Livre Blanc

Marc LEVESQUE

Gestion automatisée et transparente de l'audit de MAJ de tables dans SQL Server à l'aide de procédures stockées et de triggers – Avantages / Inconvénients

Devolis
Les couturiers de l'informatique

Table des matières

1. Introduction

Quel développeur ou équipe de support n'a pas été un jour confronté à un problème de ce genre ?

Premier cas :

- Un utilisateur remonte un bug
- L'analyse des données (grâce à quelques champs techniques d'audit au niveau de chaque table) montre qu'il ne s'agit probablement pas d'un bug mais d'une mauvaise utilisation de l'application, ou tout du moins d'une mise à jour hasardeuse de métadonnées qui changent le comportement de celle-ci.
- L'utilisateur, de mauvaise foi ou n'étant tout simplement pas informé de ce changement (ou même encore ne se souvenant tout simplement plus de la manipulation), soutient que celles-ci n'ont pas été modifiées depuis longtemps, et surtout bien avant que le problème ne survienne.

- Et pourtant, c'est une évidence pour le développeur ou l'équipe de support, mais il n'y a pas de preuve tangible à lui fournir (même si l'on peut prouver à l'aide des champs techniques que l'enregistrement a été modifié, potentiellement à quelle date et par qui, pour la dernière fois, on n'est pas en mesure de dire quelle modification a effectivement été apportée et ainsi montrer que c'est cette modification qui est à l'origine du problème), si bien que son ressenti vis-à-vis de l'application et de ses contacts en cas de problème peut s'en trouver entaché.
- Sans compter que c'est uniquement l'expérience, dans ce cas, qui fait dire que le problème vient de telle ou telle modification de données puisque même le développeur ou l'équipe support ne peut pas avoir la certitude absolue de la modification de la donnée.

Un exemple concret ?

Supposons qu'un traitement calcule automatiquement une remise commerciale de 2% lors de l'établissement d'une facture pour peu que l'utilisateur ait coché une case dans l'entête de la facture.

Supposons que ce taux de 2% soit indiqué dans une table gérant les différentes remises possibles, table comportant d'autres informations - telles que le code et le libellé de la remise, les conditions (optionnelles) sur montant ou nombre d'articles d'une commande, par exemple, devant être remplies pour que la remise s'applique… - et ne pouvant être mise à jour que par un administrateur.

Un jour, un utilisateur qui fait aveuglément confiance à l'application et ne sait pas que le taux de la remise provient de ladite table, tombe sur un cas simple de facture pour laquelle il n'y a qu'une ligne avec un montant « rond » de 100€. Pour le coup, le montant de la remise est facile à calculer et devrait, dans son esprit, être de 2€. Or, il se trouve être en fait de 1,8€ car le taux enregistré dans la base a été modifié par un administrateur.

Pour l'utilisateur, dans un premier réflexe, il s'agit d'un bug. Du coup il commence à vérifier plusieurs factures et s'aperçoit que cela fait plusieurs semaines que le montant de remise est « faux ». Il alerte le service support qui lui explique ledit changement effectué, a priori, 2 mois plus tôt (d'après le champ technique de date de dernière modification). L'utilisateur n'en reste pas là et vérifie quelques factures plus anciennes. Et, manque de chance, le montant était déjà, d'après lui, erroné. Pour l'équipe support et les développeurs cela ne fait aucun doute. La règle métier et correctement appliquée avec un taux que l'on va bien chercher dans la table, donc si celui-ci est de 1,8% alors qu'il était initialement de 2%, c'est qu'il a bel et bien été modifié, et si cette modification est antérieure à 2 mois c'est que la dernière modification portait sur un ou plusieurs autres champs de la table.

Mais comment rassurer l'utilisateur et lui prouver la cause de la soi-disant erreur puisque l'on ne peut lui fournir que la date de la dernière modification de l'enregistrement qui ne portait pas sur le taux.

Pire encore : imaginons que la modification du taux soit effectivement une erreur de saisie de la part de l'administrateur (qui se serait trompé, par exemple, d'enregistrement et aurait mis à jour la remise en question à la place d'une autre) et que la direction décide, pour assurer son image commerciale, d'effectuer une reprise des factures erronées pour proposer un avoir à ses clients lésés pendant le laps de temps durant lequel le taux était faux. Comment retrouver la date à partir de laquelle les factures établies étaient fausses ?

Second cas :

- Un utilisateur remonte un problème
- L'analyse de ce problème n'est pas, à première vue, évidente et l'on aimerait bien pouvoir rejouer le scénario, au niveau des mises à jour des données, qui a amené à cette situation puisque, à première vue, le problème semble aléatoire car variant dans le temps.
- Malheureusement, on ne possède que la dernière version des enregistrements de chaque table si bien que, sans historique, il n'est pas possible de rejouer toutes les étapes entre la version stable à un instant T et la situation problématique remontée par l'utilisateur.
- La correction d'un tel problème peut s'avérer longue, compliquée et sans garantie qu'elle couvre parfaitement le problème décrit puisque l'on suppute une cause sans pouvoir en avoir la certitude et en apporter la preuve.

Un exemple concret ?

Supposons qu'un traitement automatisé déclenche l'émission d'une commande auprès d'un fournisseur dès que le niveau minimum de stock défini au niveau d'un article est atteint (afin de ne pas avoir de rupture de stock).

Le problème c'est que la règle n'est pas évidente à suivre et à vérifier puisque le déclenchement de la commande dépend à la fois du niveau de stock initial, du stock minimum défini pour l'article et de la quantité de pièces sortant du stock.

Imaginons maintenant que la direction décide d'auditer ce mécanisme à la suite d'un ressenti de déclenchement « aléatoire » des commandes de réapprovisionnement sur un article pour lequel la consommation est irrégulière et pour lequel le seuil minimum de stock a été ajusté plusieurs fois par un responsable, malheureusement en congés, pour essayer de l'optimiser.

Sans l'historique de ces modifications de seuil minimum, et en l'absence du responsable qui aurait pu apporter la lumière sur la situation, l'équipe support ou développement aura sans doute bien du mal à justifier chaque commande de réapprovisionnement en montrant que le seuil minimum « de l'époque » a été atteint au moment de l'élaboration automatique de celle-ci. Le problème pourra alors être assimilé à un bug « aléatoire » et faire perdre énormément de temps en recherche sans forcément pouvoir être solutionné.

Et en supposant, par exemple, qu'une autre règle métier mal implémentée indique que la commande de réapprovisionnement doit être soumise à validation lorsque son montant dépasse une certaine somme (elle-même configurée par article), et que cette valeur, elle aussi, ait été modifiée plusieurs fois pendant la phase d'ajustement des valeurs pour l'article.

La correction d'un tel bug, sur la base des constations d'un utilisateur, peut s'avérer quasi impossible sans avoir l'historique des modifications pour rejouer l'intégralité du scénario jusqu'à tomber sur la situation déclenchant le bug

Troisième cas :

- Un utilisateur ne comprend pas pourquoi il ne retrouve pas des métadonnées qu'il est pourtant certain d'avoir saisi dans l'application et signale donc un bug « à l'enregistrement »
- Les règles métiers mises en œuvre prévoient que les métadonnées sont uniquement archivées (à l'aide d'un flag de suppression logique dans la table) en cas de référencement dans des tables « enfant » mais qu'une suppression physique a lieu lorsqu'elles ne sont pas encore « utilisées » (référencées) afin de ne pas polluer la base de données en cas de simple erreur de saisie qu'il est alors possible d'annuler sans laisser de trace.
- En pareil cas, même avec quelques champs techniques, aucun moyen de démontrer à l'utilisateur que sa donnée avait effectivement été correctement enregistrée (donc, pas de bug) puisqu'elle a été supprimée (par erreur ou non).

Un exemple concret ?

Un administrateur prépare en urgence, avant son départ en congés (ou arrêt maladie) des métadonnées qui seront à utiliser dans quelques jours.
Un collègue, mal informé, fait le ménage durant son absence afin de supprimer toutes les données « polluantes ».

Par erreur, il présume que les données ainsi préparées n'ont jamais été utilisées et sont de vieilles erreurs de saisie et décide donc de les supprimer.

À son retour, l'administrateur reçoit des plaintes d'utilisateurs lui remontant que les données de références censées être disponibles depuis plusieurs jours n'existent pas.
L'administrateur, certain de sa manipulation avant son départ suppose, tout d'abord, qu'il s'agit d'un potentiel bug au niveau de la restitution des métadonnées et contacte l'équipe support qui lui répond que les données n'existent tout simplement pas.

Là, l'administrateur, sûr de son coup, remonte alors un bug « à l'enregistrement » et tous se retrouvent dans une impasse, puisqu'il n'y a aucun moyen de prouver que

l'enregistrement a bel et bien été créé puis supprimé. Chacun peut alors douter de la parole de l'autre, ce qui peut entrainer un climat de défiance entre les personnes peu propice à une bonne collaboration.

Que retenir de ces quelques exemples ?

D'abord, la traçabilité des modifications effectuées sur les données n'est pas vaine, bien au contraire et en particulier dès que l'on rencontre une suspicion de bug.

En tant que développeur, on est en effet fréquemment confronté à des retours utilisateurs remontant une erreur sur une application pourtant installée depuis longtemps, et n'ayant pas subi de mise à jour depuis un bon moment. Généralement, de tels bugs (avérés ou non) apparaissent à la suite de modifications de données, faisant basculer l'application dans des situations nouvelles et inattendues (donc non gérées) et bien sûr non envisagées en phase de test ou d'UAT.

En pareil cas, toute information de traçabilité ou d'historique des modifications apportées aux tables (structure ou données) est la bienvenue pour aider à comprendre la situation.

Mais souvent, et dans le meilleur des cas, on dispose uniquement de quelques champs dans chaque enregistrement.

N'imaginant pas spécialement, lors de la création d'une ou plusieurs table(s), les problèmes que l'on pourrait être amené à rencontrer dans l'avenir, et devant la complexité supposée de mise en œuvre (manuelle) de solution visant à historiser les modifications effectuées, on se contente souvent d'ajouter à toute table 2 champs d'audits que sont :

- La date de dernière modification
- L'utilisateur ayant réalisé la dernière modification

Parfois, on ajoute également un ou deux champs concernant la création de l'enregistrement :

- L'utilisateur ayant créé l'enregistrement
- La date de création de l'enregistrement

Ainsi, qu'un ou plusieurs champs pour gérer la suppression logique de l'enregistrement :

- Flag indiquant que l'enregistrement est archivé (ad-minima)
- La date de l'archivage
- L'utilisateur ayant réalisé l'archivage

Dans bien des cas, cela suffit effectivement à assurer un minimum d'audit et de traçabilité des modifications de données mais généralement, tôt ou tard, on regrette de ne pas avoir à sa disposition plus d'informations sur les modifications effectuées pour répondre à tel ou tel besoin, question, ou suspicion de bug remonté par un utilisateur.

2. Vers une traçabilité totale des mises à jour de données

Une solution « mono-table »

Une solution permettant cette traçabilité totale (ou quasi-totale) des mises à jour de données consiste à gérer une notion de version d'enregistrement dans les tables à historiser. Pour une traçabilité complète, il faudra également prendre en compte la notion de « suppression » qui ne pourra alors qu'être logique par le biais d'un flag.

La solution, côté base de données, est assez simple à mettre en œuvre puisqu'elle consiste en l'ajout d'un champ technique « Version » devant faire partie de la clé primaire ou d'un index unique.

Ainsi, un même élément sera défini par une clé représentant l'élément (id d'élément, à ne pas confondre avec un id d'enregistrement, auto-incrémenté ou non) + version de l'élément (incrémenté de 1 à chaque modification).
Bien entendu, cette solution requiert que l'on s'interdise tout usage de requête UPDATE ou DELETE.

Les seules requêtes autorisées (outre le SELECT) seront donc des requêtes INSERT afin de gérer manuellement et intelligemment la clé (ou index unique) composé de l'ID de l'élément (pouvant donc être présent plusieurs fois dans la table) et le n° de version (que l'on devra incrémenter systématiquement).

Ainsi la table sera porteuse à la fois de tout l'historique de modification, depuis sa création jusqu'à son éventuelle suppression physique, d'un élément et de sa toute dernière version représentée par l'enregistrement ayant le numéro de version le plus élevé.

Toutefois, si la solution est simple côté BDD, sa mise en œuvre est plutôt assez complexe côté applicatif, puisque toute la gestion doit être faite en amont, dans l'application, perdant ainsi tous les avantages proposés par le moteur de base de données :

- Pour un ajout, si l'on veut gérer un id d'élément s'apparentant à un id auto-incrémenté, il faut en premier lieu interroger la table pour récupérer le plus grand id d'élément possible avant de gérer la requête insert avec n° version égale à zéro.
  L'alternative est d'avoir recours à une séquence (géré par la plupart des moteurs de base données)
  L'alternative
- Pour toute modification ultérieure (ou suppression logique), il faut interroger la table afin de retrouver le plus grand n° de version pour l'élément à mettre à jour afin de l'incrémenter de 1 dans la requête d'insert.
- Pour toute requête d'interrogation d'un élément (SELECT), il faut penser à prendre uniquement l'enregistrement dont le n° de version est le plus élevé pour ledit élément, ce qui complexifie énormément la requête.
- Pour une requête d'interrogation portant sur plusieurs éléments, c'est encore plus compliqué, bien entendu.
- La notion de clé étrangère référençant une table ainsi historisée est impossible techniquement et ne peut qu'être gérée par l'application ou par l'écriture de requête complexe.
  En effet, la clé primaire de la table historisée étant composée et comportant, dans sa

composition, un n° de version, il n'est pas possible de référencer un élément dans une autre table (référencer un éventuel id d'enregistrement serait une erreur puisque cela figerait la version référencée, de même que se base sur la clé primaire ou index unique composé du n° de version).
Ainsi la gestion de référencement ne peut être faite que de manière applicative ou au moyen de requêtes complexes, en s'affranchissant de la déclaration et de la gestion, pourtant fort utile, de clés étrangères.

- Bien entendu, la conséquence directe de cette dernière remarque, est la plus grande complexité des requêtes de sélection intégrant des jointures vers des tables ainsi historisées.

Il y a cependant un moyen de simplifier un peu la récupération du dernier enregistrement d'un élément, en ajoutant un autre champ à la table permettant de connaitre instantanément l'enregistrement « actif » en cours de l'élément (celui correspondant au plus grand n° de version).

La conséquence, cependant, de la mise en œuvre de ce champ supplémentaire (pouvant s'avérer très pratique pour toute opération de relecture et jointure au niveau des requêtes SELECT), c'est que les opérations d'écriture (en dehors de la phase initiale d'ajout d'un nouvel élément) s'en trouvent complexifiées, puisqu'il ne faut pas oublier de mettre à jour l'ancien enregistrement « actif » de l'élément mis à jour pour le rendre « inactif » et ne pas oublier de flaguer comme « actif » l'enregistrement inséré pour refléter la modification de l'élément. En effet, pour que ce système fonctionne, il faut absolument qu'il n'y ait qu'un seul élément actif à un instant T.

Un autre petit inconvénient de cette solution (pas si petit que cela, d'ailleurs), c'est que l'on doit également gérer une sorte de mécanisme de verrou lors des mises à jour de la table (ou tout du moins penser à systématiquement travailler avec une transaction) puisque les opérations d'ajout et de modification incluent quasi-systématiquement une phase de lecture de la table pour retrouver des informations nécessaires à la constitution de l'enregistrement à insérer effectivement dans la table.

Un autre inconvénient porte sur les mises à jour de masse puisque les mécanismes à mettre en œuvre à chaque modification d'enregistrement sont potentiellement complexes et multiples, avec lecture de la table avant écriture dans celle-ci, il peut s'avérer très complexe, voire impossible, de concevoir des requêtes visant à mettre à jour des données en masse (aussi bien pour de la création que pour de la mise à jour).

Enfin, une table ainsi historisée deviendra vite très grosse, ce qui aura forcément un impact sur les temps d'exécution des requêtes, tant en lecture qu'en écriture. Il y aura également un impact sur la durée et la taille des sauvegardes, ainsi que leurs éventuelles restaurations, sans parler du journal de logs.

Bien entendu, il est possible de traiter les mises à jour de données via des triggers pour garantir les règles métier vis-à-vis des champs « Id Element » (ajout), « Version » et « Élément Actif ». Cependant, il n'est pas aisé de mettre en place un mécanisme automatique de génération de tels triggers puisque le code à exécuter est dépendant de champs dont la sémantique doit être connue. Il est donc nécessaire de créer manuellement de tels triggers, ce qui rend la tâche de création et mise à disposition de table un peu plus longue et complexe.

En conclusion, même si cette solution peut sembler séduisante, a priori, et donc être choisie, parfois, pour des tables en bout de chaîne, elle apporte tout de même son lot de problèmes et de complexités qu'il conviendra de résoudre. Au fond, cela la rend peut efficace et un peu risquée puisque le moindre oubli, par les développeur, des quelques règles de bases permettant à cette mécanique de fonctionner correctement, peut avoir des conséquences non négligeables sur les performances et/ou l'intégrité des données, d'autant qu'il y a souvent confusion, pour les développeurs peu habitués à cette méthodologie, entre id technique (de l'enregistrement) et id métier (de l'élément) si bien que bon nombre de requêtes (avec jointure, notamment) peuvent aisément être faussées.

Une solution avec table d'historique dédiée

La meilleure solution, pour assurer une traçabilité totale des mises à jour de données et ainsi ne pas regretter le manque d'information évoqué en introduction, consiste à doubler toute table de production d'une table « historique » reprenant l'ensemble des colonnes de la table principale (sans contrainte ni gestion de clé étrangère, pour éviter tout risque de conflit) auxquelles il suffit d'ajouter quelques champs de suivi :

- Type de modification (ajout, modification ou suppression)
- Date/heure réelle de la modification (date système, par opposition à l'éventuelle date de création ou dernière modification présente dans l'enregistrement de la table principale qui est généralement fournie en entrée par l'application ou la requête de mise à jour au point de pouvoir être modifiée après coup, si l'on n'y prend pas garde)
- Utilisateur réel à l'origine de la modification, au sens utilisateur de la base de données. Ici, il s'agit du login utilisé pour se connecter à la base de données, par opposition à l'éventuel « utilisateur ayant créé ou modifié en dernier l'enregistrement » présent dans l'enregistrement de la table principale. Ce dernier est généralement fourni en entrée par l'application ou la requête de mise à jour et peut être modifié après coup, si l'on n'y prend pas garde.
- Id de la ligne d'historique (optionnel, mais bien pratique)
- N° de version de l'enregistrement. Celui-ci est optionnel mais peut aussi s'avérer utile, même si la date/heure réelle de la modification devrait théoriquement suffire.

Bien entendu, cette solution apporte son lot d'avantages et d'inconvénients.

Commençons par les avantages de cette solution :

- Les données de la table principale peuvent être gérées, côté BDD, sans se soucier des problématiques d'historique, puisque celles-ci sont gérées via une table miroir enrichie. On conserve donc bien l'utilisation possible des 3 modes de mise à jour (INSERT, UPDATE, DELETE), la possibilité de gérer comme il se doit les clés étrangères, etc.
- La table principale ne grossit pas « artificiellement » à cause de la gestion d'historique
- Les tables « historique » peuvent être gérées dans une seconde base de données afin de limiter l'impact sur les sauvegardes et restaurations de la base de données contenant les tables principales. En prérequis, les droits suffisants doivent avoir été accordés aux utilisateurs manipulant les données devant être historisées.

À noter : dans le cas où l'on préfèrerait tout de même conserver les tables d'historique dans la même base, il peut être souhaitable de les séparer en termes de schéma.

- Il est facile de visualiser dans le temps les évolutions appliquées à la valeur de n'importe quel champ d'une table.
- Il est donc également possible et relativement facile de rejouer n'importe quel scénario applicatif à partir d'un état des données à un instant T.
- Les applications exploitant la BDD n'ont pas nécessairement lieu de gérer « pleinement » les tables d'historique (voir « automatisation » plus loin dans cet article)
- Il est possible de purger ponctuellement les données d'historique les plus anciennes sans risquer de perte de données au niveau de leur table principale.

Pour ce qui est des moins de cette solution :

- Les tables d'historique devant être le reflet des différentes mises à jour de leur table principale, elles doivent toujours être à jour en termes de structure, ce qui veut dire que toute modification de type « *ajout de colonne* » ou « *modification de type de données d'une colonne* » apportée à la structure de la table principale doit nécessairement être reportée également dans la table historique. Ceci implique des manipulations supplémentaires, en particulier lorsqu'une application utilise EntityFrameWork en mode code-first. Il faudra alors créer/modifier manuellement les tables d'historiques, à moins de les intégrer totalement dans l'application. Cette méthode aura pour conséquence d'alourdir le modèle de données uniquement pour intégrer la gestion de l'historique.
  Également, il est conseillé de reporter tout changement du nom de la table principale ou de l'une de ses colonnes pour lever toute ambiguïté auprès des équipes de développement ou de support en cas de consultation des données de la table d'historique.
  A contrario, pour correctement suivre l'historique des données d'une table, il peut être préjudiciable de reporter une suppression de colonne de la table principale dans la table d'historique. À terme, il peut donc être logique qu'une table d'historique comporte plus de colonnes que sa table principale.
- Sans mécanisme d'automatisation (voir chapitre suivant), la mise à jour des données des tables d'historique, pour être efficace et ne rien laisser passer, nécessite une gestion systématique par tout intervenant sur les données des tables principales.
  Cela implique que les développeurs pensent bien à doubler les enregistrements effectués en table « principale » par des enregistrements d'historique, en précisant systématiquement le contexte (ajout, modification, suppression) et la date réelle de la modification.
  Ces opérations doivent donc être intégrées dans l'application, bien sûr, directement par requête SQL ou utilisation d'EntityFramework, ou indirectement, par recours à des procédures stockées. Celles-ci se chargeant de la mise à jour à la fois de la table principale et de sa table d'historique, mais aussi lorsqu'une éventuelle action de mise à jour est réalisée directement en base de données (sans passer par une application), faute de quoi l'historique ne peut pas être considéré comme fiable.

En conclusion, même si cette solution offre un certain nombre d'avantages, en particulier en regard de la première solution évoquée, elle n'en demeure pas moins contraignante : pour remplir correctement son rôle, les tables d'historiques doivent systématiquement refléter les évolutions

apportées à leur table principale respective, tant en termes de données que de structure. Ceci impose de mettre en place des mécanismes ou procédures devant être respectés par tous.

Heureusement, il est possible d'automatiser tout cela à l'aide de triggers et de procédure stockées. Voyons comment…

3. Vers l'automatisation de la traçabilité

La seconde solution évoquée serait idéale, en exploitation de données, si elle ne nécessitait pas de recours à des procédures stockées pour la mise à jour des données ni ne nécessitait que les développeurs ou autres intervenants sur les données de la base, n'aient à penser et à mettre en œuvre la mise à jour des données d'historique en même temps que celle des données de la base principale.

Automatisation du stockage de l'historique

Eh bien ceci est réalisable grâce à des triggers (déclencheurs) pouvant être mis en place sur chaque table à historiser. Ces triggers pourront se déclencher avant ou après toute opération d'insertion, de mise à jour ou de suppression des données de la table principale.

Étant des outils proposés par le moteur de base de données, leur utilisation est alors transparente pour les utilisateurs et même pour les développeurs, dans le cadre de la réalisation des applications qui n'ont alors pas besoin de connaitre les mécanismes d'historisation mis en œuvre (et donc de se trouver alourdies par eux).

Le trigger est une sorte de procédure stockée un peu particulière : cette fonction s'exécute automatiquement, comme un événement, en réponse à une action de mise à jour des données d'une table.
Il est alors facile de renseigner la table « historique » dès que des données de la table « principale » sont mises à jour.

Avec cette méthode, la seule contrainte est de mettre à jour le trigger : il faut modifier le code exécutable, en plus de la structure de la table d'historique, lorsque la structure de la table principale change, afin de prendre en compte ce changement (par exemple, en cas d'ajout, suppression ou renommage d'un champ dans la table principale).

Mais comment savoir s'il s'agit d'une insertion, d'une suppression ou d'une modification ? Il y a 2 moyens pour cela :

- Soit avoir un trigger dédié par type d'action (un trigger en cas d'insertion, un trigger en cas de modification et un trigger en cas de suppression).
  Dans ce cas, il est facile de déterminer le type d'action, mais en cas de modification de structure de la table principale, il faudra intervenir sur 3 triggers qui, globalement, font sensiblement la même chose à savoir écrire dans la table d'historique. Ce n'est pas très efficace…
  De même, si pour un usage spécifique ponctuel et temporaire, on devait désactiver l'historisation, il y aurait potentiellement 3 triggers à désactiver puis à réactiver. Comme il s'agit d'opérations très ponctuelles, il y a un gros risque d'oubli.
- Soit utiliser les 2 pseudo-tables (« inserted » et « deleted ») mises à disposition dans le code d'un trigger pour fournir la liste des enregistrements devant être ajoutés et supprimés pour déterminer le type d'action.
  En effet, l'une seulement de ces 2 pseudo-tables n'est renseignée pour les 2 situations d'INSERT (« deleted » est alors vide) et de DELETE (« inserted » est alors vide) tandis que les 2 pseudo-tables sont renseignées en cas d'UPDATE (« inserted » représentant la nouvelle version des enregistrements tandis que « deleted » représente la versions avant modification

de ceux-ci) .
On peut donc se baser sur les 3 combinaisons possibles de ces pseudo-tables pour déterminer la situation.

L'avantage du trigger n'est donc plus à démontrer.
Une fois le code écrit et testé, il n'y a plus besoin de se soucier du remplissage de la table d'historique, que ce soit au travers d'une application ou directement via une requête SQL en base de données.
La règle métier d'historisation est alors garantie en toute occasion.

Sauf que…

Les données de la table principale ne sont pas mises à jour si le trigger « plante ». Et qu'est-ce qui pourrait faire planter le trigger ? N'importe quelle erreur remontée par la base de données lors de la tentative d'écriture (au sein du trigger) dans la table d'historique. Il faut donc bien tester et valider le code du trigger. Soit, mais cela ne suffit pas.
En effet, avec un mécanisme transparent sur un système dont les modifications de structures sont potentiellement rares, le risque d'oublier la mise à jour de la structure de la table d'historique et/ou du code du trigger en cas de changement de la structure de la table principale (renommage de la table proprement dite ou de l'un de ses champs, ajout d'un champ et surtout suppression d'un champ, modification du type de données d'un champ…).
Ce risque se trouve naturellement renforcé si l'équipe de développeurs est amenée à changer régulièrement et que tous ne sont pas informés de la présence de ce mécanisme, d'autant plus si l'application développée utilise EntityFramework en mode code-first.

Or, si la table historique ou le trigger ne prend pas correctement en compte ces changements, le risque de « plante » lors de l'écriture est alors élevé.

En conclusion, le recours aux triggers pour la mise à jour systématique des données de la table d'historique semble une bonne solution, mais on se retrouve tout de même un peu dans une impasse, en particulier dans certaines situations telles que l'usage d'EntityFramework en mode code-first, puisqu'en termes de maintenance et d'évolution de la structure de la base de données, on est contraint de penser à intervenir pour la mise à niveau de cette table d'historique et du trigger.

La solution réellement idéale serait de pouvoir gérer également de façon transparente cette mise à niveau de la table d'historique et du trigger.

Automatisation de mise en œuvre du mécanisme d'automatisation

Eh bien là encore, c'est possible grâce à des triggers !

Il ne s'agit pas de triggers portant sur les tables, cette fois, mais de triggers un peu spéciaux portant sur la base de données proprement dite.

Quid de déclencheurs capables d'intercepter les événements de modification de structure de la base de données, et en particulier de modifications apportées aux tables ou aux colonnes de celles-ci ? Pour peu qu'on soit en mesure d'écrire un code qui auto-générerait la table historique à partir de la structure de la table principale (ou qui la modifierait) et qui soit également capable d'auto-générer

le code du trigger de table à appliquer pour toute action d'iNSERT, UPDATE ou DELETE de la table principale, l'affaire serait gagnée, non ?

C'est là tout l'intérêt du trigger de base de données qui va se déclencher sur des événements tels que :

- Création de table
- Modification de table
- Renommage (de table ou de champ)

Ainsi, le trigger va nous permettre d'écrire du code propre à créer ou modifier la structure de la table d'historique associée et générer le code du trigger de table visant à mettre à jour cette table d'historique, en s'appuyant sur certaines tables/vues système (sys.tables, sys.columns, …) grâce auxquelles il est possible de récupérer la structure de la table principale et de sa clé primaire.

Il y a tout de même deux bémols à cette solution :

- Une fois qu'il a été mis en œuvre, le trigger se déclenchera bien pour toute évolution à venir de la structure de la base de données, mais ne permettra pas une reprise de l'existant (afin de générer automatiquement les tables historiques et triggers correspondant aux tables « principales » déjà existantes).
- Il ne faudrait pas que le trigger entre dans une boucle infernale en générant des tables d'historique et des triggers de tables pour les tables d'historique elles-mêmes créées depuis une table principale (ou d'une précédente table d'historique) dans le cadre du trigger et ce à l'infini (du moins jusqu'à ce que le système finisse par planter).

Mais pas de panique ! Il y a moyen de pallier ces 2 petits problèmes en passant par une ou plusieurs procédures stockées appelées depuis le trigger, mais pouvant également, de ce fait, être appelée(s) manuellement pour la génération d'une table d'historique et du trigger de table pour une table principale donnée et en contrôlant l'appel à cette ou ces procédure(s) stockée(s) en fonction du nom de la table pour laquelle le trigger est déclenché en création, en le comparant, par exemple, à un pattern de nom de table défini pour les tables d'historique, ou en se basant sur son schéma d'appartenance ou encore en s'appuyant sur une table destinée à gérer le statut des tables d'historique.

À noter que si les tables d'historique sont stockées dans une autre base de données, ce second problème est inexistant puisque le trigger de base de données ne s'applique alors que pour les tables de la BDD principale d'exploitation (qui ne contient alors que celles-ci). Cependant, dans ce cas, l'automatisation de la création des tables d'historique et triggers sera un peu plus complexe car SQL Server pose une limite, pour la création de trigger, à la base de données en cours. Or, idéalement, il est souhaitable de créer également un trigger pour les tables historisées afin d'en empêcher la modification ou suppression de données (seule l'insertion doit être autorisée). Mais là-encore on peut s'en sortir avec un trigger de base de données au niveau de la base dédiée à l'historisation.

Il est temps de passer à cas concret !

4. Mise en œuvre de la traçabilité (cas pratique)

Plusieurs étapes vont devoir être réalisées « manuellement » (ou presque) pour initier le système qui pourra ensuite fonctionner de façon autonome.

Dans l'exemple suivant, nous allons travailler avec 2 bases de données, l'une contenant les données vivantes d'exploitation et l'autre dédiée à l'historisation des tables de la première base.

Création de la base de données « Historique »

Tout d'abord, en supposant que la base de données « principale » existe déjà, il faut créer la base de données dédiée à son historisation. Pour aider à la compréhension et à la maintenance, nous reprendrons le nom de la base « principale » à laquelle nous ajouterons le suffixe « _Histo ». Imaginons donc que notre base principale s'appelle « ProdData », nous commençons par créer la base de données « ProdData_Histo ».

```
CREATE DATABASE ProdData_Histo
```

Cette base étant dédiée à l'historisation des données de la base ProdData, il n'y a pas lieu, théoriquement, de trouver d'autres tables que celles utilisées pour l'historisation. On va donc créer un trigger de base de données qui créera systématiquement 2 triggers pour chaque table ajoutée, l'un pour en empêcher la modification des données et l'autre pour en empêcher la suppression de données. On peut éventuellement rendre ce trigger moins systématique en vérifiant que le nom des tables ainsi créée se trouve préfixé ou suffixé d'un terme choisi (« histo », par exemple), mais il faudra alors systématiquement reprendre cette règle dans tous les autres traitements. Le plus simple est donc de ne pas vérifier le nom des tables et simplement supprimer les triggers qui auront été créés automatiquement si cela s'avère nécessaire pour une table ne représentant pas des données d'historisation (table de travail propre à la BDD d'historisation par exemple).

Le code de ce trigger de base est le suivant :

```
CREATE TRIGGER TRIG_On_Table_Creation
ON DATABASE
FOR CREATE_TABLE
AS BEGIN
	SET NOCOUNT ON;

	-- On commence par récupérer le nom de la table créée ainsi que son schéma
	DECLARE @Data XML = EVENTDATA();
	DECLARE	@SchemaName	VARCHAR(256)	=	@Data.value('(/EVENT_INSTANCE/SchemaName)[1]',
'varchar(256)');
	DECLARE	@TableName	VARCHAR(256)	=	@Data.value('(/EVENT_INSTANCE/ObjectName)[1]',
'varchar(256)');

	-- Puis on déclare 2 variables destinées à recevoir les 2 scripts de création des triggers
	DECLARE @TrigOnUpdate VARCHAR(MAX) = 'CREATE TRIGGER ' + @SchemaName + '.TRIG_' + @TableName +
'_OnUpdate
ON ' + @SchemaName + '.' + @TableName + '
INSTEAD OF UPDATE
AS BEGIN
	RAISERROR (''Les modifications sont interdites pour une table d''''audit !'', 16, 1);
END';

	DECLARE @TrigOnDelete VARCHAR(MAX) = 'CREATE TRIGGER ' + @SchemaName + '.TRIG_' + @TableName +
'_OnDelete
ON ' + @SchemaName + '.' + @TableName + '
INSTEAD OF DELETE
AS BEGIN
	RAISERROR (''Les suppressions sont interdites pour une table d''''audit !'', 16, 1);
END';
```

```
        -- Pour finir, on exécute les 2 scripts ainsi préparés
        EXEC (@TrigOnUpdate);
        EXEC (@TrigOnDelete);
END
```

Désormais, chaque fois qu'une table sera créée dans la base de données d'historisation, 2 triggers seront également créés automatiquement pour en empêcher la modification ou la suppression de données.

Création automatique des tables d'historique

Comme évoqué précédemment, on peut être amené à mettre en place l'historique de tables existantes en plus des futures tables que seront créées à l'avenir.

C'est pourquoi il préférable de dédier le travail de création des tables d'historique et des triggers visant à les alimenter à chaque modification de données à une procédure stockée qui pourra alors être appelée manuellement (ou par script) pour reprise des tables existantes ou automatiquement au travers d'un trigger de base interceptant la création de table (comme nous l'avons déjà fait dans la base d'historisation).

Nous pouvons maintenant créer la procédure stockée de création de table d'historique, qui prend 3 arguments en entrée (le nom de la table à historiser, son schéma d'appartenance, dbo étant le schéma par défaut, et le nom, optionnel, de la base de données « principale », la base courante étant prise en compte, par défaut, lorsque l'argument n'est pas renseigné.

À noter qu'il n'est pas possible de créer un schéma ou un trigger dans une autre base que la base courante, c'est pourquoi cette procédure stockée est placée dans la base « historisation » (pour permettre le report des schémas de la base « principale » et ainsi conserver la logique de regroupement des tables) et ne contient pas d'instruction pour créer le trigger sur la table principale nécessaire à l'automatisation de l'historisation.

À noter également que, dans la mesure où la base « historique » est dédié à cette gestion d'historique, on pourrait choisir de placer cette procédure stockée dans le schéma dbo. Cela dit, pour aider à la compréhension et à la portabilité de cette procédure stockée dans une version « monobase » de la gestion d'historisation, on va plutôt la placer dans un schéma Audit (à créer si inexistant dans la base historique). Pour la même raison, on va systématiquement suffixer le nom de la table historique avec « _Histo ».
Voici son script :

```
CREATE OR ALTER PROCEDURE Audit.CREATE_AUDIT_ON_TABLE
(
        @TableName SYSNAME,
        @SchemaName SYSNAME = 'dbo',
        @DatabaseName SYSNAME = NULL
)
AS BEGIN

        DECLARE @HistoDatabaseName SYSNAME = DB_NAME();
        DECLARE @HistoTableName SYSNAME = @TableName + '_Histo';

        IF @DatabaseName IS NULL
        BEGIN
                -- Lorsque le nom de la base de données où doit se trouver la table à historiser n'est
pas fournie, on prend la base de données à partir de laquelle la procédure stockée a été appelée
                SELECT TOP 1 @DatabaseName = DB_NAME(resource_database_id)
```

```
                        FROM sys.dm_tran_locks
                        WHERE request_session_id = @@SPID
                                AND resource_type = 'DATABASE'
                                AND request_owner_type = 'SHARED_TRANSACTION_WORKSPACE'
                        ORDER BY IIF(resource_database_id != DB_ID(), 0, 1);
        END

        -- Il faut tout d'abord vérifier que la table à historiser existe bien
        DECLARE @IsNOK BIT = 0
        DECLARE @checkSql NVARCHAR(MAX) = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.tables
WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
        BEGIN
                RAISERROR (N''La table %s.%s n''''existe pas dans la base de données %s !'', 16, 1,
@SchemaName, @TableName, @DatabaseName);
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @DataBaseName,
                        @SchemaName = @SchemaName,
                        @TableName = @TableName;
        If @IsNOK = 1
                RETURN;

        -- Il faut ensuite vérifier que la table d'historique n'a pas déjà été créée
        SET @checkSql = 'IF EXISTS(SELECT * FROM ' + @HistoDatabaseName + '.sys.tables WHERE object_id
= OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
        BEGIN
                RAISERROR (N''La table d''''historique %s.%s existe déjà dans la base de données %s !'',
16, 1, @SchemaName, @TableName, @DatabaseName);
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @HistoDatabaseName,
                        @SchemaName = @SchemaName,
                        @TableName = @HistoTableName;
        If @IsNOK = 1
                RETURN;

        -- On peut maintenant préparer le script de création de la table d'historique
        DECLARE @CreateTable VARCHAR(MAX) = 'CREATE TABLE ' + @SchemaName + '.' + @HistoTableName + ' (
        CHANGE_ID INT NOT NULL IDENTITY(1,1)
        , CHANGE_DATE DATETIME NOT NULL CONSTRAINT DF_' + (CASE WHEN @SchemaName = 'dbo' THEN '' ELSE
@SchemaName + '_' END) + @HistoTableName + '_CHANGE_DATE DEFAULT (GETDATE())
        , CHANGE_LOGIN VARCHAR(256) NOT NULL CONSTRAINT DF_' + (CASE WHEN @SchemaName = 'dbo' THEN ''
ELSE @SchemaName + '_' END) + @HistoTableName + '_CHANGE_LOGIN DEFAULT (SUSER_NAME())
        , CHANGE_OPERATION CHAR(3) NOT NULL';

        -- On ajoute une instruction pour création du schéma, si nécessaire
        IF NOT EXISTS(SELECT * FROM sys.schemas WHERE name = @SchemaName)
                SET @CreateTable = 'CREATE SCHEMA ' + @SchemaName + ' ' + @CreateTable;

        -- Puis on déclare un cursor pour énumérer les colonnes de la table principale afin de les
ajouter à la table historique
        DECLARE @SqlColumns NVARCHAR(MAX) = 'DECLARE colsCursor CURSOR FOR SELECT name, system_type_id,
max_length, precision, scale, collation_name FROM ' + @DatabaseName + '.sys.columns WHERE object_id =
OBJECT_ID(''' + @DatabaseName + '.' + @SchemaName + '.' + @TableName + ''')';
        EXEC sp_executesql @SqlColumns;

        DECLARE @colName varchar(256), @colSystype_id int, @colMax_length int, @colPrecision int,
@colScale int, @colCollation varchar(256);

        OPEN colsCursor;

        FETCH NEXT FROM colsCursor INTO @colName, @colSystype_id, @colMax_length, @colPrecision,
@colScale, @colCollation;
```

```
        WHILE @@FETCH_STATUS = 0
        BEGIN
                SET @CreateTable = @CreateTable + N'
        , ' + @colName;
                IF TYPE_NAME(@colSystype_id) = 'TIMESTAMP'
                        SET @CreateTable = @CreateTable + ' VarBinary(8)';
                ELSE
                        SET @CreateTable = @CreateTable + ' ' + TYPE_NAME(@colSystype_id);

                IF TYPE_NAME(@colSystype_id) = 'CHAR'
                OR TYPE_NAME(@colSystype_id) = 'NCHAR'
                OR TYPE_NAME(@colSystype_id) = 'VARCHAR'
                OR TYPE_NAME(@colSystype_id) = 'NVARCHAR'
                OR TYPE_NAME(@colSystype_id) = 'BINARY'
                OR TYPE_NAME(@colSystype_id) = 'VARBINARY'
                BEGIN
                        SET @CreateTable = @CreateTable + '(';
                        IF @colMax_Length = -1
                                SET @CreateTable = @CreateTable + 'MAX';
                        ELSE
                                SET @CreateTable = @CreateTable + CONVERT(VARCHAR(MAX), @colMax_length);
                        SET @CreateTable = @CreateTable + ')';
                END

                IF TYPE_NAME(@colSystype_id) = 'DECIMAL'
                OR TYPE_NAME(@colSystype_id) = 'NUMERIC'
                OR TYPE_NAME(@colSystype_id) = 'FLOAT'
                BEGIN
                        SET @CreateTable = @CreateTable + '(';
                        SET @CreateTable = @CreateTable + CONVERT(VARCHAR(MAX), @colPrecision);
                        IF TYPE_NAME(@colSystype_id) <> 'FLOAT'
                                SET  @CreateTable  =  @CreateTable  +  ','  +  CONVERT(VARCHAR(MAX),
@colScale);
                        SET @CreateTable = @CreateTable + ')';
                END

                IF @colCollation IS NOT NULL
                        SET @CreateTable = @CreateTable + ' COLLATE ' + @colCollation;

                FETCH NEXT FROM colsCursor INTO @colName, @colSystype_id, @colMax_length, @colPrecision,
@colScale, @colCollation;
        END

        CLOSE colsCursor;
        DEALLOCATE colsCursor;

        -- On finalise le script en ajoutant quelques contraintes à la table historique (clé primaire,
valeurs possibles pour le champ opération, valeurs par défaut pour les champs CHANGE_LOGIN et CHANGE_DATE)
        SET @CreateTable = @CreateTable + '
        , CONSTRAINT PK_' + (CASE WHEN @SchemaName = 'dbo' THEN '' ELSE @SchemaName + '_' END) +
@HistoTableName + ' PRIMARY KEY (CHANGE_ID)
        , CONSTRAINT CHK_' + (CASE WHEN @SchemaName = 'dbo' THEN '' ELSE @SchemaName + '_' END) +
@HistoTableName + '_CHANGE_OPERATION CHECK (CHANGE_OPERATION = ''INS'' OR CHANGE_OPERATION = ''UPD'' OR
CHANGE_OPERATION = ''DEL'')
);';

        -- Pour finir on exécute les scripts
        EXEC (@CreateTable);
END
```

Pour rendre cette création de table d'historique automatique, il ne reste plus qu'à définir un trigger de base dans la base de données principale pour intercepter les événements de création de table. Nous verrons ces points un peu plus loin.
Mais en attendant, il faut également s'intéresser à la création automatique du trigger - sur la table principale – chargé de mettre à jour les données de la table d'historique.

Création automatique du trigger d'insertion dans la table d'historique

Sans vouloir anticiper trop sur la suite de cet article, là-encore, il est à noter que la mise à jour du trigger pourra être nécessaire dans plusieurs situations telle que la création d'une table, l'ajout ou la modification d'une colonne d'une table suivie, le renommage d'une colonne ou même d'une table proprement dite, son changement de schéma…
Il est donc là-encore préférable d'avoir recours à une procédure stockée pouvant ainsi être appelée manuellement ou en automatique et depuis différentes situations.
Cette procédure stockée prendra plusieurs arguments en entrée permettant d'interroger comme il se doit les tables système sys.columns et sys.indexes afin de récupérer les informations nécessaires à la définition du trigger pour une table donnée d'un schéma donné.

SQL Server imposant une contrainte empêchant la création de trigger pour une autre base de données que la base courante, il est nécessaire de placer cette procédure stockée dans la base de données principale. Cependant, pour éviter de polluer celle-ci, et afin de regrouper tout ce qui concerne l'historisation dans cette base, la procédure stockée sera située d'un schéma « Audit » de la base principale. Commençons donc par créer ce schéma (s'il n'existe pas encore) :

```
USE ProdData
GO
CREATE SCHEMA Audit
```

Nous pouvons maintenant créer la procédure stockée qui prend 3 arguments en entrée (le nom de la table à historiser, son schéma d'appartenance – dbo étant le schéma par défaut - et le nom – optionnel - de la base de données contenant les tables d'historique) :

```
CREATE OR ALTER PROCEDURE Audit.CREATE_TRIGGER_ON_TABLE
(
	@TableName SYSNAME,
	@SchemaName SYSNAME = 'dbo',
	@HistoDatabaseName SYSNAME = null
)
AS
BEGIN
	SET NOCOUNT ON;

	DECLARE @DatabaseName SYSNAME = DB_NAME();

	-- Lorsque la base d'historique n'est pas précisée, on considère qu'il s'agit du nom de la base en cours suffixe de '_Histo'
	IF @HistoDatabaseName IS NULL
		SET @HistoDatabaseName = DB_NAME() + '_Histo';

	DECLARE @HistoTableName SYSNAME = @TableName + '_Histo';

	-- Tout d'abord on vérifie l'existence de la table donnée en argument dans le schéma donné en argument.
	DECLARE @IsNOK BIT = 0;
	DECLARE @checkSql NVARCHAR(MAX) = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.tables WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
	BEGIN
		RAISERROR (N''La table %s.%s n''''existe pas dans la base de données %s !'', 16, 1, @SchemaName, @TableName, @DatabaseName);
		SET @IsNOK = 1;
	END';
	EXEC sp_executesql
		@query = @checkSql,
		@params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName SYSNAME',
			@IsNOK = @IsNOK OUTPUT,
			@DatabaseName = @DatabaseName,
			@SchemaName = @SchemaName,
			@TableName = @TableName;
```


```
        If @IsNOK = 1
                RETURN;

        -- Puis on vérifie l'existence de la table d'historique
        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName + '.sys.tables WHERE object_id
= OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
        BEGIN
                RAISERROR (N''La table d''''historique %s.%s n''''existe pas dans la base de données %s
!'', 16, 1, @SchemaName, @TableName, @DatabaseName);
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @HistoDatabaseName,
                        @SchemaName = @SchemaName,
                        @TableName = @HistoTableName;
        If @IsNOK = 1
                RETURN;

        -- On peut maintenant commencer la préparation du script de création du trigger
        -- Pour se faire, nous allons avoir besoin de 2 informations :
        --              1 : la liste des colonnes de la table principale
        --              2 : les colonnes constituant la clé primaire (s'il y a lieu) de la table
principale
        -- On a donc recours à 2 cursors et à quelques variables pour récupérer et préparer ces
informations

        DECLARE @TrigIntoSql VARCHAR(MAX) = '';
        DECLARE @TrigInsSql VARCHAR(MAX) = '';
        DECLARE @TrigDelSql VARCHAR(MAX) = '';

        DECLARE colsCursor CURSOR FOR SELECT name FROM SYS.columns WHERE object_id =
OBJECT_ID(@SchemaName + '.' + @TableName) ORDER BY column_id
        DECLARE @colName varchar(256);

        OPEN colsCursor;

        FETCH NEXT FROM colsCursor INTO @colName;

        WHILE @@FETCH_STATUS = 0
        BEGIN
                SET @TrigIntoSql = @TrigIntoSql + '
                        , ' + @colName;
                SET @TrigInsSql = @TrigInsSql + '
                        , i.' + @colName;
                SET @TrigDelSql = @TrigDelSql + '
                        , d.' + @colName;

                FETCH NEXT FROM colsCursor INTO @colName;
        END

        CLOSE colsCursor;
        DEALLOCATE colsCursor;


        DECLARE keysCursor CURSOR FOR SELECT c.Name as ColumnName
        FROM sys.indexes i
        INNER JOIN sys.index_columns ic ON i.object_id = ic.object_id AND i.index_id = ic.index_id
        INNER JOIN sys.columns c ON ic.object_id = c.object_id and ic.column_id = c.column_id
        INNER JOIN sys.objects o ON i.object_id = o.object_id
        INNER JOIN sys.schemas sc ON o.schema_id = sc.schema_id
        WHERE i.is_primary_key = 1
                AND sc.name = @SchemaName
                AND o.name = @TableName
        ORDER BY ic.key_ordinal;

        DECLARE @hasKey BIT = 0;
        DECLARE @keyColName varchar(256);
        DECLARE @KeyJoin VARCHAR(MAX) = '';
        DECLARE @KeyJoinAnd VARCHAR(10) = ' ON ';
```

```
        DECLARE @FirstKeyColumn SYSNAME = null;

        OPEN keysCursor;

        FETCH NEXT FROM keysCursor INTO @keyColName;

        WHILE @@FETCH_STATUS = 0
        BEGIN
                SET @hasKey = 1;
                SET @KeyJoin = @KeyJoin + @KeyJoinAnd + '{0}.' + @keyColName + ' = {1}.' + @keyColName;
                SET @KeyJoinAnd = ' AND ';
                IF @FirstKeyColumn IS NULL
                        SET @FirstKeyColumn = @KeyColName;
                FETCH NEXT FROM keysCursor INTO @keyColName;
        END

        CLOSE keysCursor;
        DEALLOCATE keysCursor;

        -- Il ne reste plus qu'à écrire le script de création ou modification du trigger
        DECLARE @CreateTrigger VARCHAR(MAX) = N'CREATE OR ALTER TRIGGER ' + @SchemaName + '.TRIG_' +
@TableName + '_AuditTrigger
ON ' + @SchemaName + '.' + @TableName + '
AFTER INSERT, UPDATE, DELETE
AS BEGIN
        SET NOCOUNT ON;

        INSERT INTO ' + @HistoDatabaseName + '.' + @SchemaName + '.' + @HistoTableName + '
                (
                        CHANGE_OPERATION'
                        + @TrigIntoSql + '
                )
                SELECT
                        ''INS'''
                        + @TrigInsSql +  '
                  FROM inserted i'
         + (CASE WHEN @hasKey = 1 THEN '
                          LEFT JOIN deleted d ' + REPLACE(REPLACE(@KeyJoin, '{0}', 'i'), '{1}', 'd') +
'
                          WHERE d.' + @FirstKeyColumn + ' IS NULL'
                ELSE '
                                WHERE (SELECT COUNT(*) FROM deleted) = 0'
                END) + '
                UNION ALL
                SELECT
                        ''DEL'''
                        + @TrigDelSql + '
                  FROM deleted d'
         + (CASE WHEN @hasKey = 1 THEN '
                  LEFT JOIN inserted i ' + REPLACE(REPLACE(@KeyJoin, '{0}', 'd'), '{1}', 'i') + '
                  WHERE i.' + @FirstKeyColumn + ' IS NULL'
                ELSE '
                                WHERE (SELECT COUNT(*) FROM inserted) = 0'
                END) + '
                UNION ALL
                SELECT
                        ''UPD'''
                        + @TrigInsSql + '
                  FROM inserted i'
         + (CASE WHEN @hasKey = 1 THEN '
                  INNER JOIN deleted d ' + REPLACE(REPLACE(@KeyJoin, '{0}', 'i'), '{1}', 'd')
                ELSE '
                WHERE (SELECT COUNT(*) FROM inserted) > 0
                  AND (SELECT COUNT(*) FROM deleted) > 0'
                END) + '
END';

        -- Et, pour finir, exécuter ce script de création/mise à jour du trigger
        EXEC(@CreateTrigger);

END
```

Grâce aux 2 procédures stockées ainsi crées, il est possible et facile de mettre en œuvre l'historisation des tables existantes ou même de toute nouvelle table créée en les exécutant consécutivement (depuis la base principale) pour une même table comme dans l'exemple ci-dessous :

```
USE ProdData;
EXEC ProdData_Histo.Audit.CREATE_AUDIT_ON_TABLE Person;
EXEC Audit.CREATE_TRIGGER_ON_TABLE Person;
```

Pour rendre cette opération automatique en cas de création de table, il ne reste plus qu'à écrire le trigger de base (dans la base principale) interceptant l'événement de création de table afin d'exécuter ces 2 appels automatiquement.

Mais il n'y a pas que dans le cas de la création de table que nous devons agir. En effet, si une table de la base principale est modifiée, en particulier pour lui ajouter une colonne, modifier le type de données d'une colonne ou renommer une colonne ou même encore renommer la table elle-même ou la déplacer vers un autre schéma. Dans toutes ces situations, il est nécessaire que la base d'historique soit également modifiée pour refléter les changements. Le trigger de base de données doit donc prendre en compte toutes ces situations afin d'appeler les opérations nécessaires.

Si le traitement de mise à jour du trigger est commun à tous ces cas, il n'en n'est pas de même pour les actions à réaliser sur la table historique proprement dite. Là-encore, pour éviter d'alourdir inutilement le trigger de base de données (qu'il n'est pas aisé de mettre à jour puisqu'il faut systématiquement le supprimer avant de le recréer, contrairement aux triggers de table qui autorisent l'usage de l'instruction CREATE OR ALTER), et parce qu'il peut être utile ou nécessaire d'exécuter « manuellement » ces traitements (ou de les appeler à plusieurs reprises de plusieurs endroits différents) ou même encore parce que, pour certains tout du moins, ils ne peuvent être exécutés que dans la base courante, il est préférable d'avoir recours à des procédures stockées dédiées.

Elles sont au nombre de 3 (l'une pour mise à jour de la table historique lors d'un ALTER de la table principale, l'une pour mise à jour de la table historique en cas de renommage d'un champ de la table principale ou même renommage de la table principale elle-même, et la dernière pour le transfert d'une table dans une autre) devant toutes être accompagnées, après exécution, d'une mise à jour du trigger de la table principale pour prendre en compte la changement constaté. Nous devons donc écrire d'autres procédures stockées dédiées à ces différentes situations, à commencer par la procédure stockée permettant la mise à jour de la table historique en cas d'ALTER sur la table principale associée et que l'on place dans la base de données « historique ». Comme pour la procédure de création, nous plaçons cette procédure dans le schéma Audit de la base d'historique. Voici son script :

```
CREATE OR ALTER PROCEDURE Audit.ALTER_AUDIT_ON_TABLE
(
        @TableName SYSNAME,
        @SchemaName SYSNAME = 'dbo',
        @HistoDatabaseName SYSNAME = null
)
AS
BEGIN
        SET NOCOUNT ON;
```


```
        DECLARE @DatabaseName SYSNAME = DB_NAME();

        -- Lorsque la base d'historique n'est pas précisée, on considère qu'il s'agit du nom de la base
en cours suffixe de '_Histo'
        IF @HistoDatabaseName IS NULL
                SET @HistoDatabaseName = DB_NAME() + '_Histo';

        DECLARE @HistoTableName SYSNAME = @TableName + '_Histo';

        -- Tout d'abord on vérifie l'existence de la table donnée en argument dans le schéma donné en
argument.
        DECLARE @IsNOK BIT = 0
        DECLARE @checkSql NVARCHAR(MAX) = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.tables
WHERE object_id = OBJECT_ID(@DatabaseName ''.'' + @SchemaName + ''.'' + @TableName))
        BEGIN
                RAISERROR (N''La table %s.%s n''''existe pas dans la base de données %s !'', 16, 1,
@SchemaName, @TableName, @DatabaseName);
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @DatabaseName,
                        @SchemaName = @SchemaName,
                        @TableName = @TableName;
        If @IsNOK = 1
                RETURN;

        -- Puis on vérifie l'existence de la table d'historique
        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName + '.sys.tables WHERE object_id
= OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
        BEGIN
                RAISERROR (N''La table d''''historique %s.%s n''''existe pas dans la base de données %s
!'', 16, 1, @SchemaName, @TableName, @DatabaseName);
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @HistoDatabaseName,
                        @SchemaName = @SchemaName,
                        @TableName = @HistoTableName;
        If @IsNOK = 1
                RETURN;

        -- On peut maintenant commencer la préparation du script de création du trigger
        -- Pour se faire, nous allons avoir besoin de 2 informations :
        --              1 : la liste des colonnes de la table principale
        --              2 : les colonnes constituants la clé primaire (s'il y a lieu) de la table
principale
        -- On a donc recours à 2 cursors et à quelques variables pour récupérer et préparer ses
informations

        DECLARE @TrigIntoSql VARCHAR(MAX) = '';
        DECLARE @TrigInsSql VARCHAR(MAX) = '';
        DECLARE @TrigDelSql VARCHAR(MAX) = '';

        DECLARE   colsCursor   CURSOR   FOR   SELECT   name   FROM   SYS.columns   WHERE   object_id   =
OBJECT_ID(@SchemaName + '.' + @TableName) order by column_id
        DECLARE @colName varchar(256);

        OPEN colsCursor;

        FETCH NEXT FROM colsCursor INTO @colName;

        WHILE @@FETCH_STATUS = 0
        BEGIN
                SET @TrigIntoSql = @TrigIntoSql + '
```

```
                        , ' + @colName;
                SET @TrigInsSql = @TrigInsSql + '
                        , i.' + @colName;
                SET @TrigDelSql = @TrigDelSql + '
                        , d.' + @colName;

                FETCH NEXT FROM colsCursor INTO @colName;
        END

        CLOSE colsCursor;
        DEALLOCATE colsCursor;


        DECLARE keysCursor CURSOR FOR SELECT c.Name as ColumnName
        FROM sys.indexes i
        INNER JOIN sys.index_columns ic ON i.object_id = ic.object_id AND i.index_id = ic.index_id
        INNER JOIN sys.columns c ON ic.object_id = c.object_id and ic.column_id = c.column_id
        INNER JOIN sys.objects o ON i.object_id = o.object_id
        INNER JOIN sys.schemas sc ON o.schema_id = sc.schema_id
        WHERE i.is_primary_key = 1
                AND sc.name = @SchemaName
                AND o.name = @TableName
        ORDER BY ic.key_ordinal;

        DECLARE @hasKey BIT = 0;
        DECLARE @keyColName varchar(256);
        DECLARE @KeyJoin VARCHAR(MAX) = '';
        DECLARE @KeyJoinAnd VARCHAR(10) = ' ON ';
        DECLARE @FirstKeyColumn SYSNAME = null;

        OPEN keysCursor;

        FETCH NEXT FROM keysCursor INTO @keyColName;

        WHILE @@FETCH_STATUS = 0
        BEGIN
                SET @hasKey = 1;
                SET @KeyJoin = @KeyJoin + @KeyJoinAnd + '{0}.' + @keyColName + ' = {1}.' + @keyColName;
                SET @KeyJoinAnd = ' AND ';
                IF @FirstKeyColumn IS NULL
                        SET @FirstKeyColumn = @KeyColName;
                FETCH NEXT FROM keysCursor INTO @keyColName;
        END

        CLOSE keysCursor;
        DEALLOCATE keysCursor;

        -- Il ne reste plus qu'à écrire le script de création ou modification du trigger
        DECLARE @CreateTrigger VARCHAR(MAX) = N'CREATE OR ALTER TRIGGER ' + @SchemaName + '.TRIG_' +
@TableName + '_AuditTrigger
ON ' + @SchemaName + '.' + @TableName + '
AFTER INSERT, UPDATE, DELETE
AS BEGIN
        SET NOCOUNT ON;

        INSERT INTO ' + @HistoDatabaseName + '.' + @SchemaName + '.' + @HistoTableName + '
                (
                        CHANGE_OPERATION'
                        + @TrigIntoSql + '
                )
                SELECT
                        ''INS'''
                        + @TrigInsSql +  '
                  FROM inserted i'
         + (CASE WHEN @hasKey = 1 THEN '
                        LEFT JOIN deleted d ' + REPLACE(REPLACE(@KeyJoin, '{0}', 'i'), '{1}', 'd') +
'
                        WHERE d.' + @FirstKeyColumn + ' IS NULL'
                ELSE '
                              WHERE (SELECT COUNT(*) FROM deleted) = 0'
                END) + '
                UNION ALL
                SELECT
```

```
                        ''DEL'''
                        + @TrigDelSql + '
                  FROM deleted d'
         + (CASE WHEN @hasKey = 1 THEN '
                  LEFT JOIN inserted i ' + REPLACE(REPLACE(@KeyJoin, '{0}', 'd'), '{1}', 'i') + '
                  WHERE i.' + @FirstKeyColumn + ' IS NULL'
                ELSE '
                                WHERE (SELECT COUNT(*) FROM inserted) = 0'
                END) + '
                UNION ALL
                SELECT
                        ''UPD'''
                        + @TrigInsSql + '
                  FROM inserted i'
         + (CASE WHEN @hasKey = 1 THEN '
                  INNER JOIN deleted d ' + REPLACE(REPLACE(@KeyJoin, '{0}', 'i'), '{1}', 'd')
                ELSE '
                WHERE (SELECT COUNT(*) FROM inserted) > 0
                  AND (SELECT COUNT(*) FROM deleted) > 0'
                END) + '
END';

        -- Et, pour finir, exécuter ce script de création/mise à jour du trigger
        PRINT @CreateTrigger;
        EXEC(@CreateTrigger);
END
```

Pour les situations de renommage de colonne ou de table, la procédure stockée à utiliser, afin de conserver la cohérence côté historique, prend plusieurs paramètre en entrée permettant d'indiquer le nom d'origine de l'objet (colonne ou table) renommé, le type (COLUMN ou TABLE) de l'objet renommé, le nom de l'objet parent (le nom de la table, ou de l'index, par exemple, en cas de changement de nom de colonne), le type de l'objet parent (seul le type TABLE nous intéresse ici) le nouveau nom de l'objet renommé, le nom du schéma d'appartenance de la table (renommée ou contenant la colonne renommée), le nom de la base de donnée « principale ». Voici le script obtenu :

```
CREATE OR ALTER PROCEDURE Audit.RENAME_AUDIT_ON_TABLE
(
        @ObjectName SYSNAME,
        @ObjectType VARCHAR(25),
        @TargetObjectName SYSNAME,
        @TargetObjectType VARCHAR(25),
        @ObjectNewName SYSNAME,
        @SchemaName SYSNAME = 'dbo',
        @DatabaseName SYSNAME = null
)
AS BEGIN

        DECLARE @HistoDatabaseName SYSNAME = DB_NAME();

        IF @DatabaseName IS NULL
        BEGIN
                -- Lorsque le nom de la base de données où doit se trouver la table à historiser n'est
pas fourni, on prend la base de données à partir de laquelle la procédure stockée a été appelée
                SELECT TOP 1 @DatabaseName = DB_NAME(resource_database_id)
                        FROM sys.dm_tran_locks
                        WHERE request_session_id = @@SPID
                                AND resource_type = 'DATABASE'
                                AND request_owner_type = 'SHARED_TRANSACTION_WORKSPACE'
                        ORDER BY IIF(resource_database_id != DB_ID(), 0, 1);
        END

        DECLARE @IsNOK BIT = 0
        DECLARE @checkSql NVARCHAR(MAX);

        IF @ObjectType = 'COLUMN' AND @TargetObjectType = 'TABLE'
                BEGIN
                        -- Le renommage porte sur une colonne d'une table.
```

```
                        DECLARE @TableName SYSNAME = @TargetObjectName;
                        DECLARE @HistoTableName SYSNAME = @TargetObjectName + '_Histo';
                        DECLARE @NewColumnName SYSNAME = @ObjectNewName;
                        DECLARE @OldColumnName SYSNAME = @ObjectName;

                        -- Tout d'abord il faut s'assurer que la table historisée existe bien
                        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.tables
WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
                        BEGIN
                                RAISERROR (N''La table %s.%s n''''existe pas dans la base de données %s
!'', 16, 1, @SchemaName, @TableName, @DatabaseName);
                                SET @IsNOK = 1;
                        END';
                        EXEC sp_executesql
                                @query = @checkSql,
                                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName
SYSNAME, @TableName SYSNAME',
                                        @IsNOK = @IsNOK OUTPUT,
                                        @DatabaseName = @DatabaseName,
                                        @SchemaName = @SchemaName,
                                        @TableName = @TableName;
                        If @IsNOK = 1
                                RETURN;

                        -- Il faut ensuite vérifier que la table d'historique existe bien elle aussi
                        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName +
'.sys.tables WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
                        BEGIN
                                RAISERROR (N''La table d''''historique %s.%s n''''existe pas dans la
base de données %s !'', 16, 1, @SchemaName, @TableName, @DatabaseName);
                                SET @IsNOK = 1;
                        END';
                        EXEC sp_executesql
                                @query = @checkSql,
                                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName
SYSNAME, @TableName SYSNAME',
                                        @IsNOK = @IsNOK OUTPUT,
                                        @DatabaseName = @HistoDataBaseName,
                                        @SchemaName = @SchemaName,
                                        @TableName = @HistoTableName;
                        If @IsNOK = 1
                                RETURN;

                        -- On s'assure ensuite que la colonne existe bien dans la table historisée (sous
son nouveau nom)
                        Set @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.columns
WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName) AND name =
@ColumnName)
                        BEGIN
                                RAISERROR (N''La colonne %s n''''existe pas dans la table %s.%s de la
base de données %s !'', 16, 1, @ColumnName, @SchemaName, @TableName, @DatabaseName);
                                SET @IsNOK = 1;
                        END';
                        EXEC sp_executesql
                                @query = @checkSql,
                                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName
SYSNAME, @TableName SYSNAME, @ColumnName SYSNAME',
                                        @IsNOK = @IsNOK OUTPUT,
                                        @DatabaseName = @DatabaseName,
                                        @SchemaName = @SchemaName,
                                        @TableName = @TableName,
                                        @ColumnName = @NewColumnName;
                        If @IsNOK = 1
                                RETURN;

                        -- Et qu'elle existe également bien dans la table d'historique (sous son ancien
nom)
                        Set @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName +
'.sys.columns WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName) AND
name = @ColumnName)
                        BEGIN
                                RAISERROR (N''La colonne %s n''''existe pas dans la table d''''historique
%s.%s de la base de données %s !'', 16, 1, @ColumnName, @SchemaName, @TableName, @DatabaseName);
```

```
                                SET @IsNOK = 1;
                        END';
                        EXEC sp_executesql
                                @query = @checkSql,
                                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName
SYSNAME, @TableName SYSNAME, @ColumnName SYSNAME',
                                        @IsNOK = @IsNOK OUTPUT,
                                        @DatabaseName = @HistoDatabaseName,
                                        @SchemaName = @SchemaName,
                                        @TableName = @HistoTableName,
                                        @ColumnName = @OldColumnName;
                        If @IsNOK = 1
                                RETURN;

                        DECLARE @oldColumnFullName NVARCHAR(1035) = @HistoDatabaseName + '.' +
@SchemaName + '.' + @HistoTableName + '.' + @OldColumnName;
                        EXEC sp_rename @oldColumnFullName, @NewColumnName, 'COLUMN';
                END

        IF @ObjectType = 'TABLE'
                BEGIN
                        DECLARE @oldHistoTableName SYSNAME = @ObjectName + '_Histo';
                        DECLARE @newTableName SYSNAME = @ObjectNewName;
                        DECLARE @newHistoTableName SYSNAME = @newTableName + '_Histo';

                        -- Tout d'abord il faut s'assurer que la table historisée existe bien (sous son
nouveau nom)
                        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.tables
WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
                        BEGIN
                                RAISERROR (N''La table %s.%s n''''existe pas dans la base de données %s
!'', 16, 1, @SchemaName, @TableName, @DatabaseName);
                                SET @IsNOK = 1;
                        END';
                        EXEC sp_executesql
                                @query = @checkSql,
                                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName
SYSNAME, @TableName SYSNAME',
                                        @IsNOK = @IsNOK OUTPUT,
                                        @DatabaseName = @DatabaseName,
                                        @SchemaName = @SchemaName,
                                        @TableName = @NewTableName;
                        If @IsNOK = 1
                                RETURN;

                        -- Il faut ensuite vérifier que la table d'historique existe bien elle aussi
(sous son ancien nom)
                        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName +
'.sys.tables WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
                        BEGIN
                                RAISERROR (N''La table d''''historique %s.%s n''''existe pas dans la
base de données %s !'', 16, 1, @SchemaName, @TableName, @DatabaseName);
                                SET @IsNOK = 1;
                        END';
                        EXEC sp_executesql
                                @query = @checkSql,
                                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName
SYSNAME, @TableName SYSNAME',
                                @IsNOK = @IsNOK OUTPUT,
                                @DatabaseName = @HistoDataBaseName,
                                @SchemaName = @SchemaName,
                                @TableName = @OldHistoTableName;
                        If @IsNOK = 1
                                RETURN;

                        DECLARE @OldHistoTableFullName NVARCHAR(1035) = @HistoDatabaseName + '.' +
@SchemaName + '.' + @OldHistoTableName;
                        EXEC sp_rename @oldHistoTableFullName, @NewHistoTableName, 'OBJECT';
                END
END
```

Enfin, il nous faut gérer la procédure stockée permettant le transfert de table depuis un schéma vers un autre (avec possible besoin de création du schéma au préalable). Dans ce cas particulier, malheureusement, les informations fournies par le XML que nous pouvons intercepter depuis le trigger de base de données sont insuffisantes pour un contrôle complet de l'événement et une recopie de celui-ci dans la base historique. Mais on peut tout de même envisager une utilisation. En effet, les informations récupérables sont le nom du schéma cible, le nom de l'objet transféré, le type de l'objet transféré et la commande SQL ayant été exécutée pour réaliser le transfert. Cela dit on connait la syntaxe d'une instruction ALTER SCHEMA pour transférer une table. La seule information qu'il nous manque est le nom du schéma source, mais nous pouvons l'obtenir à moindre frais en remplaçant tous les termes connus de l'instruction à l'aide d'une instruction SELECT reposant sur une fonction système SPLIT_STRING et renseignant une variable @SourceSchemaName à partir de la commande SQL récupérée depuis le XML dans une variable @TSQLCommand.
Il suffira d'exécuter cette extraction du nom du schéma source au sein du trigger de base de données avant d'appeler la procédure stockée placée dans la base historique pour gérer le transfert de table entre schéma et dont voici le script de création :

```
CREATE OR ALTER PROCEDURE Audit.TRANSFER_AUDIT_ON_TABLE
(
	@TableName SYSNAME,
	@SourceSchemaName SYSNAME = NULL,
	@TargetSchemaName SYSNAME = NULL,
	@DatabaseName SYSNAME = NULL
)
AS BEGIN
	DECLARE @HistoDatabaseName SYSNAME = DB_NAME();
	DECLARE @HistoTableName SYSNAME = @TableName + '_Histo';

	IF @DatabaseName IS NULL
	BEGIN
		-- Lorsque le nom de la base de données où doit se trouver la table à historiser n'est pas fourni, on prend la base de données à partir de laquelle la procédure stockée a été appelée
		SELECT TOP 1 @DatabaseName = DB_NAME(resource_database_id)
			FROM sys.dm_tran_locks
			WHERE request_session_id = @@SPID
				AND resource_type = 'DATABASE'
				AND request_owner_type = 'SHARED_TRANSACTION_WORKSPACE'
			ORDER BY IIF(resource_database_id != DB_ID(), 0, 1);
	END

	-- Il faut tout d'abord vérifier que le "nouveau" schéma existe bien dans la base "principale"
	DECLARE @IsNOK BIT = 0
	DECLARE @checkSql NVARCHAR(MAX) = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.schemas WHERE name = @SchemaName)
	BEGIN
		RAISERROR (N''Le schéma %s n''''existe pas dans la base de données %s !'', 16, 1, @SchemaName, @DatabaseName);
		SET @IsNOK = 1;
	END';
	EXEC sp_executesql
		@query = @checkSql,
		@params = N'@IsNOK BIT OUTPUT, @SchemaName SYSNAME, @DatabaseName SYSNAME',
			@IsNOK = @IsNOK OUTPUT,
			@DatabaseName = @DatabaseName,
			@SchemaName = @TargetSchemaName;
	If @IsNOK = 1
		RETURN;

	-- Il faut ensuite vérifier que la table principale (à historiser) existe bien dans le "nouveau" schéma
	SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @DatabaseName + '.sys.tables WHERE object_id = OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
	BEGIN
		RAISERROR (N''La table %s n''''existe pas dans le schéma %s de la base de données %s !'', 16, 1, @TableName, @SchemaName, @DatabaseName);
```


```
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @DatabaseName,
                        @SchemaName = @TargetSchemaName,
                        @TableName = @TableName;
        If @IsNOK = 1
                RETURN;

        -- Il faut ensuite vérifier si la table d'historique existe bien dans l'ancien schéma
        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName + '.sys.tables WHERE object_id
= OBJECT_ID(@DatabaseName + ''.'' + @SchemaName + ''.'' + @TableName))
        BEGIN
                RAISERROR (N''La table d''''historique %s n''''existe pas dans le schéma %s de la base
de données %s !'', 16, 1, @TableName, @SchemaName, @DatabaseName);
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @DatabaseName SYSNAME, @SchemaName SYSNAME, @TableName
SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @DatabaseName = @HistoDatabaseName,
                        @SchemaName = @SourceSchemaName,
                        @TableName = @HistoTableName;
        If @IsNOK = 1
                RETURN;

        -- Il ne reste plus qu'à vérifier si le "nouveau" schéma existe déjà dans la base d'historique
et le créer le cas échéant.
        SET @checkSql = 'IF NOT EXISTS(SELECT * FROM ' + @HistoDatabaseName + '.sys.schemas WHERE name
= @SchemaName)
        BEGIN
                SET @IsNOK = 1;
        END';
        EXEC sp_executesql
                @query = @checkSql,
                @params = N'@IsNOK BIT OUTPUT, @SchemaName SYSNAME',
                        @IsNOK = @IsNOK OUTPUT,
                        @SchemaName = @TargetSchemaName;
        If @IsNOK = 1
        BEGIN
                EXEC ('CREATE SCHEMA ' + @TargetSchemaName);
        END

        -- Pour finir, on exécute le transfert de table depuis le schéma source vers le schéma cible
        DECLARE @AlterSchemaSql VARCHAR(MAX) = 'ALTER SCHEMA ' + @TargetSchemaName + ' TRANSFER ' +
@SourceSchemaName + '.' + @HistoTableName;
        EXEC (@alterSchemaSql);
END
```

Voilà ! Tout est prêt pour la mise en place de l'automatisation des créations/mises à jour de tables d'historique et la mise à niveau des triggers de table à partir d'un trigger de base de données qui reste le dernier maillon à écrire.
On le place dans la base « principale ». En voici le script de création :

```
CREATE TRIGGER [TRIG_TableAudit_AutoInscription]
ON DATABASE
FOR CREATE_TABLE,
    ALTER_TABLE,
        RENAME,
        ALTER_SCHEMA
AS BEGIN
        SET NOCOUNT ON;

        DECLARE @HistoDatabaseName SYSNAME = 'ProdData_Histo';
```

```
        -- On commence par récupérer le nom de la table créée ainsi que son schéma
        DECLARE @Data XML = EVENTDATA();

        --DECLARE   @DatabaseName   VARCHAR(256)   =     @data.value('(/EVENT_INSTANCE/DatabaseName)[1]',
'varchar(256)');
        DECLARE     @SchemaName     VARCHAR(256)     =     @data.value('(/EVENT_INSTANCE/SchemaName)[1]',
'varchar(256)');
        DECLARE @EventType VARCHAR(50) = @data.value('(/EVENT_INSTANCE/EventType)[1]', 'varchar(50)');
        DECLARE     @ObjectName     VARCHAR(256)     =     @data.value('(/EVENT_INSTANCE/ObjectName)[1]',
'varchar(256)');
        DECLARE @ObjectType VARCHAR(25) = @data.value('(/EVENT_INSTANCE/ObjectType)[1]', 'varchar(25)');
        DECLARE  @TargetObjectName  VARCHAR(256)  =  @data.value('(/EVENT_INSTANCE/TargetObjectName)[1]',
'varchar(256)');
        DECLARE  @TargetObjectType  VARCHAR(25)  =  @data.value('(/EVENT_INSTANCE/TargetObjectType)[1]',
'varchar(25)');
        DECLARE    @NewObjectName    VARCHAR(256)    =    @data.value('(/EVENT_INSTANCE/NewObjectName)[1]',
'varchar(256)');
        DECLARE     @TSQLCommand     VARCHAR(MAX)     =     @data.value('(/EVENT_INSTANCE/TSQLCommand)[1]',
'varchar(256)');

        IF @EventType = 'CREATE_TABLE'
        BEGIN
                -- Création de la table historique
                EXEC ProdData_Histo.Audit.CREATE_AUDIT_ON_TABLE
                                @SchemaName = @SchemaName,
                                @TableName = @ObjectName
                                --, @DatabaseName = @DatabaseName
                                ;

                -- Ajout du trigger de mise à jour de la table d'historique
                EXEC Audit.CREATE_TRIGGER_ON_TABLE
                                @SchemaName = @SchemaName,
                                @TableName = @ObjectName,
                                @HistoDatabaseName = @HistoDatabaseName;
        END

        IF @EventType = 'ALTER_TABLE'
        BEGIN
                -- Modification de la table historique
                EXEC ProdData_Histo.Audit.ALTER_AUDIT_ON_TABLE
                        @SchemaName = @SchemaName,
                        @TableName = @ObjectName
                        --, @DatabaseName = @DatabaseName
                        ;

                -- Modification du trigger de mise à jour de la table d'historique
                EXEC Audit.CREATE_TRIGGER_ON_TABLE
                                @SchemaName = @SchemaName,
                                @TableName = @ObjectName,
                                @HistoDatabaseName = @HistoDatabaseName;
        END

        IF @EventType = 'RENAME' AND @ObjectType IN ('TABLE', 'COLUMN')
        BEGIN
                -- Renommage de la table ou de la colonne
                EXEC ProdData_Histo.Audit.RENAME_AUDIT_ON_TABLE
                        --@DatabaseName = @DatabaseName,
                        @SchemaName = @SchemaName,
                        @ObjectName = @ObjectName,
                        @ObjectType = @ObjectType,
                        @TargetObjectName = @TargetObjectName,
                        @TargetObjectType = @TargetObjectType,
                        @ObjectNewName = @NewObjectName;

                -- Modification du trigger de mise à jour de la table d'historique
                EXEC Audit.CREATE_TRIGGER_ON_TABLE
                                @SchemaName = @SchemaName,
                                @TableName = @ObjectName,
                                @HistoDatabaseName = @HistoDatabaseName;
        END

        IF @EventType = 'ALTER_SCHEMA' AND @ObjectType = 'TABLE'
```

```
        BEGIN
                -- Récupération du nom du schéma source, depuis la commande SQL, que l'on place dans la
variable @SourceSchemaName
                DECLARE @SourceSchemaName SYSNAME;
                -- On récupère la position du point séparant le nom du schéma source du nom de la table
(qui doit être unique dans la syntaxe de la commande)
                DECLARE @POS INT = CHARINDEX('.', @TSQLCommand);
                IF @POS > 0
                BEGIN
                        -- Puis on supprime tout ce qui se trouve à partir de ce potentiel point (y-
compris)
                        SET @TSQLCommand = STUFF(@TSQLCommand, @POS, LEN(@TSQLCommand) - @POS + 1, '');
                        SELECT @SourceSchemaName  = RTRIM(LTRIM(value)) FROM STRING_SPLIT(@TSQLCommand,
' ')
                                WHERE LTRIM(RTRIM(value)) NOT IN ('', 'ALTER', 'SCHEMA', 'TRANSFER',
@SchemaName);
                END
                ELSE
                        -- Si le nom de schéma source n'est pas fourni dans la commande, c'est qu'il
s'agit de 'dbo'.
                        SET @SourceSchemaName = 'dbo';

                -- Exécution du transfert de table dans la base historique
                EXEC ProdData_Histo.Audit.TRANSFER_AUDIT_ON_TABLE
                        @TargetSchemaName = @SchemaName,
                        @SourceSchemaName = @SourceSchemaName,
                        --@DatabaseName = @DatabaseName,
                        @TableName = @ObjectName;

                -- Modification du trigger de mise à jour de la table d'historique
                EXEC Audit.CREATE_TRIGGER_ON_TABLE
                                @SchemaName = @SchemaName,
                                @TableName = @ObjectName,
                                @HistoDatabaseName = @HistoDatabaseName;
        END
END
```

Mise en place dans une base « à historiser »

Pour une nouvelle base de données

Nous venons de le voir, pour une mise en place dans une nouvelle base de données qui sera à historiser, la meilleure solution, avant de réaliser toute autre opération, en particulier de création de table, est d'exécuter, dans l'ordre, les actions suivantes :

1. Créer la base de données servant à gérer les tables d'historique
2. Se placer sur la base de données « historique »
3. Créer le trigger de base de données servant à créer automatiquement les triggers de tables visant à bloquer la modification et suppression sur celles-ci (TRIG_On_Table_Creation).
4. Créer le schéma Audit
5. Créer les 4 procédures stockées :
    a. Audit.CREATE_AUDIT_ON_TABLE
    b. Audit.ALTER_AUDIT_ON_TABLE
    c. Audit.RENAME_AUDIT_ON_TABLE
    d. Audit.TRANSFER_AUDIT_ON_TABLE
6. Se placer sur la base de données « principale »
7. Créer le schéma Audit dans la base de données « principale »
8. Créer la procédure stockée Audit.CREATE_TRIGGER_ON_TABLE

9. Créer le trigger de base de données visant à intercepter les événements nécessaires afin d'appeler les procédures stockées adéquates pour la mise en place automatique de l'historisation des tables de la base de données « principale.

Pour une base de données existante

Pour une base de données existante, le processus est pratiquement le même, bien que pouvant nécessiter quelques petites adaptations.

Surtout, pour mettre en place l'historique des tables existantes, il faudra exécuter manuellement, à la fin du processus présenté précédemment, et pour chaque table pour laquelle on souhaite mettre en place l'historisation, les 2 procédures stockées <Historique>.Audit.CREATE_AUDIT_ON_TABLE et <Principale>.Audit.CREATE_TRIGGER_ON_TABLE.

On peut aussi réaliser cette dernière opération au travers d'un script reposant sur un curseur bouclant sur les données de la vue système sys.tables (avec ce qu'il faut de filtres dans la clause WHERE pour ne lister que les tables « utilisateur ») et exécutant ainsi pour chaque table listée les 2 procédures.

5. Vers encore plus de traçabilité : traçabilité des mises à jour de structure d'une base de données

De la théorie…

Maintenant que l'on est en mesure de tracer efficacement et automatiquement tout changement de données apporté à l'une des tables de la base principale, il pourrait également être intéressant de se servir des triggers de base de données pour tracer tout changement de structure sur la base de données.

En effet, il arrive parfois que l'on cherche à retrouver quand a eu lieu telle ou telle modification de structure (création ou suppression de table, ajout, suppression ou modification de colonne sur une table, création ou modification ou suppression de vue, procédure stockée, fonction, schéma, type, renommages… la liste peut être longue.

Et puis souvent, en pareil cas, on aimerait bien retrouver quel était l'état de la structure (table, vue) ou du code (procédure stockée, fonction) avant modification.

Pour se faire, on peut envisager une table dédiée au stockage de ces modifications de structure qui serait stockée dans le schéma Audit de la base « historique » et qui contiendrait les colonnes nécessaires (nom de la base de données, nom du schéma, nom de la table, type d'action réalisée, script exécuté…). En somme, si l'on ne veut perdre aucunes informations liées à ces modifications, il faudrait tout bonnement intercepter, dans un trigger de base de données dédiée, les différents événements que l'on souhaite tracer et stocker les informations récupérées depuis le xml lu à cette occasion dans la fameuse table de traçabilité des modifications de structures. Les colonnes de cette table devront alors représenter toutes les combinaisons possibles de champs du XML (donc reposer sur la structure du XML pour chacun des événements interceptés). Pour plus d'informations à ce sujet voici 2 liens utiles : https://docs.microsoft.com/fr-fr/sql/t-sql/functions/eventdata-transact-sql?view=sql-server-ver15 et http://schemas.microsoft.com/sqlserver/2006/11/eventdata/

… à la pratique

Voici un exemple concret mais non exhaustif de ce qu'il est possible de faire pour assurer cette traçabilité des modifications de structure.

Pour commencer, il faut créer la table servant à enregistrer ces modifications. Nous la nommerons DB_CHANGE_LOG et la placerons dans le schéma Audit de la base de données d'historique :

```
CREATE TABLE Audit.DB_CHANGE_LOG(
        LogId INT IDENTITY(1,1) NOT NULL,
        DatabaseName VARCHAR(256) NOT NULL,
        SchemaName VARCHAR(256) NULL,
        EventType VARCHAR(50) NOT NULL,
        ObjectName VARCHAR(256) NOT NULL,
        ObjectType VARCHAR(25) NOT NULL,
        TargetObjectName VARCHAR(256) NULL,
        TargetObjectType VARCHAR(25) NULL,
        NewObjectName VARCHAR(256) NULL,
        TSqlCommand VARCHAR(max) NOT NULL,
        EventDate DATETIME NOT NULL CONSTRAINT DF_AUDIT_DB_CHANGE_LOG_EventDate DEFAULT (GETDATE()),
        LoginName   VARCHAR(256)   NOT   NULL   CONSTRAINT   DF_AUDIT_DB_CHANGE_LOG_LoginName   DEFAULT
(SUSER_NAME()),
        TS TIMESTAMP NOT NULL,
        CONSTRAINT PK_Audit_DB_CHANGE_LOG PRIMARY KEY CLUSTERED (LogId ASC)
)
```

Ensuite, il ne reste plus qu'à créer un trigger de base de données, à placer dans toute base de données pour laquelle on souhaite tracer les changements de structure et visant à alimenter cette table DB_CHANGE_LOG en réponse aux différentes actions réalisées. Pour chaque nouvelle action que l'on souhaitera, par la suite, tracer, il suffira de consulter le schéma XML correspondant dans EVENTDATA pour éventuellement récupérer des champs non encore tracés et adapter alors la procédure stockée (ci-dessus) et le trigger de base de données (ci-dessous) en conséquence.

Pour commencer, il faut créer ledit trigger dans la base de données « principale ». On pourra éventuellement, si on le souhaite, le créer dans la base de données « historique », le même code étant applicable pour toute base de données (seule l'instruction « INSERT INTO ProdData_Histo.Audit.DB_CHANGE_LOG » sera potentiellement à adapter si la base de données, le schéma ou le nom de la table à utiliser pour le stockage ne sont pas ceux utilisés ici) :

```
CREATE TRIGGER TRIG_On_DB_Change
ON DATABASE
FOR CREATE_PROCEDURE,
    ALTER_PROCEDURE,
    DROP_PROCEDURE,
    CREATE_TABLE,
    ALTER_TABLE,
    DROP_TABLE,
    CREATE_FUNCTION,
    ALTER_FUNCTION,
    DROP_FUNCTION,
    CREATE_VIEW,
    ALTER_VIEW,
    DROP_VIEW,
        CREATE_SCHEMA,
        ALTER_SCHEMA,
        DROP_SCHEMA,
        CREATE_TRIGGER,
        ALTER_TRIGGER,
        DROP_TRIGGER,
        CREATE_INDEX,
        ALTER_INDEX,
        DROP_INDEX,
        RENAME
AS
BEGIN

        SET NOCOUNT ON;

        DECLARE @data XML;
        SET @data = EVENTDATA();

        DECLARE   @DatabaseName   VARCHAR(256)   =     @data.value('(/EVENT_INSTANCE/DatabaseName)[1]',
'varchar(256)');
        DECLARE    @SchemaName    VARCHAR(256)    =    @data.value('(/EVENT_INSTANCE/SchemaName)[1]',
'varchar(256)');
        DECLARE @EventType VARCHAR(50) = @data.value('(/EVENT_INSTANCE/EventType)[1]', 'varchar(50)');
        DECLARE    @ObjectName    VARCHAR(256)    =    @data.value('(/EVENT_INSTANCE/ObjectName)[1]',
'varchar(256)');
        DECLARE @ObjectType VARCHAR(25) = @data.value('(/EVENT_INSTANCE/ObjectType)[1]', 'varchar(25)');
        DECLARE  @TargetObjectName  VARCHAR(256)  =  @data.value('(/EVENT_INSTANCE/TargetObjectName)[1]',
'varchar(256)');
        DECLARE  @TargetObjectType  VARCHAR(25)  =  @data.value('(/EVENT_INSTANCE/TargetObjectType)[1]',
'varchar(25)');
        DECLARE    @TSQLCommand    VARCHAR(MAX)    =    @data.value('(/EVENT_INSTANCE/TSQLCommand)[1]',
'varchar(max)');
        DECLARE @LoginName VARCHAR(256) = @data.value('(/EVENT_INSTANCE/LoginName)[1]', 'varchar(256)');
        DECLARE   @NewObjectName   VARCHAR(256)   =   @data.value('(/EVENT_INSTANCE/NewObjectName)[1]',
'varchar(256)');

        INSERT INTO ProdData_Histo.Audit.DB_CHANGE_LOG
        (
```


```
                DatabaseName,
                SchemaName,
                EventType,
                ObjectName,
                ObjectType,
                TSqlCommand,
                LoginName,
                TargetObjectName,
                TargetObjectType,
                NewObjectName
        ) VALUES (
                @DatabaseName,
                @SchemaName,
                @EventType,
                @ObjectName,
                @ObjectType,
                @TSQLCommand,
                @LoginName,
                @TargetObjectName,
                @TargetObjectType,
                @NewObjectName
        );
END
```

6. Conclusion

Il est tout à fait possible, à peu de frais, de mettre en place une traçabilité efficace, totale et automatisée des modifications, tant des données des tables que de la structure d'une base de données en s'appuyant sur quelques triggers et quelques procédures stockées. Trois tous petits bémols cependant :

- Il est toujours possible de désactiver ou supprimer un trigger (pour un besoin ponctuel et court de maintenance, par exemple). Dans ce cas, bien entendu, et à moins que l'on pense à le réactiver ou à le recréer une fois l'opération terminée, la politique de traçabilité peut s'en trouver compromise.
- Chaque enregistrement dans une table entraine systématiquement un enregistrement, en parallèle, dans la table d'historique ce qui augmente un petit peu le temps de traitement. C'est très négligeable et plutôt transparent pour l'utilisateur lorsque l'on travaille sur des tables de petites taille et/ou des enregistrements espacés (cas de la saisie), mais cela peut avoir un petit impact pour de très grosses tables ou lors de mises à jour en masse. À noter toutefois que dans les tables d'historique il n'y a pas lieu d'ajouter de la complexité ni des contraintes d'aucune sorte (hormis la clé primaire dont on pourrait même se passer, dans l'absolu) : pas de contrainte « check » et surtout pas de clé étrangère ni d'indexes… Si bien que l'enregistrement dans ces tables reste somme toute relativement rapide.
- L'automatisation de la gestion des historiques fait que l'on risque de ne pas penser à contrôler la santé de la base d'historique et ainsi de remplir, sans y prêter attention, le journal de logs. Il est dont préférable de mettre en place une politique efficace de gestion des logs et des sauvegardes sur la base « historique » afin de ne pas risquer le blocage des enregistrements pour faute de disque plein en particulier si l'on gère les tables d'historique dans la base de données principale.
  Il faut d'ailleurs également bien penser que la base de donnée d'historique va grossir bien plus que la base de données principale (car il y aura fatalement, au niveau de chaque entité, plusieurs enregistrements d'historique pour un même enregistrement actif de la base « principale ». Il faudra donc calibrer la taille du disque devant stocker les données de la base d'historique en conséquence et peut-être également envisager une politique d'archivage et de purge des données d'historique. En effet, il est regrettable de ne pas avoir l'historique des dernières modifications (disons les 10 dernières ou celles de moins de 2 ans, par exemple), mais est-il vraiment nécessaire de conserver toute l'historique de modification d'un enregistrement dont la dernière modification date de plusieurs années… ?

Quoi qu'il en soit, et en dépit des quelques inconvénients qu'il peut y avoir à mettre en place une telle traçabilité des modifications de données, le jeu en vaut souvent la chandelle, pour les équipes de développement ou de support, de pouvoir suivre et consulter l'historique des modifications pour des données issues des applications qu'elles doivent maintenir.

Et pour finir, il ne faut pas perdre de vue que la solution présentée ici vise à permettre une gestion entièrement automatique (avec auto-inscription des tables historisées) mais qu'il serait tout à fait possible de mettre en place quelques mécanismes supplémentaires afin de ne pas systématiquement historiser l'intégralité des tables de la base principale (certaines tables de travail n'ont effectivement pas nécessairement besoin d'être historisées).