**Auditing Who Changed Temporal Table Data in SQL Server**

By: [Aaron Bertrand](https://www.mssqltips.com/sqlserverauthor/49/aaron-bertrand/)

FROM: <https://www.mssqltips.com/sqlservertip/4722/auditing-who-changed-temporal-table-data-in-sql-server/>

**Problem**

Temporal tables, introduced in SQL Server 2016, provide an easy way to track the history of data as it changes over time (also called "system versioning"). You can essentially use a "time machine" to see exactly what a table or row looked like at a specific point in time, or over a range. One piece of data that isn't tracked, however, is *who made the change*. It can be great to see that Jaime's salary doubled yesterday, but did Jaime update the table, or is she being sabotaged?

**Solution**

There are already plenty of existing ways to track which users have changed data, such as SQL Server Audit, custom application logging, and triggers. Several people who would like to use temporal tables want to know if there are ways to extend the feature to also track who changed the data, so that they don't have to implement two different features or an inordinate amount of additional development. There are at least a couple of ways to do this. One involves changing the base table and its history table to include an additional "modified by" column, and the other involves creating a separate table to link each history row to the modifier. Both assume that the application code cannot be changed, which is often the case, especially when retrofitting temporal tables to an existing application. If you can control all data access, then you could simply change the application to write the modifier to the base table directly. If you can do that, great! If you can't, here are your options...

Option 1 : Change the tables

Let's start with a very simple pair of tables that have already been set up to use the new temporal tables feature:

CREATE TABLE dbo.EmployeeHistory

(

EmployeeID int NOT NULL,

FirstName sysname,

ValidFrom datetime2 NOT NULL,

ValidTo datetime2 NOT NULL

);

GO

CREATE TABLE dbo.Employees

(

EmployeeID int CONSTRAINT PK\_Employees PRIMARY KEY,

FirstName sysname,

ValidFrom datetime2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,

ValidTo datetime2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,

PERIOD FOR SYSTEM\_TIME (ValidFrom, ValidTo)

)

WITH (SYSTEM\_VERSIONING = ON (HISTORY\_TABLE = dbo.EmployeeHistory));

Now, we can insert a few rows to get some existing history data:

INSERT dbo.Employees(EmployeeID, FirstName)

VALUES(1,N'Frank'),(2,N'Luke'),(3,N'Samantha');

UPDATE dbo.Employees SET FirstName = N'Franky' WHERE EmployeeID = 1;

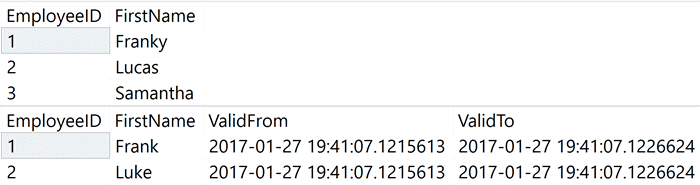
UPDATE dbo.Employees SET FirstName = N'Lucas' WHERE EmployeeID = 2;

Now, we can look at the tables to see exactly what we have right now:

SELECT \* FROM dbo.Employees;

SELECT \* FROM dbo.EmployeeHistory;

Your dates/times will vary, obviously, but you should see three rows in the base table, and two rows in history representing the updates:



In this case, in order to capture who made each change going forward, we need to do a number of things:

* Turn off system versioning (can't modify base or history tables while it's on)
* Add a ModifyingUser column to both the base table and the history table
* Update existing rows to show ? or N/A (we can't fix those retroactively, unless we have existing logging we can rely on)
* Make the column NOT NULL and add a default
* Turn system versioning back on
* Create a trigger to record the modifying user for future updates

BEGIN TRANSACTION;

GO

ALTER TABLE dbo.Employees SET (SYSTEM\_VERSIONING = OFF);

-- add the column to the base table

ALTER TABLE dbo.Employees ADD ModifyingUser nvarchar(128);

GO

-- update existing rows

UPDATE dbo.Employees SET ModifyingUser = N'?'; -- or N/A or <unknown>

-- then make it not NULLable

ALTER TABLE dbo.Employees ALTER COLUMN ModifyingUser nvarchar(128) NOT NULL;

-- then add a default constraint for new rows

ALTER TABLE dbo.Employees ADD

CONSTRAINT df\_Employees\_ModifyingUser DEFAULT

(COALESCE(SUSER\_NAME(SUSER\_ID()),USER\_NAME()))

FOR ModifyingUser;

GO

-- change the history table too

-- (less scaffolding here because we don't need a default for future rows)

ALTER TABLE dbo.EmployeeHistory

ADD ModifyingUser nvarchar(128) NOT NULL

CONSTRAINT df\_EmployeeHistory\_ModifyingUser DEFAULT (N'?');

ALTER TABLE dbo.Employees SET

(

SYSTEM\_VERSIONING = ON

(

HISTORY\_TABLE = dbo.EmployeeHistory,

DATA\_CONSISTENCY\_CHECK = ON

)

);

GO

COMMIT TRANSACTION;

Now, we need to create the trigger for updates to the base table. We can't add a trigger to the history table, but we can add it to the base table:

CREATE TRIGGER dbo.Employee\_AuditModifier

ON dbo.Employees

AFTER UPDATE

AS

BEGIN

-- note this assumes EmployeeID PK is immutable

UPDATE e

SET e.ModifyingUser = COALESCE(SUSER\_NAME(SUSER\_ID()),USER\_NAME())

FROM dbo.Employees AS e

INNER JOIN inserted AS i

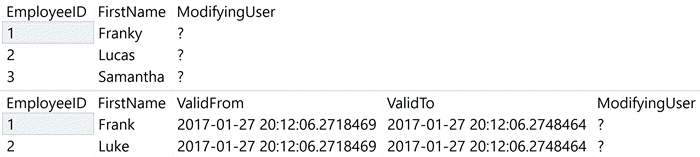
ON e.EmployeeID = i.EmployeeID;

END

GO

COMMIT TRANSACTION;

Before we do anything, let's take a look at the base and history tables now:



Next, to validate that both tables maintain who created or modified a row, let's add a login and a user:

CREATE LOGIN audited\_login WITH PASSWORD = 'x', CHECK\_POLICY = OFF;

CREATE USER audited\_login FROM LOGIN audited\_login;

CREATE USER audited\_user WITHOUT LOGIN;

GRANT SELECT, INSERT, UPDATE ON dbo.Employees

TO audited\_login, audited\_user;

Then we can perform some activities against the base table as these two principals (let's have the login insert a row and update another, and have the user update an existing row and the new row just added by the login):

EXEC

(

'INSERT dbo.Employees(EmployeeID, FirstName)

VALUES(4, N''Julio'');

UPDATE dbo.Employees SET FirstName = N''Francis''

WHERE EmployeeID = 1;'

) AS LOGIN = N'audited\_login';

EXEC

(

'UPDATE dbo.Employees SET FirstName = N''Lukey''

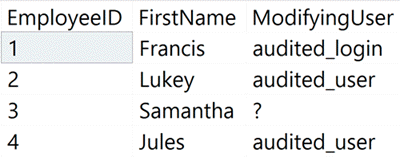
WHERE EmployeeID = 2;

UPDATE dbo.Employees SET FirstName = N''Jules''

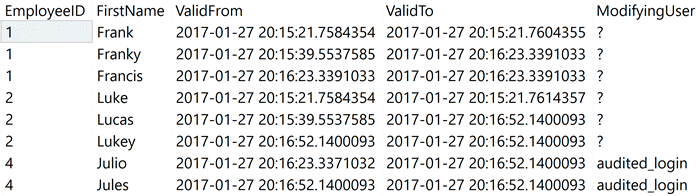
WHERE EmployeeID = 4;'

) AS USER = N'audited\_user';

Then take a look at the data after these changes. The base table looks right (three of the rows indicate the principal who last touched them, which Samantha still shows unknown):



However, the history table has additional rows - for every row we updated, the trigger wrote an extra row, so we have both a before and after picture of the modified row in the history table. We end up with 8 rows in the history table, when we should have only 5 (we started with 2, then inserted one row and updated 3, which should lead to 3 new history rows for the old versions of the updates).



If we look closer, we can see that the rows where ValidFrom and ValidTo are equal represent the data in some transitional state - they contain the *previous* ModifyingUser value, but the *new* FirstName value (which, logically, shouldn't show up in the history table yet). This means the history table will grow faster than we expected - up to twice as fast, depending on the DML mix. Obviously, we can discard those rows that aren't meaningful. We can identify those rows with the following query:

;WITH x AS

(

SELECT EmployeeID, FirstName, ModifyingUser, ValidFrom, ValidTo,

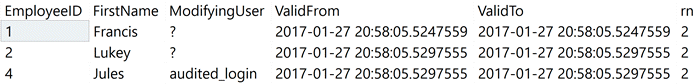
rn = ROW\_NUMBER() OVER (PARTITION BY EmployeeID, ValidTo ORDER BY ValidFrom)

FROM dbo.EmployeeHistory

)

SELECT \* FROM x WHERE rn > 1 AND ValidFrom = ValidTo;

The "bad" rows:



Now that you can easily identify which rows are disposable, you can perform periodic deletes to keep the history table growth under control (see [this tip](https://www.mssqltips.com/sqlservertip/4674/managing-temporal-table-history-in-sql-server-2016/) for ideas about that, as well as potential caveats). You could run something like this as frequently as desired to clean up those additional rows:

-- turn versioning off:

ALTER TABLE dbo.Employees SET (SYSTEM\_VERSIONING = OFF);

GO

-- remove redundant rows:

;WITH x AS

(

SELECT EmployeeID, FirstName, ModifyingUser, ValidFrom, ValidTo,

rn = ROW\_NUMBER() OVER (PARTITION BY EmployeeID, ValidTo ORDER BY ValidFrom)

FROM dbo.EmployeeHistory

)

DELETE x WHERE rn > 1 AND ValidFrom = ValidTo;

GO

-- turn versioning back on:

ALTER TABLE dbo.Employees SET

(

SYSTEM\_VERSIONING = ON

(

HISTORY\_TABLE = dbo.EmployeeHistory,

DATA\_CONSISTENCY\_CHECK = ON

)

);

Let's try one more update just to make sure we are capturing things correctly:

EXEC

(

'UPDATE dbo.Employees SET FirstName = N''Francisco''

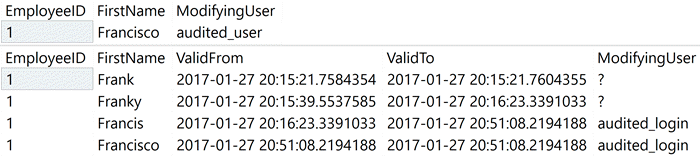
WHERE EmployeeID = 1;'

) AS USER = N'audited\_user';

SELECT \* FROM dbo.Employees WHERE EmployeeID = 1;

SELECT \* FROM dbo.EmployeeHistory WHERE EmployeeID = 1;

The output:



We can see that the base table reflects the most recent modifier, and the history table has 4 rows - 2 from before our auditing was implemented, the third row showing the data and modifier from just before this most recent change, and the fourth row will be cleaned up the next time our maintenance script runs.

*As an aside, I thought this would be a clever use for an INSTEAD OF trigger, but* [*that loophole was closed before RTM*](https://connect.microsoft.com/SQLServer/feedback/details/2769923/can-easily-work-around-instead-of-trigger-restriction-on-temporal-tables)*.*

Option 2 : Add a separate table

The first option was pretty complicated, and assumes you have the luxury of modifying the base tables and suspending system versioning occasionally. Either of those things might not be possible in all environments.

Another option would be to record the users responsible for modifying data in a separate table. This sounds a lot like a completely independent solution (rolling your own logging with triggers), but there will be a way to tie to the history table so that temporal queries can accurately reflect the modifier for each version of a row.

Let's start again with a simple system versioned table:

CREATE TABLE dbo.InvoiceHistory

(

InvoiceID int NOT NULL,

Amount decimal(8,2) NOT NULL,

ValidFrom datetime2 NOT NULL,

ValidTo datetime2 NOT NULL

);

CREATE TABLE dbo.Invoices

(

InvoiceID int CONSTRAINT PK\_Invoices PRIMARY KEY,

Amount decimal(8,2) NOT NULL,

ValidFrom datetime2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,

ValidTo datetime2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,

PERIOD FOR SYSTEM\_TIME (ValidFrom, ValidTo)

)

WITH (SYSTEM\_VERSIONING = ON (HISTORY\_TABLE = dbo.InvoiceHistory));

Then we can create a separate table to tie users to specific invoice changes:

CREATE TABLE dbo.InvoiceAuditing

(

InvoiceID int NOT NULL,

ValidFrom datetime2 NOT NULL,

ValidTo datetime2 NOT NULL,

ModifyingUser sysname

);

Next, we can create triggers to capture the inserts and updates. First, the insert trigger, which adds the creation time as both the ValidFrom and ValidTo time:

CREATE TRIGGER dbo.Invoice\_AuditCreator

ON dbo.Invoices

AFTER INSERT

AS

BEGIN

INSERT dbo.InvoiceAuditing

(

InvoiceID,

ValidFrom,

ValidTo,

ModifyingUser

)

SELECT

InvoiceID,

ValidFrom,

ValidFrom,

COALESCE(SUSER\_NAME(SUSER\_ID()),USER\_NAME())

FROM inserted;

END

GO

And then one to capture subsequent updates - this one bridges the gap from the previous version to the new version by storing the old start time as the ValidFrom value, and the new start time as the ValidTo time:

CREATE TRIGGER dbo.Invoice\_AuditModifier

ON dbo.Invoices

AFTER UPDATE

AS

BEGIN

INSERT dbo.InvoiceAuditing

(

InvoiceID,

ValidFrom,

ValidTo,

ModifyingUser

)

SELECT

i.InvoiceID,

d.ValidFrom, -- the "old" valid start time

i.ValidFrom, -- the "new" valid start time

COALESCE(SUSER\_NAME(SUSER\_ID()),USER\_NAME())

FROM inserted AS i

INNER JOIN deleted AS d

ON i.InvoiceID = d.InvoiceID;

END

GO

To demonstrate the triggers in action, I'll create a couple of new users:

CREATE USER Sales1 WITHOUT LOGIN;

CREATE USER Sales2 WITHOUT LOGIN;

GRANT INSERT, UPDATE, SELECT ON dbo.Invoices TO Sales1, Sales2;

Running as GORDIE\Administrator, I'll insert a single invoice, then each Sales user will update the invoice once, and then we can look at the relevant tables:

INSERT dbo.Invoices(InvoiceID, Amount) SELECT 12, 50;

EXEC(N'UPDATE dbo.Invoices SET Amount += 10 WHERE InvoiceID = 12;') AS USER = N'Sales1';

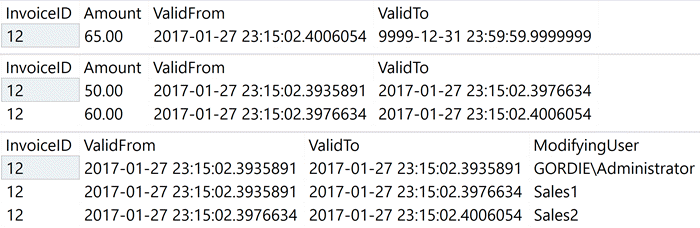
EXEC(N'UPDATE dbo.Invoices SET Amount += 5 WHERE InvoiceID = 12;') AS USER = N'Sales2';

SELECT InvoiceID, Amount, ValidFrom, ValidTo FROM dbo.Invoices;

SELECT InvoiceID, Amount, ValidFrom, ValidTo FROM dbo.InvoiceHistory;

SELECT InvoiceID, ValidFrom, ValidTo, ModifyingUser FROM dbo.InvoiceAuditing;

Results:



Now, to tie these together in traditional temporal queries, we can use a temporal range query for "today" and match up the starting time in the base table to the ending time for the modification, in order to reflect who was responsible for that change. We use a LEFT OUTER JOIN so that the temporal query still returns invoices that were created before the auditing was implemented.

DECLARE @start date = SYSDATETIME(), @end date = DATEADD(DAY,1,SYSDATETIME());

SELECT

i.InvoiceID,

i.Amount,

ia.ModifyingUser,

i.ValidFrom,

i.ValidTo

FROM dbo.Invoices

FOR SYSTEM\_TIME FROM @start TO @end AS i

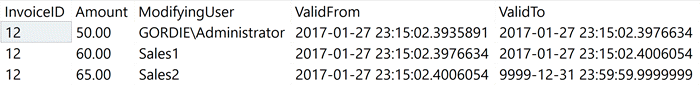
LEFT OUTER JOIN dbo.InvoiceAuditing AS ia

ON i.InvoiceID = ia.InvoiceID

AND i.ValidFrom = ia.ValidTo

ORDER BY i.InvoiceID, i.ValidFrom;

The results accurately reflect that GORDIE\Administrator inserted the invoice with a value of $50, Sales1 updated it to $60, and Sales2 updated it to $65. Note that the additional auditing table only needs to track the immutable key, and not all of the other values in the row (those will still all be maintained in the history table):



If, over time, you will be pruning the history table, you will need to include the extra auditing table in those routines as well.

Summary

While temporal tables don't natively support any forensics other than actual data changes, I have shown that it only takes a little effort to correlate auditing data to the history table through either additional columns or an additional table. You could easily extend this approach to also log other details, such as host name, application name, and other typical diagnostics.

*Note that these approaches only handle inserts and updates; deletes would need to be handled a little differently. I will put that off for a future tip, leaving it as an exercise for the reader in the meantime.*

**Next Steps**

* See these previous tips and other resources:
  + [Tip #4674: Managing Temporal Table History in SQL Server 2016](https://www.mssqltips.com/sqlservertip/4674/managing-temporal-table-history-in-sql-server-2016/)
  + [Tip #3680 : Introduction to SQL Server 2016 Temporal Tables](https://www.mssqltips.com/sqlservertip/3680/introduction-to-sql-server-2016-temporal-tables)
  + [Temporal Tables](https://msdn.microsoft.com/en-us/library/dn935015.aspx) (MSDN)
  + [Tip #3682 : SQL Server 2016 T-SQL Syntax to Query Temporal Tables](https://www.mssqltips.com/sqlservertip/3682/sql-server-2016-tsql-syntax-to-query-temporal-tables)
  + [All SQL Server 2016 Tips](https://www.mssqltips.com/sql-server-tip-category/210/sql-server-2016)
  + [Manage Retention of Historical Data in System-Versioned Temporal Tables](https://msdn.microsoft.com/en-us/library/mt637341.aspx) (MSDN)
  + [Temporal Table Considerations and Limitations](https://msdn.microsoft.com/en-us/library/mt604468.aspx) (MSDN)

Last Update: 2/27/2017