Daten-Historisierung im Data Warehouse

Dr. Kurt Franke, debitel AG

In einem Data Warehouse ist es üblich, zeitliche Entwicklungen aller möglichen Datenarten zu betrachten. Dazu müssen die betreffenden Daten nicht nur, wie in rein operativen System, für den aktuellen Zeitpunkt, sondern für den gesamten zu betrachtenden Zeitraum vorliegen. Je nach Datenart werden diese in geeigneter Form abgelegt. In diesem Artikel werden die Erfordernisse und Möglichkeiten zur historisierten Datenhaltung betrachtet

Die für die vorliegende Betrachtung einzig relevante Unterscheidung nach Datenarten betrifft die Gültigkeit von Datensätzen entweder für einen genauen Zeitpunkt oder für einen Zeitraum von einem Zeitpunkt A bis zu einem Zeitpunkt B sowie das Faktum, ob ein Set von Daten – im Allgemeinen eine komplette Lieferung – denselben Gültigkeitszeitraum für alle Datensätze hat oder diese jeweils einen individuellen Gültigkeitszeitraum aufweisen. Für Datensätze, die sich auf einen Zeitpunkt beziehen, ist letztere Unterscheidung für die Betrachtung der Historisierung nicht relevant.

Eventbezogene Daten

Einerseits gibt es Datenarten über Vorgänge, sogenannte Events, die einfach nur einen Zeitstempel beziehungsweise mehrere davon für ein geordnetes Reporting benötigen, um damit den Zeitpunkt des Auftretens, den Zeitpunkt der Anlieferung ins Data Warehouse (DWH), den Zeitpunkt, der fürs Reporting verwendet werden soll etc., festzulegen. Daten in dieser Form benötigen kein spezielles Handling zur Historisierung und werden einfach mit den erforderlichen Zeitstempeln als neue Sätze gespeichert. Veränderungen von Sätzen gibt es hier nicht – und damit auch keine Updates. Zusammenfassende Auswertungen über einen Zeitraum sind einfach möglich, indem über eine BETWEEN-Klausel verlangt wird, dass der jeweilige Zeitstempel im zu betrachtenden Zeitraum liegt. Aus Performance-Gründen ist eine partitionierte Speicherung in geeigneter Granularität mit einem oder mehreren Zeitstempeln als Partitionierungs-Key anzustreben.

Setweise für einen Zeitraum gültige Daten

Periodisch auftretende Daten, die einen bestimmten Sachverhalt für einen vordefinierten Zeitraum beschreiben, sei es Tag, Woche, Monat oder eine beliebige andere Periode, sind eine weitere Datenart im DWH. Dazu gehören sämtliche im DWH oder in einem Vorsystem berechneten Aggregationen, darunter auch Daten über periodisches Billing, aber auch einfache Zusammenfassungen von Eventdaten über eine Periode oder einen Anfangs- beziehungsweise Endzustand für die betreffende Periode von zustandsbeschreibenden Daten. Man könnte eine Anfangs- und einen End-Zeitpunkt für die Gültigkeit eines jeden Satzes speichern, jedoch erweist sich dies als überflüssig, weil bereits durch einen der Zeitpunkte der jeweils korrespondierende über die vordefinierte Periodizität festgelegt ist. Damit genügt es, den Anfangszeitpunkt der Periode, für die die Daten gelten, mit abzulegen. Hier ist also ebenfalls kein besonderes Handling zur Historisierung erforderlich – die Datensätze werden einmal gespeichert und nie wieder geändert. Hier ist eine partitionierte Speicherung, partitioniert nach der vordefinierten Periodizität, besonders empfehlenswert. Damit werden nicht nur die Auswertungen, die im allgemeinen nur ein Subset aller existierenden Perioden betreffen, oft auch nur eine einzige, in ihrer Performance wesentlich verbessert. Dies bringt auch einen entscheidenden Vorteil für die Entsorgung nicht mehr benötigter Daten aus früheren Perioden, die hier einfach via DROP PARTITION hochperformant ohne Schreiben von Undo- und Redo-Logs entfernt werden können (Rolling-Window-Verfahren).

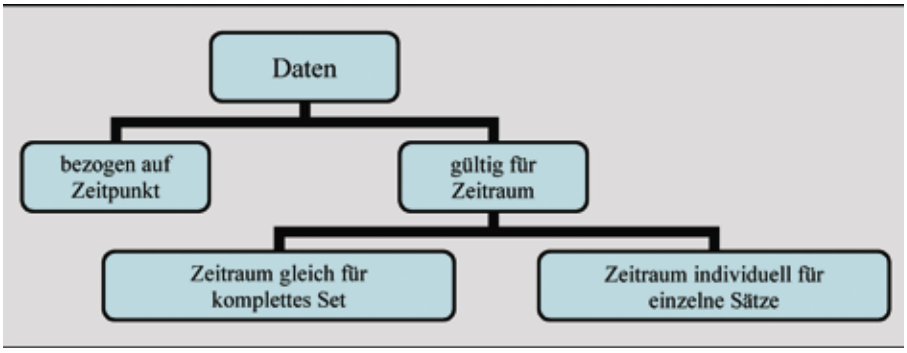

Abbildung 1: Betrachtung der Daten

Statusbeschreibende Daten

Unter diese Art von Daten fallen alle, bei denen der Zustand einer Entität beschrieben wird. Jede Änderung dieses Zustandes durch die Änderung eines oder mehrerer Attribute erfordert in einer historischen Sichtweise einen neuen Datensatz für die geänderte Ausprägung der Entität, wobei die Zusammengehörigkeit von Vorgänger und Nachfolger erkennbar sein muss,

Wir bringen die Oracle-Community zusammen

2008 DOAG Konferenz + Ausstellung

- 1. – 3. Dezember 2008
- CongressCenter Nürnberg
- Mehr als 300 Vorträge
- Prominente Keynote-Speaker aus den Oracle-Headquarters
- Demo-Grounds
- Große Ausstellung
- Schulungstag im Anschluss
- Weitere Informationen und Anmeldung unter www.doag.org/doag2008

Erleben Sie die bedeutendste Kompetenz-Präsentation des Oracle-Marktes in Zentraleuropa, erweitern Sie Ihre Netzwerke, profitieren Sie von den Erfahrungen und vom Know-how aller Teilnehmer.

und die Abgrenzung durch nicht überschneidende dicht aufeinanderfolgende Gültigkeitszeiträume erfolgt. Stammdaten wie Kundendaten etc. oder Referenzdaten sind komplett in dieser Klasse von Daten zu finden. Beachtenswert ist, dass ein Gültigkeitszeitraum durch ein abgeschlossenes Intervall beschrieben wird, damit die BETWEEN-Klausel in WHERE-Bedingungen einfach anwendbar ist. Bei Verwendung des Datentyps DATE beginnt also der Nachfolger immer genau eine Sekunde nach dem Vorgänger, beim Einsatz von TIMESTAMP ist dies je nach Genauigkeitsdefinition ein Wert bis hinunter zu einer Nanosekunde. Aus diesem Unterschied folgt, dass bei einer eventuellen Migration mit Änderung des Datentyps für die Festlegung des Gültigkeitszeitraums gegebenenfalls auch ein UPDATE aller Gültigkeitsendzeitpunkte erfolgen muss, damit die lückenlose Versions-Abfolge erhalten bleibt.

Bei dieser Art von Daten ist insbesondere zu sehen, dass Änderungen der Attribute im Allgemeinen individuell für die einzelnen Ausprägungen einer Entität erfolgen und nicht komplett für alle. Für alle in einer Lieferung eintreffenden Sätze ist jedoch der Anfangszeitpunkt ihrer Gültigkeit gleich. Letztendlich lassen sich alle derartigen Zustandsdaten als periodische Daten speichern, wenn man eine ausreichende zeitliche Granularität dafür wählt, um alle erforderlichen Zustandsänderungen erfassen zu können. Da dies jedoch immer eine Komplettspeicherung für alle existierenden Ausprägungen in jeder Periodenausprägung darstellt – unabhängig von der tatsächlichen Gültigkeit der einzelnen Sätze, wäre dies in vielen Fällen eine äußerst resourcenintensive Methode vor allem bei Plattenplatz und IO-Last.

Deshalb ist in allen Fällen, bei denen sich immer nur ein kleiner Teil aller Ausprägungen über den erforderlichen Granularitätszeitraum ändert, ein individuelles Handling der Gültigkeitszeiträume für einzelne Sätze weitaus besser geeignet. Dabei wird für jede einzelne Ausprägung der betreffenden Datenarten, also für jeden einzelnen Datensatz, ein Gültigkeitszeitraum mit abgespeichert, wobei der zeitlich letzte Satz nach der Zukunft hin offen ist. Realisiert wird dies nicht durch einen NULL-Wert für das Ende des Gültigkeitszeitraums, sondern wegen der Nutzung als Partitionierungs-Key und als Index-Key durch ein sehr großes einheitliches Datum – analog zum Jobqueue-Handling von Oracle wurde dafür der 01.01.4000 gewählt. Wird eine geänderte Variante eines Satzes angeliefert, so wird diese an die zeitliche Reihe angefügt, während bei identischer Lieferung wie zuvor keine neue Version eines Satzes erzeugt wird (Inkrement-Bildung), wodurch ein einziger Verarbeitungspfad sowohl für Inkremente als auch für Komplett-Abzüge der Entität vom Vorsystem ermöglicht wird. Eine derartige Form der Historisierung ist die Methode der Wahl für alle Formen von Stammdaten und Referenzdaten – und in einem Data Warehouse deshalb nahezu unverzichtbar. Auch für Daten mit individuellen Gültigkeitszeiträumen ist eine zeitlich partitionierte Speicherung sinnvoll, wenn die betreffende Entität eine große Datenmenge enthält. Es hat sich jedoch gezeigt, dass die Abfragen auf die einzelnen Partitionen deutlich selektiver sind, wenn das Ende des Gültigkeitszeitraum und nicht dessen Anfang die Partitionierung bestimmt. Aus einer derartigen Partitionierung ergibt sich insbesondere, dass gerade die aktuellen Versionen – und je nach Granularität der zeitlichen Partitionen auch noch einige wenige Versionen von Sätzen aus der kurz zurückliegenden Vergangenheit – in einer Partition zusammengefasst sind. Die große Masse der vergangenen Versionen von Sätzen ist jedoch in Vorgänger-Partitionen zu finden und hat damit keinen Einfluss auf die Performance bei reinen Abfragen auf die aktuelle Sicht. Mit dem Einsatzmöglichkeit der RANGE-RANGE-Partitionierung ab Oracle 11*g* und damit beiden Grenzwerten des Gültigkeitszeitraumes kann die Möglichkeit der Partionsvorherbestimmung durch den Optimizer auch für verschiedenartige Abfragen noch verbessert werden. Im Folgenden werden Anforderungen und Realisierungsmöglichkeiten dieser Methode detaillierter beschrieben.

Anforderungen an eine Statusdaten-Historisierung

Die wichtigste Anforderung ist, dass für aufeinanderfolgende Sätze der Ausprägung einer Entität zur Identifizierung der Zusammengehörigkeit eine gemeinsame ID verwaltet wird, und dass die jeweilige zeitliche Gültigkeit dicht anschließend und ohne Überschneidungen bestimmt wird.

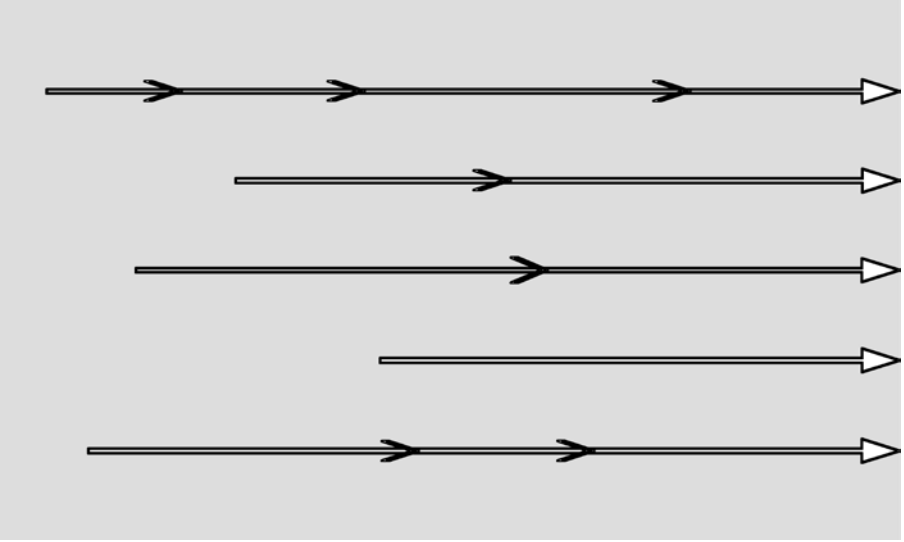

Abbildung 2: Zeitliche Gültigkeitsbereiche geänderter Versionen verschiedener Ausprägungen einer Entität

Im Prinzip reicht es wegen der Forderung des dichten Anschließens aus, nur jeweils ein Gültigkeitsdatum – entweder das Anfangs- oder das Ende-Datum – bei einem Satz zu speichern. Dies würde aber erfordern, dass zur Bestimmung des jeweils gültigen Satzes zu einem bestimmten Zeitpunkt eine komplexere Abfrage erforderlich ist. Vermieden wird dies durch eine Speicherung beider Datumswerte in einem Satz. Das hat jedoch zur Folge, dass nach zukünftigen Zeiten hin offene Sätze bei Anlieferung einer geänderten Version abgeschlossen und damit durch ein Update modifiziert werden müssen.

Zusätzlich zur versionsübergreifenden ID sollte jeder Satz für sich allein eine ID – im folgenden zur Unterscheidung als Ovid (Object-Version-ID) bezeichnet – besitzen, die dann auch für Referenzen in allen berechneten Aggregationen verwendet werden kann, wenn dort ein dem Aggregationszeitraum zugeordneter Zeitpunkt für die Dimensionen betrachtet werden soll. In diesen Fällen erübrigt sich bei Abfragen auf eine Aggregationsperiode ein Datumsvergleich in den Join-Bedingungen mit den Dimensionen, wobei durch die Reduktion der Anzahl der Join-Bedingungen auch eine Performanceverbesserung erreicht wird. Bei einer zusammenhängenden Betrachtung mehrerer Aggregationsperioden und Betrachtung der Dimensionen zu einem gemeinsamen Zeitpunkt für bessere Vergleichbarkeit ist jedoch die wegen der Erhöhung der Anzahl der Join-Bedingungen nicht ganz so performante Variante mit Referenzieren auf die versionsübergreifende ID und einen gemeinsamen Zeitpunkt für alle Perioden – empfehlenswert ist hier der Endzeitpunkt der höchsten verwendeten Aggregationsperiode, weil nur für diesen Zeitpunkt alle vorkommenden Dimensionsausprägungen auch garantiert existieren – erforderlich. Für eine effektive Suche nach Vorgänger- beziehungsweise Nachfolger-Sätzen ist die Verwaltung einer weiteren Größe sinnvoll, die sozusagen eine Link auf den Vorgänger oder Nachfolger (oder beide) in Form der jeweiligen Ovid darstellt. Für manche Abfragen ist es sinnvoll, auch noch eine fortlaufende Versionsnummer, beginnend mit 1, zu verwalten, womit dann auch die Abfrage der ersten Version eines Satzes besonders einfach wird.

Es sind also zur Speicherung derartig historisierter Daten eine Reihe zusätzlicher technischer Felder erforderlich, wovon einige später beim Einfügen eines Nachfolgesatzes modifiziert werden müssen. Damit wird im Vergleich zu nicht historisierten Daten doch ein deutlich aufwändigeres Verfahren für die Integration neuer Daten erforderlich.

Möglichkeiten zur Realisierung

An dieser Stelle werden die möglichen Methoden zur Integration neu zu historisierender Daten betrachtet, die wiederum abhängig sind vom eingesetzten Modell zur Speicherung dieser Daten. Die Speicherung selbst kann dabei in einer oder mehreren Tabellen pro Entität erfolgen, wobei der Zugriff – das betrifft ebenso die DML zur Datenintegration – auch über eine View erfolgen kann.

a) Historisierung in einer Tabelle mit direktem Zugriff

Die scheinbar einfachste Möglichkeit besteht darin, sämtliche zu historisierenden Daten einer Entität in einer Tabelle mit direktem Zugriff zu speichern. Damit sind Zugriffe für Auswertungen relativ einfach, weil alle Daten in einem Object zu finden sind. Andererseits bedeutet dies jedoch, dass die komplette Befüllung und auch spätere Updates der technischen Historisierungsfelder für jede Entität explizit in der Befülllungs-Routine codiert werden müssen. Wegen des dann auftretenden Fehlers "ORA-04091 table xxx is mutating, trigger/function may not see it" ist es nämlich nicht möglich, bei Verwendung eines Triggers auf schon existierende Sätze zuzugreifen. Auf der Entwicklerseite bedeutet diese Variante also einen vermeidbaren Overhead an Aufwand und zusätzlich noch erhöhte Fehleranfälligkeit gegenüber einer immer wieder verwendbaren automatisierten Methode. Zusätzlich muss noch erwähnt werden, dass Foreign-Key-Referenzen auf die versionsübergreifende ID direkt nicht möglich sind, weil diese ja vielfach vorkommt und deshalb kein UNIQUE-Constraint erlaubt. Dies ist nur mit einer Primary-Key-Hilfstabelle für die IDs möglich, in der diese einmalig gespeichert werden. Zur Absicherung wird dann auch ein Foreign-Key-Constraint von der Tabelle mit den historisierten Daten auf die ID-Tabelle erzeugt. Die Verwaltung zur Befüllung der ID-Tabelle bei Bedarf bedeutet eine weitere Erhöhung der Entwickler-Aufwände und der Fehleranfälligkeit. Insbesondere ist diese Methode absolut unzureichend, wenn Daten manuell gepflegt werden müssen, weil dazu ja der Mechanismus genau bekannt sein und auch exakt eingehalten werden muss.

b) Getrennte Speicherung historischer und aktueller Daten

Die beschriebene Einschränkung beim Einsatz von Triggern zur Verwaltung der technischen Historisierungsfelder lässt sich umgehen, wenn zwei Tabellen verwendet werden. In der ersten sind nur die jeweils aktuellen Versionen einer Entität gespeichert. Nur auf dieser Tabelle findet auch DML statt. Ein Trigger führt ausgelöst durch die DML die erforderlichen Aktionen auf einer zweiten Tabelle durch, die sämtliche historischen Versionen der Daten enthält. Da in der Tabelle mit den aktuellen Versionen die versionsübergreifende ID eindeutig ist, kann sie ein UNIQUE-Constraint erhalten und ist damit auch als Ziel für Foreign-Key-Constraints geeignet. Zur Absicherung wird auch ein Foreign-Key-Constraint von der zweiten Tabelle mit den historischen Daten auf die aktuelle Tabelle gelegt. Die Speicherung der aktuellen Sicht in einer separaten Tabelle bringt bei Auswertungen auf diese Sicht Performance-Vorteile gegenüber dem Zugriff auf die aktuelle Teilmenge einer Tabelle mit der kompletten Historie, sofern nicht eine Partitionierung nach dem Ende des Gültigkeitszeitraums wie oben beschrieben auf der Tabelle mit der kompletten Historie eingesetzt wird. An dieser Stelle gibt es noch eine

Unterscheidung zwischen zwei Varianten:

- *Aktuelle Version nicht in Historientabelle*
  In dieser Variante gibt es kein Einzelobjekt mit allen Versionen verschiedener Ausprägungen einer Entität. Insbesondere existiert die Verknüpfung zwischen der aktuellen Version und der Vorgängerversion nur objektübergreifend. Eine Auswertung mit Historiendaten ist nur durch Zugriff auf beide Tabellen möglich. Der Zugriff kann jedoch vereinfacht werden, indem eine View, die die beiden Tabelle mit UNION ALL verknüpft, eingesetzt wird. Dies reduziert jedoch nicht die Anzahl der Tabellen, die in Join-Bedingungen tatsächlich vorkommen, und kann deshalb in Statements mit vielen in Joins verknüpften Entitäten schon früher als beim Einsatz von Einzeltabellen für die Entitäten zu Performance-Problemen führen.

- *Aktuelle Version zusätzlich in Historientabelle*
  Wird zusätzlich auch die aktuelle Version mit in der Historientabelle abgespeichert, dann genügt es, bei einer Auswertung von Historiendaten nur diese Historientabelle zu betrachten. Damit werden die Möglichkeiten für eventuell zu viele Tabellen in einem Statement auf den Wert ohne zusätzliche technische Tabellen reduziert und das Statement ist auch ohne zusätzliche View weniger komplex, allerdings um den Preis der doppelten Datenhaltung für die aktuellen Versionen. Für im Trigger abgebildete Funktionalität wird damit die Konsistenz garantiert, allerdings kann es auch vorkommen, dass anderweitige manuelle Korrekturen wegen fehlerhafter Datenlieferungen erforderlich sind. In solchen Fällen ist bei den Ausführenden entsprechendes Know-how und entsprechende Sorgfalt unabdingbar. Der zusätzliche Speicherplatzbedarf wird nur bei großen Stammdaten-Tabellen eine merkliche Auswirkung haben, während er bei Referenzdaten in der Regel zu vernachlässigen ist.

c) Historisierung in einer Tabelle mit DML-Zugriff über View

Die Einschränkung bezüglich des Zugriffs auf bereits existierende Daten via Trigger lässt sich auch umgehen, wenn die neuen Daten in eine 1:1-View auf der Historientabelle eingefügt werden. Der Trigger wird dann als INSTEAD-OF-Trigger auf der View erzeugt und hat unbeschränkten Zugriff auf der zugrundeliegenden Tabelle. Gleichzeitig ist auch ohne doppelte Datenhaltung gewährleistet, dass die komplette Historie einschließlich der aktuellen Version in einer Tabelle gespeichert ist. Falls eine separate Sicht nur auf die aktuellen Versionen gewünscht ist, kann dies durch eine einfache weitere View mit zeitlicher Einschränkung auf die aktuell gültigen Versionen erreicht werden. Wenn bei größeren Datenmengen die beschriebene Partitionierung mit dem Ende des Gültigkeitszeitraumes eingesetzt wird, dann selektiert eine solche aktuelle Sicht auch gerade die Partition mit den aktuellen Daten und sollte damit auch über eine hinreichende Performance verfügen, solange nicht ein Zugriffspfad über globale Indices zum Einsatz kommt. Außerdem bietet sich immer noch die Möglichkeit, zur Performance-Verbesserung statt der normalen View eine Materialized View einzusetzen, die dann auch separate Indices erhalten kann.

Da in dieser Variante die versionsübergreifende ID wie in a) ebenfalls nicht eindeutig ist, wird auch hier analog eine reine Primary-Key-Hilfstabelle für die IDs eingesetzt, um Foreign-Key-Constraints zu ermöglichen. Zur Absicherung wird auch gleich ein Foreign-Key-Constraint von der Tabelle mit den historisierten Daten auf die ID-Tabelle angelegt.

Gerade diese Variante erlaubt es auch, dem Trigger eine Form zu geben, die es ermöglicht, grundsätzlich alle neuen Versionen mit INSERT einzufügen, unabhängig davon, ob es schon eine Vorgängerversion gibt oder nicht. Der Unterschied ist lediglich, dass im Falle einer existierenden Vorgängerversion eben schon eine existierende ID verwendet wird.

Flashback Data Archive ab Oracle 11*g*

Mit 11*g* steht endlich auch von Seiten der Datenbank ein Mechanismus zur Verfügung, der eine Form der Historisierung ermöglicht: Flashback Data Archive. Allerdings ist dieser Mechanismus wohl eher dafür gedacht, frühere Versionen von Sätzen in ansonsten nicht historisierenden System vorzuhalten, um Revisionssicherheit zu erreichen. Die Historisierung erfolgt dabei immer für den Zeitpunkt des SQL-Statements, das sie verursacht. Dabei wird transparent eine Tabelle im Hintergrund verwaltet, die nur die historischen Daten enthält, während die aktuelle Version in der normalen benutzerdefinierten Tabelle gespeichert ist. Damit ist die Speicherung vergleichbar mit der Realisierungsmöglichkeit, die in b) beschrieben wurde, der Zugriff erfolgt jedoch grundsätzlich über die benutzerdefinierte Tabelle, gegebenenfalls mit der „AS OF TIMESTAMP / SCN"-Klausel oder der „VERSIONS"-Klausel für Zeitpunkte der Vergangenheit.

Gerade für Arbeit mit historischen Daten gibt es jedoch einige Einschränkungen, die den Einsatz von Flashback Data Archive für die Historisierung nur in Fällen sinnvoll erscheinen lässt, bei denen diese Einschränkungen irrelevant sind. Zum einen ist es nicht möglich, zusätzlich technische Columns automatisch mitverwalten zu lassen – wenn denn sowieso schon eine eigene Verwaltung in Form eines Triggers erforderlich ist, erübrigt sich der Einsatz von Flashback Data Archive. Außerdem gibt es Einschränkungen bei den Änderungsmöglichkeiten: Beim Versuch, weitere Spalten an die benutzerdefinierte Tabelle anzuhängen, erhält man den Oracle-Fehler „ORA-55610: Invalid DDL statement on history-tracked table". Eine Einführung zusätzlicher Attribute bedeutet in diesem Falle also, dass ein kompletter Neuaufbau erfolgen muss, wobei es noch notwendig ist, die historischen Daten auf unüblichem Wege mit DBA-Privilegien unter Einbeziehung der Verwaltungs-Columns direkt von

dem Flashback-Archiv der bisherigen Tabelle in das von Oracle verwaltete Flashback-Archiv der neu erzeugten Tabelle zu kopieren, weil sonst die korrekte Timestamp-Information verloren gehen würde und damit die Historie nicht mehr vorhanden wäre.

Eine Historisierung unter Verwendung dieses Oracle-Features würde auch grundsätzlich eine Gültigkeit der neuen Version eines Satzes ab sysdate des Änderungszeitpunkts bedeuten – es ist damit aber nicht möglich, die Gültigkeitszeiträume der Vorsysteme abzubilden, weil ja jede Lieferung mit einer gewissen Zeitverzögerung erfolgt und diese Zeitverzögerung auch nicht einmal einen festen Wert hat. Eine Korrektur fehlerhaft angelieferter Daten ist bei dieser Form der Historisierung ebenfalls nicht möglich, was ja auch durchaus dem Ziel Revisionssicherheit für diesen Mechanismus entspricht.

Man sieht also, das „Flashback Data Archive“ in seiner ersten Version nur sehr eingeschränkt bis überhaupt nicht für die Historisierung von Daten geeignet ist, wenn die Historiendaten für beliebige Zugriffe via benutzerdefinierte zusätzliche technische Columns verfügbar sein müssen oder die Gültigkeitszeiträume der Vorsysteme exact abgebildet werden müssen. Auch wenn absehbar ist, dass in Zukunft eine Erweiterung um zusätzliche Attribute erforderlich sein wird, ist ein Einsatz nicht empfehlenswert. Da man ja im Regelfall eine Standardmethode für alle zu historisierenden Daten anstrebt und immer irgendwo weitere Attribute erforderlich werden, wird diese Aussage allgemeingültig.

Automatisierungsmöglichkeiten

Der Historisierung von Daten verschiedener Entitäten ist gemeinsam, dass – bei standardisiertem Einsatz der gleichen Methodik für alle Entitäten – für alle betroffenen Tabellen die gleichen technischen Columns verwaltet werden müssen. Mit Ausnahme der eigentlichen Daten-Columns, die in einem INSERT-SQL-Statement natürlich mit angegeben werden müssen, werden sich die zur Historisierung eingesetzten Trigger also sehr ähnlich sein. Daraus folgt, dass diese Trigger einfach mit einem Generator erzeugt werden können. Über Parameter werden Triggername, Tabellenname, View-Name, Eigentümer, Namen der speziellen technischen Column usw. angegeben. Der Generator liest dann die Namen der Daten-Columns aus dem Oracle Data Dictionary und fügt sie in ein Trigger-Template ein und führt via „EXECUTE IMMEDIATE“-Aufruf direkt ein CREATE TRIGGER aus. Ohne besonderen Aufwand können im Template auch noch weitere technische Columns wie CREATED_AT, MODIFIED_AT, MODIFIED_BY etc. untergebracht werden und gegebenenfalls zur Erhöhung der Flexibilität des Generators dort auch

noch via Parameter einen beliebigen Namen erhalten. Bei Einsatz der Variante mit DML-Zugriff über eine View kann auch gleich noch die View selbst erzeugt werden. Auf der Tabelle können einige unsinnige Modifikationen via direkte Statements, die die Historie zerstören würden, durch einen miterzeugten Verbots-Trigger auf der Tabelle selbst automatisch verhindert werden. Bei allen Datenarten mit hinreichend kleinem Lieferumfang, wie Referenzdaten, lässt sich somit durch die Verwendung der generierten Trigger die Historisierung nahezu ohne zusätzlichen Aufwand zur eigentlichen Integration ins DWH durchführen.

An dieser Stelle zeigt sich auch, dass es wenig sinnvoll ist, nur einzelne Columns zu historisieren, während dies für andere nicht durchgeführt wird. Im Sinne einer Automatisierung wäre dies absolut kontraproduktiv, weil ein Trigger-Generator dadurch erheblich komplexer wird – um den Preis einer geringen Reduktion von Datensätzen. Insbesondere bei Referenzdaten ist dies im Vergleich zu den sonst in einem DWH vorhandenen Datenmengen absolut irrelevant. Bei Stammdaten mit schnell wechselnden Attributen – darunter ist ein um mindestens eine Größenordnung häufigerer Wechsel als sonst zu verstehen – ist die Abspaltung solcher Attribute in eine separat historisierte Tabelle die Methode der Wahl, um unter Beibehaltung von generierten Standard-Triggern die Datenmengen zu reduzieren.

Grenzen des Einsatzes von Triggern zur Historisierung

Trigger zur Historisierung von Daten müssen vom Prinzip her auf Einzelsatz-Ebene arbeiten. Bei sehr großen zu verarbeitenden Datenmengen kann dies erhebliche Performance-Einbußen zu Folge haben. In der Regel sind davon jedoch allenfalls einige wenige Stammdaten-Tabellen betroffen, niemals die Vielzahl der Referenz-Tabellen. Dann kann es eventuell erforderlich werden, die im Trigger verwendete Historisierungsmethode für Massen-SQL-Statements explizit nachzubilden. Aber auch in diesen Fällen ist es sinnvoll, trotzdem einen Trigger zu generieren, um ggf. eine einfache manuelle Datenpflege zu ermöglichen. In der Massenverarbeitung wird dann zuerst ein Exclusivlock auf die Datentabelle angefordert. Danach werden in einer AUTONOMOUS TRANSACTION direkt auf der Tabelle liegende Trigger abgeschaltet. Die AUTONOMOUS TRANSACTION vermeidet, dass der soeben allokierte Lock sofort wieder freigegeben wird. In diesem Fall darf man aus Performance-Gründen ein besonderes Handling für Sequences nicht vergessen, bei dem in einer speziellen Funktion ausgehend vom letzten verwendeten Wert explizit hochgezählt und die Sequence danach wieder auf diesen neuen Wert synchronisiert wird. Dazu wird – wiederum jeweils in einer AUTONOMOUS TRANSACTION – der INCREMENT-Wert der Sequence auf die gerade verbrauchte Anzahl an Werten gesetzt, ein einzelner Sequence-Wert mit NEXTVAL gezogen und danach der INCREMENT-Wert wieder restauriert. Nach Abschluss aller erforderlichen DML-Statements auf der Datentabelle wird zunächst ebenfalls in einer AUTONOMOUS TRANSACTION der Trigger wieder eingeschaltet. Erst danach wird die Haupt-Transaction mit COMMIT beendet und damit auch der Lock wieder freigegeben. Damit ist sichergestellt, dass bei ausgeschalteten Triggern keine manuellen Änderungen erfolgen können, die sonst zu Inkonsistenzen führen würden.

Fazit

Von den verschiedenen, in einem DWH vorkommenden Datenarten ist insbesondere für die statusbehafteten Daten mit individuellem Gültigkeitszeitraum eine spezielle Methodik für das Handling einer Historie sinnvoll. Als optimale Speicherungsmethode hat sich die Variante mit der kompletten Speicherung in einer Tabelle und DML-Zugriff über eine 1:1-View dieser Tabelle mit automatisch generiertem INSTEAD_OF-Trigger für das Handling der technischen Spalten mit Historisierungsinformation herauskristallisiert. Bei Entitäten mit großen zu verarbeitenden Datenmengen muss parallel dazu eine Massendaten-Methode individuell codiert werden.

Kontakt:
Dr. Kurt Franke
kurt.franke@de.debitel.com

Impressum

Herausgeber:
DOAG – Deutsche ORACLE Anwendergruppe e.V.
Tempelhofer Weg 64, 12347 Berlin
Tel.: 0700 11 36 24 38
Fax: 0700 11 36 24 39
E-Mail: office@doag.org
www.doag.org

Verlag:
DOAG Dienstleistungen GmbH
Fried Saacke, Geschäftsführer
info@doag-dienstleistungen.de

Chefredakteur (VisdP):
Wolfgang Taschner, redaktion@doag.org

Chef vom Dienst (CvD):
Carmen Al-Youssef, office@doag.org

Gestaltung und Satz:
Claudia Wagner

Anzeigen:
Carmen Al-Youssef, office@doag.org
Ralf Rutkat, Agentur CrossMarketeam

Mediadaten und Preise finden Sie unter:
www.doag.org/publikationen/

Druck:
Parzeller Druck- und Mediendienstleistungen GmbH & Co. KG
www.parzeller.de