Kontrolliertes Schema-Evolutionsmanagement für NoSQL-Datenbanksysteme

Uta Störl[1], Meike Klettke[2], Stefanie Scherzinger[3]

[1] Hochschule Darmstadt, uta.stoerl@h-da.de
[2] Universität Rostock, meike.klettke@uni-rostock.de
[3] OTH Regensburg, stefanie.scherzinger@oth-regensburg.de

Zusammenfassung. In der agilen Entwicklung von Anwendungen werden neue Software-Versionen häufig und regelmäßig veröffentlicht. Relationale Datenbanksysteme mit ihrem rigiden Schema-Management werden dabei oft als unflexibel empfunden. Schemalose NoSQL-Datenbanksysteme bieten zwar die nötige Flexibilität, unterstützen aber kein systematisches Release- und Schema-Evolutionsmanagement.
Dieser Artikel stellt entsprechende Konzepte vor: Schema-Evolutionsschritte werden deklarativ spezifiziert, ihre Umsetzung erfolgt für die Anwendung transparent *eager* oder *lazy*. Während eine *eager* Migration sämtliche Datensätze erfasst, werden *lazy* persistierte Objekte nur bei Zugriff durch die Anwendung aktualisiert. Wir diskutieren eine effiziente *lazy* Migration selbst für den Fall, dass eine Migration über mehrere Evolutionsschritte und mehrere persistierte Objekte hinweg erfolgt.

1 Einführung

NoSQL-Datenbanksysteme werden in der Anwendungsentwicklung nicht nur bei sehr großen Datenmengen eingesetzt: Die Flexibilität in der Verwaltung heterogen strukturierter Daten macht NoSQL-DBMS gerade in der agilen Entwicklung attraktiv [5]. Das Schema wird typischerweise in der Anwendungsschicht mit Hilfe von Objekt-NoSQL Mapper Bibliotheken deklariert. Diese unterstützen mitunter auch weitere Aufgaben des Schema-Managements, wie etwa die *lazy* Migration von vorhandenen Daten im Produktionssystem [11]. Letztlich stellen Mapper aber nur eine Programmierschnittstelle bereit, die Ausimplementierung bleibt Aufgabe der Entwickler. Während sich Objekt-NoSQL Mapper in der Entwickler-Community großer Beliebtheit erfreuen, findet aus Sicht der Datenbank-Community eine gravierende Schichtverletzung statt.

Eine Schichtverschiebung des Schema-Managements aus der Anwendung in die Datenbank ist (nicht nur aus Gründen der Performance) wünschenswert. Das NoSQL-DBMS F1 [6] ist ein Schritt in diese Richtung: F1 verwaltet ein relationales Schema und implementiert ein rigides Protokoll, um hochfrequent

 In: R. Bergmann, S. Görg, G. Müller (Eds.): Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB. Trier, Germany, 7.-9. October 2015, published at http://ceur-ws.org

Schemaänderungen in einem verteilten System zu propagieren. Schemaänderungen werden hier zwar asynchron, aber *eager* ausgeführt.

Database-as-a-Service Kunden sind allerdings sehr daran interessiert, unnötige (kostenpflichtige) Lese- und Schreiboperationen gegen die Datenbank zu vermeiden. Das macht eine *lazy* Datenmigration besonders interessant, da persistierte Objekte nur dann migriert werden, wenn die Anwendung auch auf sie zugreift.

KVolve [7] vollzieht *lazy* Schemaänderungen in NoSQL-DBMS mit einem nachweislich niedrigen Overhead. Allerdings werden nur einfache Operationen unterstützt, wie das Hinzufügen und Entfernen von Attributen. Da die meisten NoSQL-DBMS keine Join-Operationen unterstützen, stellen Denormalisierungsoperationen, und damit komplexere Schema-Änderungen wie `copy` oder `move` Operationen, wichtige Schema-Evolutionsschritte dar.

Unsere deklarative Evolutionssprache aus [8] unterstützt entsprechend das Kopieren von Attributen zwischen persistierten Objekten. Wir zeigen in diesem Artikel, dass sich dadurch neue Herausforderungen an die Korrektheit einer *lazy* Migration stellen (Kapitel 2). In Kapitel 3 präsentierten wir das *Darwin* Projekt[4] mit der *lazy* Implementierung unserer Evolutionssprache. Die Zusammenfassung und ein Ausblick auf weitere Vorhaben folgen am Ende des Artikels.

2 Lazy Migration

Bei der *lazy* Migration wird ein Entity (d.h. ein persistiertes Objekt) erst zum Zeitpunkt seiner Verwendung in das aktuelle Schema migriert. Dabei bleibt die Datenbank für die Anwendung verfügbar. Aus Sicht der Anwendung muss transparent bleiben, ob die Daten *eager* oder *lazy* migriert werden; das stellt eine Herausforderung bei der Entwicklung von *lazy* Migrationsprotokollen dar.

Beispiel: Abbildung 1 zeigt die Daten eines Online-Rollenspiels über mehrere Versionen der Anwendung hinweg. In der NoSQL-DB werden Player und ihre Missionen persistiert. Das Schema entwickelt sich mit der Anwendung, so wird in Version 2 ein neues Attribut SCORE zur Klasse Player hinzugefügt. Bei einer *lazy* Migration werden persistierte Entities nicht unmittelbar bei der Veröffentlichung einer neuen Anwendungsversion aktualisiert. Erst wenn Player Lisa von Version 2 der Anwendung geladen wird, erfolgt das Hinzufügen des Attributes SCORE. Beim Übergang zu Schema-Version 3 soll das Attribut SCORE von der Klasse Player zur Klasse Mission kopiert[5] werden. Diese Operation wird für Mission 100 erst dann ausgeführt, wenn diese in die Anwendung geladen wird.

Die analoge Vorgehensweise führt bei Mission 101 zu einem inkorrekten Ergebnis: In Abbildung 1 wird die `copy` Operation mit einer noch nicht migrierten Version von Player Bart ausgeführt. Dementsprechend wird kein SCORE-Attribut kopiert. Das geladene Objekt unterscheidet sich von dem Objekt, das

[4] In einer früheren Implementierung wurde unsere Sprache aus [8] in der *Cleager* Konsole *eager* mit Hilfe von MapReduce Prozessen umgesetzt [9].

[5] Wie in Abbildung 1 zu sehen, erfolgt die Auswahl der *target* Entites bei der `copy` Operation durch die Angabe einer geeigneten `where`-Klausel (analoges gilt für `move`).

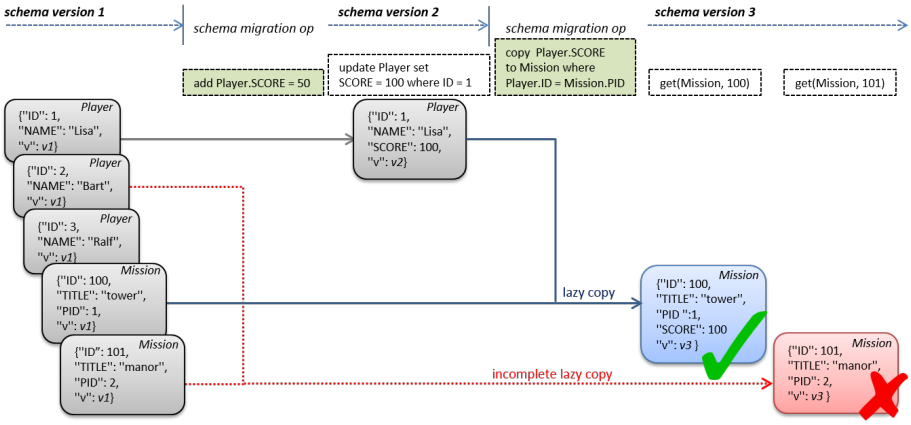

Abb. 1. Mission 100 wird *lazy* migriert, indem das SCORE-Attribut des Spielers kopiert wird. Bei Mission 101 führt diese Vorgehensweise zu einem inkorrekten Ergebnis.

durch eine *eager* Migration geladen worden wäre. Wenn mehrere Evolutionsschritte *lazy* nachzuvollziehen sind und mehr als ein Entity an der Migration beteiligt ist (etwa bei `copy` oder `move` Operationen), stellen sich Herausforderungen an die Korrektheit einer *lazy* Migration.

Abbildung 2 zeigt eine korrekte, zweistufige Migration von Mission 101, bei der Player Bart zunächst migriert wird, bevor sein SCORE kopiert wird.

Kaskadierender Implementierungsansatz: Ein erster Ansatz für die korrekte Ausführung der *lazy* Migration basiert auf folgender Vorgehensweise: Bei einer `copy` oder `move` Operation werden alle korrespondierenden *source* bzw. *target* Entities, die in der gleichen oder einer früheren Version im Vergleich zum zu migrierenden Entity vorliegen, ebenfalls in die aktuelle Version des Entity migriert. Sofern dabei eine weitere `copy` oder `move` Operation ausgeführt werden muss, wird diese analog durchgeführt und ggf. rekursiv fortgesetzt. Damit wird nachträglich der Zustand einer *eager* Migration für die betroffenen Entities sichergestellt.

Dieser *kaskadierende* Ansatz stellt die Korrektheit der *lazy* Migration sicher, führt allerdings dazu, dass beim Laden eines einzelnen Entity ggf. weitere, unbeteiligte Entities migriert werden, was zu Einbußen in der Laufzeit führt. Im Folgenden skizzieren wir erste Ideen für die Optimierung der *lazy* Migration.

Optimierungsansätze: Bei der *kaskadierenden* Implementierung werden bei der Migration von Entities, die *source* einer `copy` oder `move` Operation sind, auch die *target* Entities (kaskadierend) migriert, da sonst ggf. die Informationen der *source* Entities später nicht mehr zur Verfügung stehen. Sind hingegen alte Versionen der Entities verfügbar (wie in vielen NoSQL-DBMS implementiert), kann die Migration der *target* Entities *lazy* ausgeführt werden, also erst beim Zugriff. Dies reduziert die Anzahl der (zu einem Zeitpunkt) zu migrierenden Entities.

Bei einer *lazy* Migration liegen Entities, die über längere Zeit nicht verwendet wurden, in einer älteren Version vor (Version i). Werden diese von der Anwen-

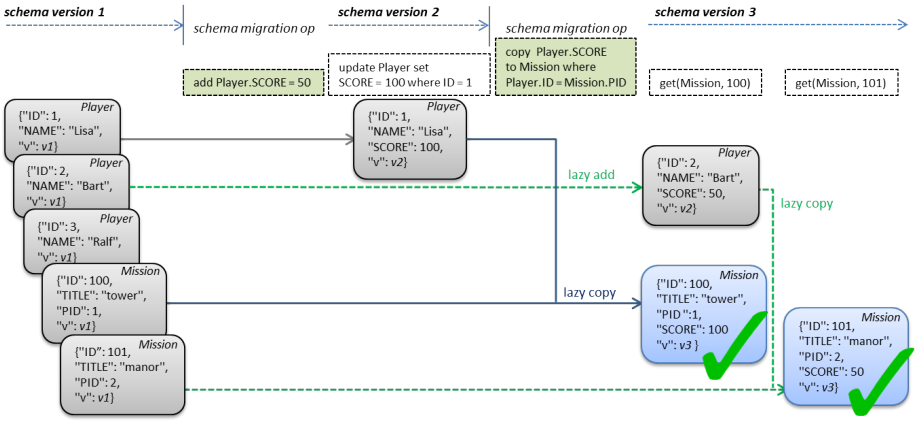

Abb. 2. Korrekte Ausführung der *lazy* Migration von Mission 101.

dung gelesen, dann erfolgt die Migration in die aktuelle Version $(i+x)$. Über der *Folge von Update-Operationen* $u_{i+x}(u_{i+x-1}(..(u_{i+1}(entity_i))))$ sind äquivalente Zusammenfassungen möglich [1]. In der NoSQL-DB wird dann nur das Ergebnis der Migration (das Entity in der Version $i + x$) persistiert. Entities, die durch Zwischenschritte entstanden sind, werden nicht dauerhaft gespeichert, sodass die Anzahl der Schreiboperationen erheblich reduziert werden kann.

In [10] präsentieren wir einen Ansatz, der die Migration durch Datalog-Regeln spezifiziert. Eine inkrementelle top-down Auswertung stellt sicher, dass die Ergebnisse einer *lazy* Migration aus Sicht des Anwendungsprogramms mit dem Ergebnis übereinstimmt, das bei der Durchführung der *eager* Migration (bzw. der äquivalenten bottom-up Auswertung) entsteht.

3 Schema-Evolutionsmanagement mit *Darwin*

In [3] wurden als Anforderungen für eine Schema-Management-Komponente die *Definition eines Schemas*, die *Validierung von Entities* gegen ein Schema sowie die Unterstützung der *Schema-Evolution inklusive Datenmigration* definiert. Die dort vorgeschlagene Schema-Management-Komponente wurde inzwischen prototypisch implementiert: *Darwin* ist eine Schema-Management-Komponente, die zwischen der Applikation bzw. dem Objekt-NoSQL Mapper und dem NoSQL-DBMS angesiedelt ist und die oben stehenden Funktionalitäten unterstützt.

Das Schema wird als JSON-Schema [2] gespeichert. Damit lassen sich sowohl Schemata von Dokumentenorientierten als auch Column-Family-Datenbanksystemen verwalten. Aktuell unterstützt *Darwin* die NoSQL-DBMS MongoDB und Couchbase. Durch die bereitgestellte abstrakte Datenbank-Schnittstelle ist es aber einfach möglich, weitere DBMS anzubinden.

Die Schema-Evolutionsoperationen können in *Darwin* direkt auf einer Konsole (CLI) eingegeben oder über eine Web-Applikation generiert werden. Die Migration der Daten erfolgt *eager* oder *lazy*. *Darwin* ist damit die erste uns bekann-

te Schema-Management-Komponente für NoSQL-DBMS, die ein kontrolliertes Schema-Management für NoSQL-DBMS (inklusive `copy` und `move` Operationen) und *lazy* Migration unterstützt.

4 Zusammenfassung und Ausblick

In der vorgestellten Schema-Management-Komponente werden verschiedene Datenbanktechniken für NoSQL-Datenbanksysteme eingesetzt, die Schema-Evolution in hochverfügbaren Anwendungen orchestrieren:

- Eine deklarative Sprache zur Schemaevolution
- Definition der Semantik der Datenmigrations-Operationen über Datalog
- Versionierung von Daten zur Konsistenzsicherung bei *lazy* Migration

Es wurden weitere Datenbanktechniken für NoSQL-Daten adaptiert, wie die Schema-Extraktion aus vorhandenen Datensätzen über Strukturgraphen [4]. Die Integration dieser Implementierung in *Darwin* ist einer der nächsten Schritte.

Um die Schema-Management-Komponente komfortabler für den Anwendungsentwickler zu gestalten, ist die Entwicklung eines IDE Plugins geplant, das bei Veränderungen an der Klassenstruktur die korrespondierenden Schema-Evolutionsoperationen automatisch generiert.

Danksagung: Wir danken den Studierenden O. Haller, T. Landmann, T. Lehwalder, D. Müller, H. Nkwinchu, M. Richter und M. Shenavai der Hochschule Darmstadt für die Implementierung von *Darwin*.

Literatur

1. M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Relational and XML Data Exchange*. Synthesis Lectures on Data Management. Morgan & Claypool, 2010.
2. JSON Schema Community. *JSON Schema*, June 2015. `http://json-schema.org`.
3. M. Klettke, S. Scherzinger, and U. Störl. "Datenbanken ohne Schema? - Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schema-flexiblen NoSQL-Datenbanksystemen". *Datenbank-Spektrum*, 14(2), 2014.
4. M. Klettke, U. Störl, and S. Scherzinger. "Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores". In *Proc. BTW'15*, 2015.
5. Z. H. Liu and D. Gawlick. "Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL". In *CIDR'15*, 2015.
6. I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. "Online, Asynchronous Schema Change in F1". In *Proc. VLDB'13*, 2013.
7. K. Saur, T. Dumitra, and M. Hicks. "Evolving NoSQL Databases without Downtime". Technical report, University of Maryland, College Park, Apr. 2015. `http://www.cs.umd.edu/~ksaur/pubs/kvolve-submitted.pdf`.
8. S. Scherzinger, M. Klettke, and U. Störl. "Managing Schema Evolution in NoSQL Data Stores". *Proc. DBPL'13*, arXiv:1308.0514 [cs.DB], 2013.
9. S. Scherzinger, M. Klettke, and U. Störl. "Cleager: Eager Schema Evolution in NoSQL Document Stores". In *Proc. BTW'15*, 2015.
10. S. Scherzinger, U. Störl, and M. Klettke. "A Datalog-based Protocol for Lazy Data Migration in Agile NoSQL Application Development". In *Proc. DBPL'15*, 2015.
11. U. Störl, T. Hauff, M. Klettke, and S. Scherzinger. "Schemaless NoSQL Data Stores Object-NoSQL Mappers to the Rescue?". In *Proc. BTW'15*, 2015.