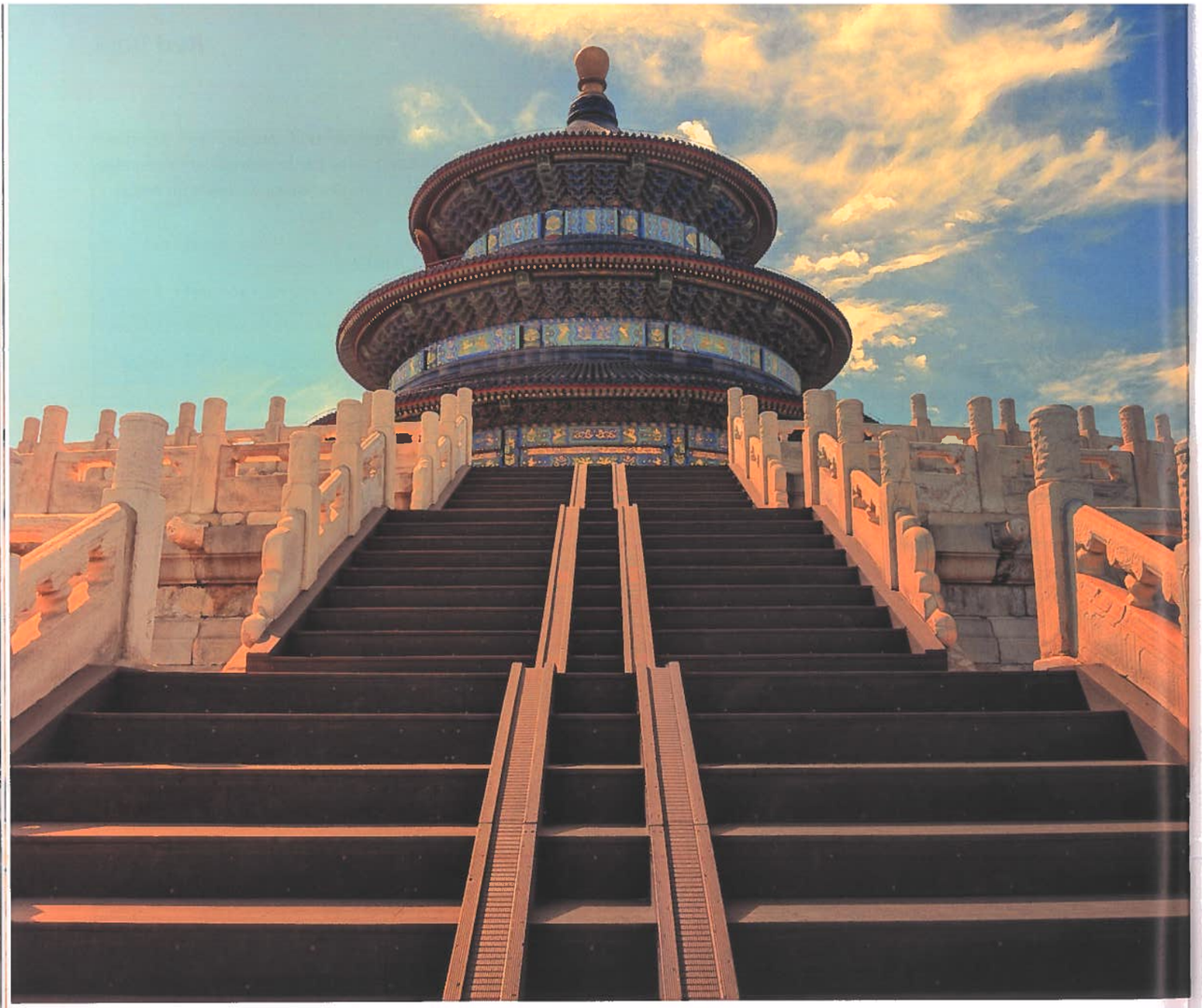

# Daten im Wandel der Zeit: Aufbau eines Historisierungs-Frameworks

Wolf G. Beckmann, TEAM GmbH

In der schnelllebigen Welt der Daten ist es entscheidend, nicht nur den aktuellen Zustand zu erfassen, sondern auch seine Entwicklung im Laufe der Zeit zu verstehen. Daten können sich kontinuierlich ändern und oft ist es unerlässlich, auf frühere Datenkonstellationen zurückzugreifen. Die Datenhistorisierung ermöglicht es uns, vergangene Zustände jederzeit wiederherzustellen und so wertvolle Erkenntnisse zu gewinnen. In diesem Artikel werde ich die Bedeutung dieser Herangehensweise genauer untersuchen und Methoden aufzeigen, wie Organisationen ihre Datenbestände erfolgreich historisieren können.





## Arten der Historisierung

Am Anfang der Historisierung sollte man sich erst einmal die Frage stellen, was konkret das Ziel der Historisierung ist. Geht es um die Protokollierung aller Änderungen, um die Reproduzierbarkeit von Datenzuständen und Konstellationen oder um eine organisierte Historisierung in Revisionen beziehungsweise Revisionierung?

Abhängig vom Ziel der Datenhistorisierung ist die Vorgehensweise unterschiedlich. Wenn es um die Protokollierung von Änderungen und die Reproduzierbarkeit von Datenzuständen geht, empfiehlt es sich, mit Triggern zu arbeiten. Diese erfassen automatisch jede Änderung und speichern den vorherigen Zustand ab. Dadurch wird die Rekonstruktion vergangener Zustände oder zumindest die Auflistung der Änderungen ermöglicht.

Bei einer Revisionierung hingegen wird die Historisierung gezielt durch den Benutzer aufgerufen. Dies geschieht in dem Moment, in dem eine bestimmte Revision erreicht werden soll. Hierbei werden nur bestimmte Zustände historisiert, die über mehrere Transaktionen hinweg entstanden sind.

Der Fokus dieses Artikels liegt auf der Möglichkeit, einen beliebigen Datenbestand zu reproduzieren, und das sowohl über beliebige Tabellen als auch über ganze Tabellenstrukturen hinweg.

In diesem Artikel möchte ich Schritt für Schritt anhand eines einfachen Beispiels die Konzepte hinter der automatisierten Historisierung erklären, die jeden Zustand restaurieren lässt.

Oracle bietet grundsätzlich ein Feature dafür an, das sich Flashback Query Archive nennt. Allerdings ist dieses Feature nicht in der Autonomous Database integriert. Deshalb stelle ich dar, wie grundsätzlich ein Historien-Framework aufgebaut werden kann. Das Konzept kann auch auf andere Datenbanken übertragen werden.

## Aufbau von Historien-Daten

Um dies zu veranschaulichen, nehmen wir eine typische Tabelle mit den dazugehörigen Standard-Trigger wie in *Listing 1* dargestellt.

Wenn man Tabellen historisieren möchte, benötigt man in erster Linie zwei Informationen: Den Zeitpunkt, ab dem eine Datenzeile gültig ist, und den Zeitpunkt, ab dem sie nicht mehr gültig ist. Ein weiterer Aspekt ist, dass in einer Tabelle, die historische Informationen aufnimmt, der Primärschlüssel erweitert werden muss, da er nur noch in Kombination mit dem Gültigkeitszeitraum eindeutig ist. Alternativ kann der ehemalige Primärschlüssel entfernt oder zu einem Index umgewandelt werden, vorausgesetzt, die Daten ändern sich nicht allzu häufig.

Bei der Historisierung speichere ich typischerweise auch, welche konkrete Transaktion die Zeile geändert hat, also ob sie hinzugefügt, geändert oder gelöscht wurde.

Als Zusatzinformation kann ich auch noch speichern, wer die Aktion durch-

```
CREATE SEQUENCE hd_customers_seq;

CREATE TABLE hd_customers (
    hd_customer_id NUMBER DEFAULT ON NULL
                   hd_customers_seq.NEXTVAL
                   CONSTRAINT hd_customers_id_pk PRIMARY KEY,
    customer_name VARCHAR2(100 CHAR),
    contact_number VARCHAR2(15 CHAR),
    created DATE NOT NULL,
    created_by VARCHAR2(255 CHAR) NOT NULL,
    updated DATE NOT NULL,
    updated_by VARCHAR2(255 CHAR) NOT NULL
);

CREATE OR REPLACE TRIGGER hd_customers_biu
BEFORE INSERT OR UPDATE ON hd_customers
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        :new.created := SYSDATE;
        :new.created_by :=
            COALESCE(SYS_CONTEXT('APEX$SESSION','APP_USER')
                     ,USER);
    END IF;
    :new.updated := SYSDATE;
    :new.updated_by :=
            COALESCE(SYS_CONTEXT('APEX$SESSION','APP_USER')
                     ,USER);
END hd_customers_biu;
```

*Listing 1: Erstellen einer Demo-Tabelle*

```
CREATE TABLE hd_customers_hist (
    hd_customer_id NUMBER,
    customer_name VARCHAR2(100 CHAR),
    contact_number VARCHAR2(15 CHAR),
    created DATE,
    created_by VARCHAR2(255 CHAR),
    updated DATE,
    updated_by VARCHAR2(255 CHAR),
    hist_trans VARCHAR2(1 CHAR),
    valid_from TIMESTAMP,
    valid_until TIMESTAMP,
    invalidated_by VARCHAR2(255 CHAR),
    invalidation_trans VARCHAR2(255 CHAR)
);
```

*Listing 2: Erstellen einer Tabelle für die Historisierung mit Gültigkeitszeitraum*

```
CREATE OR REPLACE TRIGGER HD_CUSTOMERS_BUD_HIST
BEFORE UPDATE OR DELETE ON HD_CUSTOMERS
FOR EACH ROW
DECLARE
   v_hist_trans VARCHAR2(1);
BEGIN
   IF UPDATING THEN
      v_hist_trans := 'U';
   END IF;

   IF DELETING THEN
      v_hist_trans := 'D';
   END IF;

   INSERT INTO HD_CUSTOMERS_HIST
      (hd_customer_id, customer_name, contact_number, created,
       created_by, updated, updated_by, hist_trans, valid_from,
       valid_until, invalidated_by, invalidation_trans)
   VALUES
      (:old.hd_Customer_ID, :old.Customer_Name, :old.Contact_Number,
       :old.created, :old.created_by, :old.updated, :old.updated_by,
       v_hist_trans,
       NVL(:old.updated, :old.created),
       SYSDATE,
       NVL(SYS_CONTEXT('APEX$SESSION', 'APP_USER'), USER),
       v('APP_PAGE_ID') || ' (' || v('APP_PAGE_ALIAS') || ')');
END;
/
```

*Listing 3: Erstellen eines Triggers für eine Historisierungstabelle mit Gültigkeitszeitraum*

```
CREATE OR REPLACE PACKAGE TEAM_HIST AS
  PROCEDURE SET_HIST_DATE(p_date IN TIMESTAMP);
  FUNCTION GET_HIST_DATE RETURN TIMESTAMP;
END TEAM_HIST;
/
```

*Listing 4: Erstellen eines Package-Headers zum Setzen und Lesen eines Gültigkeitsdatums*

```
CREATE OR REPLACE PACKAGE BODY TEAM_HIST AS
  g_hist_date TIMESTAMP;

  PROCEDURE SET_HIST_DATE(p_date IN TIMESTAMP) IS
  BEGIN
    g_hist_date := p_date;
  END SET_HIST_DATE;

  FUNCTION GET_HIST_DATE RETURN TIMESTAMP IS
  BEGIN
    RETURN coalesce(g_hist_date,sysdate+1);
  END GET_HIST_DATE;

END TEAM_HIST;
/
```

*Listing 5: Erstellen eines Package-Bodies zum Setzen und Lesen eines Gültigkeitsdatums*

geführt hat und in welchem Rahmen die Transaktion stattgefunden hat. Um zu verhindern, dass die Applikation beim Arbeiten mit den aktuellen Daten beeinflusst wird, halte ich diese historischen Daten üblicherweise in einer eigenen Tabelle.

Somit würde die „Customers"-Tabelle für die historischen Daten wie in *Listing 2* aussehen.

Des Weiteren benötige ich einen Trigger, der genau diese historische Tabelle befüllt. Es ist anzumerken, dass der Trigger nur für Updates und Deletes zuständig ist, da ich den Create-Fall ausschließlich in den aktuellen Daten verwalte. Darauf werde ich später näher eingehen. In diesem Fall möchte ich das Framework im APEX-Umfeld einsetzen. Daher nehme ich als invalidation_trans die APEX-Applikation und -Seite *(siehe Listing 3)*.

Um die Daten aus der Tabelle unter Berücksichtigung eines bestimmten Zeitpunkts abzufragen, benötige ich einen View. Dieser View soll extern gesteuert werden können, um den Zeitpunkt festzulegen, zu dem die Daten sichtbar sein sollen.

Um das Testen zu vereinfachen, erstelle ich ein kleines Package, in dem ich eine Variable für das Datum setzen und auslesen kann *(siehe Listing 4 und 5)*.

Der View sieht wie in *Listing 6* aus.

Wenn ich jetzt Beispieldaten einfüge *(siehe Listing 7)* liefert der Customers View folgende Daten zurück *(siehe Listing 8)*.

Dann ändere ich bei Kunde A die Daten *(siehe Listing 9)* und lösche ihn nach einer kurzen Wartezeit gänzlich, wie in *Listing 10* dargestellt.

Jetzt liefert der Customers View folgende Daten zurück *(siehe Listing 11)*.

Setze ich aber das Abfrage-Datum auf „19.07.2023 13:44:24", ändert sich das Ergebnis, wie in *Listing 12* dargestellt.

Zu dem Zeitpunkt gab es noch nicht den Kunden C und A hatte noch die Nummer 123456789.

Das VALID_UNTIL-Datum und die HIST_TRANS haben sich geändert. Diese geben an, dass der Datensatz nicht mehr aktuell ist, sondern durch ein Update invalidiert wurde.

So sieht das Ergebnis kurz vor dem Delete *(siehe Listing 13)* aus.

Ich sehe, dass die CONTACT_NUMBER 111111111 ist und dieser Satz wiederum durch ein Löschen invalidiert wurde.

```
CREATE OR REPLACE VIEW V_HD_CUSTOMERS_HIST AS
SELECT *
FROM (
  SELECT t.*,
    'A' AS hist_trans,
    NVL(t.updated, t.created) AS valid_from,
    to_date('01.01.4000','dd.mm.yyyy') AS valid_until,
    NULL AS invalidated_by,
    NULL AS invalidation_trans
  FROM HD_CUSTOMERS t
  UNION ALL
  SELECT *
  FROM HD_CUSTOMERS_HIST
)
WHERE TEAM_HIST.GET_HIST_DATE >= valid_from
  AND TEAM_HIST.GET_HIST_DATE < valid_until;
```

*Listing 6: Erstellen eines Views um historische Daten anzuzeigen*

```
INSERT INTO HD_CUSTOMERS (CUSTOMER_NAME, CONTACT_NUMBER)
VALUES ('Kunde A', '123456789');

INSERT INTO HD_CUSTOMERS (CUSTOMER_NAME, CONTACT_NUMBER)
VALUES ('Kunde B', '987654321');

INSERT INTO HD_CUSTOMERS (CUSTOMER_NAME, CONTACT_NUMBER)
VALUES ('Kunde C', '555555555');
```

*Listing 7: Beispieldaten*

```
CUSTOMER_NAME CONTACT_NUMBER  HIST_TRANS VALID_FROM          VALID_UNTIL
------------- --------------- ---------- ------------------- ----------
Kunde A       123456789       A          19.07.2023 13:44:21 01.01.4000
Kunde B       987654321       A          19.07.2023 13:44:24 01.01.4000
Kunde C       555555555       A          19.07.2023 13:44:28 01.01.4000
```

*Listing 8: Ausgabe der Beispieldaten*

```
UPDATE HD_CUSTOMERS
SET CONTACT_NUMBER = '111111111'
WHERE CUSTOMER_NAME = 'Kunde A';
```

*Listing 9: Ändern der Beispieldaten*

```
DELETE FROM HD_CUSTOMERS
WHERE CUSTOMER_NAME = 'Kunde A';
```

*Listing 10: Ändern der Beispieldaten*

```
CUSTOMER_NAME   CONTACT_NUMBER  HIST_TRANS VALID_FROM          VALID_UNTI
--------------- --------------- ---------- ------------------- ----------
Kunde B         987654321       A          19.07.2023 13:44:24 01.01.4000
Kunde C         555555555       A          19.07.2023 13:44:28 01.01.4000
```

*Listing 11: Ausgabe der Beispieldaten*

```
/* Der nächste Befehl muss in einer Zeile stehen!
   Er passte hier nur nicht ein eine Zeile! */
exec team_hist.set_hist_date(
    to_date('19.07.2023 13:44:24','dd.mm.yyyy hh24:mi:ss')
);

CUSTOMER_NAME   CONTACT_NUMBER  HIST_TRANS VALID_FROM          VALID_UNTI
--------------- --------------- ---------- ------------------- ----------
Kunde A         123456789       U          19.07.2023 13:44:21 19.07.2023
Kunde B         987654321       A          19.07.2023 13:44:24 01.01.4000
```

*Listing 12: Ausgabe der Beispieldaten*

```
exec team_hist.set_hist_date(
    to_date('19.07.2023 14:07:36','dd.mm.yyyy hh24:mi:ss')
);

CUSTOMER_NAME   CONTACT_NUMBER  HIST_TRANS VALID_FROM          VALID_UNTI
--------------- --------------- ---------- ------------------- ----------
Kunde A         111111111       D          19.07.2023 14:07:36 19.07.2023
Kunde B         987654321       A          19.07.2023 13:44:24 01.01.4000
Kunde C         555555555       A          19.07.2023 13:44:28 01.01.4000
```

*Listing 13: Ausgabe der Beispieldaten*

```
CREATE SEQUENCE hd_transaction_seq;

CREATE TABLE hd_transaction (
    transaction_num    NUMBER,
    transaction_trans  VARCHAR2(255),
    created            DATE NOT NULL,
    created_by         VARCHAR2(255) NOT NULL,
    updated            DATE NOT NULL,
    updated_by         VARCHAR2(255) NOT NULL
);

CREATE OR REPLACE TRIGGER hd_transaction_biu
    BEFORE INSERT OR UPDATE
    ON hd_transaction
    FOR EACH ROW
BEGIN
    IF INSERTING THEN
        :new.created := sysdate;
        :new.created_by :=
              COALESCE(sys_context('APEX$SESSION','APP_USER'),
                       user);
    END IF;
    :new.updated := sysdate;
    :new.updated_by :=
              COALESCE(sys_context('APEX$SESSION','APP_USER'),
                       user);
END;
```

*Listing 14: Erstellen einer Transaktionstabelle*

```
ALTER TABLE HD_CUSTOMERS ADD transaction_num NUMBER;

DROP TABLE HD_CUSTOMERS_HIST;

CREATE TABLE HD_CUSTOMERS_HIST AS
SELECT t.*,
       CAST(NULL AS VARCHAR2(1))   AS hist_trans,
       CAST(NULL AS TIMESTAMP)     AS valid_from,
       CAST(NULL AS TIMESTAMP)     AS valid_until,
       CAST(NULL AS VARCHAR2(255)) AS invalidated_by,
       CAST(NULL AS VARCHAR2(255)) AS invalidation_trans,
       CAST(NULL AS NUMBER)        AS until_transaction_num
FROM HD_CUSTOMERS t
WHERE 1=0;
```

*Listing 15: Erweitern der Tabellen um die Transaktion*

```
/* Globale Variablen */
g_transaction_id VARCHAR2(255);
g_transaction_num NUMBER;

FUNCTION GET_TRANSACTION_NUM RETURN NUMBER IS
BEGIN
  IF DBMS_TRANSACTION.local_transaction_id IS NULL OR
     NVL(g_transaction_id,'#*') !=
                        DBMS_TRANSACTION.local_transaction_id
  THEN
        g_transaction_id :=
                        DBMS_TRANSACTION.local_transaction_id;
        INSERT INTO hd_transaction (transaction_num,
                                    transaction_trans)
        VALUES (hd_transaction_seq.NEXTVAL,
                v('APP_PAGE_ID')
                || ' (' || v('APP_PAGE_ALIAS') || ')')
        RETURNING transaction_num INTO g_transaction_num;
  END IF;
  RETURN g_transaction_num;
END;
```

*Listing 16: Funktion zum Ermitteln der Transaktionsnummer*

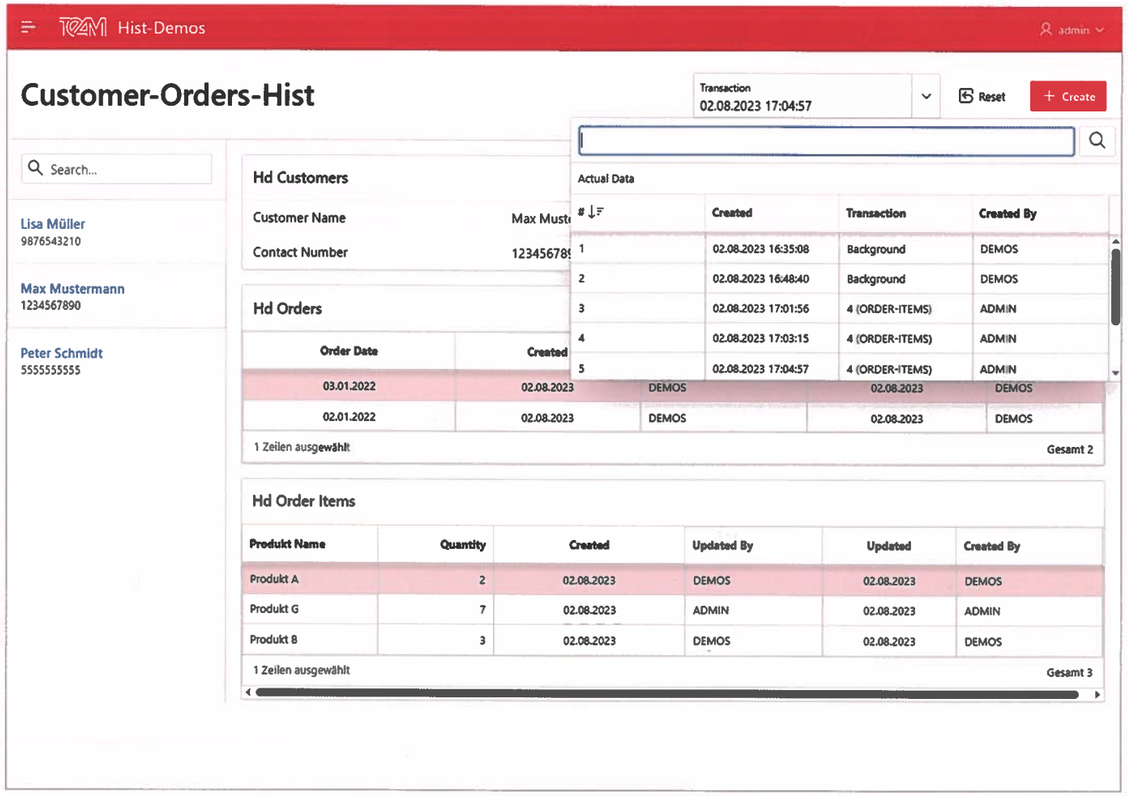


*Abbildung 1: APEX-Demo mit Transaktionsauswahl (Quelle: TEAM GmbH)*



Es sieht gut aus, aber was passiert, wenn eine Transaktion über mehrere Zeilen oder Tabellen hinweg etwas länger dauert?

Beispielsweise verändern wir in einer Tabellenstruktur eine neue Bestellung (eine Tabelle ORDERS) und erst in der nächsten Sekunde wird eine Position (eine Tabelle ORDER_ITEMS) gelöscht, jedoch wird beides gemeinsam committed.

Nutze ich dann das VALID_FROM-Datum der Bestellung und setze es als Hist-Datum, würde die Position immer noch angezeigt werden.

Tatsächlich handelt es sich um ein komplexes Problem, da Oracle das Datum, an dem das Commit stattgefunden hat, nicht ohne weiteres preisgibt. Die eigentliche Frage ist, was benötigt wird. In der Praxis geht es selten um den sekundengenauen Commit-Zeitpunkt, sondern eher um konsistente Daten und dabei kann uns Oracle relativ einfach helfen. Ich kann herausfinden, welche Daten in einer Transakti-

```
CREATE OR REPLACE TRIGGER HD_CUSTOMERS_BIUD_HIST
BEFORE INSERT OR UPDATE OR DELETE ON HD_CUSTOMERS
FOR EACH ROW
DECLARE
   v_hist_trans VARCHAR2(1);
BEGIN

   IF INSERTING OR UPDATING THEN
      :NEW.transaction_num := team_hist.GET_TRANSACTION_NUM;
   END IF;

   IF UPDATING THEN
      v_hist_trans := 'U';
   END IF;

   IF DELETING THEN
      v_hist_trans := 'D';
   END IF;

   IF NOT INSERTING THEN
       INSERT INTO HD_CUSTOMERS_hist
          (HD_CUSTOMER_ID, CUSTOMER_NAME, CONTACT_NUMBER,
           CREATED, CREATED_BY, UPDATED, UPDATED_BY,
           TRANSACTION_NUM,
           hist_trans, valid_from, valid_until,
           invalidated_by, invalidation_trans,
           until_transaction_num)
       VALUES
          (:old.HD_CUSTOMER_ID, :old.CUSTOMER_NAME,
           :old.CONTACT_NUMBER, :old.CREATED, :old.CREATED_BY,
           :old.UPDATED, :old.UPDATED_BY,
           :old.TRANSACTION_NUM,
           v_hist_trans,
           NVL(:old.updated, :old.created),
           SYSDATE,
           NVL(SYS_CONTEXT('APEX$SESSION', 'APP_USER'), USER),
           v('APP_PAGE_ID') || ' (' || v('APP_PAGE_ALIAS')
           || ')',
           team_hist.GET_TRANSACTION_NUM);
   END IF;
END;
```

*Listing 17: Trigger der Hist-Tabelle mit Transaktionsnummer*

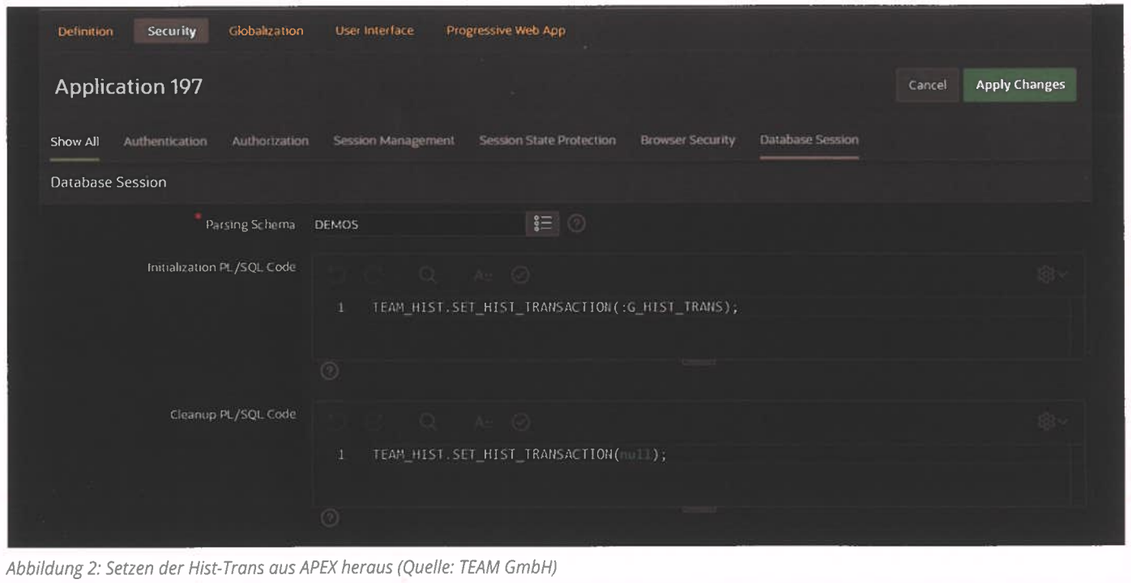


*Abbildung 2: Setzen der Hist-Trans aus APEX heraus (Quelle: TEAM GmbH)*

on gemeinsam bearbeitet wurden. Das funktioniert über die Funktion DBMS_TRANSACTION.local_transaction_id (in PostgeSQL wäre übrigens das Äquivalent die txid_current(), falls man das Konzept übertragen möchte). Wenn ich mich in einer Transaktion befinde, also wenn Daten in der aktuellen Session geändert wurden, wird hier eine ID gefunden, die so lange gleichbleibt, bis ein Commit erfolgt.

Um das für mich auszunutzen, gehe ich wie folgt vor:

1. Ich lege eine Sequence für Transaktionsnummern an.
2. Ich erweitere die eigentlichen Daten-Tabellen um eine Transaktionsnummer.
3. In meinem Package erstelle ich eine Funktion, die die aktuelle Transaktionsnummer liefert. Diese zieht nur eine neue Nummer, wenn sich die local_transaction_id geändert hat.
4. Bei jeder Änderung speichere ich die Transaktionsnummer, mit der die Daten erzeugt wurden und mit welcher sie invalidiert wurden.
5. Um eine Übersicht über alle Transaktionen zu bekommen, legen wir noch eine Transaktionstabelle an, in der die Transaktionsnummern und das Datum, zu dem sie erzeugt wurden, hinterlegt sind

```
/* Globale Variablen */
g_hist_transaction_num NUMBER;

PROCEDURE SET_HIST_DATE(p_date IN TIMESTAMP) IS
BEGIN
  g_hist_date := p_date;
  g_hist_transaction_num := NULL;
END SET_HIST_DATE;

PROCEDURE SET_HIST_TRANSACTION(p_transaction_num IN NUMBER) IS
BEGIN
   g_hist_date := NULL;
   g_hist_transaction_num := p_transaction_num;
END SET_HIST_TRANSACTION;

FUNCTION GET_HIST_TRANSACTION_NUM RETURN NUMBER IS
BEGIN
  RETURN g_hist_transaction_num;
END;
```

*Listing 18: Hist-Package um Abfrage der Transaktion erweitern*

```
CREATE OR REPLACE VIEW V_HD_CUSTOMERS_HIST AS
SELECT *
  FROM (
    SELECT t.*,
      'A' AS hist_trans,
      NVL(t.updated, t.created) AS valid_from,
      TO_DATE('01.01.4000','dd.mm.yyyy') AS valid_until,
      NULL AS invalidated_by,
      NULL AS invalidation_trans,
      1.0E125 AS until_transaction_num
    FROM HD_CUSTOMERS t
  UNION ALL
    SELECT *
      FROM HD_CUSTOMERS_HIST)
     WHERE (    TEAM_HIST.GET_HIST_TRANSACTION_NUM IS NULL
           AND TEAM_HIST.GET_HIST_DATE >= valid_from
           AND TEAM_HIST.GET_HIST_DATE < valid_until)
       OR  (    TEAM_HIST.GET_HIST_TRANSACTION_NUM IS NOT NULL
           AND TEAM_HIST.GET_HIST_TRANSACTION_NUM >=
                                          transaction_num
           AND TEAM_HIST.GET_HIST_TRANSACTION_NUM <
                                       until_transaction_num)
```

*Listing 19: View zum Abfragen historischer Werte mit Transaktionsnummer*

```
BEGIN
  -- Transaktion 1
  INSERT INTO HD_CUSTOMERS (CUSTOMER_NAME, CONTACT_NUMBER)
                    VALUES ('Kunde C', '555555555');
  COMMIT;
  DBMS_SESSION.SLEEP(5);

  -- Transaktion 2
  INSERT INTO HD_CUSTOMERS (CUSTOMER_NAME, CONTACT_NUMBER)
                    VALUES ('Kunde A', '123456789');
  DBMS_SESSION.SLEEP(5);
  INSERT INTO HD_CUSTOMERS (CUSTOMER_NAME, CONTACT_NUMBER)
                    VALUES ('Kunde B', '987654321');
  COMMIT;
END;
```

*Listing 20: Erstellung der Testdaten*

6. Abschließend wird der View zum Abfragen der historischen Daten um die Transaktionsnummer erweitert. So kann man nach Datum oder Transaktionsnummer abfragen.

Zuerst die Transaktionstabelle *(siehe Listing 14)*.

Dann die Customers-Tabellen erweitern/passend erstellen *(siehe Listing 15)*.

Dann erweitere ich das Package, um die Transaktionsnummer zu füllen *(siehe Listing 16)*.

Und ich passe den Trigger an, der jetzt auch bei einem Insert aktiv wird, um die TRANSACTION_NUM zu füllen *(siehe Listing 17)*.

Zum Abfragen erweitere/ändere ich erneut das Package *(siehe Listing 18)*.

Für die Abfragen gehe ich somit entweder über das Datum oder die Transaktionsnummer. Damit komme ich zum View *(siehe Listing 19)*.

Jetzt teste ich das Ganze mit folgendem Script *(siehe Listing 20)*.

Ich schaue mir die Customers an, bevor ich sie lösche *(siehe Listing 21)*.

Wenn ich jetzt den Zustand der Datenbank über das Datum setze, passiert Folgendes *(siehe Listing 22)*.

Tatsächlich war dieser Zustand in der Datenbank für niemanden sichtbar, da ich Kunde A und B in derselben Transaktion committet habe.

Über die Transaktionsnummer kann ich jetzt konsistente Zustände abrufen *(siehe Listing 23)*.

Über die Hist-Views kann ich jetzt komplette Datenstrukturen genauso abfragen, wie mit den Original-Tabellen. Der typische Weg ist einen Query mit den Tabellen zu erstellen und Aliase zu verwenden. Anschließend werden in dem Query die Tabellen durch die Views ersetzt.

Durch das Setzen der Hist-Transaktionsnummer oder des Hist-Datums im Package kann ich die Datenkonstellation zu einem beliebigen historischen Zeitpunkt wiederherstellen *(siehe Abbildung 1)*.

## Einbinden des Frameworks in APEX

In APEX besteht die Herausforderung darin, dass das Setzen einer Package Variable wenig Nutzen hat, wenn ich die Variable nicht innerhalb eines Requests setze

```
select customer_name,contact_number,transaction_num,
       to_char(valid_from,'dd.mm.yy hh24:mi:ss') valid_from
  from v_hd_customers_hist
  order by valid_from;

CUSTOMER_NAME CONTACT_NUMBE TRANSACTION_NUM VALID_FROM
------------- ------------- --------------- -----------------
Kunde C       555555555                  61 25.07.23 15:45:06
Kunde A       123456789                  62 25.07.23 15:45:11
Kunde B       987654321                  62 25.07.23 15:45:16

delete from hd_customers;
commit;

select …

Keine Zeilen ausgewählt
```

*Listing 21: Abfragen der historischen Stände*

```
exec team_hist.set_hist_date(
   to_date('25.07.2023 15:45:11','dd.mm.yyyy hh24:mi:ss')
);

select * from V_HD_CUSTOMERS_HIST

CUSTOMER_NAME CONTACT_NUMBE TRANSACTION_NUM VALID_FROM
------------- ------------- --------------- -----------------
Kunde C       555555555                  61 25.07.23 15:45:06
Kunde A       123456789                  62 25.07.23 15:45:11
```

*Listing 22: Abfragen mit Zeitstempel*

```
exec team_hist.set_hist_transaction(62);

select * from V_HD_CUSTOMERS_HIST

CUSTOMER_NAME CONTACT_NUMBE TRANSACTION_NUM VALID_FROM
------------- ------------- --------------- -----------------
Kunde C       555555555                  61 25.07.23 15:45:06
Kunde A       123456789                  62 25.07.23 15:45:11
Kunde B       987654321                  62 25.07.23 15:45:16
```

*Listing 23: Abfragen mit Transaktionsnummer*

und sofort verwende. Dies liegt daran, dass ORDS bei jedem Zugriff des Frontends auf die Datenbank eine beliebige Datenbankverbindung aus einem Pool auswählt.

Glücklicherweise bietet APEX einen Mechanismus, mit dem Sessions für einen Request vorbereitet und anschließend bereinigt werden können.

Das Vorgehen besteht daher darin, ein Application Item zu erstellen, in das die History-Transaktion eingetragen wird. Jeder Request wird dann vorbereitet und anschließend bereinigt.

Gelöst wird das ganze dadurch, dass die Hist-Transaktionsnummer oder das Hist-Datum in ein globales Item geschrieben wird und dieses bei der Session-Behandlung verwendet wird.

Die Konfiguration für die Session-Behandlung findet man unter Shared Components > Security > Security Attributes *(siehe Abbildung 2)*.

Damit man nicht alles selbst abtippen muss, stelle ich das Package mit der Demo-Applikation über folgendes Git-Repository zur Verfügung: *https://github.com/Team-wb/TEAM_HIST.*

Das Erstellen der History-Views und Trigger ist je nach Größe des Datenmodells aufwendig, kann aber hervorragend durch dynamisches SQL automatisiert werden. Ich habe die TEAM_HIST Package daher um Prozeduren erweitert, die automatisch die Hist-Tabelle, -Trigger und -Views erstellen.

## Transferleistung

Das Konzept der globalen Variable, die Views unterschiedlicher Daten zurückliefern lässt, ist grundlegend und kann für verschiedene Zwecke verwendet werden. Ich habe mit diesem Konzept beispielsweise eine VPD-Lösung (Virtual Privat Database) „für Arme" entwickelt, um eine Mandantenfähigkeit umzusetzen.

## Über den Autor

Wolf G. Beckmann ist Bereichsleiter des Software- und Consulting-Teams und Consultant bei der TEAM GmbH sowie leidenschaftlicher Entwickler.

Wolf G. Beckmann
wb@team-pb.de