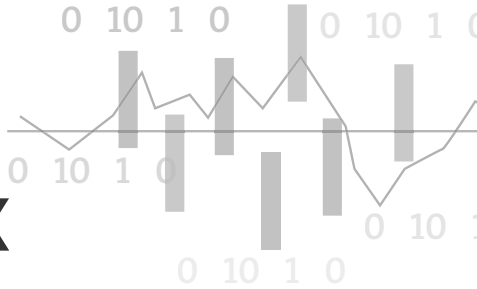
KAPITEL 2

Einführung in DAX



In diesem Kapitel geht es nun ans Eingemachte. Sie werden die DAX-Syntax, den Unterschied zwischen einer berechneten Spalte und einem Measure (das in älteren Excel-Versionen noch als »berechnetes Feld« bezeichnet wurde) und die meistverwendeten Funktionen in DAX kennenlernen.

Da es sich um ein einleitendes Kapitel handelt, werden viele Funktionen anfangs noch nicht ausführlich behandelt. Wir werden jedoch in späteren Kapiteln genauer darauf eingehen. Vorerst soll es genügen, eine Einführung in die Funktionen von DAX und die Sprache im Allgemeinen zu erhalten. Wenn wir in Power BI, Power Pivot oder Analysis Services auf Merkmale des Datenmodells verweisen, verwenden wir den Begriff *tabellarisch* auch dann, wenn das Merkmal nicht in allen Produkten vorhanden ist. Beispielsweise bezeichnet »tabellarisches DirectQuery« den DirectQuery-Modus, der in Power BI und Analysis Services, nicht aber in Excel verfügbar ist.

DAX-Berechnungen verstehen

Bevor wir uns mit komplexeren Formeln auseinandersetzen, müssen Sie erst einmal die DAX-Grundlagen kennen. Hierzu gehören die DAX-Syntax, die verschiedenen Datentypen, die in DAX verarbeitet werden können, die grundlegenden Operatoren und die Referenzierung von Spalten und Tabellen. Alle diese Konzepte werden wir in den kommenden Abschnitten behandeln.

Wir verwenden DAX, um Werte über Spalten in Tabellen zu berechnen. Wir können Zahlen zwar aggregieren, berechnen und nach ihnen suchen, aber am Ende beinhalten doch alle Berechnungen Tabellen und Spalten. Am Anfang der Syntaxbeschreibung soll daher die Referenzierung einer Spalte in einer Tabelle stehen.

Grundsätzlich wird der Tabellenname zu diesem Zweck in einfachen Anführungszeichen notiert, gefolgt vom Spaltennamen in eckigen Klammern:

```
'Sales'[Quantity]
```

Wir können die einfachen Anführungszeichen weglassen, wenn der Tabellenname nicht mit einer Zahl beginnt, keine Leerzeichen enthält und kein reserviertes Wort (wie *Date* oder *Sum*) ist.

Der Tabellenname ist auch dann optional, wenn wir eine Spalte oder ein Measure innerhalb der Tabelle referenzieren, in der wir die Formel definieren. Deswegen ist *[Quantity]* eine gültige Spaltenreferenz, sofern es in einer berechneten Spalte oder in einem in der Tabelle *Sales* definierten Measure steht. Aber auch wenn es möglich ist, raten wir Ihnen dringend davon ab, den Tabellennamen wegzulassen. Wir wollen an dieser Stelle nicht erläutern, warum das so wichtig ist; der Grund wird Ihnen aber einleuchten, wenn Sie Kapitel 5, »CALCULATE und CALCULATETABLE verstehen«, gelesen haben. Trotzdem ist es von größter Bedeutung, beim Lesen des DAX-

Codes zwischen den noch zu beschreibenden Measures und den Spalten unterscheiden zu können. Der De-facto-Standard sieht vor, den Tabellennamen in Spaltenreferenzierungen grundsätzlich zu verwenden, ihn aber in Measureverweisen immer wegzulassen. Je früher Sie sich an dieses Prinzip gewöhnen, desto einfacher wird Ihr Leben mit DAX. Daher sollten Sie sich gleich mit dieser Art der Referenzierung von Spalten und Measures vertraut machen:

```
Sales[Quantity] * 2          -- Dies ist ein Spaltenverweis.
[Sales Amount] * 2           -- Dies ist ein Measureverweis.
```

Den Grund für diesen Standard werden Sie erfahren, nachdem Sie Kontextübergänge kennengelernt haben (auch das kommt in Kapitel 5 auf Sie zu). Vorläufig sollten Sie uns einfach vertrauen und sich an diesen Standard halten.

Kommentare in DAX

Das vorhergehende Codebeispiel zeigt Ihnen erstmals, wie Kommentare in DAX aussehen. DAX unterstützt einzeilige und mehrzeilige Kommentare. Einzeilige Kommentare beginnen mit -- oder //. Der gesamte nachfolgende Teil der Zeile wird dann als Kommentar betrachtet.

```
= Sales[Quantity] * Sales[Net Price]   -- Einzeiliger Kommentar
= Sales[Quantity] * Sales[Unit Cost]   // Noch ein einzeiliger Kommentar
```

Ein mehrzeiliger Kommentar beginnt mit /* und endet mit */. Alles, was zwischen diesen Markern steht, wird vom DAX-Parser als Kommentar betrachtet und folglich ignoriert.

```
= IF (
    Sales[Quantity] > 1,
    /* Erstes Beispiel für einen mehrzeiligen Kommentar.
       Alles, was hier steht, wird von DAX ignoriert.
    */
    "Multi",
    /* Ein häufiger Anwendungsfall für mehrzeilige Kommentare ist das Auskommentieren
       von Codeabschnitten.
       Die nächste IF-Anweisung wird ignoriert, da sie in einem mehrzeiligen
Kommentar steht.
        IF (
            Sales[Quantity] = 1,
            "Single",
            "Special note"
        )
    */
    "Single"
)
```

Sehen Sie davon ab, Kommentare ans Ende eines DAX-Ausdrucks in ein Measure, eine berechnete Spalte oder die Definition einer berechneten Tabelle zu setzen. Solche Kommentare sind anfangs möglicherweise nicht sichtbar und werden von Tools wie DAX Formatter (siehe weiter hinten in diesem Kapitel) unter Umständen nicht unterstützt.

DAX-Datentypen

DAX kann Berechnungen mit verschiedenen numerischen Typen durchführen, von denen es insgesamt sieben gibt. Im Laufe der Zeit hat Microsoft unterschiedliche Bezeichnungen für dieselben Datentypen eingeführt, was ein wenig Verwirrung gestiftet hat. Tabelle 2.1 enthält die verschiedenen Namen, unter denen Sie die einzelnen DAX-Datentypen finden können.

DAX-Datentyp	Power BI-Datentyp	Power Pivot- und Analysis Services-Datentyp	Entsprechender konventioneller Datentyp (z. B. in SQL Server)	TOM-Datentyp (Tabular Object Model)
Integer	Ganze Zahl	Ganze Zahl	Integer/INT	int64
Dezimal	Dezimalzahl	Dezimalzahl	Gleitkommazahl/DOUBLE	Double
Währung	Feste Dezimalzahl	Währung	Währung/MONEY	Dezimal
DateTime	DateTime, Datum, Uhrzeit	Datum	Datum/DATETIME	dateTime
Boolesch	Wahr/Falsch	Wahr/Falsch	Boolesch/BIT	Boolesch
Zeichenfolge	Text	Text	Zeichenfolge/ NVARCHAR(MAX)	String
Variant	-	-	-	Variant
Binärzahl	Binärzahl	Binärzahl	Blob/VARBINARY(MAX)	Binärzahl

Tabelle 2.1 Datentypen

In diesem Buch verwenden wir die Namen aus der ersten Spalte von Tabelle 2.1, die den De-facto-Standards in der Datenbank- und BI-Community entsprechen. In Power BI beispielsweise würde eine Spalte, die entweder *WAHR* oder *FALSCH* enthält, *WAHR/FALSCH* heißen, während sie in SQL Server als *BIT* bezeichnet wird. Trotzdem ist der historische und noch immer gebräuchlichste Name hierfür »boolescher Wert«.

DAX verfügt über ein leistungsfähiges Typenbehandlungssystem, weswegen Sie sich in Sachen Datentypen keine Sorgen machen müssen. In einem DAX-Ausdruck basiert der resultierende Typ auf dem Typ des im Ausdruck verwendeten Terms. Das müssen Sie für den Fall wissen, dass der von einem DAX-Ausdruck zurückgegebene Typ nicht Ihren Erwartungen entspricht. In diesem Fall müssten Sie sich den Datentyp der im Ausdruck selbst verwendeten Termen genauer ansehen.

Wenn beispielsweise einer der Termen in einer Summe ein Datum ist, ist das Ergebnis auch ein Datum. Ähnlich ist, wenn derselbe Operator mit ganzen Zahlen verwendet wird, das Ergebnis ebenfalls eine ganze Zahl. Dieses Verhalten wird als *Operatorüberladung* bezeichnet. Ein Beispiel sehen Sie in Abbildung 2.1, wo die Spalte *OrderDatePlusOneWeek* (Auftragsdatum plus eine Woche) durch Hinzuaddieren von 7 zum Wert der Spalte *Order Date* (Auftragsdatum) berechnet wird.

```
Sales[OrderDatePlusOneWeek] = Sales[Order Date] + 7
```

Das Ergebnis ist ein Datum.

Order Date	OrderDatePlusOneWeek
10/08/2008	10/15/2008
10/10/2008	10/17/2008
10/12/2008	10/19/2008
09/05/2008	09/12/2008
09/07/2008	09/14/2008
09/23/2008	09/30/2008
11/05/2008	11/12/2008
11/07/2008	11/14/2008
11/09/2008	11/16/2008
11/17/2008	11/24/2008

Abbildung 2.1 Wird ein Integer zu einem Datum hinzuaddiert, dann ist das Ergebnis ebenfalls ein Datum, das um die entsprechende Anzahl von Tagen später liegt.

DAX führt nicht nur eine Operatorüberladung durch, sondern wandelt Strings auch automatisch in Zahlen um und umgekehrt, sofern der Operator dies erfordert. Wenn wir beispielsweise den *&*-Operator verwenden, der Strings verknüpft, dann konvertiert DAX seine Argumente in Strings. Die folgende Formel gibt »54« als String zurück:

```
= 5 & 4
```

Im Gegensatz dazu liefert folgende Formel als Ergebnis einen Integer mit dem Wert 9:

```
= "5" + "4"
```

Der resultierende Wert hängt vom Operator ab und nicht von den Quellspalten, die je nach Anforderungen des Operators umgewandelt werden. Das wirkt zwar alles sehr bequem, aber Sie werden im weiteren Verlauf dieses Kapitels sehen, welche Fehler bei diesen automatischen Konvertierungen auftreten können. Außerdem ist dieses Verhalten nicht allen Operatoren gemein. Beispielsweise können Vergleichsoperatoren Strings nicht mit Zahlenwerten vergleichen. Das bedeutet, dass Sie zwar eine Zahl an einen String anhängen können, aber das Vergleichen einer Zahl mit einem String ist nicht möglich. Eine vollständige Referenz finden Sie unter *https://docs.microsoft.com/de-de/power-bi/desktop-data-types*. Da die Regeln so komplex sind, empfehlen wir Ihnen, automatische Konvertierungen ganz zu vermeiden. Wenn eine Konvertierung stattfinden muss, sollten Sie diese aktiv steuern und sie explizit vornehmen. Zur Verdeutlichung: Das obige Beispiel sollte wie folgt formuliert werden:

```
= VALUE ( "5" ) + VALUE ( "4" )
```

Benutzer, die daran gewöhnt sind, mit Excel oder anderen Sprachen zu arbeiten, kennen die DAX-Datentypen vielleicht schon. Einige Eigenschaften der Datentypen hängen von der Engine ab und sind bei Power BI, Power Pivot und Analysis Services eventuell unterschiedlich. Ausführliche Informationen zu den DAX-Datentypen von Analysis Services finden Sie unter *https://docs.microsoft.com/de-de/analysis-services/tabular-models/data-types-supported-ssas-tabular*,

die Datentypen in Power BI sind unter *https://docs.microsoft.com/de-de/power-bi/desktop-data-types* beschrieben. Trotzdem ist es sinnvoll, ein paar Überlegungen zu den Datentypen anzustellen.

Integer

DAX hat nur einen *Integer*-Datentyp, der einen 64-Bit-Wert speichern kann. Alle internen Berechnungen zwischen ganzen Zahlen verwenden in DAX ebenfalls einen 64-Bit-Wert.

Decimal

Ein *Decimal* (Dezimalzahl) wird immer als Gleitkommazahl mit doppelter Genauigkeit gespeichert. Verwechseln Sie diesen DAX-Datentyp nicht mit dem Datentyp *decimal* oder *numeric* in *Transact-SQL*. Der entsprechende Datentyp einer DAX-Dezimalzahl in SQL ist *float*.

Currency

Der Datentyp *Currency* (Währung, in Power BI auch als *feste Dezimalzahl* bezeichnet) speichert eine feste Dezimalzahl. Er kann vier Dezimalstellen darstellen und wird intern als 64-Bit Ganzzahl geteilt durch 10.000 gespeichert. Beim Addieren oder Subtrahieren von *Currency*-Datentypen werden alle Dezimalstellen ignoriert, die über die vierte Dezimalstelle hinausgehen, während Multiplikation und Division einen Gleitkommawert erzeugen und so die Genauigkeit des Ergebnisses erhöhen. Grundsätzlich müssen Sie, wenn Sie eine höhere Genauigkeit als die vier angegebenen Stellen benötigen, den *Decimal*-Datentyp verwenden.

Das Standardformat des Datentyps *Currency* beinhaltet das Währungssymbol. Sie können die Währungsformatierung auch auf *Integers* und Dezimalzahlen anwenden, aber auch ein Format ohne Währungssymbol für einen *Currency*-Datentyp verwenden.

DateTime

DAX speichert Datumsangaben als *DateTime*-Datentyp. Dieses Format verwendet intern eine Gleitkommazahl, wobei die ganze Zahl der Anzahl der seit dem 30. Dezember 1899 verstrichenen Tage entspricht, während der Nachkommateil den Anteil eines Tages benennt. Stunden, Minuten und Sekunden werden in Bruchteile eines Tages umgewandelt. Der folgende Ausdruck gibt also das aktuelle Datum plus einen Tag (genau 24 Stunden) zurück:

```
= TODAY () + 1
```

Das Ergebnis ist das Datum von morgen zum Zeitpunkt der Auswertung. Wenn Sie nur den Datumsteil einer *DateTime*-Angabe verwenden möchten, denken Sie immer daran, dass Sie die Nachkommastellen mit *TRUNC* entfernen können.

Power BI bietet zwei weitere Datentypen an: *Date* und *Time*. Intern handelt es sich dabei nur um einfache Varianten von *DateTime*. Tatsächlich speichern *Date* und *Time* lediglich den ganzzahligen bzw. den Nachkommateil von *DateTime*.

Der Schaltjahresfehler

Lotus 1-2-3, eine beliebte Tabellenkalkulation, die 1983 auf den Markt kam, enthielt einen Bug bei der Behandlung des Datentyps *DateTime*. Das Jahr 1900 wurde nämlich als Schaltjahr betrachtet, obwohl es das gar nicht war. Jahrhunderte sind nämlich nur dann ein Schaltjahr, wenn die ersten beiden Ziffern glatt durch 4 teilbar sind, wie 2000. Seinerzeit baute das Entwicklerteam von Microsoft diesen Fehler in der ersten Excel-Version gezielt nach, um für Kompatibilität mit Lotus 1-2-3 zu sorgen. Damit nicht genug: Samt und sonders jede nachfolgende Excel-Version enthält aus Kompatibilitätsgründen diesen Bug. Zum Zeitpunkt der Drucklegung im Jahr 2020 ist er auch in DAX immer noch vorhanden, um die Abwärtskompatibilität mit Excel zu gewährleisten. Dieser Bug (den man mittlerweile durchaus als Feature bezeichnen könnte) kann bei Berechnungen im Zusammenhang mit Zeiträumen vor dem 1. März 1900 zu Fehlern führen. Und aus diesem Grund ist das erste von DAX offiziell unterstützte Datum eben jener 1. März 1900. Datumsberechnungen, die für Zeiträume vor diesem Datum durchgeführt werden, können zu Fehlern führen und sollten mit viel Vorsicht genossen werden.

Boolean

Der Datentyp *Boolean* (boolescher Wert) wird zum Formulieren von Logikbedingungen verwendet. Beispielsweise hat eine berechnete Spalte, die durch den folgenden Ausdruck definiert ist, den Typ *Boolean*:

```
= Sales[Unit Price] > Sales[Unit Cost]
```

Es gibt auch *Boolean*-Datentypen in numerischer Form, wobei *TRUE* den Wert 1 und *FALSE* den Wert 0 hat. Diese Schreibweise ist manchmal zum Sortieren ganz nützlich, da *TRUE* > *FALSE* gilt.

String

Jeder String in DAX wird als *Unicode*-Zeichenfolge gespeichert. Dabei erfolgt die Speicherung jedes Zeichens im 16-Bit-Format. Standardmäßig wird die Groß-/Kleinschreibung beim Vergleich zwischen Strings nicht unterschieden, sodass die beiden Zeichenfolgen *Power BI* und *POWER BI* als gleichwertig angesehen werden.

Variant

Der *Variant*-Datentyp wird für Ausdrücke verwendet, die je nach Bedingung unterschiedliche Datentypen zurückgeben können. Die folgende Anweisung beispielsweise könnte entweder einen Integer oder einen String zurückgeben, das heißt, die Rückgabe erfolgt als Variant:

```
IF ( [measure] > 0, 1, "N/A" )
```

Der *Variant*-Datentyp kann nicht als Datentyp für eine Spalte einer regulären Tabelle verwendet werden. Ein DAX-Measure und grundsätzlich ein DAX-Ausdruck können ein *Variant* sein.

Binary

Der Datentyp *Binary* (Binärzahl) wird im Datenmodell verwendet, um Bilder und sonstige unstrukturierte Informationsformen zu speichern. Er steht in DAX selbst nicht zur Verfügung. Er wurde hauptsächlich in Power View verwendet, ist aber möglicherweise in anderen Tools wie Power BI nicht verfügbar.

DAX-Operatoren

Sie haben bereits gesehen, wie wichtig Operatoren für die Festlegung eines Ausdruckstyps sind. In Tabelle 2.2 finden Sie nun eine Liste der in DAX verfügbaren Operatoren.

Operatortyp	Symbol	Verwendung	Beispiel
Klammern	( )	Verarbeitungsreihenfolge und Gruppierung von Argumenten	(5 + 2) * 3
Rechenoperatoren	+	Addition	4 + 2
	–	Subtraktion/Negation	5 – 3
	*	Multiplikation	4 * 2
	/	Division	4 / 2
Vergleichs-operatoren	=	Gleich	[CountryRegion] = "USA"
	<>	Ungleich	[CountryRegion] <> "USA"
	>	Größer als	[Quantity] > 0
	>=	Größer oder gleich	[Quantity] >= 100
	<	Kleiner als	[Quantity] < 0
	<=	Kleiner oder gleich	[Quantity] <= 100
Textverkettung	&	Verkettung von Zeichenfolgen	"Wert ist" & [Betrag]
Logische Operatoren	&&	UND-Bedingung zweier boolescher Ausdrücke	[CountryRegion] = "USA" && [Quantity]>0
	||	ODER-Bedingung zweier boolescher Ausdrücke	[CountryRegion] = "USA" || [Quantity] > 0
	IN	Vorhandensein eines Elements in einer Liste	[CountryRegion] IN {"USA", "Canada"}
	NOT	Boolesche Negation	NOT [Quantity] > 0

Tabelle 2.2 Operatoren

Darüber hinaus stehen die logischen Operatoren auch als DAX-Funktionen zur Verfügung, deren Syntax der von Excel ähnelt. Wir können beispielsweise Ausdrücke wie die folgenden schreiben:

```
AND ( [CountryRegion] = "USA", [Quantity] > 0 )
OR ( [CountryRegion] = "USA", [Quantity] > 0 )
```

Diese Beispiele sind jeweils äquivalent zu den folgenden:

```
[CountryRegion] = "USA" && [Quantity] > 0
[CountryRegion] = "USA" || [Quantity] > 0
```

Die Verwendung von Funktionen anstelle von Operatoren für die boolesche Logik ist hilfreich beim Formulieren komplexer Bedingungen. Funktionen sind beim Strukturieren umfangreicher Codeabschnitte viel einfacher zu formatieren und zu lesen als Operatoren. Allerdings haben sie den großen Nachteil, dass Sie nur zwei Parameter gleichzeitig übergeben können. Daher müssen Sie Funktionen verschachteln, wenn mehr als zwei Bedingungen auszuwerten sind.

Tabellenkonstruktoren

In DAX können Sie anonyme Tabellen direkt im Code definieren. Hat die Tabelle nur eine Spalte, dann braucht die DAX-Syntax lediglich eine Liste mit Werten (nämlich einen je Zeile), die durch geschweifte Klammern voneinander getrennt sind. Sie können mehrere Zeilen mit Klammern begrenzen, die optional sind, wenn die Tabelle aus nur einer Spalte besteht. Die beiden folgenden Definitionen sind beispielsweise gleichwertig:

```
{ "Red", "Blue", "White" }
{ ( "Red" ), ( "Blue" ), ( "White" ) }
```

Hat die Tabelle dagegen mehrere Spalten, dann sind Klammern obligatorisch. Jede Spalte muss für alle Zeilen denselben Datentyp verwenden, andernfalls konvertiert DAX die Spalte automatisch in einen Datentyp, der alle Datentypen aufnehmen kann, die in den verschiedenen Zeilen für dieselbe Spalte angegeben werden.

```
{
    ( "A", 10, 1.5, DATE ( 2017, 1, 1 ), CURRENCY ( 199.99 ), TRUE ),
    ( "B", 20, 2.5, DATE ( 2017, 1, 2 ), CURRENCY ( 249.99 ), FALSE ),
    ( "C", 30, 3.5, DATE ( 2017, 1, 3 ), CURRENCY ( 299.99 ), FALSE )
}
```

Der Tabellenkonstruktor wird häufig mit dem *IN*-Operator verwendet. So sind beispielsweise folgende Syntaxen in einem DAX-Prädikat gültig:

```
'Product'[Color] IN { "Red", "Blue", "White" }

( 'Date'[Year], 'Date'[MonthNumber] ) IN { ( 2017, 12 ), ( 2018, 1 ) }
```

Das zweite Beispiel zeigt die Syntax, die verwendet wird, um mehrere Spalten (ein sogenanntes Tupel) mit dem *IN*-Operator zu vergleichen. Bei Vergleichsoperatoren kann eine solche Syntax dagegen nicht verwendet werden. Folgende Syntax wäre mithin ungültig:

```
( 'Date'[Year], 'Date'[MonthNumber] ) = ( 2007, 12 )
```

Wir können sie allerdings wie im folgenden Beispiel gezeigt mithilfe des *IN*-Operators mit einem Tabellenkonstruktor, der nur eine Zeile umfasst, neu schreiben:

```
( 'Date'[Year], 'Date'[MonthNumber] ) IN { ( 2007, 12 ) }
```

Bedingungsanweisungen

In DAX formulieren wir Bedingungsausdrücke mit der *IF*-Funktion. So können wir beispielsweise einen Ausdruck schreiben, der je nachdem, ob der Mengenwert größer als eins oder nicht ist, MULTI oder SINGLE zurückgibt.

```
IF (
    Sales[Quantity] > 1,
    "MULTI",
    "SINGLE"
)
```

Die *IF*-Funktion hat drei Parameter, wobei nur die ersten beiden obligatorisch sind. Die dritte ist optional und wird standardmäßig auf *BLANK* festgelegt. Betrachten Sie folgenden Code:

```
IF (
    Sales[Quantity] > 1,
    Sales[Quantity]
)
```

Dies entspricht der folgenden expliziten Version:

```
IF (
    Sales[Quantity] > 1,
    Sales[Quantity],
    BLANK ()
)
```

Berechnete Spalten und Measures verstehen

Nachdem Sie nun einen ersten Eindruck von der DAX-Syntax erhalten haben, möchten wir Sie jetzt mit einem der wichtigsten Konzepte in DAX vertraut machen: dem Unterschied zwischen berechneten Spalten und Measures. Zwar mögen berechnete Spalten und Measures auf den ersten Blick einander recht ähnlich erscheinen, weil man mit beiden bestimmte Berechnungen durchführen kann, doch in Wirklichkeit sind sie unterschiedlich. Das Begreifen dieses Unterschieds ist der Schlüssel, mit dem Sie sich DAX in seiner vollen Leistungsfähigkeit erschließen.

Berechnete Spalten

Sie können eine berechnete Spalte abhängig vom verwendeten Tool auf unterschiedliche Weise erstellen. Das Konzept bleibt dabei jedoch immer dasselbe: Eine berechnete Spalte ist eine neue Spalte, die Ihrem Modell hinzugefügt wird, aber sie wird nicht aus einer Datenquelle geladen, sondern durch Einsatz einer DAX-Formel erstellt.

Berechnete Spalten unterscheiden sich nicht von anderen Spalten in einer Tabelle, und Sie können sie in Zeilen, Spalten, Filtern oder Werten einer Matrix oder eines anderen Berichts verwenden. Außerdem können Sie eine berechnete Spalte verwenden, um ggf. eine Beziehung zu

definieren. Der für eine berechnete Spalte definierte DAX-Ausdruck arbeitet im Kontext der aktuellen Zeile der Tabelle, zu der die berechnete Spalte gehört. Jegliche Referenz auf eine Spalte gibt deren Wert für die aktuelle Zeile zurück. Auf die Werte anderer Zeilen können Sie dagegen nicht direkt zugreifen.

Wenn Sie anstelle von DirectQuery den standardmäßigen tabellarischen *Importmodus* verwenden, dürfen Sie im Zusammenhang mit berechneten Spalten keinesfalls übersehen, dass diese Spalten während der Datenbankverarbeitung berechnet und dann im Modell gespeichert werden. Dieses Konzept mag seltsam erscheinen, wenn Sie SQL-berechnete (d. h. nicht persistierte) Spalten gewohnt sind, die zur Abfragezeit ausgewertet werden und keinen Speicher verwenden. Beim tabellarischen Modell hingegen belegen alle berechneten Spalten Platz im Speicher und werden während der Tabellenverarbeitung berechnet.

Dieses Verhalten ist vor allem dann nützlich, wenn wir komplexe berechnete Spalten erstellen. Die Zeit, die für die Berechnung komplexer berechneter Spalten benötigt wird, ist immer Prozesszeit und nicht Abfragezeit. Dies führt zu einer Steigerung der Benutzerfreundlichkeit. Nichtsdestoweniger sollten Sie beachten, dass eine berechnete Spalte kostbaren Arbeitsspeicher verbraucht. Nehmen wir beispielsweise an, wir hätten eine komplexe Formel für eine berechnete Spalte. In diesem Fall könnten wir versucht sein, die Berechnungsschritte auf verschiedene »Zwischenspalten« zu verteilen. Ein solcher Ansatz ist zwar bei der Projektentwicklung nützlich, in der Produktion dagegen nicht empfehlenswert, da jede Zwischenberechnung im RAM gespeichert wird und wertvollen Platz vergeudet.

Basiert ein Modell dagegen auf DirectQuery, dann unterscheidet sich das Verhalten erheblich. Im DirectQuery-Modus erfolgt die Berechnung solcher Spalten quasi »auf Zuruf«, wenn die tabellarische Engine die Datenquelle abfragt. Dies kann zu umfassenden Abfragen durch die Datenquelle führen – das Ergebnis sind langsame Modelle.

Lieferzeitraum berechnen

Angenommen, wir haben eine Tabelle *Sales*, die sowohl Bestell- als auch Liefertermine enthält. Anhand dieser beiden Spalten können wir die Anzahl der Tage berechnen, die für die Lieferung der Bestellung benötigt werden. Da Daten als Anzahl von seit dem 30.12.1899 verstrichenen Tagen gespeichert werden, brauchen wir lediglich eine einfache Subtraktion, um die Differenz zwischen zwei Datumsangaben in Tagen zu berechnen:

```
Sales[DaysToDeliver] = Sales[Delivery Date] - Sales[Order Date]
```

Da die beiden Spalten, die für die Subtraktion verwendet werden, jedoch Datumsangaben sind, ist auch das Ergebnis ein Datum. Um ein numerisches Ergebnis zu erhalten, konvertieren Sie es deswegen wie folgt in eine ganze Zahl:

```
Sales[DaysToDeliver] = INT ( Sales[Delivery Date] - Sales[Order Date] )
```

Abbildung 2.2 zeigt das Ergebnis.

Order Date	Delivery Date	DaysToDeliver
01/02/2007	01/08/2007	6
01/02/2007	01/09/2007	7
01/02/2007	01/10/2007	8
01/02/2007	01/11/2007	9
01/02/2007	01/12/2007	10
01/02/2007	01/13/2007	11
01/02/2007	01/14/2007	12

Abbildung 2.2 Durch Subtraktion von zwei Datumsangaben und Umwandlung des Ergebnisses in einen Integer berechnet DAX die Anzahl der Tage zwischen den beiden Daten.

Measures

Berechnete Spalten sind nützlich, aber Sie können Berechnungen in einem DAX-Modell auch auf andere Weise definieren. Wann immer Sie nicht für jede Zeile Werte berechnen, sondern diese aus vielen Zeilen einer Tabelle aggregieren wollen, werden Sie eine Berechnungsform praktisch finden, die als *Measure* bezeichnet wird.

Sie können beispielsweise in der Tabelle *Sales* einige berechnete Spalten definieren, um den Betrag der Bruttogewinnspanne zu berechnen:

```
Sales[SalesAmount] = Sales[Quantity] * Sales[Net Price]
Sales[TotalCost] = Sales[Quantity] * Sales[Unit Cost]
Sales[GrossMargin] = Sales[SalesAmount] – Sales[TotalCost]
```

Was geschieht nun, wenn Sie die Bruttogewinnspanne als prozentualen Anteil des Umsatzbetrags ausweisen möchten? Sie können eine berechnete Spalte mit der folgenden Formel erstellen:

```
Sales[GrossMarginPct] = Sales[GrossMargin] / Sales[SalesAmount]
```

Diese Formel berechnet den korrekten Wert auf Zeilenebene (Abbildung 2.3), aber das Ergebnis ist auf der Ebene der Gesamtsumme eindeutig falsch.

Der als Gesamtbetrag angezeigte Wert ist die Summe der einzelnen prozentualen Anteile, die zeilenweise innerhalb der berechneten Spalte berechnet werden. Wenn wir den Gesamtwert eines Prozentsatzes berechnen wollen, sind berechnete Spalten keine zuverlässige Lösung. Vielmehr müssen wir den Prozentwert auf der Grundlage der Summe der einzelnen Spalten berechnen. Wir müssen also den aggregierten Wert als Summe der Bruttogewinnspanne geteilt durch die Summe des Umsatzbetrags berechnen. In diesem Fall müssen wir das Verhältnis der aggregierten Werte ermitteln – eine Aggregation berechneter Spalten können Sie schlicht nicht verwenden. Wir berechnen folglich das Verhältnis der Summen, nicht die Summe der Verhältnisse.

SalesKey	SalesAmount	TotalCost	GrossMargin	GrossMarginPct
20070104611301-0002	$72.19	$38.74	$33.45	46.34%
20070104611301-0003	$23.75	$11.50	$12.25	51.58%
20070104611320-0006	$216.57	$116.22	$100.35	46.34%
20070104611320-0007	$23.75	$11.50	$12.25	51.58%
20070104611506-0002	$72.19	$38.74	$33.45	46.34%
20070104611506-0003	$23.75	$11.50	$12.25	51.58%
20070104611914-0002	$64.59	$38.74	$25.85	40.02%
20070104611914-0003	$21.25	$11.50	$9.75	45.88%
20070104611952-0004	$64.59	$38.74	$25.85	40.02%
20070104611952-0005	$21.25	$11.50	$9.75	45.88%
20070104611998-0002	$64.59	$38.74	$25.85	40.02%
20070104611998-0003	$63.75	$34.50	$29.25	45.88%
Total	**$732.23**	**$401.92**	**$330.31**	**551.46%**

Abbildung 2.3 Die Spalte *GrossMarginPct* zeigt in jeder Zeile den korrekten Wert an, aber die Gesamtsumme ist falsch.

Ebenso falsch wäre es, die Aggregation der Spalte *GrossMarginPct* einfach in einen Durchschnitt zu ändern und sich darauf zu verlassen, dass das Ergebnis schon richtig sein wird, denn hierdurch würde es zu einer fehlerhaften Auswertung des Prozentwerts kommen, die die Unterschiede zwischen den Beträgen nicht berücksichtigen würde. Das Ergebnis dieses gemittelten Werts ist in Abbildung 2.4 zu sehen. Sie können hier ganz leicht feststellen, dass 330,31 ÷ 732,23 nicht gleich dem angezeigten Wert (45,96 %) ist – vielmehr sollte dieser 45,11 % betragen.

SalesKey	SalesAmount	TotalCost	GrossMargin	Average of GrossMarginPct
20070104611301-0002	$72.19	$38.74	$33.45	46.34%
20070104611301-0003	$23.75	$11.50	$12.25	51.58%
20070104611320-0006	$216.57	$116.22	$100.35	46.34%
20070104611320-0007	$23.75	$11.50	$12.25	51.58%
20070104611506-0002	$72.19	$38.74	$33.45	46.34%
20070104611506-0003	$23.75	$11.50	$12.25	51.58%
20070104611914-0002	$64.59	$38.74	$25.85	40.02%
20070104611914-0003	$21.25	$11.50	$9.75	45.88%
20070104611952-0004	$64.59	$38.74	$25.85	40.02%
20070104611952-0005	$21.25	$11.50	$9.75	45.88%
20070104611998-0002	$64.59	$38.74	$25.85	40.02%
20070104611998-0003	$63.75	$34.50	$29.25	45.88%
Total	**$732.23**	**$401.92**	**$330.31**	**45.96%**

Abbildung 2.4 Die Änderung der Aggregationsmethode in *AVERAGE* liefert nicht das richtige Ergebnis.

Die korrekte Umsetzung für *GrossMarginPct* erfolgt am besten mit einem Measure:

```
GrossMarginPct := SUM ( Sales[GrossMargin] ) / SUM (Sales[SalesAmount] )
```

Wie bereits erwähnt, kann mit einer berechneten Spalte das richtige Ergebnis nicht bestimmt werden. Wenn Sie nicht zeilenweise, sondern mit aggregierten Werten arbeiten müssen, müssen Sie Measures erstellen. Sie haben vielleicht festgestellt, dass ein Measure mit »:=« statt mit dem Gleichheitszeichen (»=«) definiert wird. Diesen Standard haben wir im gesamten Buch verwendet, um die Unterscheidung zwischen Measures und berechneten Spalten im Code zu erleichtern. Nachdem Sie *GrossMarginPct* als Measure definiert haben, ist das Ergebnis korrekt (Abbildung 2.5).

Sowohl Measures als auch berechnete Spalten verwenden DAX-Ausdrücke; der Unterschied liegt im Kontext der Auswertung. Ein Measure wird im Kontext eines visuellen Elements oder einer DAX-Abfrage ausgewertet. Allerdings wird eine berechnete Spalte auf Zeilenebene der Tabelle berechnet, zu der sie gehört. Der Kontext des visuellen Elements (im weiteren Verlauf des Buchs werden Sie erfahren, dass es sich hierbei um einen Filterkontext handelt) hängt von der Auswahl des Benutzers im Bericht oder vom Format der DAX-Abfrage ab. Wenn wir also *SUM(Sales[SalesAmount])* in einem Measure verwenden, dann meinen wir die Summe aller Zeilen, die unter einer Visualisierung zusammengefasst werden. Verwenden wir dagegen *Sales[SalesAmount]* in einer berechneten Spalte, dann beziehen wir uns auf den Wert der Spalte *SalesAmount* in der aktuellen Zeile.

SalesKey ▲	SalesAmount	TotalCost	GrossMargin	GrossMarginPct
20070104611301-0002	$72.19	$38.74	$33.45	46.34%
20070104611301-0003	$23.75	$11.50	$12.25	51.58%
20070104611320-0006	$216.57	$116.22	$100.35	46.34%
20070104611320-0007	$23.75	$11.50	$12.25	51.58%
20070104611506-0002	$72.19	$38.74	$33.45	46.34%
20070104611506-0003	$23.75	$11.50	$12.25	51.58%
20070104611914-0002	$64.59	$38.74	$25.85	40.02%
20070104611914-0003	$21.25	$11.50	$9.75	45.88%
20070104611952-0004	$64.59	$38.74	$25.85	40.02%
20070104611952-0005	$21.25	$11.50	$9.75	45.88%
20070104611998-0002	$64.59	$38.74	$25.85	40.02%
20070104611998-0003	$63.75	$34.50	$29.25	45.88%
Total	**$732.23**	**$401.92**	**$330.31**	**45.11%**

Abbildung 2.5 *GrossMarginPct*, als Measure definiert, zeigt die korrekte Gesamtsumme an.

Ein Measure muss in einer Tabelle definiert werden. Dies ist eine der Anforderungen der DAX-Sprache. Allerdings gehört das Measure eigentlich gar nicht zur Tabelle. Wir können ein Measure vielmehr aus einer Tabelle in eine andere Tabelle verschieben, ohne seine Funktionalität einzubüßen.

Unterschiede zwischen berechneten Spalten und Measures

Obwohl sie ähnlich aussehen, gibt es einen beträchtlichen Unterschied zwischen berechneten Spalten und Measures. Der Wert einer berechneten Spalte wird bei der Datenaktualisierung ermittelt und verwendet die aktuelle Zeile als Kontext. Das Ergebnis ist unabhängig von Benutzerhandlungen im Bericht. Ein Measure dagegen bearbeitet Aggregationen von Daten, die durch den aktuellen Kontext definiert sind. In einer Matrix oder in einer Pivot-Tabelle werden z. B. Quelltabellen entsprechend den Koordinaten von Zellen gefiltert; Daten werden dann mit diesen Filtern aggregiert und berechnet. Mit anderen Worten: Ein Measure arbeitet immer mit Aggregationen von Daten im Auswertungskontext. Das Konzept des Auswertungskontexts wird in Kapitel 4, »Auswertungskontexte verstehen«, näher erläutert.

Zwischen berechneten Spalten und Measures auswählen

Nachdem Sie nun den Unterschied zwischen berechneten Spalten und Measures gesehen haben, sollte es nun darum gehen, wann Sie das eine, wann das andere verwenden. Manchmal steht tatsächlich beides zur Auswahl, aber in den meisten Situationen bestimmen die Anforderungen der Berechnung Ihre Entscheidung.

Als Entwickler müssen Sie immer dann eine berechnete Spalte definieren, wenn Sie Folgendes tun möchten:

- Berechnete Ergebnisse in einem Slicer anordnen, Ergebnisse in Zeilen oder Spalten einer Matrix oder einer Pivot-Tabelle (statt in einem Wertebereich) anzeigen oder die berechnete Spalte als Filterbedingung in einer DAX-Abfrage verwenden
- Einen Ausdruck definieren, der strikt an die aktuelle Zeile gebunden ist. Beispielsweise funktioniert *Price * Quantity* nicht für den Durchschnitt oder die Summe dieser beiden Spalten.
- Text oder Zahlen kategorisieren. Dies könnte beispielsweise ein Wertebereich für ein Measure, ein Altersbereich für Kunden (z. B. 0–18, 18–25 usw.) oder Ähnliches sein Diese Kategorien werden häufig als Filter oder für Slice-and-Dice-Analysen von Werten verwendet.

Unverzichtbar ist die Definition eines Measures, wenn Sie Berechnungswerte anzeigen möchten, die eine Benutzerauswahl wiedergeben, und die Werte etwa in einem Bericht als Aggregate dargestellt werden müssen, um beispielsweise

- den Gewinnprozentsatz einer Berichtsauswahl zu berechnen oder
- Verhältnisse eines Produkts im Vergleich zu allen anderen Produkten zu berechnen, aber beide nach Jahr und Region zu filtern.

Sie können viele Berechnungen sowohl mit berechneten Spalten als auch mit Measures ausdrücken, auch wenn Sie jeweils unterschiedliche DAX-Ausdrücke verwenden müssen. Beispielsweise kann man *GrossMargin* als berechnete Spalte definieren:

```
Sales[GrossMargin] = Sales[SalesAmount] - Sales[TotalProductCost]
```

Genauso gut könnte man aber auch als Measure definieren:

```
GrossMargin := SUM ( Sales[SalesAmount] ) - SUM ( Sales[TotalProductCost] )
```

Wir empfehlen Ihnen, in diesem Fall ein Measure zu verwenden, da es weder Arbeits- noch Festplattenspeicher verbraucht, weil es erst zur Abfragezeit ausgewertet wird. In der Regel sind Measures zu bevorzugen, wenn Sie eine Berechnung in beide Richtungen ausdrücken können. Berechnete Spalten sollten Sie nur in den (relativ seltenen) Fällen verwenden, in denen Sie nicht umhinkommen. Benutzer mit Excel-Erfahrung bevorzugen in der Regel berechnete Spalten gegenüber Measures, da sie Berechnungen in Excel sehr ähnlich sind. Trotzdem bleiben wir dabei: Die beste Möglichkeit, einen Wert in DAX zu berechnen, ist ein Measure.

Measures in berechneten Spalten verwenden

Naheliegenderweise kann ein Measure eine oder mehrere berechnete Spalten referenzieren. Und auch wenn das weniger intuitiv ist, stimmt auch das Gegenteil: Eine berechnete Spalte kann ein Measure referenzieren. Auf diese Weise erzwingt die berechnete Spalte die Berechnung eines Measures für den durch die aktuelle Zeile definierten Kontext. Dieser Vorgang transformiert und konsolidiert das Ergebnis eines Measures in eine Spalte, die nicht durch Benutzerhandlungen beeinflusst wird. Natürlich können nur bestimmte Operationen sinnvolle Ergebnisse liefern, da ein Measure normalerweise Berechnungen durchführt, die in hohem Maße von der Auswahl des Benutzers in der Visualisierung abhängen. Darüber hinaus greifen Sie als Entwickler immer dann, wenn Sie Measures in einer berechneten Spalte verwenden, auf ein Merkmal namens *Kontextübergang* zurück. Hierbei handelt es sich um eine fortgeschrittene Berechnungstechnik in DAX. Damit Sie ein Measure in einer berechneten Spalte korrekt verwenden, empfehlen wir Ihnen dringend, Kapitel 4, das Auswertungskontexte und Kontextübergänge ausführlich erläutert, zu lesen und zu verstehen.

Variablen

Mit Variablen können Sie beim Schreiben eines DAX-Ausdrucks Redundanzen vermeiden und die Lesbarkeit des Codes erheblich verbessern. Betrachten Sie beispielsweise den folgenden Ausdruck:

```
VAR TotalSales = SUM ( Sales[SalesAmount] )
VAR TotalCosts = SUM ( Sales[TotalProductCost] )
VAR GrossMargin = TotalSales - TotalCosts
RETURN
    GrossMargin / TotalSales
```

Variablen werden mit dem Schlüsselwort *VAR* definiert. Wenn Sie eine Variable definiert haben, müssen Sie in einem *RETURN*-Abschnitt den Ergebniswert des Ausdrucks definieren. Sie können beliebig viele Variablen definieren. Diese gelten stets lokal für den Ausdruck, in dem sie definiert sind.

Eine in einem Ausdruck definierte Variable kann dagegen außerhalb des Ausdrucks nicht verwendet werden. Es gibt also keine globale Variablendefinition. Mithin können Sie keine Variablen definieren, die im gesamten DAX-Code des Modells verwendet werden könnten.

Variablen werden mittels Lazy Evaluation berechnet. Das bedeutet nichts anderes, als dass eine Variable nie ausgewertet wird, wenn sie zwar definiert, aber aus irgendeinem Grund nicht im Code ausgewertet wird. Muss die Berechnung hingegen erfolgen, dann geschieht das nur einmal. Bei allen nachfolgenden Abrufen der Variablen wird der zuvor berechnete Wert ausgelesen. Insofern sind Variablen auch als Optimierungstechnik nützlich, wenn sie in einem komplexen Ausdruck mehrfach verwendet werden.

Variablen sind in DAX ein wichtiges Tool. Wie Sie in Kapitel 4 erfahren werden, sind sie äußerst nützlich, da sie nicht den Kontext, in dem sie verwendet werden, sondern den Kontext der Definitionsauswertung nutzen. In Kapitel 6, »Variablen«, werden wir uns eingehend mit Variablen und ihrer Verwendung befassen. Insgesamt werden wir Variablen im gesamten Buch ausgiebig nutzen.

Fehlerbehandlung in DAX-Ausdrücken

Nachdem Sie nun einige Grundlagen der Syntax kennengelernt haben, wollen wir uns an dieser Stelle einmal der Frage widmen, wie man mit ungültigen Berechnungen sinnvoll umgeht. Ein DAX-Ausdruck kann ungültige Berechnungen enthalten, wenn die Daten, auf die er verweist, für die Formel nicht gültig sind. Möglich sind beispielsweise eine Division durch null oder die Referenzierung eines Spaltenwerts, der in einer Rechenoperation (z. B. Multiplikation) verwendet werden soll, aber gar keine Zahl ist. Es ist gut zu wissen, wie diese Fehler standardmäßig behandelt werden und wie man diese Bedingungen für eine solche Behandlung abfängt.

Bevor es jedoch um den Umgang mit Fehlern geht, wollen wir erst einmal die verschiedenen Arten von Fehlern benennen, die bei der Auswertung einer DAX-Formel auftreten können. Es sind dies:

- Konvertierungsfehler
- Fehler bei Rechenoperationen
- Leere oder fehlende Werte

Konvertierungsfehler

Der erste zu beschreibende Fehlertyp ist der Konvertierungsfehler. Wie Sie in diesem Kapitel bereits gesehen haben, konvertiert DAX String- und Zahlenwerte automatisch ineinander, wann immer der Operator es erfordert. Alle folgenden Beispiele sind gültige DAX-Ausdrücke:

```
"10" + 32 = 42
"10" & 32 = "1032"
10 & 32 = "1032"
DATE (2010,3,25) = 3/25/2010
DATE (2010,3,25) + 14 = 4/8/2010
DATE (2010,3,25) & 14 = "3/25/201014"
```

Diese Formeln sind immer korrekt, da sie mit konstanten Werten arbeiten. Aber wie sieht es mit der folgenden Formel aus, wenn *VatCode* ein String ist?

```
Sales[VatCode] + 100
```

Da der erste Operand dieser Summe eine Spalte vom Datentyp *Text* ist, müssen Sie als Entwickler darauf vertrauen, dass DAX alle Werte in dieser Spalte in Zahlen umwandeln kann. Gelingt es DAX nicht, einen Teil der Inhalte entsprechend den Anforderungen des Operators umzuwandeln, dann tritt ein Konvertierungsfehler auf. Hier einige typische Situationen:

```
"1 + 1" + 0 = Der Wert "1 + 1" vom Typ "Text" kann nicht in den Typ "Zahl" konvertiert
werden.
DATEVALUE ("25/14/2010") = Typenkonflikt
```

Wenn Sie diese Fehler vermeiden wollen, ist es wichtig, in DAX-Ausdrücken eine Fehlererkennungslogik hinzuzufügen, um Fehlerbedingungen abzufangen und ein sinnvolles Ergebnis zurückzugeben. Dasselbe Ergebnis erzielen Sie, wenn Sie den Fehler nachträglich abfangen oder die Operanden vorab auf eine Fehlersituation prüfen. Dennoch ist es besser, proaktiv nach der Fehlersituation zu suchen, als den Fehler geschehen zu lassen und dann erst abzufangen.

Fehler bei Rechenoperationen

Die zweite Fehlerkategorie sind bestimmte Rechenoperationen, z. B. eine Division durch null oder das Ziehen der Quadratwurzel einer negativen Zahl. Es handelt sich hierbei nicht um Konvertierungsfehler: DAX löst sie vielmehr aus, wenn Sie versuchen, eine Funktion mit ungültigen Werten aufzurufen oder einen Operator mit falschen Werten zu verwenden.

Die Division durch null erfordert eine besondere Behandlung, da das Verhalten nicht intuitiv ist (zumindest nicht, sofern Sie nicht Mathematiker sind). Wenn man eine Zahl durch null teilt, gibt DAX den Sonderwert *Infinity* zurück. In den Sonderfällen 0 geteilt durch 0 oder *Infinity* geteilt durch *Infinity* gibt DAX dagegen den Sonderwert *NaN* (»not a number«, also keine Zahl) zurück.

Da es sich um ein ungewöhnliches Verhalten handelt, ist es in Tabelle 2.3 zusammengefasst.

Ausdruck	Ergebnis
10 / 0	Infinity
7 / 0	Infinity
0 / 0	NaN
(10 / 0) / (7 / 0)	NaN

Tabelle 2.3 Spezielle Ergebniswerte für die Division durch null

Es ist wichtig zu beachten, dass *Infinity* und *NaN* keine Fehler im eigentlichen Sinne sind, sondern spezielle Werte in DAX. Tatsächlich erzeugt, wenn man eine Zahl durch *Infinity* teilt, der Ausdruck keinen Fehler. Stattdessen gibt er 0 zurück:

```
9954 / ( 7 / 0 ) = 0
```

Abgesehen von dieser Sondersituation kann DAX beim Aufruf einer Funktion mit einem falschen Parameter, wie z. B. der Quadratwurzel einer negativen Zahl, Rechenfehler zurückgeben:

```
SQRT ( -1) = Ein Argument der Funktion "SQRT" weist den falschen Datentyp auf oder das
Ergebnis ist zu lang oder zu kurz.
```

Erkennt DAX solche Fehler, dann wird die weitere Berechnung des Ausdrucks gesperrt und ein Fehler ausgelöst. Mit der Funktion *ISERROR* können Sie prüfen, ob ein Ausdruck zu einem Fehler führt. Wir zeigen dieses Szenario weiter hinten in diesem Kapitel.

Beachten Sie, dass Sonderwerte wie *NaN* auf der Benutzeroberfläche diverser Tools wie z. B. Power BI als normale Werte angezeigt werden. In anderen Clienttools wie etwa einer Excel-Pivot-Tabelle können sie dagegen als Fehler behandelt werden. Schließlich werden diese Sonderwerte auch von den Fehlererkennungsfunktionen als Fehler erkannt.

Leere oder fehlende Werte

Die dritte Kategorie, die wir untersuchen, ist keine konkrete Fehlerbedingung, sondern das Vorhandensein von Leerwerten. Diese können, wenn sie mit anderen Elementen in einer Berechnung kombiniert werden, zu unerwarteten Ergebnissen oder Berechnungsfehlern führen.

DAX behandelt fehlende Werte, Leerwerte oder leere Zellen auf die gleiche Weise: mit dem Wert *BLANK*. *BLANK* ist kein echter Wert, sondern vielmehr eine spezielle Methode, um solche Bedingungen zu identifizieren. Wir können den Wert *BLANK* in einem DAX-Ausdruck erhalten, indem wir die Funktion *BLANK* aufrufen, die sich von einem Leerstring unterscheidet. So gibt beispielsweise der folgende Ausdruck immer einen Leerwert zurück, der in verschiedenen Clienttools entweder als leerer String oder als »(blank)« angezeigt werden kann:

```
= BLANK ()
```

Für sich genommen ist dieser Ausdruck nutzlos, aber die *BLANK*-Funktion selbst erweist sich jedes Mal als nützlich, wenn es darum geht, einen leeren Wert zurückzugeben. Nehmen wir beispielsweise an, Sie wollten anstelle von 0 ein leeres Ergebnis anzeigen. Der folgende Ausdruck berechnet den gesamten Rabatt für eine Verkaufstransaktion und lässt den Wert leer, wenn der Rabatt 0 beträgt:

```
=IF (
    Sales[DiscountPerc] = 0,            -- Überprüfung auf Rabatt
    BLANK (),                           -- Gibt leeren Wert zurück, wenn kein Rabatt
    Sales[DiscountPerc] * Sales[Amount]   -- Berechnet andernfalls den Rabatt
)
```

BLANK ist an sich kein Fehler, sondern nur ein leerer Wert. Daher kann ein Ausdruck, der *BLANK* enthält, je nach erforderlicher Berechnung einen Wert oder einen Leerwert zurückgeben. Der folgende Ausdruck etwa gibt *BLANK* zurück, wenn *Sales[Amount] BLANK* ist:

```
= 10 * Sales[Amount]
```

Anders formuliert: Das Ergebnis eines Rechenprodukts ist *BLANK*, wenn mindestens ein Term *BLANK* ist. Dies kann problematisch werden, wenn es notwendig ist, auf einen leeren Wert zu prüfen. Aufgrund der impliziten Konvertierungen ist es nicht möglich zu unterscheiden, ob bei Verwendung eines Gleichheitsoperators ein Ausdruck 0 (oder ein Leerstring) oder aber *BLANK* ist. Tatsächlich sind die folgenden logischen Bedingungen immer wahr:

```
BLANK () = 0     -- Gibt immer TRUE zurück
BLANK () = ""    -- Gibt immer TRUE zurück
```

Wenn also die Spalten *Sales[DiscountPerc]* oder *Sales[Clerk]* leer sind, geben die folgenden Bedingungen auch dann *TRUE* zurück, wenn die Prüfung auf 0 bzw. einen Leerstring erfolgt:

```
Sales[DiscountPerc] = 0  -- Gibt TRUE zurück, wenn DiscountPerc BLANK oder 0 ist
Sales[Clerk] = ""        -- Gibt TRUE zurück, wenn Clerk BLANK oder "" ist
```

In solchen Fällen kann man mit der Funktion *ISBLANK* darauf prüfen, ob ein Wert *BLANK* ist:

```
ISBLANK ( Sales[DiscountPerc]) -- Gibt TRUE nur dann zurück, wenn DiscountPerc BLANK ist
ISBLANK ( Sales[Clerk] )         -- Gibt TRUE nur dann zurück, wenn Clerk BLANK ist
```

Die Fortpflanzung von *BLANK* in einem DAX-Ausdruck erfolgt in verschiedenen weiteren Rechen- und Logikoperationen. Die folgenden Beispiele zeigen dies:

```
BLANK () + BLANK () = BLANK ()
10 * BLANK () = BLANK ()
BLANK () / 3 = BLANK ()
BLANK () / BLANK () = BLANK ()
```

Allerdings erfolgt die Fortpflanzung von *BLANK* im Ergebnis eines Ausdrucks nicht für alle Formeln. Bei manchen Berechnungen gibt es keine Fortpflanzung von *BLANK*. Stattdessen geben sie einen Wert zurück, der von den anderen Termen der Formel abhängt. Beispiele hierfür sind Addition, Subtraktion, Division durch *BLANK* und eine Logikoperation mit einem *BLANK*-Wert. Die folgenden Ausdrücke zeigen einige dieser Bedingungen und ihre Ergebnisse:

```
BLANK () – 10 = –10
18 + BLANK () = 18
4 / BLANK () = Infinity
0 / BLANK () = NaN
BLANK () || BLANK () = FALSE
BLANK () && BLANK () = FALSE
( BLANK () = BLANK () ) = TRUE
( BLANK () = TRUE ) = FALSE
( BLANK () = FALSE ) = TRUE
( BLANK () = 0 ) = TRUE
```

```
( BLANK () = "" ) = TRUE
ISBLANK ( BLANK() ) = TRUE
FALSE || BLANK () = FALSE
FALSE && BLANK () = FALSE
TRUE || BLANK () = TRUE
TRUE && BLANK () = FALSE
```

Leere Werte in Excel und SQL

Excel verwendet eine andere Methode zur Behandlung leerer Werte. In Excel werden alle leeren Werte als 0 betrachtet, wenn sie in einer Summe oder in einer Multiplikation verwendet werden. Sind sie jedoch Teil einer Division oder eines logischen Ausdrucks, dann können sie einen Fehler auslösen.

In SQL pflanzen sich NULL-Werte in einem Ausdruck anders fort als *BLANK* in DAX. Wie Sie in den obigen Beispielen gesehen haben, führt ein *BLANK* in einem DAX-Ausdruck nicht immer zum Ergebnis *BLANK*, während *NULL* in SQL oft dazu führt, dass der gesamte Ausdruck *NULL* wird. Dieser Unterschied ist immer dann relevant, wenn Sie neben einer relationalen Datenbank DirectQuery verwenden, da einige Berechnungen in SQL und andere in DAX ausgeführt werden. Die unterschiedliche Semantik von *BLANK* in den beiden Engines kann zu unerwartetem Verhalten führen.

Zur Steuerung der Ergebnisse eines DAX-Ausdrucks ist es wichtig, das Verhalten leerer oder fehlender Werte darin und die Verwendung von *BLANK* zur Rückgabe einer leeren Zelle in einer Berechnung zu verstehen. Sie können *BLANK* häufig als Ergebnis verwenden, um fehlerhafte Werte oder sonstige Fehler zu erkennen. Wir werden das im nächsten Abschnitt zeigen.

Fehler abfangen

Nachdem wir nun die verschiedenen möglichen Fehlertypen ausführlich beschrieben haben, wollen wir Ihnen Techniken zeigen, mit denen Fehler abgefangen und korrigiert werden oder zumindest eine Fehlermeldung mit aussagekräftigen Informationen erzeugt wird. Fehler in einem DAX-Ausdruck hängen häufig vom Wert der im Ausdruck selbst verwendeten Spalten ab. Daher kann es sinnvoll sein, auf diese Fehlerbedingungen zu prüfen und ggf. eine Fehlermeldung zurückzugeben. Das Standardverfahren sieht vor, zu prüfen, ob ein Ausdruck einen Fehler zurückgibt, und einen solchen Fehler ggf. durch eine bestimmte Nachricht oder einen Standardwert zu ersetzen. Für diese Aufgabe gibt es einige DAX-Funktionen.

Die erste davon heißt *IFERROR*-Funktion. Sie ähnelt der *IF*-Funktion, aber statt eine boolesche Bedingung auszuwerten, prüft sie, ob ein Ausdruck einen Fehler zurückgibt. Hier sehen Sie zwei typische Anwendungen der *IFERROR*-Funktion:

```
= IFERROR ( Sales[Quantity] * Sales[Price], BLANK () )
= IFERROR ( SQRT ( Test[Omega] ), BLANK () )
```

Wenn im ersten Ausdruck entweder *Sales[Quantity]* oder *Sales[Price]* ein String ist, der nicht in eine Zahl umgewandelt werden kann, ist der zurückgegebene Ausdruck ein leerer Wert. Andernfalls wird das Produkt aus *Quantity* und *Price* zurückgegeben.

Im zweiten Ausdruck ist das Ergebnis immer dann eine leere Zelle, wenn die Spalte *Test [Omega]* eine negative Zahl enthält.

Eine solche Verwendung von *IFERROR* entspricht einem grundsätzlicheren Muster, das *ISERROR* und *IF* erfordert:

```
= IF (
    ISERROR ( Sales[Quantity] * Sales[Price] ),
    BLANK (),
    Sales[Quantity] * Sales[Price]
)

= IF (
    ISERROR ( SQRT ( Test[Omega] ) ),
    BLANK (),
    SQRT ( Test[Omega] )
)
```

In diesen Fällen ist *IFERROR* die bessere Option. Man kann *IFERROR* nämlich immer dann verwenden, wenn das Ergebnis derselbe Ausdruck ist, der auf einen Fehler getestet wurde; es besteht also keine Notwendigkeit, den Ausdruck doppelt aufzuführen. Zudem macht dies den Code sicherer und besser lesbar. Dagegen sollte sich der Entwickler für *IF* entscheiden, wenn er das Ergebnis eines anderen Ausdrucks zurückgeben möchte.

Außerdem kann man auf diese Weise einen Fehler schon im Vorhinein vermeiden: Man testet Parameter einfach, bevor man sie verwendet. So kann man beispielsweise prüfen, ob das Argument für *SQRT* positiv ist, und für negative Werte *BLANK* zurückgeben:

```
= IF (
    Test[Omega] >= 0,
    SQRT ( Test[Omega] ),
    BLANK ()
)
```

Da das dritte Argument einer *IF*-Anweisung standardmäßig auf *BLANK* festgelegt ist, können Sie denselben Ausdruck auch knapper formulieren:

```
= IF (
    Test[Omega] >= 0,
    SQRT ( Test[Omega] )
)
```

Ein häufiges Szenario ist das Prüfen auf leere Werte. *ISBLANK* erkennt leere Werte und gibt *TRUE* zurück, wenn das Argument *BLANK* ist. Diese Fähigkeit ist insbesondere dann wichtig, wenn ein nicht vorhandener Wert nicht unbedingt bedeutet, dass er 0 ist. Das folgende Beispiel

berechnet die Versandkosten für eine Verkaufstransaktion unter Verwendung eines Standardwerts für die Produktversandkosten, wenn in der Transaktion selbst kein Gewicht angegeben ist:

```
= IF (
    ISBLANK ( Sales[Weight] ),            -- Wenn das Gewicht fehlt,
    Sales[DefaultShippingCost],           -- dann gib die Standardkosten zurück
    Sales[Weight] * Sales[ShippingPrice]  -- Andernfalls multipliziere das Gewicht mit den
Versandkosten
)
```

Würden wir einfach nur das Produktgewicht mit dem Versandpreis multiplizieren, dann erhielten wir aufgrund der Fortpflanzung von *BLANK* in Multiplikationen Leerkosten für alle Verkaufstransaktionen ohne Gewichtsdaten.

Bei der Verwendung von Variablen müssen Fehler zum Zeitpunkt der Variablendefinition und nicht erst dann überprüft werden, wenn wir sie verwenden. Tatsächlich gibt die erste Formel im folgenden Code null zurück, die zweite Formel löst immer einen Fehler aus und die letzte führt je nach Version des Produkts, das DAX verwendet, zu unterschiedlichen Ergebnissen (wobei die neueste Version ebenfalls einen Fehler auslöst):

```
IFERROR ( SQRT ( -1 ), 0 )                -- Gibt 0 zurück

VAR WrongValue = SQRT ( -1 )              -- Hier entsteht ein Fehler, das heißt, das
Ergebnis
RETURN                                    -- ist immer ein Fehler
    IFERROR ( WrongValue, 0 )             -- Diese Zeile wird nie ausgeführt

IFERROR (                                 -- Verschiedene Ergebnisse je nach Version
    VAR WrongValue = SQRT ( -1 )          -- IFERROR löst in 2017er-Versionen einen Fehler
aus
    RETURN                                -- IFERROR gibt in Versionen bis 2016 0 zurück
        WrongValue,
    0
)
```

Der Fehler tritt auf, wenn *WrongValue* ausgewertet wird. Somit wird die Engine im zweiten Beispiel niemals die *IFERROR*-Funktion ausführen, während das Ergebnis des dritten Beispiels von den Produktversionen abhängt. Wenn Sie auf Fehler prüfen müssen, treffen Sie ein paar zusätzliche Vorsichtsmaßnahmen bei der Verwendung von Variablen.