Die vierte Dimension: Temporale Objekte

Veröffentlicht in JavaSPEKTRUM 9/10/2002 Titelthema, Andreas Voigt

In vielen Anwendungsgebieten, wie z. B. der Medizin oder der Finanzwirtschaft, werden zusätzlich zu den eigentlichen Daten Zeitinformationen benötigt. Diese Informationen beschreiben meist Gültigkeitszeiträume oder Änderungszeitpunkte und sind oft von ähnlicher Bedeutung für die Gesamtanwendung wie die Daten selbst. Auf den ersten Blick erscheint eine Lösung durch die Erweiterung von Tabellen und der betroffenen SQL-Anweisungen als ausreichend. Bei genauerer Betrachtung zeigt sich jedoch, dass die entstehende Komplexität Auswirkungen auf die Anwendungslogik und Architektur hat. So würde eine Änderung der Historisierungsstrategie eine Änderung aller bestehenden SQL-Anweisungen zur Folge haben. Dies legt nahe, das Problem auf der Ebene der Persistenzschicht der Anwendung zu lösen.

Eindimensionale Historisierung

Im Folgenden werden die technischen Grundlagen temporaler Datenhaltung erörtert. Hierzu wird eine fiktive Anwendung vorgestellt, bei der Mitarbeiter sich zu ihrer jeweiligen Ansicht zu dem aktuellen Projektstatus äußern können. Ziel der Anwendung ist es, Statistikmaterial für Reviews zu liefern. Einen ersten Entwurf liefert die Datenbanktabelle in Tabelle 1 zur Erfassung der Daten.

Mitarbeiter	Status	Von	Bis
Petra	OK	01.01.02	15.01.02
Hans	OK	01.01.02	08.10.02
Hans	Schleppend	08.01.02	15.01.02
Rudi	Hervorragend	01.01.02	15.01.02

Tabelle 1: Erster Entwurf einer Datenbanktabelle zur Modellierung der Einschätzung

Die Einschätzung der Mitarbeiter zu einem bestimmten Zeitpunkt lässt sich mit der folgenden Query erfragen:

```
SELECT Mitarbeiter, Status FROM ProjektStatus WHERE Von <= 10.1.02 AND Bis > 10.1.02
```

Wie das Beispiel zeigt, enthält eine temporale Datenbank zusätzlich zu den eigentlichen Nutzdaten Zeitinformationen bezüglich der Gültigkeit der Daten. Zur Visualisierung (s. Abb. 1) kann man sich vorstellen, dass zu jedem Zeitraum eine oder mehrere virtuelle Datenbanken existieren. Betrachtet man historisierte Daten zu einem bestimmten Zeitpunkt, im Beispiel zum 10.1.02, so bezeichnet man dies als einen Snapshot.

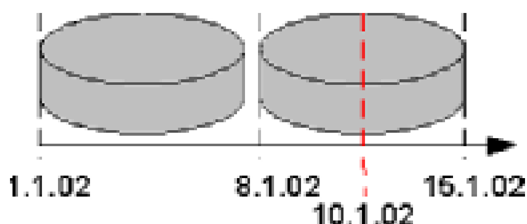

Abbildung 1: Fachliche Historie

Im Sinne historisierter Datenhaltung ist Zeit eine lineare, diskrete Reihe der kleinsten von der Anwendung unterstützten Zeitintervalle, den Chronons. Je nach Granularität kann ein Chronon eine Millisekunde, ein Tag oder auch ein Jahr betragen. Ein Zeitraum wird bestimmt durch einen Zeitpunkt, ab dem eine Information gültig ist, und einem weiteren Zeitpunkt, ab dem sie nicht mehr gültig ist. Zeiträume werden immer lückenlos gespeichert (Open/close-Semantik). Ein aktuell gültiger, d. h. offener Zeitraum wird durch einen ausgezeichneten Ende-Zeitpunkt bestimmt (z. B. NULL oder "31.12.99" in der Datenbank).

Handelt es sich bei den verwalteten Zeiträumen wie im oben vorgestellten Fall um Zeiten, die zur Anwendungsdomäne gehören, wird von einer fachlichen Historie gesprochen. Der fragliche Zeitraum wird als valid time bezeichnet. Dem eigentlichen Primärschlüssel wird ein Zeitstempel hinzugefügt. Im Beispiel wäre dies die Spalte "Von". Die Verwendung von Fremdschlüsseln auf der Datenbank wird dadurch leider unmöglich.

Bitemporale Historisierung

Die Verwendung von zwei Zeitintervallen wird als bitemporale Historisierung bezeichnet. Natürlich kann auch, je nach Anwendungsdomäne, nur eine technische Historie verwendet werden. Dies wird z. B. öfters auf Grund gesetzlicher Bestimmungen für personenbezogene Daten angewendet.

Mit Hilfe einer bitemporalen Historie kann nicht nur festgestellt werden, welche Daten für einen bestimmten Zeitpunkt fachlich gültig waren, sondern auch, welche Version einer Information zu einem fachlichen Zeitpunkt zur Verfügung stand. So ist es insbesondere möglich, die damals bekannten, aber heute geänderten Daten wiederherzustellen.

Um dies auf unser Beispiel anzuwenden, wird den Mitarbeitern die Möglichkeit eingeräumt, die vorgenommenen Einschätzungen zu revidieren. Diese Änderungen sollen aber nicht die alten Daten ersetzen, da auch diese Änderungen zu einer Statistik beitragen können. Es werden zwei weitere Zeitstempel hinzufügt, die ebenfalls ein Open/close-Intervall bilden. Statt Daten zu löschen oder zu verändern, wird eine neue Version des betroffenen Eintrags eingefügt. Zu dem Zeitpunkt der Transaktion wird ein Zeitstempel erzeugt, weshalb man diese Zeit auch als transaction time bezeichnet. In Tabelle 2 hat Rudi seine optimistische Einschätzung am 15.1.02 zurückgenommen.

Mitarbeiter	Status	Von	Bis	Editiert	Ersetzt
Petra	OK	01.01.02	15.01.02	01.01.02	31.12.99
Hans	OK	01.01.02	08.10.02	01.01.02	31.12.99
Hans	Schleppend	08.01.02	15.01.02	01.01.02	31.12.99
Rudi	Hervorragend	01.01.02	15.01.02	01.01.02	15.01.02
Rudi	OK	01.01.02	15.01.02	15.01.02	31.12.99

Tabelle 2: Zweiter Entwurf einer Datenbanktabelle zur Modellierung der Einschätzung der Mitarbeiter mit bitemporaler Historisierung

Die oben vorgestellte Abfrage wird nun erweitert, da zusätzlich zum eigentlichen Datum 10.01.02 noch der Zeitpunkt angegeben werden muss, zu dem die Daten bekannt, also technisch gültig waren, hier 15.01.02:

```
SELECT Status FROM ProjektStatus WHERE Von &LT;= 10.1.02 AND Bis &GT; 10.1.02 AND EDITIERT &LT;=15.1.02 AND ERSE
```

Um bei dem Bild virtueller Datenbanken zu bleiben, stellt man sich ein zweidimensionales Feld (s. Abb. 2) virtueller Datenbanken vor. Für einen Snapshot werden dann zwei Zeitpunkte benötigt: einer für den fraglichen fachlichen Zeitpunkt und ein weiterer technischer Zeitpunkt für die Version eines fachlichen Zeitpunktes.

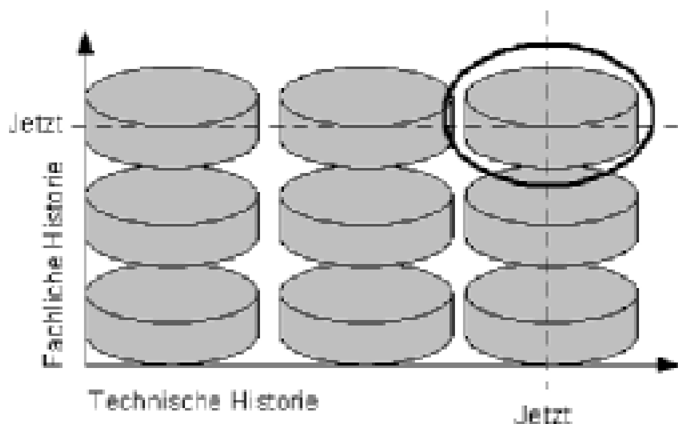

Abbildung 2: Bitemporale Historie Abbildung

Schwebe

Ein Spezialfall einer Historie, der oftmals nicht als solcher zu erkennen ist, ist die Schwebeverarbeitung. Häufig soll die Bearbeitung mehrere Objekte in einer lang laufenden fachlichen Transaktion zusammengefasst werden. Damit sich diese Bearbeitung auch über längere Zeiträume erstrecken kann, müssen Zwischenstände der Daten persistiert werden. Alternativ können zustandsbehaftete Server wie stateful Session-Beans oder HTTPSessions verwendet werden, die jedoch nicht immer stabil gegen Timeouts und Fehler sind.

Es wird zusätzlich zu der technischen Datenbanktransaktion eine langlaufende fachliche Transaktion, Schwebe genannt, benötigt. Innerhalb einer Schwebe werden die Daten mittels kurzlebiger Datenbanktransaktionen persistiert. Hiermit wird die Unterbrechbarkeit einer Anwendung ermöglicht, was hilfreich ist, um zustandslose Server zu realisieren.

Was passiert bei einer Schwebeverarbeitung? In Abbildung 3 wird eine langlaufende Transaktion dargestellt, die zum Zeitpunkt t0 gestartet und zum Zeitpunkt t4 übernommen wird. In der Schwebe wird das Bestandsobjekt o1 zum Zeitpunkt t1 verändert, das Objekt o3 zum Zeitpunkt t2 neu angelegt und das Bestandsobjekt o2 zum Zeitpunkt t3 gelöscht. Wie von einer Transaktion gefordert, sind diese Änderungen in der Bestandsdatenbank - und damit auch für andere Benutzer - erst sichtbar, wenn die langlaufende Transaktion übernommen wird. Wird die Schwebe verworfen, so werden auch alle Änderungen, die darin vorgenommen wurden, gelöscht.

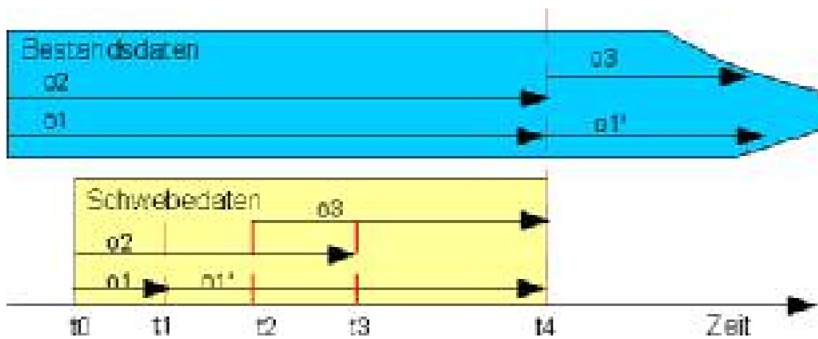

Abbildung 3: Schematische Darstellung einer Schwebe

Eine häufig verfolgte Strategie, eine Schwebe zu realisieren, ist, alle Daten in eine zweite Datenbank zu kopieren und dort zu bearbeiten. Ist die Bearbeitung erfolgreich beendet, werden die Daten in die Originaldatenbank zurück kopiert. Dies ist jedoch aufwändig und erschwert Joins zwischen Bestands- und Schwebedaten. Bei genauer Betrachtung ist eine Schwebe im Prinzip eine besondere Form der Historisierung. Durch Hinzufügen eines Markers kann man zwischen Bestandsdaten und Schwebedaten unterscheiden. Somit können Schwebe- und Bestandsdaten in der gleichen Tabelle gespeichert werden und das Vorab-Kopieren entfällt. Wird darüber hinaus eine Versionsnummer verwendet, so kann eine hierarchische Schwebe realisiert werden. Dies erlaubt die Implementierung einer Undo-Funktionalität auf Arbeitsschritt-Ebene.

Historisierung und Schwebeverarbeitung sind keine orthogonalen Konzepte, sie können sogar recht gut miteinander kombiniert werden. Eine einfache technische Historie auf Basis von Versionsnummern statt Zeiträumen kann mit einem Trick um eine mehrstufige Schwebe erweitert werden. Ein Schwebestand wird dabei als eine spezielle Variante einer Version betrachtet, die noch nicht in das Bestandsystem übernommen ist. Somit bietet es sich an, die beiden Verfahren auch auf technischer Ebene zusammenzulegen, z. B. dadurch, dass der Wertebereich der Versionsnummern in drei Teilbereiche (s. Abb. 4) aufgeteilt wird: Min-Integer bis -1: Versionsnummern historisierter Stände, Versionsnummer 0: der aktuell gültige Stand, 1 bis Max-Integer: Versionsnummern von Schwebeständen. Mit diesem Konstrukt kann die Anwendung sehr schnell auf Schwebe- und Bestandsdaten zugreifen. Auch andere Operationen, wie die Übernahme in die Bestandsdaten, sind durch einfache Kommandos realisierbar. Das Lesen der Daten wird jedoch komplexer, da hier unter anderem Löschungen in der Schwebe berücksichtigt werden müssen.

Abbildung 4: Teilbereiche der Versionsnummern

Probleme des Straigth-Forward-Ansatzes

Wie gezeigt, werden SQL-Befehle insbesondere bei einer mehrdimensionalen Historie schnell komplex. Es gilt allerdings bei einer korrekten Implementierung noch weitere Punkte zu berücksichtigen:

- Ist die verwendete Historisierungsstrategie nicht zentral gekapselt, wirken sich Änderungen auf weite Teile der Anwendung aus. Alle Entwickler müssen die verwendete Historisierung genau verstehen, um korrekte Datenbankoperationen ausführen zu können.
- Man kann nur schwer - wenn überhaupt - referenzielle Integrität aufrecht erhalten, da der Einsatz von Datenbank-Constraints nicht möglich oder aufwändig ist.
- Da je nach Strategie Daten nicht aus der Datenbank gelöscht werden, sondern deren Gültigkeitszeitraum verändert wird, können Löschkaskaden nicht auf einfache Weise realisiert werden.
- Die Zeitstempel betreffen immer die Tupelversion als Ganzes und nicht etwa einzelne Attribute. Dennoch ändert sich die Gültigkeit der einzelnen zeitveränderlichen Attribute i.d.R. asynchron. Das führt vor allem bei Tabellen mit einer Vielzahl von Attributen zu beträchtlichen Redundanzen. Verbesserungen kann man nur durch eine Restrukturierung der Tabellenstruktur erhalten.

Temporale Datenbanken

Da die direkte Verwendung von temporal angereicherten SQL-Kommandos kompliziert und fehlerträchtig ist, wird eine einfachere Möglichkeit benötigt, mit dieser Art von Daten umzugehen. Eine naheliegende Lösung (s. Abb. 5a) bieten temporale Datenbanken. Obwohl es Vorschläge gibt, SQL3 um temporale Sprachkonstrukte zu erweitern, stehen leider in absehbarer Zeit keine kommerziellen Implementierungen zur Verfügung. Die Open-Source-Software TimeDB [TimeDB] bietet jedoch einen interessanten Ansatz. Sie arbeitet auf einer relationalen Datenbank und wandelt SQL/Temporal-Befehle "on-the-fly" in SQL-Befehle um. Für eine kommerzielle Lösung wird jedoch in der Regel ein anderer Weg verfolgt werden müssen.

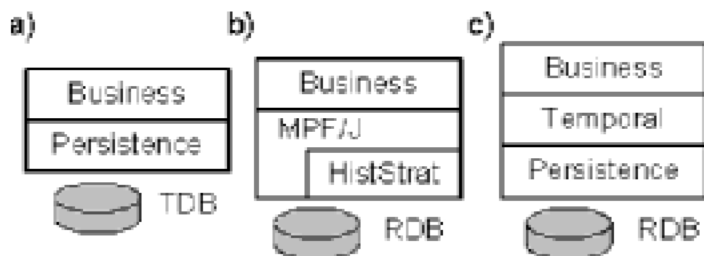

Abbildung 5: Vergleich der vorgestellten Architekturmodelle

Historisierungsschicht in Java

Für die Entwicklung mit Java bietet es sich an, eine temporale Zugriffsschicht bereitzustellen, welche die Transformation der Daten in Historien-Daten durchführt. Diese sollte oberhalb oder im Rahmen der üblichen Persistenzschicht einer Anwendung erfolgen. Eine Persistenzschicht hat die Aufgabe, Objekte transparent in relationale Daten zu transformieren und so den Zugriff auf die Datenbank zu kapseln. Diese Aufgabe wird häufig von kommerziellen O/R-Mapping Tools übernommen. Einige, wie beispielsweise MPF/J von MicroDoc [MPFJ], unterstützen historisierte Datenhaltung direkt. Mit Hilfe von MPF/J wird im Folgenden exemplarisch vorgeführt, wie eine Historisierungsstrategie realisiert werden kann. In der Praxis ist es schwierig, eine allgemeingültige Lösung anzubieten, da viele Anwendungen spezielle Anforderungen an eine Historie stellen. Deshalb delegiert MPF/J die Aufgabe, Daten um Historisierungsinformationen anzureichern, an eine Historisierungsstrategie, die in das Framework eingehängt wird (s. Abb. 5b). Die Strategie wird so weit möglich, implizit durch das Framework selbst aufgerufen, sodass keine neue API von den Entwicklern erlernt werden muss. Das folgende Beispiel verdeutlicht dies anhand einer einfachen technischen Historie.

```
public void mpfDelete(HistorizingTransaction transaction, IHistorizable object) {
        java.sql.Date now = getCreationDate();
        OneDHistorizationInformation info = (OneDHistorizationInformation)object.getHistorizationInformation();
        info.setValidTo(now);
        transaction.getMpfTransaction().update(objects);
}
```

Die Methode implementiert das "Löschen" eines Objekts aus der Datenbank. Dabei wird der Gültigkeitszeitraum des Objekts geändert und ein Update auf der Datenbank ausgeführt.

Die Strategie enthält den für die temporale Transformation nötigen Code und kennt auch die für die Historisierung nötigen Attribute. Das Datenbank-Mapping historisierter Objekte sowie die zugehörigen Tabellen werden automatisch um die für die Strategie spezifischen Attribute angereichert. Damit werden alle die Historie betreffenden Aspekte in der Strategie gekapselt, mit der Folge, dass dem Businesscode die temporalen Aspekte der Daten weitgehend verborgen bleiben. O/R Mapping Tools mit Unterstützung zur Historisierung sind aber leider relativ selten. In diesem Fall sollte man eine weitere Abstraktionsschicht (s. Abb. 5c) entwickeln, welche die Historisierung emuliert. Hierbei treten jedoch folgende Schwierigkeiten auf:

- Der Primärschlüssel wird um bestimmte Teile der Historisierungsdaten erweitert. Dies erschwert den Einsatz von Datenbank-Constraints sowie das Caching der Objekte, da dies typischerweise mit dem Primärschlüssel verbunden ist.
- "Read on demand", also das dynamische Nachlesen von Objekten an Hand von Schlüsselbeziehungen, wird erschwert bis unmöglich gemacht. Es wird eine neue API notwendig, die den Wartungs- und Lernaufwand erhöht.

Fazit

In bestimmten Anwendungsgebieten ist eine temporale Datenhaltung notwendig. Obwohl es sich um ein wissenschaftlich gründlich erforschtes Gebiet handelt (siehe auch [Sno99] und [TimeC]), stehen in absehbarer Zeit keine temporalen Datenbanken für den kommerziellen Einsatz zur Verfügung. Eine Emulation mittels SQL ist möglich und wird in vielen existierenden Anwendungen erfolgreich eingesetzt, hält aber eine Reihe von Fallstricken bereit. Daher sollte man auf jeden Fall eine Applikationsschicht verwenden, welche die Historisierungsmechanismen so weit wie möglich kapselt. Die angenehmste Art, dies zu tun, ist die Verwendung eines O/R-Mapping-Produkts mit Historisierungsunterstützung.

In diesem Artikel konnte das Thema temporaler Datenbanken nicht erschöpfend erörtert werden. Neben den betrachteten Snapshots bieten temporale Datenbanken gerade auch die Möglichkeit, mit den Zeiträumen selbst zu arbeiten. Zudem können bei rückwirkenden Veränderungen fachlicher Daten sehr spezielle Probleme auftreten. Die nachfolgenden Literaturangaben enthalten hierzu weitere Informationen.

Literatur

- [MPFJ] O/R-Mapping Tool MPF/J von MicroDoc, http://www.microdoc.de
- [Sno99] R. T. Snodgrass, Developing Time-Oriented Database Applications in SQL, Morgan Kaufmann Publishers, Inc., 1999
- [TimeC] TimeCenter, http://www.cs.auc.dk/TimeCenter [TimeDB] Open-Source-Software TimeDB, http://www.timeconsult.com

Exkurs O/R-Mapping-Tools

Wenn Objekte in relationalen Datenbanken abgelegt werden sollen, so ergeben sich semantische Probleme, die unter der Bezeichnung impedance mismatch bekannt sind. Während in Java Referenzen durch Zeiger aufgelöst werden, stehen in relationalen Datenbanken Fremdschlüssel zur Verfügung. Objekte zeichnen sich durch eine verdeckte Identität aus, die in einer Datenbank durch Primärschlüssel explizit gemacht wird. Das Konzept der Vererbung muss emuliert werden. Hinzu kommen häufig Anforderungen in den Bereichen Performance und Konfigurierbarkeit.

Andreas Voigt arbeitet als Softwareentwickler und Architekt bei MicroDoc Computersysteme GmbH in München in den Bereichen J2EE und Persistenz. E-Mail: avo@microdoc.de.