendet. In SQL gibt es keine Möglichkeit, um mehrere Anweisungen zu verknüpfen oder eine Ablauflogik zu definieren.

Erweiterung des SQL-Standards

Viele Datenbankhersteller haben deshalb den SQL-Standard für ihre Produkte um diese Funktionen erweitert. So hat z. B. Oracle die Sprache PL/SQL und IBM die Sprache SQL PL entwickelt. Das Kürzel PL steht in beiden Fällen für *Procedural Language*. Bei diesen Sprachen können Blöcke von Anweisungen als Funktion oder Prozedur gespeichert und immer wieder ausgeführt werden. Bei SQLScript handelt es sich ebenfalls um eine solche Erweiterung des *ANSI-SQL-Standards*, den SAP für die Datenbank SAP HANA definiert hat.

Auch ASE spricht SQLScript

SQLScript wird fast ausschließlich als Programmiersprache für die SAP-HANA-Datenbank wahrgenommen und auch so vermarktet. Dabei hat SAP mit der Firmentochter *Sybase* eine weitere Datenbank im Portfolio, die auch SQLScript versteht: *Adaptive Server Enterprise* (ASE).

Welche SQLScript-Anweisungen und SQL-Funktionen für Ihre Version der ASE-Datenbank funktionieren, erfahren Sie in dem jeweiligen SAP-Dokument »SAP ASE SQLScript Reference«.

Die Erweiterungen des SQL-Standards durch SQLScript betreffen u. a. die folgenden Bereiche:

- *Prozeduren*, *Funktionen* und *anonyme Blöcke* als logische Container für den SQLScript-Code
- Erweiterung der Datentypen um *Tabellentypen* ohne zugehörige Datenbanktabelle
- *Deklarative Logik* zur Formulierung komplexer, aber trotzdem sehr performanter Datenbankabfragen
- *Imperative Logik* zur Ablaufsteuerung, wie z. B. `IF`/`ELSE` oder `FOR`-Schleifen

Es handelt sich bei SQLScript also um eine Erweiterung des SQL-Standards. Das bedeutet, dass SQL-Anweisungen direkt im SQLScript-Code eingebettet sind. Sie bilden gemeinsam eine Sprache. Deshalb beschränkt sich dieses Buch auch nicht darauf, nur diese Erweiterungen zu besprechen, sondern betrachtet SQLScript als Einheit.

Kategorien von SQL-Anweisungen

Traditionell werden die SQL-Anweisungen in die drei Kategorien *Data Manipulation Language* (DML), *Data Definition Language* (DDL) und *Data Control Language* (DCL) eingeteilt.

- **DML**
  DML umfasst alle Anweisungen, die den Datenbestand in den Datenbanktabellen lesen oder ändern. Die meisten Anwendungen verwenden im laufenden Betrieb ausschließlich DML-Anweisungen. Typische Anweisungen sind `SELECT`, `INSERT` und `UPDATE`. Der lesende Zugriff mittels DML wird in Kapitel 3, »Deklarative Programmierung in SQLScript«, und der ändernde Zugriff in Kapitel 5, »Schreibender Zugriff auf die Datenbank«, beschrieben.
- **DDL**
  Die Definition des Datenmodells erfolgt mit diesen Anweisungen. Dies geschieht normalerweise während der Entwicklung oder zum Installations- bzw. Upgrade-Zeitpunkt einer Anwendung. Typische Anweisungen sind die `CREATE`-, `ALTER`- und `DROP`-Anweisungen. Sie werden in Kapitel 7, »Datenbankobjekte anlegen, löschen und verändern«, ausführlich besprochen.
- **DCL**
  Dieser Teil der SQL-Sprache ist für die Vergabe von Schreib- und Leseberechtigungen zuständig. Die Anweisungen sind also für die Administration der Datenbank relevant. Typische Anweisungen sind `GRANT` und `REVOKE`. DCL wird in diesem Buch nicht weiter besprochen. Informationen zur Administration der SAP-HANA-Datenbank mit SQL finden Sie im SAP-Dokument »SAP HANA SQL and System Views Reference«.

In Abbildung 2.1 sehen Sie eine Übersicht über die Sprache SQLScript mit dem SQL-Standard und den Erweiterungen von SAP.

Code-to-Data-Paradigma

Durch das Anlegen von wiederausführbaren Prozeduren und Funktionen ist es möglich, auch größere Aufgaben, die über die Ausführung einer einzelnen SQL-Anweisung hinausgehen, an die Datenbank zu delegieren. Diese Technik wird von SAP das *Code-to-Data-Paradigma* genannt. Es besagt, dass komplexe und datenintensive Berechnungen direkt in der Datenbank durchgeführt werden sollen.

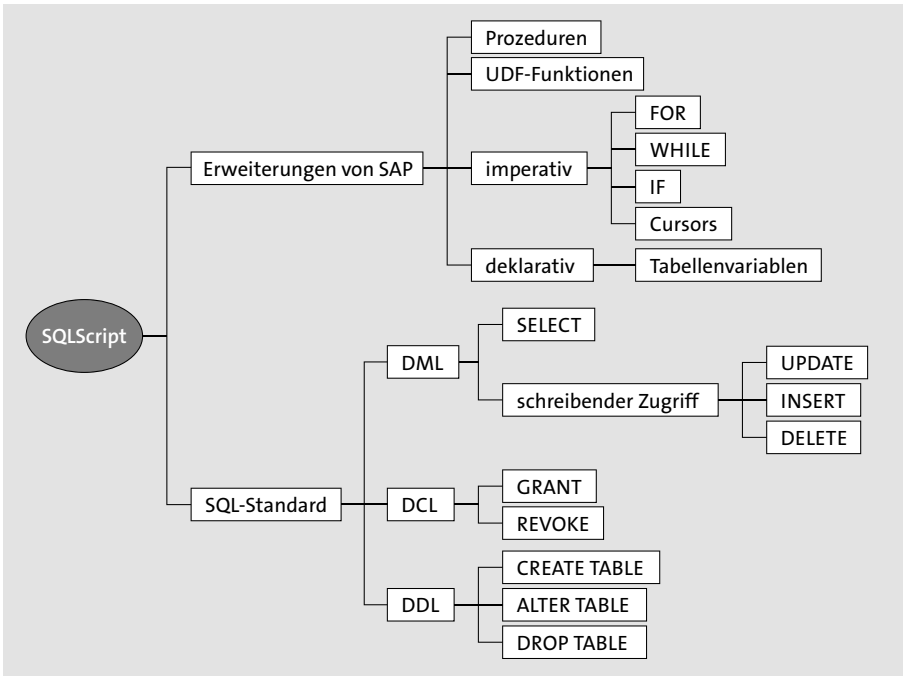


Abbildung 2.1 Typische SQLScript-Anweisungen im Überblick

Damit spart man sich die Kosten für das Kopieren der Daten auf den Anwendungs-Server und kann gegebenenfalls noch von der Parallelisierung und weiteren Optimierungen von SAP HANA profitieren. Dies wird erst durch die Erweiterungen in der Sprache SQLScript ermöglicht, da sich mit reinem SQL immer nur einzelne Anweisungen absetzen lassen. Abbildung 2.2 zeigt die Unterschiede zwischen der klassischen Drei-Schichten-Architektur und dem Code-to-Data-Paradigma.

Gerade wenn sich die Probleme mit deklarativem Code lösen lassen, sind erhebliche Geschwindigkeitsverbesserungen durch Code-Pushdown möglich. Allerdings hat dieses Vorgehen auch Nachteile. Der Anwendungsentwickler muss z. B. mehrere Programmiersprachen beherrschen. Außerdem erhöht der Wechsel der Sprache auch die Komplexität bei der Entwicklung und Fehlersuche in der Anwendung. Darüber hinaus ist die logische Trennung zwischen Applikations-Server und Datenbank-Server nicht mehr so sauber, wie es nach der klassischen *Drei-Schichten-Architektur* sein sollte. Diese Trennung von *Präsentationsschicht*, *Anwendungsschicht* und *Datenbankschicht* galt lange Zeit als wichtige Errungenschaft der Softwarearchitektur. Die Trennung der unteren beiden Schichten war so scharf, dass man SAP-Systeme auf unterschiedlichen Datenbanksystemen laufen lassen

konnte. Dies war für viele Kunden und Partner ein wichtiger Aspekt bei der Entscheidung für diese Softwareplattform.

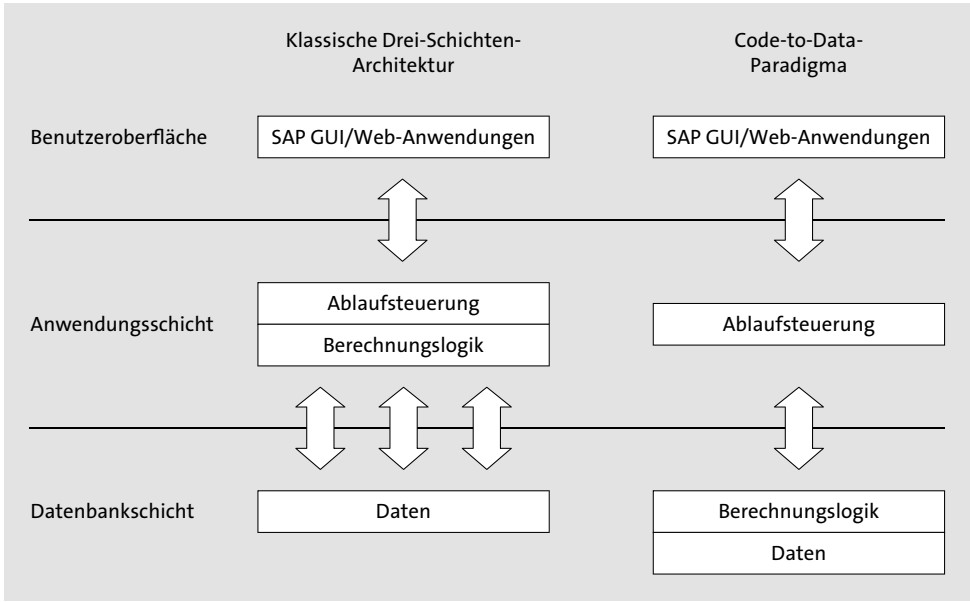


Abbildung 2.2 Vergleich der Drei-Schichten-Architektur mit dem Code-to-Data-Paradigma

Diese saubere Trennung wird mit dem Code-to-Data-Paradigma aufgehoben. Formal bestehen die drei Schichten weiterhin, da ein Applikations-Server und ein Datenbank-Server existieren. Aber die logische Trennung zwischen den Systemen wird verwässert, da die Datenbank einen Teil der Aufgaben des Anwendungs-Servers übernimmt.

Drittanbieter

Für Drittanbieter von Produkten auf der SAP-NetWeaver-Plattform bringt dieser Paradigmenwechsel erhebliche Nachteile mit sich. Wenn man sein Produkt weiterhin für alle Datenbankplattformen offenhalten möchte, kann man von den Vorzügen der SAP-HANA-Datenbank nur begrenzt profitieren. Alternativ könnte man sich für eine zweigleisige Entwicklung entscheiden, die aber mit entsprechendem Mehraufwand verbunden ist.

2.2 Grundlegende Sprachelemente

Wahrscheinlich konnten Sie Ihre Muttersprache schon fließend und fehlerfrei sprechen, bevor Sie in der Grundschule das erste Mal etwas über Grammatik gelernt haben. Für das Erlernen von Fremdsprachen ist es aber sehr

hilfreich, sich mit den grundlegenden Regeln auseinanderzusetzen, die für diese Sprache gelten. Und das gilt genauso für Programmiersprachen.

ASCII-Zeichensatz

Damit der Compiler unseren Quellcode auch richtig interpretieren kann, müssen wir zuerst verstehen, wie die einzelnen Sprachelemente aufgebaut sind. Der Quellcode von SQLScript wird im *Unicode-Zeichensatz* geschrieben. Dabei handelt es sich um einen Zeichensatz, der die Zeichen von nahezu allen Schriftkulturen abbildet. So können z. B. kyrillische, chinesische oder auch koreanische Schriftzeichen für den Quelltext verwendet werden. Zugunsten einer besseren Lesbarkeit und Wartbarkeit sollten Sie sich jedoch unbedingt bei allen Bezeichnern auf die Zeichen des *ASCII-Zeichensatzes* ohne Umlaute beschränken.

2.2.1 Anweisungen

Die Sprache SQLScript besteht in erster Linie aus *Anweisungen*. Eine Anweisung beginnt, außer bei Zuweisungen, mit einem Schlüsselwort und endet mit einem Semikolon.

Anweisungsarten

In SQLScript gibt es unterschiedliche Anweisungsarten:

- *SQL-Anweisungen* wie z. B. `INSERT`, `CREATE PROCEDURE` oder `COMMIT`
- *Zuweisungen* durch Gleichheitszeichen oder mit der `INTO`-Klausel einer `SELECT`-Abfrage (siehe Abschnitt 3.1.2, »Verwendung von Tabellenvariablen«, und Abschnitt 6.1.1, »Lokale skalare Variablen«)
- *Schleifen* mit `FOR` oder `WHILE` oder als Cursor (siehe Abschnitt 6.3, »Schleifen«, und Abschnitt 6.4, »Cursors«)
- *Bedingte Verzweigungen* mit `IF` (siehe Abschnitt 6.2, »Ablaufsteuerung mit IF und ELSE«)
- *Prozeduraufrufe* (siehe Abschnitt 2.3.2, »Prozeduren«)
- *Deklarationen* von Variablen

Verschachtelte Anweisungen

Manche Anweisungen können selbst wieder Anweisungen enthalten. So bildet z. B. eine `IF`-Anweisung eine Klammer um einen oder mehrere Anweisungsblöcke. Das Konzept der Blöcke lernen Sie in Abschnitt 2.3.1, »Blöcke«, kennen.

In Listing 2.1 beginnt die `IF`-Anweisung in Zeile 1 und endet mit dem Semikolon in Zeile 4. Die `INSERT`-Anweisung beginnt und endet in Zeile 3.


```
1  IF lv_counter > 0
2  THEN
3     INSERT INTO farben VALUES ('Violett');
4  END IF;
```

Listing 2.1 IF-Anweisung enthält eine INSERT-Anweisung

2.2.2 Whitespace

Als *Whitespace* bezeichnet man alle Zeichen im Quellcode, die als Leerflächen in Weiß dargestellt werden. Dazu gehören z. B. Leerzeichen, Tabulatoren und Zeilenumbrüche. Diese Zeichen haben in Programmiersprachen unterschiedliche Aufgaben. In SQLScript ist Whitespace nur dort erforderlich, wo eine eindeutige Trennung aufeinanderfolgender Schüsselwörter, Felder, Variablen usw. sonst nicht möglich wäre. Whitespace darf aber auch überall sonst zwischen den einzelnen Sprachelementen eingefügt werden.

Interpretation

Der Compiler unterscheidet nicht zwischen den unterschiedlichen Whitespace-Zeichen. Ebenfalls unerheblich ist die Anzahl aufeinanderfolgender Whitespace-Zeichen. Damit können Sie Whitespace auch zur Formatierung einsetzen, um z. B. die Lesbarkeit des Quelltextes durch Einrückungen zu verbessern. Die Einrückungen haben, im Gegensatz zu einzelnen anderen Programmiersprachen, wie z. B. Python, keine Auswirkungen auf die Semantik. Es ist nicht erforderlich, dass jede Anweisung in einer neuen Zeile steht, auch wenn das den Code deutlich übersichtlicher macht.

Das Beispiel in Listing 2.2 zeigt zwei identische `SELECT`-Anweisungen, die unterschiedliche Formatierung spielt aber semantisch keine Rolle.

```
SELECT col1,col2 FROM T1;
SELECT col1,
       col2
    FROM T1 ;
```

Listing 2.2 Beispiel für unterschiedliche Formatierung

2.2.3 Kommentare

Kommentare sind Bestandteile des Quelltextes, die vom System vollständig ignoriert werden. Der Inhalt eines Kommentars dient also nur dem menschlichen Leser zum Verständnis des Codes. SQLScript unterscheidet zwei Varianten von Kommentaren: *Zeilenendkommentare* und *Blockkommentare*.

Zeilenend-kommentare

Ein Zeilenendkommentar beginnt mit einem doppelten Bindestrich. Alles was danach bis zum Ende der Zeile geschrieben steht, wird nicht als Anweisung interpretiert. Vor dem Zeilenendkommentar können Teile von Anweisungen stehen. Es kann eine Anweisung, die über mehrere Zeilen geht, auch von einem Zeilenendkommentar unterbrochen werden.

Blockkommentare

Blockkommentare beginnen mit den beiden Zeichen Schrägstrich und Stern und enden mit Stern und Schrägstrich:

```
/* Das ist ein Blockkommentar */
```

Ein Blockkommentar kann sich auch über mehrere Zeilen erstrecken und mitten in Anweisungen stecken. In Abbildung 2.3 sehen Sie einige Beispiele für die Anwendung von Kommentaren in SQLScript.

```
/*Dieses Listing ist ein Beispiel für
  die unterschiedlichen Kommentare in
  SQLScript*/
SELECT a.id, --Feld ID
       a.titel --Feld TITEL
FROM aufgaben /*Tabelle mit Aufgaben*/ AS a
--Einschränkungen:
WHERE bearbeiter = 1 --Mr. Beams
```

Abbildung 2.3 Beispiele für (überflüssige) Kommentare im Quelltext

Kommentare sollten nicht das Offensichtliche im Quelltext beschreiben, sondern dem Leser das Verständnis des Quellcodes erleichtern, z. B. durch:

- Hintergrundwissen
- Bezug zur Spezifikation
- Gliederung des Quelltextes

Vor diesem Hintergrund sind alle Kommentare in dem obigen Beispiel überflüssig. Es soll nur zeigen, wo im Quelltext Kommentare möglich sind.

ABAP-Zeilenkommentare mit * in AMDP

In den *ABAP Managed Database Procedures* (AMDP), siehe Abschnitt 8.1, »AMDP-Prozeduren«, sind auch Zeilenkommentare wie in ABAP erlaubt. Diese beginnen mit dem Stern (*) als erstes Zeichen einer Zeile.

Es ist sehr verlockend, Zeilenkommentare auch in SQLScript zu verwenden, da man sie mit den gewohnten Tastenkombinationen Strg + > bzw. Strg + < erzeugen bzw. entfernen kann. Trotzdem rate ich von deren Verwendung ab, da sie nicht in der SQL-Konsole oder Prozeduren außerhalb der AMDP erlaubt sind. Denn um den Code in der SQL-Konsole zu testen, müssen Sie die Kommentare erst entfernen.

2.2.4 Literale

Literale repräsentieren konstante Werte im Quellcode, die direkt eingegeben werden. Literale können an unterschiedlichen Stellen verwendet werden, wie z. B. in Zuweisungen, als Feldwert oder als Vergleichswert in einer Bedingung. Die Literale können unterschiedliche Datentypen haben, die sich aus ihrem Wert ableiten.

Tabelle 2.1 zeigt Beispiele für verschiedene Datentypen.

Datentypen

Bezeichnung	Format	Beispiel
einfache Zeichenketten	in Hochkomma	`'Peter'`
Unicode-Zeichenketten	in Hochkomma, mit einem großen N als Präfix	`N'Jörg'`
Binärzeichenketten	In Hochkomma, mit X als Präfix	`X'FF'`
Ganzzahlen	Ziffernfolge	`123`
Dezimalzahlen	Ziffernfolge mit Dezimalpunkt	`123.456`
Gleitkommazahlen	Mantisse und Exponent, getrennt durch ein großes E	`123E2`
Hexadezimalzahlen	Präfix `0x`	`0xFF`
Datum	Präfix `DATE`	`DATE'2017-11-10'`
Uhrzeit	Präfix `TIME`	`TIME'15:42:04.123'`
Zeitstempel	Präfix `TIMESTAMP`	`TIMESTAMP'2011-12-31 23:59:59'`

Tabelle 2.1 Literale der unterschiedlichen Datentypen

Für die numerischen Literale ist auch ein negatives Vorzeichen in Form eines Minuszeichens vorab möglich.

Literale im Quelltext

In Listing 2.3 sehen Sie die unterschiedlichen Literale in einer einfachen `SELECT`-Anweisung. Wenn Sie diese in der SQL-Konsole der SAP HANA Web-based Development Workbench oder in der SAP Web IDE ausführen, können Sie neben der Spaltenüberschrift der Ergebnistabelle jeweils ein Symbol für den Datentyp sehen.

```
SELECT 'Jörg'                          AS string,
       N'Jörg'                         AS unicode,
       x'fff'                          AS binary,
       -10                             AS integer,
       - 1.2345                        AS decimal,
       - 17.126E30                     AS float,
       0xff                            AS hex,
       '2010-01-01'                    AS date_as_string,
       DATE'2017-11-10'                AS date,
       '15:42:04.123'                  AS time_as_string,
       TIME'15:42:04.123'              AS time,
       '2011-12-31 23:59:59'           AS timestamp_string,
       TIMESTAMP'2011-12-31 23:59:59' AS timestamp
FROM   dummy;
```

Listing 2.3 Beispiele für Literale im Quelltext

Zeichenkettenliterale

Das Ergebnis der *Zeichenkettenliterale* sehen Sie in Abbildung 2.4. Es kann sein, dass die Umlaute in der Datenvorschau in den unterschiedlichen Entwicklungsumgebungen unterschiedlich dargestellt werden, wenn Sie Nicht-Unicode-Zeichenketten verwenden.

STRING	UNICODE	BINARY
Jörg	Jörg	0FFF

Abbildung 2.4 Ergebnis der Zeichenkettenliterale

Numerische Literale

Die *numerischen Literale* sehen Sie in Abbildung 2.5. Links neben der Spaltenüberschrift findet man ein kleines Symbol für die dargestellten Datentypen. Hier ist durchweg die Zahl 12 zu erkennen, die für numerische Datentypen steht. Sie können sehen, dass das Gleitkommaliteral in die wissenschaftliche Notation mit nur einer Vorkommastelle umgewandelt wurde. Das hexadezimale Literal wurde von `0xFF` in die Zahl 255 umgewandelt.

INTEGER	DECIMAL	FLOAT	HEX
-10	-1.2345	-1.7126e+31	255

Abbildung 2.5 Ergebnis der numerischen Literale

Zeit- und Datumsliterale

Bei den *Zeit-* und *Datumsliteralen* wurde im Listing jeweils auch eine Zeichenkettendarstellung erzeugt. Diese lässt sich in der Ausgabe nicht von der korrekten Darstellung von Datums- und Zeitwerten unterscheiden. Nur anhand der Symbole für die Datentypen in der Kopfzeile der Spalten erkennen Sie die tatsächlich verwendeten Datentypen. Durch die Präfixe `TIME`, `DATE` und `TIMESTAMP` wurden die Daten mit dem jeweils korrekten Datentyp erzeugt (siehe Abbildung 2.6).

DATE_AS_STRING	DATE	TIME_AS_STRING	TIME	TIMESTAMP_STRING	TIMESTAMP
2010-01-01	2017-11-10	15:42:04.123	15:42:04	2011-12-31 23:59:59	2011-12-31 23:59:59

Abbildung 2.6 Ergebnis der Zeit- und Datumsliterale

2.2.5 Bezeichner

Bezeichner sind Namen für Objekte in SAP HANA, z. B. für Tabellen, Views und Spalten. Diese Bezeichner sind grundsätzlich *case-sensitive*, das heißt, es wird zwischen Groß- und Kleinschreibung unterschieden. Es ist wichtig, wie der Quelltext vom System interpretiert wird. Dabei gibt es zwei unterschiedliche Notationen.

Einfache Notation von Bezeichnern

In der *einfachen Notation* wird ein Bezeichner im Quelltext nicht in Gänsefüßchen angegeben. Damit wird er intern automatisch in Großbuchstaben konvertiert. Außerdem gelten dann für den Bezeichner die folgenden Einschränkungen:

- Der Bezeichner muss mit einem Buchstaben oder einem Unterstrich beginnen.
- Der Bezeichner darf nur aus den folgenden Zeichen bestehen:
  - Buchstaben des lateinischen Alphabets, keine Umlaute
  - Ziffern
  - Unterstrich (_)
  - Dollar-Zeichen ($)
  - Doppelkreuz (#)

Für die Namen von Parametern und Variablen ist nur diese einfache Notation zulässig.

In Listing 2.4 sehen Sie ein Beispiel für die Verwendung der einfachen Notation. Die Bezeichner werden intern als `ID`, `STATUS`, `TITEL` und `AUFGABEN` interpretiert.

```
SELECT id,
       status,
       titel
       FROM aufgaben;
```

Listing 2.4 Beispiel für die einfache Notation

Spezielle Notation von Bezeichnern

In der *speziellen Notation* werden die Bezeichner in Gänsefüßchen eingerahmt. Dabei sind *alle* Unicode-Zeichen an *jeder* Position erlaubt. Das bedeutet, dass Leerzeichen, Sonderzeichen, wie z. B. Punkt und Komma, und auch alle anderen Zeichen erlaubt sind.

Das folgende Beispiel zeigt eindrücklich, dass mit der speziellen Notation die Inhalte zwischen den Gänsefüßchen exakt 1:1 als Bezeichner verwendet werden.

```
CREATE TABLE id_with_space("ID" int, " ID" int , "ID " int);
```

Ungünstige Bezeichner erschweren die Lesbarkeit erheblich. Hier wird eine Tabelle angelegt, deren Spalten sich nur durch die Position von Leerzeichen unterscheiden. Das ist syntaktisch zwar erlaubt und lässt sich ohne Fehler ausführen; allerdings sehen Sie im Ergebnis in Abbildung 2.7 eine Tabelle, deren Spaltennamen für das menschliche Auge nicht mehr auseinanderzuhalten sind.

Table Name	Schema	Type
ID_WITH_SPACE	JBRANDEIS	ROW

Columns | Indexes

	Name	SQL Data Type	Column Store Data Type
1	ID	INTEGER	INT
2	ID	INTEGER	INT
3	ID	INTEGER	INT

Abbildung 2.7 Resultierende Tabellendefinition

Verwenden Sie möglichst immer die einfache Notation

Dieses Beispiel zeigt, wie gefährlich die Verwendung von schlecht gewählten Bezeichnern ist. Deshalb empfehle ich Ihnen, für eigene Datenbankobjekte stets Großbuchstaben ohne Umlaute zu verwenden und auf Leer- und Sonderzeichen zu verzichten. Am besten verwenden Sie immer die einfache Notation, um Probleme zu vermeiden. Mit dieser sind mehrdeutige Namen fast unmöglich.

Wenn wir auf existierende Tabellen zugreifen, haben wir aber nicht die Wahl, welche Tabellen- und Spaltennamen wir verwenden möchten. In den Demodaten des SHINE-Datenmodells von SAP finden sich u. a. Spaltennamen mit Punkt, wie z. B. `"NAME.FIRSTNAME"`, die uns zur Verwendung der speziellen Notation für diese Spalten zwingen. Im Quelltext lässt sich die Verwendung der einfachen und der speziellen Notation aber auch beliebig kombinieren.

Bezeichner von generierten Datenbankobjekten von SAP BW

SAP Business Warehouse (SAP BW) generiert die Datenbanktabellen für die Datenmodelle in die sogenannten *Generierungsnamensräume* `/BIC/` bzw. `/BIO/`. Das bedeutet, dass jedes Datenbankobjekt mit diesem Präfix beginnt. Auch manche Feldnamen haben diese Präfixe. Um in SQLScript darauf zuzugreifen, ist die spezielle Notation mit Gänsefüßchen notwendig. Hier ist aber wiederum zu beachten, dass dann alles in Großbuchstaben angegeben werden muss. Ein Beispiel für die Attributstabelle des InfoObjects `OMAT_PLANT` ist:

```
"SAPNPL"."/BIO/PMAT_PLANT"
```

2.2.6 Zugriff auf lokale Variablen und Parameter

Lesender Zugriff

In SQLScript kann man auf lokale Tabellenvariablen und Tabellenparameter genauso mit einer `SELECT`-Anweisung zugreifen wie auf Datenbanktabellen. Somit ist bei *lesendem* Zugriff eine Unterscheidung notwendig, damit der Quellcode eindeutig bleibt. Hierzu wird ein Doppelpunkt vor die entsprechende Variable gesetzt, wie in Listing 2.5 zu sehen.

```
CREATE PROCEDURE get_name(IN id INT)
AS BEGIN
   tmp = SELECT id, name, first_name FROM test;
   SELECT * FROM :tmp WHERE id = :id;
END;
CALL get_name(1);
```

Listing 2.5 Auf lokale Felder und Parameter zugreifen

Schreibender Zugriff

Bei *schreibendem* Zugriff ist der Doppelpunkt nicht erforderlich, da hier keine Verwechslungsgefahr mit den Datenbankobjekten besteht. Diese können nicht durch eine einfache Zuweisung verändert werden, sondern nur durch die ändernden Anweisungen, die Sie in Kapitel 5, »Schreibender Zugriff auf die Datenbank«, kennenlernen.

2.2.7 Systemvariablen

In SQLScript gibt es einige *Systemvariablen*, die in Ihrem Kontext jeweils nützliche Informationen liefern können. Sie sind vor allem für das Analysie-

ren und Protokollieren von Fehlersituationen nützlich. Die Systemvariablen beginnen mit zwei führenden Doppelpunkten.

- **::CURRENT_OBJECT_NAME**: Name der Prozedur oder Funktion, in der die Systemvariable abgefragt wird. In anonymen Blöcken hat die Variable den Wert `NULL`. Außerhalb von logischen Containern (siehe Abschnitt 2.3, »Modularisierung und logische Container«) liefert der Zugriff auf die Variable einen Fehler.
- **::CURRENT_OBJECT_SCHEMA**: Datenbankschema der Prozedur oder Funktion, in der die Systemvariable abgefragt wird. Die Sichtbarkeit ist wie bei der Variablen `::CURRENT_OBJECT_NAME`.
- **::SQL_ERROR_CODE**: Enthält den aktuellen Fehlercode. Diese Variable ist nur innerhalb von Fehlerbehandlern (siehe Abschnitt 6.8, »Fehlerbehandlung«) sichtbar.
- **::SQL_ERROR_MESSAGE**: Enthält die Nachricht zum aktuellen Fehlercode. Die Sichtbarkeit ist analog zu der Variablen `::SQL_ERROR_CODE`.
- **::ROWCOUNT**: Die Variable enthält die Anzahl der durch die letzte ändernden Datenbankanweisung veränderten Datensätze. Lesende Datenbankanweisungen sind nicht relevant. Falls Sie die Variable abfragen, bevor eine ändernde Datenbankanweisung stattgefunden hat, wird ein Fehler erzeugt.
- **::CURRENT_LINE_NUMBER**: Enthält die Zeilennummer im Quellcode. Diese Variable gibt es erst mit SAP HANA 2.0 SPS02.

Verwendung der Systemvariablen

Praktische Beispiele für die Verwendung der Systemvariablen finden Sie in Abschnitt 6.8 über die Fehlerbehandlung.

2.2.8 Reservierte Wörter

Es gibt eine Menge von Wörtern, die nicht als Bezeichner in SQLScript verwendet werden sollten, da diese für die Sprache selbst reserviert sind. Sie können deshalb auch nur in der speziellen Notation als Bezeichner verwendet werden, da sonst keine eindeutige Semantik mehr gegeben ist.

Schlüsselwörter für SQLScript

Die Menge dieser *Schlüsselwörter* für SQLScript können Sie aus dem System-View `RESERVED_KEYWORDS` mit der Anweisung `SELECT * FROM RESERVED_KEYWORDS;` auslesen.

Außerdem ist es empfehlenswert, die Schlüsselwörter der anderen an dem Projekt beteiligten Programmiersprachen und dem aktuellsten ANSI-SQL-Standard zu meiden, damit hier keine Komplikationen auftreten.

2.2.9 Operatoren

Operatoren berechnen ein Ergebnis aus den Operanden. Beispielsweise kann man zwei Zahlen mit dem Plusoperator (+) zu ihrer Summe verknüpfen:

```
Ergebnis = Zahl1 + Zahl2;
```

Man kann die Operatoren nach der Anzahl ihrer Operanden in *unäre* und *binäre* Operatoren unterteilen. Unäre Operatoren haben genau einen Operanden, binäre genau zwei. Tabelle 2.2 zeigt Beispiele dafür.

Operatortyp	Muster	Beispiel
unär	<Operator><Operand>	–9
binär	<Operand1><Operator><Operand2>	3 + 4

Tabelle 2.2 Unterscheidung der Operatoren nach der Anzahl der Operanden

Operatoren nach Datentypen

Es gibt, nach dem Datentyp des Ergebnisses gruppiert, die folgenden Operatoren:

- Arithmetische Operatoren liefern ein numerisches Ergebnis.
  - Addition: +
  - Subtraktion: -
  - Multiplikation: *
  - Division: /
  - Negation: - als Vorzeichen
- Zeichenkettenoperatoren zum Verketten von Zeichenketten zu einer neuen Zeichenkette: ||
- Vergleichsoperatoren liefern als Ergebnis eine Aussage in boolescher Logik, die entweder `TRUE`, `FALSE` oder `UNKNOWN` sein kann. `UNKNOWN` entspricht einem `NULL`-Wert der Datenbank. Das Ergebnis der Operatoren ist immer genau dann `UNKNOWN`, wenn ein Operand `NULL` ist.
  - gleich =
  - ungleich != oder <>
  - kleiner als <
  - größer als >
  - kleiner gleich <=
  - größer gleich >=

- Logische Operatoren verknüpfen logische Aussagen zu neuen logischen Aussagen.
  - `AND`: Wenn beide Operanden `TRUE` ergeben, ist das Ergebnis auch `TRUE`.
  - `OR`: Wenn mindestens einer der beiden Operanden `TRUE` ist, ist das Ergebnis `TRUE`.
  - `NOT`: Dieser unäre Operator negiert den Wert seines Operanden. Aus `TRUE` wird `FALSE` und umgekehrt.

Für alle Operatoren gilt, dass das Ergebnis jeweils `NULL` ist, wenn ein Operand `NULL` ist.

Vorfahrtsregeln

Falls in einem Ausdruck mehrere Operatoren vorkommen, gibt es Vorfahrtsregeln, in welcher Reihenfolge die Operatoren ausgewertet werden. Damit ist sichergestellt, dass das Ergebnis eines Ausdrucks eindeutig festgelegt ist. In Tabelle 2.3 sehen Sie die Auswertungsreihenfolge der Operatoren von oben nach unten.

Gruppe	Operatoren
–	Klammern
arithmetische Operatoren	Vorzeichen
	Multiplikation und Division
	Addition und Subtraktion
String-Operatoren	Verkettung
Vergleichsoperatoren	alle
logische Operatoren	`NOT`
	`AND`
	`OR`

Tabelle 2.3 Auswertungsreihenfolge der Operatoren von oben nach unten

Selbst wenn Sie die Auswertungsreihenfolge der Operatoren kennen, sollten Sie im Falle von mehr als zwei Operatoren immer Klammern verwenden. Dies erhöht die Lesbarkeit erheblich, und es erlaubt auch weniger erfahrenen Kollegen, Ihren Quellcode korrekt zu interpretieren. In dem einfachen Beispiel in Listing 2.6 sieht man bereits, wie die Klammerung die Lesbarkeit erheblich erhöht, und das, obwohl es sich hier nur um die aus der Schulzeit geläufige »Punkt-vor-Strich-Regel« handelt.


```
SELECT
       "PURCHASEORDERID",
       "PURCHASEORDERITEM",
       "PRODUCT.PRODUCTID",
       "CURRENCY",
       case
           when netamount <> 0
--- Mit Klammerung:
---             then ((grossamount / netamount) * 100) - 100
--- Ohne Klammerung:
                then grossamount / netamount * 100 - 100
           else 0
           end     as tax,
      "GROSSAMOUNT",
      "NETAMOUNT",
      "TAXAMOUNT",
      "QUANTITY",
      "QUANTITYUNIT",
      "DELIVERYDATE"
FROM "SAP_HANA_DEMO"."sap.hana.democontent.epm.data::PO.Item";
```

Listing 2.6 Beispiel für die Klammerung von Operatoren

Häufig ist es auch sinnvoll, komplexere Ausdrücke mit mehreren Operatoren in kleinere Schritte zu zerlegen. Somit erhalten Sie Zwischenergebnisse, die die Fehlersuche im Debugger erheblich erleichtern. Die Ausführungsgeschwindigkeit wird sich durch diese Maßnahme nicht relevant verändern, da die SAP-HANA-Datenbank die Ausdrücke vor der Ausführung entsprechend optimiert.

Die Mengenoperatoren `UNION (ALL)`, `INTERSECT` und `EXCEPT` werden hier nicht beschrieben. Ich bespreche sie stattdessen in Abschnitt 3.2.9, »ORDER BY-Klausel«, im Zusammenhang mit der `SELECT`-Anweisung im Detail.

2.2.10 Ausdrücke

Ein *Ausdruck* ist ein Sprachkonstrukt, das in seinem Kontext ausgewertet wird und dabei einen Wert zurückgibt. Ausdrücke können an verschiedenen Stellen in einer SQL-Anweisung vorkommen, wie z. B. in der `SELECT`-, `WHERE`- oder `GROUP BY`-Klausel. Folgende Ausdrücke können hier verwendet werden:

- Literale
- Konstanten- und Variablennamen

- Spaltennamen
- Funktionsaufrufe
- Verknüpfung von Ausdrücken mit Operatoren
- CASE-Ausdrücke
- Aggregatfunktionen
- skalare Abfragen

Manche Ausdrücke können zum Teil selbst wiederum aus anderen Ausdrücken zusammengesetzt sein. So ist es z. B. möglich, bei einem Funktionsaufruf für die Parameter auch wieder Ausdrücke zu verwenden. Somit kann es sich bei Ausdrücken immer auch um tiefer verschachtelte Konstrukte handeln.

Kontext von Ausdrücken

Nicht alle Ausdrücke sind in jedem Kontext möglich. Beispielsweise können Aggregatfunktionen nur sinnvoll in SELECT-Anweisungen verwendet werden. Für die Zuweisung einer skalaren Variablen kann eine Aggregatfunktion nicht verwendet werden.

Im Beispiel von Listing 2.7 sehen Sie einige der oben genannten Arten von Ausdrücken, insbesondere auch die verschachtelte Verwendung von Ausdrücken in Ausdrücken.

```
SELECT
--      Feldname als Ausdruck
        id,
--      Verkettungsoperation als Ausdruck
        vorname || ' ' || nachname AS name,
--      CASE-Ausdruck ...
        CASE
--                          ... mit Funktionsaufruf als Ausdruck
           WHEN geschlecht = 'F' THEN NCHAR('9792')
           WHEN geschlecht = 'M' THEN NCHAR('9794')
           ELSE ''
        END AS MW,
--      Funktionsaufruf als Ausdruck
        COALESCE(team, 0) AS team
   FROM benutzer;
```

Listing 2.7 Ausdrücke in Feldlisten

Skalare Abfragen in skalaren Ausdrücken

Mit SAP HANA 2.0 SPS04 sind auch skalare Abfragen in skalaren Ausdrücken möglich. Die entsprechende Abfrage wird von runden Klammern umschlossen und muss exakt einen Wert zurückgeben.


```
DECLARE lv_count INTEGER;
  lv_count = COALESCE((SELECT max(id) FROM aufgaben ), 0);
```

Falls mehr als ein Wert von der Abfrage zurückgegeben wird, kommt es zu einem Laufzeitfehler. Wird kein Wert gefunden, ist der Wert der Abfrage NULL.

Eine direkte Zuweisung des Ergebnisses einer skalaren Abfrage zu einer skalaren Variablen ist jedoch nicht möglich. Dies muss immer mit SELECT … INTO (siehe Kapitel 6, »Imperative Programmierung«) geschehen.

2.2.11 Prädikate

Bei *Prädikaten* handelt es sich um Ausdrücke, die als logische Aussagen in den Selektionsbedingungen von SELECT-Anweisungen verwendet werden können. Wenn die Aussage für eine Zeile nach TRUE ausgewertet wird, wird diese Zeile in die Ergebnismenge mitübernommen. Sie werden aber auch im imperativen SQLScript in den Anweisungen IF und WHILE verwendet.

Arten von Prädikaten

In Kapitel 3, »Deklarative Programmierung in SQLScript«, behandle ich die Prädikate im Zusammenhang mit der WHERE-Klausel ausführlich. Die wichtigsten Arten von Prädikaten sind:

- **Vergleich mit Einzelwerten**
  Ein Vergleich mit Einzelwerten ist mit Vergleichsoperatoren oder dem Schlüsselwort BETWEEN möglich.
- **Vergleich mit Mengen**
  Einen Vergleich mit einer Menge können Sie entweder mit Vergleichsoperatoren und einem der Quantoren ALL, ANY und SOME oder mit dem Schlüsselwort IN durchführen.
- **Suchprädikate**
  Suchprädikate können Sie entweder nach festen Mustern mit LIKE oder LIKE_REGEXPR oder über eine unscharfe Suche in mehreren Spalten mit dem CONTAINS-Schlüsselwort realisieren.
- **Array-Vergleich**
  Mit MEMBER OF überprüfen Sie, ob ein Element in einem Array vorkommt.
- **NULL-Prüfung**
  Mit dem Prädikat IS (NOT) NULL können Sie überprüfen, ob ein Wert NULL ist oder nicht.
- **Unterabfragen**
  Mit dem EXISTS-Prädikat prüfen Sie, ob eine Unterabfrage mindestens eine Zeile liefert.

Prädikate außerhalb von Abfragen

In den Bedingungen der imperativen SQLScript-Anweisungen `WHILE` und `IF` sind die Prädikate `CONTAINS` und `MEMBER OF` nicht erlaubt. Ebenso ist die Nutzung der Quantoren `ALL`, `ANY` und `SOME` nicht möglich. Die Prädikate `EXISTS` und `IN` können hier erst ab SAP HANA 2.0 SPS04 verwendet werden.

2.2.12 Datentypen

Wir brauchen Datentypen, um Daten in einer Datenbankspalte zu speichern, um sie in Prozeduren in lokalen Variablen zu verarbeiten oder um sie als Parameter zu übergeben. Der Datentyp definiert immer, wie die Daten aussehen und welche Operationen mit diesen Daten möglich sind.

Skalare Datentypen

Unter einem *skalaren Datentyp* verstehen wir ein Konstrukt, das genau einen Wert speichern kann. Das kann z. B. eine Zahl oder eine Zeichenkette sein. Das Gegenteil eines skalaren Datentyps ist ein *zusammengesetzter Datentyp*, wie z. B. eine Tabelle.

In der Literatur über objektorientierte Programmiersprachen wird häufig auch der Begriff des *primitiven Datentyps* verwendet. Dieser entspricht weitgehend dem skalaren Datentyp. Beim primitiven Datentyp geht es um die Abgrenzung gegenüber den Referenztypen, also den Objekten, wohingegen der Begriff skalar eine Abgrenzung gegenüber den zusammengesetzten bzw. tabellenartigen Daten zum Ausdruck bringt.

Skalare Funktionen

Von skalaren Funktionen sprechen wir, wenn die Funktion genau einen Wert zurückliefert. Es gibt auch Funktionen, die tabellenartige Daten zurückliefern. Eine `SELECT`-Abfrage liefert immer eine Tabelle zurück. Selbst wenn die Ergebnistabelle genau eine Spalte und eine Zeile hat, handelt es sich um eine Tabelle. Wir nennen eine solche Abfrage dennoch skalare Abfrage. In SQLScript gibt es viele eingebaute, skalare Datentypen, die Sie in Kapitel 4, »Datentypen und ihre Verarbeitung«, kennenlernen.

Datentypen für Tabellenvariablen und -parameter

Die tabellenartigen Datentypen benötigen wir für die Definition von Tabellenvariablen und Tabellenparametern. Sie sind immer aus skalaren Datentypen zusammengesetzt.

Es gibt die Möglichkeit, Tabellentypen als Datenbankobjekte mit `CREATE TYPE` explizit anzulegen. Sie tauchen dann als eigenes Datenbankobjekt im Katalog des jeweiligen Schemas auf. Alternativ kann man sich für die Typisierung auf bestehende Datenbanktabellen beziehen. Oder man definiert den Tabellentyp direkt im Quellcode durch eine Feldliste.

Typisierung

In Listing 2.8 sehen Sie alle drei Alternativen zur Typisierung für eine lokale Tabellenvariable.


```
DO BEGIN
-- Typ einer DB-Tabelle
   DECLARE lt_tab1 aufgaben;
-- Typ eines Tabellentyps
   DECLARE lt_tab2 id_text;
-- Im Code mit TABLE definierter Tabellentyp
   DECLARE lt_tab3 TABLE( id   INT,
                          col1 NVARCHAR(12) );
   lt_tab1 = SELECT * FROM aufgaben;
   lt_tab2 = SELECT id, titel AS text FROM :lt_tab1;
   lt_tab3 = SELECT id, titel AS col1 FROM :lt_tab1;
   SELECT * FROM :lt_tab1;
   SELECT * FROM :lt_tab2;
   SELECT * FROM :lt_tab3;
END;
```

Listing 2.8 Unterschiedliche Typisierung von Tabellen

2.2.13 Der Wert NULL

Beim Wert `NULL` handelt es sich nicht um einen konkreten Wert, sondern um das Fehlen eines Wertes. Insbesondere handelt es sich nicht um einen gültigen Initialwert, wie z. B. 0 oder `SPACE`. Das bedeutet, dass man mit Spalten bzw. Feldern mit dem Wert `NULL` auch keinen sinnvollen Vergleich durchführen kann. Entsprechend reagieren die Vergleichsoperatoren zunächst unerwartet.

NULL in OpenSQL in ABAP

Auf der SAP-NetWeaver-Plattform mit der Programmiersprache ABAP kommt der Wert `NULL` so gut wie nie vor. Dies liegt an der lückenlosen Integration des verwendeten OpenSQL in die Sprache ABAP. Der schreibende Zugriff erfolgt fast ausschließlich über einen Arbeitsbereich oder eine interne Tabelle, deren Zeilen der Datenbanktabelle eins zu eins entsprechen. Innerhalb dieses Arbeitsbereichs oder der internen Tabelle kann kein Feld `NULL` sein. Damit kann `NULL` auch nicht in die Datenbanktabelle geschrieben werden.

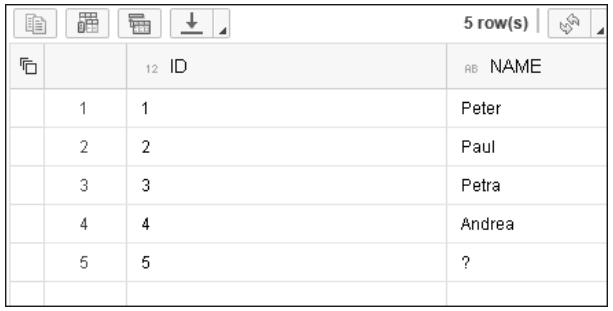
Ein anderer Mechanismus, der `NULL` in der Datenbank vermeidet, ist eine Einstellung bei der Anlage von Datenbanktabellen im Data Dictionary (DDIC). Hier kann für jede Spalte das **Kennzeichen für Initialwerte in Daten-**

banktabellen gesetzt werden. Dies entspricht einem Zusatz NOT NULL bei der Tabellendefinition in SQL und bewirkt, dass die Spalte immer einen typgerechten Initialwert hat, z. B. 0 oder SPACE.

Trotzdem kann NULL in der Datenbank vorkommen, wenn z. B. eine Datenbanktabelle nachträglich um eine Spalte erweitert wurde und bei der Definition der Spalte die NULL-Werte nicht ausgeschlossen wurden. Entsprechend gibt es auch in OpenSQL die Möglichkeit, auf NULL in einer Spalte zu selektieren.

Beispiel

Ein einfaches Beispiel in Listing 2.9 verdeutlicht die Situation. Hier wird eine Datenbanktabelle TEST_NULL mit den beiden Spalten ID und NAME angelegt, und in diese werden fünf Datensätze geschrieben, wobei im letzten Datensatz der Wert für NAME nicht angegeben wird. Entsprechend steht in der Spalte der Wert NULL (siehe Abbildung 2.8).



5 row(s)

	ID	NAME
1	1	Peter
2	2	Paul
3	3	Petra
4	4	Andrea
5	5	?

Abbildung 2.8 Darstellung von NULL als Fragezeichen in SAP HANA Studio

Interessant wird das Beispiel bei der Ausführung der beiden SELECT-Anweisungen. Man würde naiv erwarten, dass die erste Anweisung die ersten drei Datensätze findet, da die Namen alle mit P anfangen und die zweite Anweisung dann alle anderen Datensätze liefert, da sie nicht mit P anfangen. Tatsächlich wird der Datensatz mit ID=5 von keiner der beiden SELECT-Anweisungen gelesen.

```
CREATE TABLE test_null(
            id INT,
            name VARCHAR(10)
);
INSERT INTO test_null VALUES(1, 'Peter');
INSERT INTO test_null VALUES(2, 'Paul');
INSERT INTO test_null VALUES(3, 'Petra');
INSERT INTO test_null VALUES(4, 'Andrea');
INSERT INTO test_null(id) VALUES(5);
```



```
SELECT * FROM test_null WHERE name LIKE 'P%';

SELECT * FROM test_null WHERE name NOT LIKE 'P%';

DROP TABLE test_null; --Um die Tabelle wieder zu entfernen
```

Listing 2.9 Selektion auf eine Spalte mit NULL-Werten

Selektion auf NULL

Falls man also auf eine Spalte selektiert, die NULL-Werte enthalten kann, muss man diese immer mitberücksichtigen. Für das obige Beispiel wäre eine zusätzliche Abfrage mit dem Prädikat IS NULL sinnvoll:

```
SELECT * FROM test_null WHERE name NOT LIKE 'P%'
                          OR name IS NULL;
```

Es muss immer das Prädikat IS (NOT) NULL zur Selektion verwendet werden. Eine Abfrage mit NULL als Operator in einem Vergleichsprädikat ist zwar syntaktisch korrekt, liefert aber nicht das erwartete Ergebnis:

```
SELECT * FROM test_null WHERE name NOT LIKE 'P%'
                          OR name = NULL;
```

Dieser Vergleich scheitert stets, da ein Vergleich mit NULL niemals den logischen Wert TRUE ergibt.

Infektion von Ausdrücken

Wenn NULL als Operand in einem Ausdruck vorkommt, ist das Ergebnis ebenfalls NULL. Gleiches gilt auch für die meisten SQL-Funktionen, wenn NULL als Parameter übergeben wird. Beispiele hierfür sind:

```
5 + NULL = NULL
NULL || 'Zeichenkette' = NULL
```

So kann es vorkommen, dass komplexe Ausdrücke mit mehreren Operanden insgesamt zu NULL ausgewertet werden, weil ein einzelner Operand NULL ist.

Vermeidung von NULL

Häufig ist es notwendig, den Wert NULL speziell zu behandeln. Beispielsweise darf bei AMDP-Transformationsroutinen kein NULL in der OUTTAB-Tabelle vorkommen. Dafür gibt es zwei hilfreiche SQL-Funktionen, die diese Situation abfangen.

COALESCE

Der SQL-Funktion COALESCE wird eine Liste von beliebig vielen Parametern übergeben. Sie gibt den ersten (am weitesten links stehenden) Parameter zurück, der nicht NULL ist. Das Beispiel in Listing 2.10 zeigt eine typische Verwendung bei einem Left Outer Join. Falls kein passender Datensatz gefunden wird, wird stattdessen das Feld aus der Join-Bedingung verwendet. Ist dieses ebenfalls NULL, wird ein konstantes Literal verwendet.

```
SELECT a.id,
       COALESCE(b.nachname,
                to_varchar(a.bearbeiter),
                '' ) AS bearbeiter
  FROM aufgaben AS a
  LEFT OUTER JOIN benutzer AS b
  ON a.bearbeiter = b.ID
```

Listing 2.10 Beispiel für die Verwendung der SQL-Funktion COALESCE

IFNULL Die SQL-Funktion IFNULL funktioniert analog zu COALESCE, nur dass sie genau zwei Parameter erhält. Ist der erste Parameter NULL, gibt sie den Wert des zweiten Parameters zurück.

In den Entwicklungsumgebungen können Sie bei den Einstellungen festlegen, wie der Wert NULL dargestellt werden soll. Üblich ist hier entweder das Fragezeichen (?) oder die Zeichenkette NULL.

2.2.14 Die Tabelle DUMMY

Bei DUMMY handelt es sich um eine Datenbanktabelle mit genau einer Spalte namens DUMMY, die genau eine Zeile mit dem Inhalt X enthält. Da der Inhalt der Tabelle nicht geändert werden darf, können Sie sich auf diese Eigenschaften stets verlassen.

Beispiele Ein Anwendungsfall für die Tabelle DUMMY ist das Testen von Ausdrücken. Da Sie in der SQL-Konsole nicht direkt SQLScript-Code ausführen können, ohne einen anonymen Block darum zu bauen, ist DUMMY eine praktische Alternative. In Listing 2.11 sehen Sie, wie Sie z. B. die Funktion TO_DATS( ) für die Datumskonvertierung testen können.

```
-- Test mit DUMMY
SELECT TO_DATS('2016-01-01') FROM dummy;

-- Der gleiche Test mit einem anonymen Block
DO (OUT rv_result NVARCHAR(10) =>?)
BEGIN
   rv_result = TO_DATS('2016-12-31');
END;
```

Listing 2.11 Tabelle DUMMY zum Testen von Ausdrücken

Eine andere praktische Anwendung für die Tabelle DUMMY ist, dass Sie eine leere Tabelle mit einer festen Struktur erzeugen. Das ist z. B. der Fall, wenn Sie in SAP BW eine Transformationsroutine als ABAP Managed Database

Procedure implementieren. In der generierten Schnittstelle dieser Prozedur ist eine Tabelle mit dem Namen ERRORTAB definiert. Wenn dieser Tabelle kein Wert zugewiesen ist, lässt sich die Prozedur nicht aktivieren, und es wird ein Syntaxfehler angezeigt. Durch den Code in Listing 2.12 kann eine leere Tabelle mit der passenden Struktur erzeugt werden.

```
errorTab =  SELECT '' AS ERROR_TEXT,
                   '' AS SQL__PROCEDURE__SOURCE__RECORD
              FROM dummy
              WHERE dummy = 'Y';
```

Listing 2.12 Leere Tabelle mithilfe von DUMMY erzeugen

In anderen Datenbanksystemen gibt es ebenfalls solche DUMMY-Tabellen, auch wenn sich die Namen teilweise unterscheiden.