DATA ACCESS

Die Datenzugriffsberatung mit DR. HOLGER SCHWICHTENBERG

TEMPORALE TABELLEN MIT HISTORISCHEN DATENSÄTZEN IN SQL SERVER UND ENTITY FRAMEWORK CORE

Geschichtsschreibung

Temporale Tabellen merken sich alle früheren Zustände von Datensätzen.

Temporale Tabellen (englisch: Temporal Tables) sind seit SQL:2011 (ISO/IEC 9075:2011) Teil des SQL-Standards [1]. Solche Tabellen erlauben, dass ein Entwickler beim Ändern oder Löschen eines Datensatzes nichts explizit tun muss, um den vorherigen Zustand zu erhalten; das Datenbankmanagementsystem merkt sich vollautomatisch alle vorherigen Zustände. Einige moderne Datenbankmanagementsysteme bieten Unterstützung für temporale Tabellen, zum Beispiel Oracle, DB2, MySQL, MariaDB und Microsoft SQL Server. Letzteres kann dies seit Version 2016. Auch in der Cloud in SQL Azure sind die temporalen Funktionen verfügbar. Ab Entity Framework Core 6.0 gibt es dafür auch eine Unterstützung im objektrelationalen Mapper von Microsoft.

Temporale Tabellen im SQL Server aktivieren

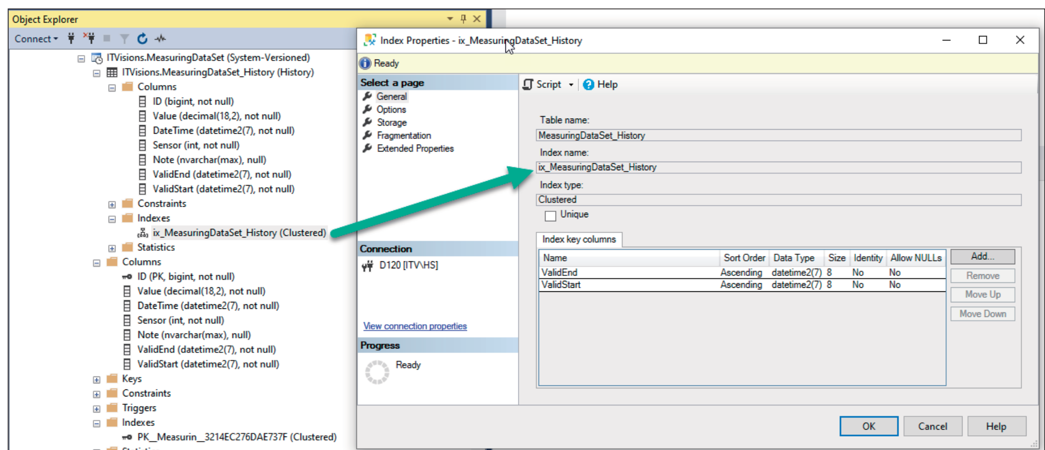
In Microsoft SQL Server heißt das Feature offiziell „system-versioned temporal tables“. Eine solche temporale Tabelle besteht tatsächlich aus zwei Tabellen: einer Tabelle mit den aktuell gültigen Datensätzen und einer Tabelle mit den früheren („historischen“) Datensätzen („History-Tabelle“). Beide Tabellen benötigen zwei zusätzliche Spalten, die den Start- und den Endzeitpunkt der Gültigkeit des Datensatzes festlegen. Diese Spalten legt man wie folgt an:

- **Startzeitpunkt:** Eine Spalte vom Typ *datetime2* mit Zusatz *GENERATED ALWAYS AS ROW START NOT NULL.*
- **Endzeitpunkt:** Eine Spalte vom Typ *datetime2* mit Zusatz *GENERATED ALWAYS AS ROW END NOT NULL.*

Listing 1: Anlegen einer temporalen Tabelle mit T-SQL

```
CREATE TABLE ITVisions.[MeasuringDataSet]
(
  [ID] [bigint] IDENTITY(1,1) PRIMARY KEY CLUSTERED,
  [Value] [decimal](18, 2) NOT NULL,
  [DateTime] [datetime2](7) NOT NULL,
  [Sensor] [int] NOT NULL,
  [Note] [nvarchar](max) NULL,
  [ValidStart] [datetime2](7) GENERATED
    ALWAYS AS ROW START NOT NULL,
  [ValidEnd] [datetime2](7) GENERATED
    ALWAYS AS ROW END NOT NULL,
    PERIOD FOR SYSTEM_TIME ([ValidStart], [ValidEnd])
)

WITH (SYSTEM_VERSIONING = ON
  (HISTORY_TABLE = ITVision
  s.MeasuringDataSet_History));
```



Die in Listing 1 angelegte temporale Tabelle im SQL Server Management Studio (Bild 1)

Ihre besondere Bedeutung gibt man diesen beiden Spalten mit dem Zusatz

```
PERIOD FOR SYSTEM_TIME (
  [Startzeitpunktspaltenname],
  [Endzeitpunktspaltenname])
```

Eine direkte Wertbelegung dieser Spalten bei *INSERT* und *UPDATE* ist dann nicht mehr möglich. Zudem muss bei *CREATE TABLE* die Angabe

```
WITH (SYSTEM_VERSIONING = ON
  (HISTORY_TABLE = Tabellenname))
```

erfolgen, um die Versionierung zu aktivieren und den Namen der History-Tabelle festzulegen. Zusätzlich ist in der aktuellen Tabelle immer ein Primärschlüssel erforderlich. Listing 1 zeigt ein Beispiel für das Anlegen einer temporalen Tabelle via T-SQL. Über das SQL Server Management Studio ist dies nicht möglich. Theoretisch kann man auch nur

```
WITH (SYSTEM_VERSIONING = ON)
```

schreiben, dann vergibt das Datenbankmanagementsystem aber für die History-Tabellen einen unschönen Namen wie *MSSQL_TemporalHistoryFor_741577680*. Eine weitere Option ist, die History-Tabelle manuell anzulegen, zum Beispiel mit einem Clustered Columnstore Index, siehe [2].

Die in Listing 1 angelegte temporale Tabelle zeigt Bild 1 im SQL Server Management Studio. Man sieht, dass die History-Tabelle eine Ebene unter der eigentlichen Tabelle dargestellt wird.

Die Spalten sind in beiden Tabellen identisch. Die History-Tabelle besitzt – im Gegensatz zur aktuellen Tabelle – keinen Primärschlüssel (die Werte in der Primärschlüsselspalte der Haupttabelle können hier ja mehrfach vorkommen, wenn der Datensatz sich mehrmals ändert), hat aber automatisch als Clustered Index die Kombination aus Startzeitpunktspalte und Endzeitpunktspalte erhalten.

Man kann auch eine bestehende, leere oder bereits mit Daten gefüllte Tabelle nachträglich in eine temporale Tabelle umwandeln, siehe Listing 2. Weitere Einschränkungen bezüglich temporaler Tabellen gibt es hinsichtlich der Tabellen- und Spaltentypen: *Filetable*, *Filestream* und *Linked Tables* werden nicht unterstützt [3].

```
21 select * from [ITVisions].[MeasuringData] where ID = 8204
22 select * from [ITVisions].[MeasuringData_History] where ID = 8204
23 select * from [ITVisions].[MeasuringData]  FOR SYSTEM_TIME as of '2021-09-27 13:42:44'  where ID = 8204
24 select * from [ITVisions].[MeasuringData]  FOR SYSTEM_TIME FROM  '2021-09-27 13:42:00' to  '2021-09-27 13:45:00' where ID = 8204
25 select * from [ITVisions].[MeasuringData]  FOR SYSTEM_TIME ALL  where ID = 8204
```

100 %

Results | Messages

	ID	Value	DateTime	Sensor	Note	ValidEnd	ValidStart
1	8204	4815.00	2021-09-27 15:44:51.8916550	4765	NULL	9999-12-31 23:59:59.9999999	2021-09-27 13:44:52.5003206

	ID	Value	DateTime	Sensor	Note	ValidEnd	ValidStart
1	8204	4765.00	2021-09-27 15:37:19.1591214	4765	NULL	2021-09-27 13:40:09.0779724	2021-09-27 13:37:19.7965585
2	8204	4775.00	2021-09-27 15:40:08.4977804	4765	NULL	2021-09-27 13:40:31.2516919	2021-09-27 13:40:09.0779724
3	8204	4785.00	2021-09-27 15:40:30.6674292	4765	NULL	2021-09-27 13:42:45.3734282	2021-09-27 13:40:31.2516919
4	8204	4795.00	2021-09-27 15:42:44.7863465	4765	NULL	2021-09-27 13:43:00.3730596	2021-09-27 13:42:45.3734282
5	8204	4805.00	2021-09-27 15:42:59.7869535	4765	NULL	2021-09-27 13:44:52.5003206	2021-09-27 13:43:00.3730596

	ID	Value	DateTime	Sensor	Note	ValidEnd	ValidStart
1	8204	4785.00	2021-09-27 15:40:30.6674292	4765	NULL	2021-09-27 13:42:45.3734282	2021-09-27 13:40:31.2516919

	ID	Value	DateTime	Sensor	Note	ValidEnd	ValidStart
1	8204	4815.00	2021-09-27 15:44:51.8916550	4765	NULL	9999-12-31 23:59:59.9999999	2021-09-27 13:44:52.5003206
2	8204	4785.00	2021-09-27 15:40:30.6674292	4765	NULL	2021-09-27 13:42:45.3734282	2021-09-27 13:40:31.2516919
3	8204	4795.00	2021-09-27 15:42:44.7863465	4765	NULL	2021-09-27 13:43:00.3730596	2021-09-27 13:42:45.3734282
4	8204	4805.00	2021-09-27 15:42:59.7869535	4765	NULL	2021-09-27 13:44:52.5003206	2021-09-27 13:43:00.3730596

	ID	Value	DateTime	Sensor	Note	ValidEnd	ValidStart
1	8204	4815.00	2021-09-27 15:44:51.8916550	4765	NULL	9999-12-31 23:59:59.9999999	2021-09-27 13:44:52.5003206
2	8204	4765.00	2021-09-27 15:37:19.1591214	4765	NULL	2021-09-27 13:40:09.0779724	2021-09-27 13:37:19.7965585
3	8204	4775.00	2021-09-27 15:40:08.4977804	4765	NULL	2021-09-27 13:40:31.2516919	2021-09-27 13:40:09.0779724
4	8204	4785.00	2021-09-27 15:40:30.6674292	4765	NULL	2021-09-27 13:42:45.3734282	2021-09-27 13:40:31.2516919
5	8204	4795.00	2021-09-27 15:42:44.7863465	4765	NULL	2021-09-27 13:43:00.3730596	2021-09-27 13:42:45.3734282
6	8204	4805.00	2021-09-27 15:42:59.7869535	4765	NULL	2021-09-27 13:44:52.5003206	2021-09-27 13:43:00.3730596

SELECT-Abfragen auf einer temporalen Tabelle (Bild 2)

Abfrage temporaler Tabellen

Die History-Tabellen kann man ebenso wie die übergeordnete aktuelle Tabelle mit den einfachen, bekannten *SELECT*-Befehlen abfragen.

In dem zweiten Resultset in Bild 2 sieht man, dass der Mess-Datensatz 8204 zu Sensor 4765 fünfmal geändert wurde, weil es in der History-Tabelle fünf verschiedene Einträge mit exakten Gültigkeitszeiträumen gibt.

Der aktuelle Datensatz hat bei *ValidEnd* das Jahresende des Jahres 9999 eingetragen, was bedeuten soll, dass er unbegrenzt gültig ist – zumindest so lange, bis ihn eine neue Version des Datensatzes ablösen wird.

Die Spalten *ValidStart* und *ValidEnd* könnte man dazu verwenden, die Daten der Vergangenheit einzugrenzen. Der SQL Server unterstützt aber verschiedene Hilfsoperationen, die dies einfacher machen:

- *FOR SYSTEM_TIME AS OF <date_time>*
- *FOR SYSTEM_TIME FROM <start_date_time> TO <end_date_time>* ▶

- *FOR SYSTEM_TIME BETWEEN <start_date_time> AND <end_date_time>*
- *FOR SYSTEM_TIME CONTAINED IN (<start_date_time> , <end_date_time>)*
- *FOR SYSTEM_TIME ALL*

So liefert eine Abfrage auf der aktuellen Tabelle mit *as of* den Datensatzzustand für einen bestimmten Zeitpunkt. Beispiel: am 27.9.2021 um 13:42:44 Uhr:

```
select * from [ITVisions].[MeasuringData]
  FOR SYSTEM_TIME as of '2021-09-27 13:42:44'
  where ID = 8204
```

Diese Abfrage liefert den Messwert *4785*, der zu diesem Zeitpunkt gültig war, entsprechend Zeile 3 im zweiten Resultset in Bild 2. Die Abfrage über einen Zeitraum

```
select * from [ITVisions].[MeasuringData]  FOR
  SYSTEM_TIME FROM '2021-09-27 13:42:00' to
  '2021-09-27 13:45:00' where ID = 8204
```

liefert die historischen Werte *4785*, *4795* und *4805* sowie den aktuellen Wert *4815* (vergleiche Bild 2). Mit

```
select * from [ITVisions].[MeasuringData]
  FOR SYSTEM_TIME ALL  where ID = 8204
```

bekommt der Datenbanknutzer den aktuellen Datensatz und alle fünf Vorgängerversionen.

Temporale Tabellen mit Entity Framework Core

Das Entity-Framework-Core-Entwicklungsteam hat die Unterstützung für temporale Tabellen in relationalen Datenbankmanagementsystemen seit Version 6.0 Release Candidate 1 (erschienen am 14. September 2021) ins Produkt eingebaut und auch im Provider für den Microsoft SQL Server implementiert. Für die Unterstützung in anderen Datenbankmanagementsystemen müssen die Providerhersteller ihre Provider aktualisieren.

Beim Forward Engineering kann ein Entwickler mit dem Aufruf *IsTemporal()* im Fluent-API eine temporale Tabelle erzwingen – sofern der Provider dies unterstützt:

```
modelBuilder.Entity<MeasuringData>().ToTable(
  tb => tb.IsTemporal());
```

Ohne weitere Festlegung erhält die temporale Tabelle den Zusatznamen *History* und die Gültigkeitsspalten heißen *PeriodStart* und *PeriodEnd*. Der Entwickler kann aber die Schema-, Tabellen- und Spaltennamen explizit setzen:

```
modelBuilder.Entity<MeasuringData>()
    .ToTable("MeasuringData","ITVisions")
    .ToTable(tb => tb.IsTemporal(ttb =>
{
  ttb.UseHistoryTable("MeasuringData_History",
    "ITVisions");
  ttb.HasPeriodStart("ValidStart");
  ttb.HasPeriodEnd("ValidEnd");
}));
```

Diese Zusatzspalten laufen bei Entity Framework Core technisch als „Shadow Property" – sie müssen und sie dürfen nicht explizit in der Entitätsklasse erscheinen. Wer dennoch die beiden Properties dort anlegt, kassiert einen Laufzeitfehler: *„Period property must be a shadow property."*

Beim Reverse Engineering einer Datenbank mit temporalen Tabellen gab es in den RC-Versionen einen Fehler im Werkzeug. Der Fehler ist in Version 6.0 RTM behoben, aber leider ignoriert Entity Framework Core nun die History-Tabelle. Es wird zwar eine Entitätsklasse für die Tabelle generiert, diese wird aber nicht mit *IsTemporal()* in der Kontextklasse ausgewiesen. Auch die Entity Framework Core Power Tools (Stand Version 2.5.827) haben dies noch nicht implementiert. Wann dies bei Reverse Engineering implementiert

Listing 2: Nachträgliche Aktivierung der temporalen Funktion für eine bestehende Tabelle

```
CREATE TABLE ITVisions.[MeasuringDataSet2]
(
  [ID] [bigint] IDENTITY(1,1) PRIMARY KEY CLUSTERED,
  [Value] [decimal](18, 2) NOT NULL,
  [DateTime] [datetime2](7) NOT NULL,
  [Sensor] [int] NOT NULL,
  [Note] [nvarchar](max) NULL
)

--- Hier könnten schon Datensätze eingefügt werden!

ALTER TABLE ITVisions.[MeasuringDataSet2]
ADD
  [ValidStart] datetime2 GENERATED ALWAYS AS ROW
    START NOT NULL
    DEFAULT SYSUTCDATETIME(),
  [ValidEnd] datetime2 GENERATED ALWAYS AS ROW END
    NOT NULL
    DEFAULT CAST('9999-12-31 23:59:59.9999999'
      AS datetime2),
  PERIOD FOR SYSTEM_TIME ([ValidStart],[ValidEnd]);

ALTER TABLE ITVisions.[MeasuringDataSet2]
SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE =
  ITVisions.[MeasuringDataSet2_History]));
```

Listing 3: LINQ-Abfragen mit den neuen temporalen Operatoren

```
using (var ctx = new Context()) {
  CUI.H2("ToList()");
  var alle = ctx.MeasuringDataSet.ToList();
  CUI.H3(alle.Count + " aktuelle Datensätze:");
  Print(ctx, alle);

  CUI.H2("TemporalAll()");
  var alleHistory =
    ctx.MeasuringDataSet.TemporalAll().ToList();
  CUI.H3(alleHistory.Count +
    " aktuelle und alte Datensätze:");
  Print(ctx, alleHistory);

  CUI.H2("SingleOrDefault()");
  var DatensatzAktuell = ctx.MeasuringDataSet
    .SingleOrDefault(x => x.ID == ID);
  Print(ctx, DatensatzAktuell);

  var vorEinemTag = ZeitpunktVorDenAenderungen
    .AddDays(-1); // Zustand vor einem Tag:
                  // das wird nichts liefern
  CUI.H2("TemporalAsOf: Datensatz zum " +
    vorEinemTag);
  var DatensatzNochAelter = ctx.MeasuringDataSet
    .TemporalAsOf(vorEinemTag.ToUniversalTime())
    .SingleOrDefault(x => x.ID == ID);
  Print(ctx, DatensatzNochAelter);

  CUI.H2("TemporalAsOf: Datensatz zum "
    + ZeitpunktVorDenAenderungen);
    // ToUniversalTime() --> Zeitzone beachten!
  var DatensatzAlt = ctx.MeasuringDataSet
    .TemporalAsOf(ZeitpunktVorDenAenderungen
    .ToUniversalTime()).SingleOrDefault(
       x => x.ID == ID);
  Print(ctx, DatensatzAlt);

  using (var ctx2 = new Context()) {
    CUI.H2("Änderung des am historischen Datensatz
      erzeugen neuen aktuellen Datensatz:");
    Print(ctx2, DatensatzAlt);
    ctx2.Attach(DatensatzAlt);
    Print(ctx2, DatensatzAlt);
    DatensatzAlt.Value--;
    Print(ctx2, DatensatzAlt);
    Console.WriteLine("Anzahl der gespeicherte
      Änderungen: " + ctx2.SaveChanges());
    Print(ctx2, DatensatzAlt);
  }

  CUI.H2($"TemporalBetween: Alle Zustände des
    Datensatzes von {vorEinemTag} bis
    {DateTime.UtcNow.AddDays(1)}:");
  var DatensaetzeInEinemZeitraum =
    ctx.MeasuringDataSet.TemporalBetween(vorEinemTag,
    DateTime.UtcNow.AddDays(1)).Where(
    x => x.ID == ID).OrderBy(x => x.DateTime)
    .ToList();
  CUI.H3(DatensaetzeInEinemZeitraum.Count +
    " Datensätze im Zeitraum:");
  Print(ctx, DatensaetzeInEinemZeitraum);
}

private static void Print(Context ctx,
  List<MeasuringData> alle) {
  foreach (var d in alle)
  {
    Print(ctx, d);
  }
}

private static void Print(Context ctx,
  MeasuringData d) {
  if (d == null) CUI.Warning(
    $"Kein Datensatz gefunden!");
  else Console.WriteLine($"Datensatz #{d.ID}
    {d.DateTime} Wert: {d.Value} State:
    {ctx.Entry(d).State} {ctx.Entry(d)
      .Property("ValidStart").CurrentValue}->
    {ctx.Entry(d).Property("ValidEnd")
      .CurrentValue}");
}
```

sein wird, kann hier nicht versprochen werden. Als schlechtes Beispiel sei erwähnt, dass Entity Framework Core zwar schon seit Version 5.0 die Abstraktion n:m-Zwischentabellen beherrscht, dieses Feature aber auch erst seit Version 6.0 RC1 beim Reverse Engineering unterstützt wird.

In LINQ-Abfragen bietet Entity Framework Core 6.0 aber bereits Unterstützung für temporale Tabellen an, nämlich mit den neuen Operationen *TemporalAsOf()*, *TemporalAll()*, *TemporalBetween()*, *TemporalFromTo()* und *TemporalContainedIn()*. Listing 3 zeigt verschiedene Abfragebeispiele mit LINQ unter Einsatz dieser Operatoren. Zu beachten ist, dass alle Zeitangaben in den temporalen Operatoren als koordinierte Weltzeit (UTC) erfolgen müssen und schon kleinste Abweichungen zwischen der Client- und der Serveruhrzeit zu unerwarteten Ergebnissen führen können.

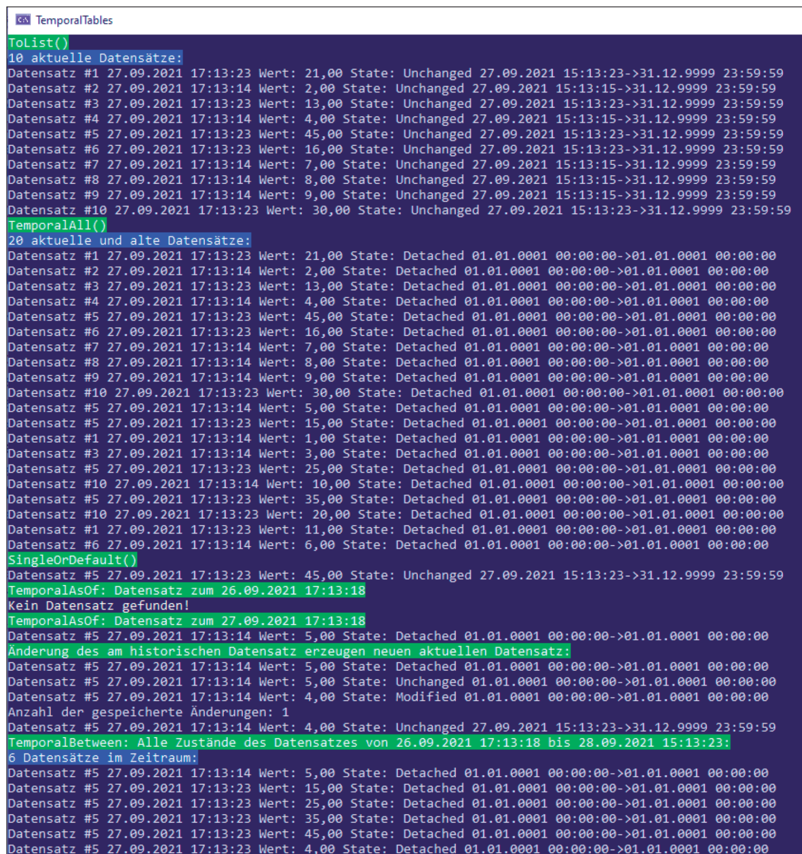
Listing 3 zeigt in der *Print()*-Methode auch, wie man eine Shadow Property abfragt. Allerdings sieht man in Bild 3, dass diese Shadow Properties nur bei den normalen Abfragen ohne die temporalen Operatoren gefüllt sind. Würde man diese Abfragen im No-Tracking-Modus ausführen, wären die ▶

Shadow Properties auch bei normalen Abfragen ebenfalls leer. Dabei sind wir schon bei einem entscheidenden Punkt: Alle Abfragen mit temporalen Operatoren liefern die Objekte im No-Tracking-Modus, also im Zustand *Detached*, bei dem keine Änderungsverfolgung von Entity Framework Core stattfindet. Anders ist es auch nicht möglich, denn Entity Framework Core verwendet für die interne Änderungsverfolgung den Primärschlüssel, der aber bei den historischen Zuständen der Datensätze nicht mehr eindeutig einen Datensatz identifiziert. Dass hier keine Änderungsverfolgung aktiv ist, ist aber auch nicht schlimm, denn die historischen Datensätze sollen ja unveränderlich sein.

Leser dieser Kolumne wissen, dass man *Detached*-Objekte dennoch zur Änderungsverfolgung heranziehen kann, indem man *Attach()* ausführt. Dadurch wird der Datensatz in den Zustand *unchanged* versetzt und nachfolgende Änderungen bringen ihn in den Zustand *modified*, was beim Aufruf von *SaveChanges()* zur Persistierung der Änderungen führt.

Wer dies mit einem historischen Datensatz macht, verändert jedoch nicht den historischen Datensatz, sondern den aktuellen. Er schreibt unter Umständen veraltete Werte zurück in die Datenbank. Das kann gewollt sein, das kann aber auch ein schwerer Programmierfehler sein.

Zudem gelingt ein Aufrufen von *Attach()* für ein zu einem historischen Datensatz gehörendes Entitätsobjekt natürlich nur, wenn es im aktuellen Kontext nicht schon eine Instanz des aktuellen Datensatzes oder eines anderen historischen Datensatzes mit dem gleichen Primärschlüssel gibt.



Ausgabe von Listing 3 für die überschaubare Menge von zehn Datensätzen (Bild 3)

Schemaänderungen

Eine Tabelle mit aktivierter temporaler Unterstützung kann man nicht im SQL Server Management Studio grafisch bearbeiten; der Eintrag *Design* im Kontextmenü ist nicht verfügbar (Stand Version 18.8). Nur Schemaänderungen per T-SQL (*ALTER TABLE*) sind möglich.

Dabei stellt sich aber die Frage, was mit der History-Tabelle bei Schemaänderungen passiert. Wenn man eine neue Spalte einfügt, egal ob mit T-SQL

```
alter table [ITVisions].[MeasuringData]
  add Source nvarchar(10)
```

oder via Entity-Framework-Core-Schemamigration, wird nicht nur die aktuelle Tabelle, sondern auch die History-Tabelle automatisch um diese Spalte erweitert. Aus diesem Grund gelingt bei bereits mit Daten gefüllter Tabelle ein Ergänzen einer Spalte mit Zusatz *not null* nicht:

```
alter table [ITVisions].[MeasuringData]
  add Source nvarchar(10) not null
```

Das führt zum Fehler: „*ALTER TABLE only allows columns to be added that can contain nulls, or have a DEFAULT definition specified, or the column being added is an identity or timestamp column, or alternatively if none of the previous conditions are satisfied the table must be empty to allow addition of this column. Column 'Source' cannot be added to non-empty table 'MeasuringData_History' because it does not satisfy these conditions.*“

Wenn man eine Spalte löscht

```
alter table [ITVisions]
  .[MeasuringData] DROP COLUMN Source
```

wird diese Spalte auch in der History-Tabelle entfernt; hier verliert man also auch die früheren Werte dieser Spalte!

Aufräumarbeiten

Das Löschen historischer Datensatzzustände mit einem *DELETE* auf der History-Tabelle gelingt ohne weitere Vorkehrungen nicht (Fehler: „*Cannot delete rows from a temporal history table*“). Dafür muss man zunächst die Historisierung kurzzeitig deaktivieren und nach der Löschoperation wieder aktivieren:

```
-- Historische Daten löschen
ALTER TABLE [ITVisions].[MeasuringData]
  SET ( SYSTEM_VERSIONING = OFF )
DELETE FROM [ITVisions]
  .[MeasuringData_History] WITH (TABLOCKX)
WHERE ID = 10
ALTER TABLE [ITVisions].[MeasuringData] SET
  (SYSTEM_VERSIONING = ON (HISTORY_TABLE
  = [ITVisions].[MeasuringData_History]))
```

Auch für einige Schemaänderungen, zum Beispiel beim Einfügen einer *Identity*-Spalte, ist eine solche vorherige temporäre Deaktivierung der temporalen Funktionen notwendig. Ein *TRUNCATE TABLE* erfordert ebenfalls das Ausschalten der temporalen Funktionen.

Seit Microsoft SQL Server 2017 gibt es für temporale Tabellen sogenannte Aufbewahrungsrichtlinien (Retention Policies), die automatisches Löschen nach definierbaren Zeiträumen ermöglicht. Der folgende Befehl legt für die Datensätze der versionierten Tabelle *Kunde* eine Aufbewahrungsdauer von 14 Tagen fest:

```
ALTER TABLE [ITVisions].[MeasuringData] SET
  (SYSTEM_VERSIONING = ON
  (HISTORY_TABLE = [ITVisions].[MeasuringData_History],
    HISTORY_RETENTION_PERIOD = 14 DAYS));
GO
```

Mögliche Maßeinheiten bei der Angabe *HISTORY_RETENTION_PERIOD* sind *DAY*, *WEEK*, *MONTH* oder *YEAR*. Es ist aber nicht möglich, anstelle der Aufbewahrungszeit die Aufbewahrung auf eine bestimmte Anzahl von Versionen pro Datensatz zu beschränken.

Festplattenverbrauch

Spannend ist auch die Frage, ob temporale Tabellen hinsichtlich des Speicherplatzverbrauchs optimiert sind. Wenn man den Werten, die SQL Server selbst liefert, Glauben schenken will, dann gibt es keine Optimierung in dem Sinne, dass das Datenbankmanagementsystem nur die geänderten Werte speichert.

Offenbar wird der ganze Datensatz abgelegt. Bei temporalen Tabellen sollte man grundsätzlich bedenken, dass große Feldtypen wie *(n)varchar(max)*, *varbinary(max)*, *(n)text* und *image* viel Festplattenplatz verbrauchen, jeweils pro Version in der Historie (vergleiche die Warnung von Microsoft in [3]).

Bild 4 zeigt den Vergleich zwischen zwei möglichen Implementierungen der Speicherung einer Reihe von sich ändernden Messdaten mit Zeitpunkt und Werten (siehe Tabellenschema in Listing 1):

- In Variante 1 wurde für 10 000 Messdaten mit 10 000 Änderungen an diesen Daten eine temporale Tabelle verwendet. Beide Tabellen zusammen verbrauchten 1294 KB für die insgesamt 20 000 Datensätze.
- In Variante 2 wurde die 10 000 Messdaten und die 10 000 Änderungen in einer normalen Tabelle abgelegt (mit verschiedenen Primärschlüsseln) und die Historisierung auf Anwendungsebene selbst gelöst. Das verbrauchte nur 1040 KB für die insgesamt 20 000 Datensätze.

Der Grund, warum die History-Tabelle weniger Platz braucht als die aktuelle Tabelle mit gleich vielen Datensätzen, liegt in der automatisch bei der History-Tabelle aktivierten Page-Kompression [4].

Dass die normale Tabelle aber weniger Platz braucht als die Summe der beiden Tabellen in der temporalen Lösung, lässt sich dadurch erklären, dass in diesem Messdatenszenario in

```
31  DECLARE @str VARCHAR(500)
32  SET @str = 'exec sp_spaceused ''?'''
33  EXEC sp_msforeachtable @command1=@str
```

100 %

Results | Messages

	name	rows	reserved	data	index_size	unused
1	[ITVisions].[MeasuringData]	10000	712 KB	680 KB	16 KB	16 KB

	name	rows	reserved	data	index_size	unused
1	[ITVisions].[MeasuringData_History]	10000	584 KB	512 KB	16 KB	56 KB

	name	rows	reserved	data	index_size	unused
1	[dbo].[MeasuringDataWithOutHistorySet]	20000	1096 KB	1040 KB	16 KB	40 KB

Ausgabe von Listing 2 für eine überschaubare Menge von zehn Datensätzen (Bild 4)

der normalen Tabelle keine Spalten für Startzeitpunkt und Endzeitpunkt gebraucht werden, denn es gibt sowieso tatsächlich eine *DateTime*-Spalte für den Zeitpunkt der Messdatengültigkeit. In anderen Szenarien kann es natürlich erforderlich sein, dass man hier eine oder zwei zusätzliche Datumsspalten verwalten muss.

Fazit

Temporale Tabellen sind sehr komfortabel, denn der Entwickler muss beim Ändern oder Löschen eines Datensatzes nichts explizit tun, um den vorherigen Zustand zu erhalten. Früher hat man dies auf Anwendungsebene oder in der Datenbank mit Triggern aufwendig implementieren müssen.

Die Entity-Framework Core-Unterstützung für temporale Tabellen in Version 6.0 ist sehr hilfreich. In älteren Versionen des OR-Mappers kann man auf eines von drei GitHub-Projekten ausweichen (siehe [5], [6] und [7]). ■

[1] *SQL-Standard 2011, www.dotnetpro.de/SL2201DataAccess1*

[2] *Microsoft Docs, Creating a system-versioned temporal table, www.dotnetpro.de/SL2201DataAccess2*

[3] *Microsoft Docs, Temporal Table Considerations and Limitations, www.dotnetpro.de/SL2201DataAccess7*

[4] *Microsoft Docs, Data compression, www.dotnetpro.de/SL2201DataAccess3*

[5] *EntityFrameworkCore.TemporalTables, www.dotnetpro.de/SL2201DataAccess4*

[6] *EfCoreTemporalTable, www.dotnetpro.de/SL2201DataAccess5*

[7] *EFCore.TimeTraveler, www.dotnetpro.de/SL2201DataAccess6*

Dr. Holger Schwichtenberg
gehört zu den bekanntesten .NET- und Web-Experten in Deutschland. Er ist Chief Technology Expert bei der MAXIMAGO-Softwareentwicklung. Mit dem 38-köpfigen Expertenteam bei www.IT-Visions.de bietet er zudem Beratung und Schulungen für andere Unternehmen an.
www.dotnet-doktor.de

dnpCode A2201DataAccess