Data evolution and migration strategies for NoSQL databases

Mark Lukas Möller
Institut für Informatik
Universität Rostock
mark.moeller2@uni-rostock.de

ABSTRACT

This article presents various data migration strategies for NoSQL databases, especially for document stores, and evaluates them in term of their efficiency. Some strategies do not execute the migration operation immediately but migrate the data on-demand (lazy and hybrid migration). The different migration strategies are classified with the use of four dimensions and will be compared to each other. The overall aim of the project is the development of an advisor which proposes an optimal migration strategy based on metrics. Various optimization goals are to be taken into account. The project Darwin is presented, which is implementing the techniques described in this article.

30^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.

Datenevolutions- und Migrationsstrategien in NoSQL-Datenbanken

Mark Lukas Möller
Institut für Informatik
Universität Rostock
mark.moeller2@uni-rostock.de

KURZFASSUNG

Die dauerhafte Verfügbarkeit von Daten stellt eine der Kernanforderungen an NoSQL-Datenbanksysteme dar.

In diesem Artikel werden verschiedene Datenmigrationsstrategien im Kontext von NoSQL-Datenbanken, insbesondere für Document Stores, vorgestellt und hinsichtlich ihrer Effizienz und ihres Einsatzspektrums bewertet. Dabei werden Strategien vorgestellt, die die Migrationsoperation nicht sofort ausführen, sondern die Daten on-demand migrieren (*Lazy Migration*). Die Migrationsvarianten werden anhand von vier Dimensionen der Datenmigration von NoSQL-Datenbanken verglichen und bewertet. Ziel ist es, einen Advisor zu entwickeln, der auf Basis von Metriken eine optimale Migrationsstrategie vorschlägt. Dabei sollen verschiedene Optimierungsziele berücksichtigt werden. Das Projekt DARWIN wird vorgestellt, in dem die in diesem Artikel genannten Techniken implementiert werden sollen.

Stichworte

NoSQL, Datenmigration, Datenevolution, Advisor, Kostenmodelle, Metriken

1. EINFÜHRUNG

Software unterliegt während ihres Entwicklungs- und Lebenszyklus nicht nur Änderungen in ihrer Logik, sondern auch Änderungen in der Persistenzschicht von Daten. Häufig kommen zur dauerhaften Speicherung von Daten *Datenbankmanagementsysteme* (DBMS) zum Einsatz, die für den operationalen oder den analytischen Einsatz optimiert sind. Datenbanken und Relationen in DBMS haben üblicherweise während ihrer Laufzeit ein definiertes Schema, das während der Laufzeit der Datenbank möglichst unverändert bleibt. Zwar unterstützen DBMS schemamodifizierende Operationen wie das Hinzufügen, Umbenennen oder Löschen einer Spalte, komplexe Evolutions- und Migrationsoperationen wie das Verschieben oder Kopieren eines Attributes einer Tabelle zu einer anderen Tabelle werden dagegen nur marginal unterstützt, indem die Schemamodifikation und die Migration der Daten durch getrennte Schritte und durch Ausformulierung der Anfragen durch den Datenbankadministrator erfolgt.

In der Softwareentwicklung mit einem Entwicklungsparadigma wie der agilen Anwendungsentwicklung ist es üblich, dass bereits entwickelte Programmteile überarbeitet werden [12]. Jim Highsmith beschrieb Agilität mit *„Deliver quickly. Change quickly. Change often“* [2]. Entsprechend ändern sich auch die Strukturen auf der Persistenzebene für Daten, bei der häufig relationale DBMS eingesetzt werden. Bei Änderungen auf dieser Ebene muss das Datenschema entsprechend evolutioniert und die Daten migriert werden [12].

Für die Realisierung einer effizienten Migration, sind Operationen notwendig, die Schemaevolutionen berücksichtigen. Daten bei jeder Schemaänderung direkt zu migrieren funktioniert, wird aber bei einer Vielzahl an Schemaänderungsoperationen und bei hohem Datenvolumen sehr teuer. Insbesondere wenn laufende Software mit vielen Daten in einer Datenbank durch eine neue Software ersetzt werden soll, die grundlegende Änderungen am Schema durchführt, sollen effiziente Operationen sicherstellen, dass keine oder nur eine minimale Ausfallzeit durch blockierende Migrationsoperationen eintritt.

Im Rahmen dieses Artikels wird vorgestellt, wie sich einzelne Migrationsoperationen realisieren lassen. Dabei werden als Datenbanksysteme insbesondere NoSQL-Document-Stores wie MongoDB verwendet, da diese im Gegensatz zu relationalen Datenbanken die für die Migration notwendige Schemaflexibilität bieten und daher bei der Lösung der genannten Probleme unterstützen. Es werden die vier Dimensionen der *Migrationszeit*, der *Art der Operationsausführung*, der *Datenmenge* und des *Migrationsortes* aus [9] verwendet, Migrationsstrategien im Rahmen dieser Dimension erläutert und bewertet. Es wird das Rahmenprojekt DARWIN vorgestellt, in welchem Migrationsoperationen auf NoSQL-Datenbanken implementiert werden.

Langfristiges Ziel ist die Entwicklung eines *Advisors* für DARWIN, der anhand von Informationen über das System automatisiert die günstigste Migrationsstrategie vorschlägt. Die Grundlagen des Advisors, welche Informationen dafür notwendig sind und welche Strategie als die Günstigste gilt, wird erläutert.

2. GRUNDLAGEN VON NOSQL-STORES

Zunächst werden grundlegende Konzepte von NoSQL-Datenbanken eingeführt und mit *relationalen Datenbanken* (RDBMS) verglichen. Nachfolgend wird das Datenmodell im Kontext von Document Stores vorgestellt, da diese haupt-

30^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.

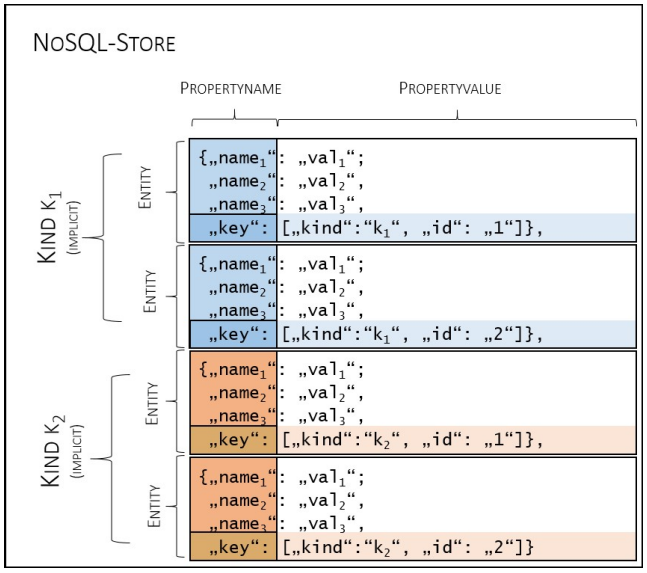

Abbildung 1: Aufbau von NoSQL-Datenbanken

sächlich im Rahmen des Artikels und der Forschungsfragestellung verwendet werden. NoSQL-Datenbanken zeichnen sich im Allgemeinen und im Gegensatz zu RDBMS durch eine *Schemalosigkeit* aus, sodass Daten ohne vorher definiertes Schema flexibel gespeichert werden können [8].[1] Schemainformationen stecken implizit in den Daten und können durch Schemaextraktion ermittelt werden.

Datenobjekte in NoSQL-Datenbanken heißen *Entitäten* und sind mit einem Tupel in einer RDBMS-Tabelle vergleichbar. Entitäten bestehen aus *Properties*, die aus einem *Property-Name* und einer *Property-Value* bestehen und eine Analogie zu Attributnamen und Attributwerten darstellen [10]. Im Gegensatz zur relationalen Speicherungsvariante[2] können die Values von Properties skalar oder mehrwertig (z.B. als Array) auftreten oder sogar andere Entitäten beinhalten (vgl. [6]). Entitäten mit dem gleichen inpliziten Schema gehören zu einem *Kind* (dt. *Art*), welches mit einer Tabelle in einem RDBMS vergleichbar ist und können einen Schlüssel besitzen. Der Schlüssel ist eindeutig und besteht aus einem Tupel, das einerseits den Namen des Kinds und andererseits eine ID enthält. Die konzeptionellen Bestandteile von NoSQL-Stores sind in Abb. 1 visualisiert.

3. EVOLUTIONSOPERATIONEN

Der Migration von Daten geht die Evolution des Schemas voraus: Während bei der Evolution das Schema der Speicherung verändert wird, werden die gespeicherten Daten bei der Migration vom alten Schema in das neue Schema überführt.

Es werden fünf grundlegende Evolutionsoperationen vorgestellt, die in NoSQL-Datenbanken auf Entitäten bzw. deren Kinds angewendet werden können. Dabei sind *Single-Entity-Operationen* von *Multi-Entity-Operationen* zu unterscheiden, die in [9] eingeführt wurden. Erstere arbeiten auf den Entities *eines* Kinds und existieren auch im Kontext von RDBMS (z.B. `Add Column`). Letztere arbeiten auf mehreren Entities und existieren im Kontext von RDBMS nicht.

Als Single-Entity-Operationen stehen nach [9] `Add`, `Delete` und `Rename` zur Verfügung. `Add A.x = default` fügt zu allen Entitäten des Kinds `A` ein Property mit dem Property-Namen `x` und einem Defaultwert hinzu. `Delete A.x` entfernt dieses Property analog. `Rename A.x To z` benennt den Property-Namen aller Entitäten über dem Kind `A` von `x` in `z` um [9] .

Neben den drei Single-Entity-Operationen existieren die beiden Multi-Entity-Operationen `Move` und `Copy`. Erstere Operation erlaubt es, unter Angabe einer Bedingung ein Attribut von Entities eines Kinds zu Entities eines anderen Kinds zu verschieben. Mit `Move A.x to B Where` *`Cond`* wird so das Property x von allen Entitäten des Kinds `A` zu den Entitäten des Kinds `B` verschoben, bei denen die Bedingung *`Cond`* (Verbundattribute) zutrifft. `Copy` unterscheidet sich von `Move` dadurch, dass die Properties von den Entitäten der Quellseite (hier: Kind `A`) nicht entfernt werden. Beispielhaft beschreibt `Copy Student.degree To Hiwi.degree Where Student.matrikel = Hiwi.matrikel` eine Kopieroperation, bei dem die Property mit dem bisherigen Abschluss eines Studierenden von der Entität *Student* zur Entität *Hiwi* kopiert und anhand der Matrikelnummer gematched wird. Die Informationen zur Matrikelnummer sind nach der Migrationsoperation weiterhin in der Entität Student vorhanden [9].

4. DIMENSIONEN DER MIGRATION

Zur Klassifikation von Migrationsstrategien werden die vier verschiedenen Dimensionen nach [9] – Migrationszeit, Operationskonsolidierung, Datenmenge und Migrationsort – eingeführt und erläutert. Diese Dimensionen ermöglichen es, die Unterschiede zwischen den Migrationsstrategien im Anschluss zu verdeutlichen.

4.1 Migrationszeit

Die Dimension der Migrationszeit definiert, wann ein Entity migriert wird. Der triviale Ansatz ist die sofortige Durchführung der Migration, sobald eine Operation zur Evolution des Schemas angewendet wird. Dieser Ansatz wird als *Eager Migration* bezeichnet. Hierbei werden mit der Änderung des Schemas zeitgleich auch die dem Schema zugrundeliegenden Daten angepasst. Dies ist auch die Art der Migration, die auch in RDBMS vorherrscht.

Als Gegenstück zur Eager Migration existiert die *Lazy Migration*. Bei der Lazy Migration wird nach einer Operation zur Schemaänderung die Migration nicht sofort ausgeführt, sondern erst dann, wenn zu einem späteren Zeitpunkt auf das von der Evolution betroffene Entity zugegriffen wird. Dies ermöglicht die Optimierung durch die Konsolidierung von mehreren, hintereinander ausgeführten Migrationsoperationen. Wird etwa ein `Add A.x = default` ausgeführt, gefolgt von einem `Delete A.x`, so heben sich beide Operationen auf, sodass keine Migrationsoperation durchgeführt werden muss. Der Zeitpunkt der Migration kann nach unterschiedlichen Kriterien festgelegt sein, erfolgt aber spätestens beim Zugriff auf das Entity.

Zwischen den beiden „Extrema“ der Eager Migration und der Lazy Migration lassen sich weitere Varianten – die *Predictive Migration* (s. Abschnitt 5.4) und die *Incremental Migration* (s. Abschnitt 5.5) – einordnen.

[1] Dies ist der allgemeine Fall für NoSQL-Datenbanken. Einige wenige Systeme wie *Cassandra* benötigen dennoch Schemainformationen [8].

[2] Unter Voraussetzung der Einhaltung der ersten Normalform.

Die Predictive Migration versucht anhand von Statistiken oder Heuristiken die von einer Migrationsoperation betroffenen Entitäten zu bestimmen, auf die mit einer hohen Wahrscheinlichkeit in naher Zukunft zugegriffen wird (*„Hot Data“*) und migriert diese sofort. Die restlichen Datensätze (*„Cold Data“*) werden zu einem späteren Zeitpunkt lazy migriert.

Die Incremental Migration migriert die Daten dann, wenn diese einen definierten Veralterungshorizont erreicht haben. Liegen Entitäten etwa 5 Evolutionsoperationen zurück, werden diese migriert, auch wenn sie als Cold Data gelten.

4.2 Operationskonsolidierung

Die zweite zu betrachtende Dimension ist die Konsolidierung von Operationen. Operationen können einerseits *schrittweise* ausgeführt werden, d.h. bei mehreren hintereinander folgenden Operationen wird nicht versucht, eine Optimierung (in Form einer Zusammenfassung) der Operationen zu erreichen (vgl. [9]). Alle Operationen werden Schritt für Schritt nacheinander angewendet.

Das Gegenstück zur schrittweisen Ausführung ist die *konsolidierte* Ausführung, bei der versucht wird, Operationen soweit wie möglich zusammenzufassen. Die Konsolidierung möglicher Operationen wurde in [9] bereits definiert.

In Abb. 2 sei die aktuelle Version $E_{t,5}$. Um von der vorherigen Version $E_{t,4}$ auf diese Version zu kommen, ist nur eine Migrations- und Evolutionsoperation nötig. Bei allen anderen Versionen wird eine konsolidierte Migration dargestellt: Um von der Version 1, in der $E_{t,1}$ liegt, auf die aktuellste Version zu kommen, sind die Evolutions- und Migrationsoperationen der Schritte 2–5 zusammenzufassen. Ohne den Umweg über die schrittweise Migration in die Versionen 2, 3 und 4 wird direkt in die neuste Version 5 migriert.

4.3 Datenmenge

Als dritte Dimension gilt die Datenmenge, die definiert, welcher Datenbestand migriert wird. Wird zum Beispiel bei einer *move* Operation ein Property `x` von `A` nach `B` verschoben, so gilt für den Fall der Lazy Migration, dass vorerst keine Migrationsoperation durchgeführt wird. Wird auf `x` in `B` zugegriffen, muss die Migration mindestens von `B` durchgeführt werden. An dieser Stelle existieren nun zwei Varianten: Entweder werden nur alle Entitäten des Kinds `B` migriert, da nur auf `B` ein schreibender Zugriff erfolgt, oder es werden auch (transitiv) betroffene Entitäten migriert, in diesem Fall alle Entitäten des Kinds `A`. Bei nicht-transitiver Migration würden die Werte von `x` in `A` erst dann entferntx werden, wenn ein Zugriff auf `A` erfolgt.

4.4 Migrationsort

Durch die letzte Dimension wird definiert, an welcher Stelle die Migration stattfindet. Die Migration der Daten wird entweder im NoSQL-Store selbst oder auf höherer Ebene, z.B. mithilfe einer Middleware, durchgeführt.

5. MIGRATIONSSTRATEGIEN

Nachfolgend werden fünf Migrationsstrategien nach [9] vorgestellt. Neben der bereits erwähnten Eager Migration und Lazy Migration wird auch die *Predictive Migration* und die *Incremental Migration* tiefergehend vorgestellt. Die Lazy Migration wird in *Lazy Composite Migration* und *Lazy Stepwise Migration* untergliedert und vorgestellt.

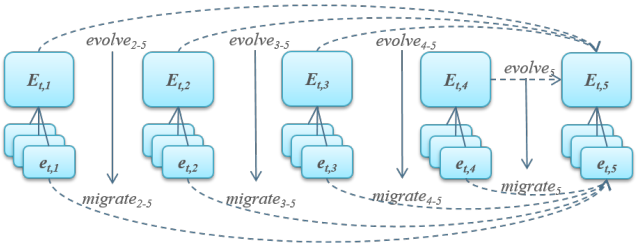

Abbildung 2: Evolution eines Schemas über fünf Versionen (entnommen aus [9])

Die Auswahl der konkreten Strategie für die Migration ist letztendlich Aufgabe eines Advisors. Innerhalb eines Systems ist es möglich, dass für die Migration gleicher oder verschiedener Entitäten unterschiedliche Migrationsstrategien angewendet werden.

5.1 Eager Migration

Die Strategie der Eager Migration spiegelt den klassischen Migrationsansatz wider, bei welcher Daten nach einer Änderung auf Schemaebene sofort migriert werden. Mehrere Evolutionsoperationen werden nicht gesammelt und gegebenenfalls optimiert ausgeführt, sondern sofortig nacheinander umgesetzt. Eine Konsolidierung der Operationen findet nicht statt. Bei der Eager Migration werden aus Sicht der Dimension der Datenmenge alle Daten migriert, d.h. bei Multi-Entity-Operationen sowohl Quell- als auch Zielseite (vgl. [9]).

Die Eager Migration kann als Migrationsstrategie in verschiedenen Szenarien genutzt werden. Nützlich ist sie in Fällen, in denen man nur sehr wenig Daten in der NoSQL-Datenbank gespeichert hat. Dabei migriert sie Daten in die neue Version, ohne relevante Ausfallzeiten durch Lese- und Schreiboperationen zu erzeugen. Weiterhin eignet sich die Eager Migration auch für die Migration von großen Datensätzen, wenn auf die zu migrierenden Daten nur sehr selten zugegriffen wird. Ist zu erwarten, dass während der Ausfallzeit durch die Migration auf die zu migrierenden Entitäten keine Lese- oder Schreiboperation angewendet wird, ist die Strategie anwendbar.

5.2 Lazy Stepwise

Die Lazy Stepwise Migration ist bezogen auf die zeitliche Dimension das andere Extremum im Vergleich zur Eager Migration. Schemaevolutionen werden nicht direkt ausgeführt, sondern erst beim Zugriff auf die betroffenen Entitäten (vgl. [9]). Aus Sicht der Datenmenge werden nur die Entitäten migriert, auf die zugegriffen wird, nicht aber transitiv betroffene Entitäten.

Die Lazy Stepwise Migration evaluiert alle Migrationsoperationen einzeln. Wird zu den Entitäten eines Kinds `A` mit der `Add A.x = default` Operation das Property mit dem Namen `x` hinzugefügt und im nächsten Evolutionsschritt mit `Delete A.x` wieder entfernt, so wird, wenn auf ein Entity des Kinds `A` zugegriffen wird, während der Migration das Property hinzugefügt und wieder entfernt. Eine Erkennung, dass sich diese Operationen gegenseitig aufheben, findet nicht statt.

Die Strategien der Lazy Migration – sowohl Lazy Stepwise als auch Lazy Composite – eignen sich für große Datenmengen, bei denen eine Eager Migration teuer und das

System durch die Migrationsoperation nicht erreichbar wäre. Weiterhin eignen sie sich für die Fälle, wenn sehr viele Schemaevolutionsoperationen ausführt werden, etwa in der agilen Anwendungsentwicklung, bei der Daten in einer eager Variante in Schemaversionen überführt werden würden, in denen nie ein Zugriff auf die Entitäten erfolgt.

Bei der Lazy Migration muss insbesondere die Performance zur Laufzeit beachtet werden, da eine Leseoperationen strategiebedingt teure Schreiboperationen verursachen kann. Diese geschehen für den Nutzer transparent, sodass im Gegensatz zur Eager Migration die Performance beim Lesen von Daten deutlich schlechter abschätzbar ist.

5.3 Lazy Composite

Die Lazy Composite Migration ist eine alternative Strategie zur Lazy Stepwise Migration. Grundlegend arbeitet sie nach dem gleichen Prinzip, ermöglicht aber die Konsolidierung von Operationen und optimiert den Migrationsprozess über mehrere Versionen hinweg entsprechend.

Die Konsolidierung der Operationen wurde im Rahmenprojekt DARWIN bereits in [9] untersucht. Wesentliche Erkenntnis war mitunter, dass sich viele der fünf genannten Operationen konsolidieren lassen, allerdings nicht alle. Eine `Add` Operation, gefolgt von einer `Copy` Operation kann beispielsweise nicht zusammengefasst werden. Entsprechend werden bei der Lazy Composite Migration die Operationen so weit wie möglich zu einer minimalen Anzahl an Operationen konsolidiert und diese dann schrittweise ausgeführt.

5.4 Predictive Migration

Die Predictive Migration ist eine Strategie, die statistische Informationen über die gespeicherten Entitäten nutzt. Sie gliedert die gespeichterten Daten in Daten, die häufig genutzt werden (Hot Data) und Daten, die vergleichsweise selten genutzt werden (Cold Data). Die Unterscheidung geschieht dabei primär auf Entitätsebene, d.h. einige Entitäten eines Kinds gelten als Hot Data, während andere Entitäten desselben Kinds als Cold Data gelten. Bei Ausführung einer Migrationsoperation wird Hot Data eager migriert, während Cold Data lazy migriert wird. Während bei Hot Data hinsichtlich der Datenmenge alles migriert wird, werden bei Cold Data nur die tatsächlich betroffenen Entitäten beim Zugriff migriert (vgl. [9]).

Als Anwendungsfall können Systeme betrachtet werden, bei denen überwiegend die neusten Datensätze hohe Relevanz haben, wie soziale Netzwerke oder Aggregationsdienste für Nachrichten. Hier gelten Beiträge, die innerhalb der letzten Stunden erstellt worden sind, als hochgradig relevant. Solche Datensätze würden sofort migriert werden, während die von Beiträgen, die vor einer Woche erstellt wurden, schon als weitgehend unrelevant gelten.

Für die Klassifizierung der Daten in Hot Data und Cold Data ist ein Advisorsystem notwendig. Je nach Strategie kann dieses System Vorschläge aufgrund von statistischen oder heuristischen Daten unterbreiten. Denkbar sind Parameter wie die Anzahl der Zugriffe in einer bestimmten Zeitspanne oder Informationen über die Entitäten, auf die zuletzt zugegriffen wurde. Ähnliche Strategien nutzen z.B. Caches, die *Last Recently Used* (LRU) Techniken anwenden (vgl. [11]). Die Aufgabe des Advisors soll es auch sein, zu entscheiden, *wieviele* Daten als Hot Data und Cold Data gelten, damit keine oder nur geringfügige Ausfallzeiten durch die Eager Migration entstehen.

Die Predictive Migration nutzt Heuristiken, um vorauszusagen, ob die Eager Migration eines Entitys sinnvoll ist. Da es sich hierbei um Schätzungen handelt, kann es sein, dass die Predictive Migration Entitäten aus dem Cold Data Bereich migriert (*„False Positive“*) oder Entitäten aus dem Hot Data Bereich nicht migriert (*„False Negative“*). Im Falle eines False Negative wird auf das Entity in naher Zukunft zugegriffen und dieses migriert. Bei einem Zugriff auf das Entity erhöht sich durch die ausstehende Migration also die Zeit der Abfrage. Im Falle eines False Positives wird ein Tupel migriert, auf das lange Zeit nicht zugegriffen wurde. Dadurch erhöht sich die Migrationszeit der Predictive Migration, stellt darüber hinaus aber kein Problem dar.

Zum Schluss soll eine begriffliche Abgrenzung der Predictive Migration erfolgen. Die Strategie bewegt sich zeitlich immer zwischen einer Eager Migration und einer Lazy Migration. Niemals sagt sie eine mögliche Schemaänderung voraus, die zeitlich vor einer möglichen Eager Migration stattfindet. Entsprechend ist auch an keiner Stelle eine Rück-Migration aufgrund falsch migrierter Entitäten notwendig.

5.5 Incremental Migration

Der Ansatz der *Incremental Migration* besteht darin, die Migration prinziell lazy durchzuführen. Es erfolgt keine sofortige Migration, wenn eine Operation zur Schemaevolution angewendet wird, sondern beispielsweise dann, wenn ein Entity um n Versionen „zurückliegt“, also bereits n Schemaevolutionsoperationen auf ein Entity angenwendet wurden, ohne, dass eine Migration stattfand.

Die Incremental Migration kann auch dazu dienen, andere Migrationsstrategien zu unterstützen, indem sie parallel ausgeführt wird. Sie kann z.B. Migrationen von Entitäten älterer Versionen im Hintergrund durchführen, wenn die Datenbank nicht ausgelastet ist. Sie hilft, dass die Migrationsoperationen nicht zu komplex und damit teuer werden, indem sie eine Grenze für die maximale Veralterung definiert.

Operationen können gegebenfalls konsolidiert werden, da es sich im Grundsatz um eine Lazy Migration handelt. Aus Sicht der Dimension der Datenmenge werden nicht alle (transitiv) betroffenen Daten migriert, es sei denn, diese liegen auch die festgelegte Anzahl an Versionen zurück.

Sowohl die Predictive Migration als auch Incremental Migration lassen sich weder als vollständig eager noch vollständig lazy klassifizieren. Beide Strategien lassen sich daher bei der hybriden Migration ansiedeln.

5.6 Alternative zur Migration

Bisher wurde definiert, dass Entities nach verschiedenen Strategien, spätestens aber beim Zugriff migriert werden. Als Alternative zur Migration ist *Query Rewriting* denkbar. Daten unterliegen dabei auch nach Anwendung einer Evolutionsoperation immer ihrem Ursprungsschema. Voraussetzung dafür ist, dass sich Evolutionsoperationen umkehren lassen, damit eine informationsverlustfreie Rückabbildung findbar ist. Wird z.B. die Evolutionsoperation `Rename A.x to z` ausgeführt, muss für die Rückabbildung die Anfrage zu `Rename A.z T o x` umgeschrieben werden, da die Entitäten noch in der Version vorliegen, in der das Property mit dem Namen x vorkommt. Zur Anwendungsschicht hin muss der Name des Property x dann durch z substituiert werden.

Es muss noch untersucht werden, wie performant sich das Query Rewriting ausführen lässt. Findet eine Vielzahl an Evolutionsoperationen statt, steigt die Komplexität der

Rückabbildung. Um die dadurch entstehenden Performanceeinbußen zu vermeiden, kann sich die Kombination mit der Predictive Migration eignen.

Anwendungsszenario für das Query Rewriting kann die Nutzung von Datenbanken sein, die in einem bestimmten Zeitfenster stark frequentiert genutzt werden. Beispielsweise kann dies die Datenbank eines Einzelhandels sein, aus der die Kassen tagsüber für jedes gescannte Produkt die Preise auslesen. Würde hier eine Migration zur kurz- oder mittelfristigen Nichterreichbarkeit der Datenbank führen, wäre dies geschäftsschädigend. Hier wäre möglich, die Entitäten in der alten Version zu belassen, erst nach Geschäftsschluss zu migrieren und bis dahin die Anfragen umzuschreiben.

6. WAHL DER MIGRATIONSSTRATEGIE

Die im vorherigen Abschnitt genannten Migrationsstrategien sollen in einem System, welches die Migration unterstützt, automatisiert angewendet werden. Im Regelfall soll der Anwender nicht vorgeben müssen, welche Strategie zur Migration von Daten angewendet wird, sondern die Auswahl wird durch einen *Migrationsadvisor* entschieden. Der Advisor ist eine Softwarekomponente, die Informationen über das System nutzt und die beste Migrationsstrategie vorschlägt. Dabei können etwa statistische, stochastische oder heuristische Verfahren oder durch den Nutzer vorkonfigurierte Werte und Gewichte zur Anwendung kommen. Ebenso soll der Nutzer vorkonfiguierte Werte und deren Gewichtung definieren können, wobei diese Möglichkeit nur zum *„tuning“* des Advisors dient und im Allgemeinen nicht notwendig sein soll.

Was genau als „beste“ Strategie gilt und welche Faktoren konkret zur Findung einer solcher herangezogen werden, hängt vom Ziel der Optimierung und von der Umgebung, in der die Datenbank liegt, ab. Als Ziel kann beispielsweise die Findung der monetär kostengünstigsten Strategie unter Inkaufnahme eines möglichen Effizienzverlustes sein, etwa, wenn die NoSQL-Datenbank in einer *Off-Premises* Umgebung liegt (*PaaS*/*IaaS*-Dienste). Alternativ kann das Ziel die Suche nach der effizientesten Migrationsstrategie sein, etwa dann, wenn monetäre Kosten nicht die primäre Rolle spielen. Dies kann beispielweise in *On-Premises* Umgebungen der Fall sein, wenn die NoSQL-Datenbank also auf eigenen Servern liegt.

In Off-Premises Umgebungen ist häufig ein Geschäftsmodell anzutreffen, bei dem Betreiber viele Parameter abrechnen. Als Beispiel für eine solche Umgebung dient etwa der *Google Cloud Datastorage*, eine NoSQL-Dokumentendatenbank von Google [5]. Abgerechnet werden neben der Anzahl an Abfragen und der Datenmenge auch Parameter wie die Anzahl der gelesenen, geschriebenen und gelöschten Entitäten [4]. *Amazon AWS* berechnet für die Nutzung der NoSQL-Datenbank DynamoDB die monatlichen Kosten auf Basis gelesener und geschriebener Daten [1]. Zur monetären Optimierung bietet sich daher an, die Anzahl der Migrationsoperationen gering zu halten. Der Advisor kann versuchen, Strategien wie die Eager Migration außen vor zu lassen, die Migrationsoperationen zu konsolidieren und lazy auszuführen. Auf mögliche Migrationen, z.B. im Hintergrund bei niedriger Prozessorauslastung, ist zu verzichten, da diese bei genanntem Geschäftsmodell Kosten verursachen kann. Ebenso kann relevant sein, ob es sich von den Kosten, sowohl monetärerweise als auch hinsichtlich Effizienz, lohnt, die Daten zu migrieren, oder ob Query Rewriting beim Zugriff auf die Daten günstiger ist.

Bei On-Premises Umgebungen sind andere Faktoren zu berücksichtigen. Hier soll bereits gekaufte Hardware weitestgehend ausgenutzt werden, bei der in der Regel keine Kosten für Lese- und Schreiboperationen entstehen. Lediglich Stromkosten müssen für die eigenen Server bezahlt werden. Es bietet sich also an, die „beste“ Migrationsstrategie als die Effizienteste zu definieren, sodass die Daten beim Zugriff schnell in der aktuellsten Version verfügbar sind. Bei einer unausgelasteten Datenbank sind auch Migrationsoperationen im Hintergrund denkbar. Als monetärer Optimierungsfaktor ist bei bestehenden On-Premises Umgebungen primär der Energieverbrauch zu betrachten.

Sind statistische Daten über gespeicherten Entitäten bekannt, kann hinsichtlich Effizienz in der Ausführung der Migrationsoperationen optimiert werden. Dabei sollen bei jeder Operation Parameter wie die Anzahl der betroffenen Entitäten oder die Ausfallzeit der NoSQL-Datenbank abgeschätzt durch Migration werden. Wichtige Kennzahl ist beispielsweise die Anzahl der gespeicherten Entitäten. Ist nur eine geringe Anzahl an Entitäten gespeichert, kann es günstig sein, Migrationsoperationen ausschließlich eager auszuführen. In so einem Fall soll der Advisor erkennen, dass die Berechnung der idealen Strategie möglicherweise mehr Zeit in Anspruch nehmen kann, als das sofortige Migrieren der Daten.

Ebenfalls kann der Advisor Wissen nutzen, inwiefern zwei Kinds in einer Wechselbeziehung zueinander stehen. Wird eine Copy-Operation angewendet, bei der der Advisor weiß, dass nur eine sehr geringe Zahl an Properties von Entitäten eines Kinds zu Entitäten eines anderen Kinds kopiert werden, weil z.B. die *Where*-Bedingung selten erfüllt ist, so ist dieses Wissen auch zur Optimierung der Operation nutzbar.

Für die Advisortechnologie können Informationen eine Rolle spielen, die der Advisor während der Laufzeit sammelt, wie die Auslastung des Systems zu bestimmten Uhrzeiten. So kann etwa die Incremental Migration dahingehend adaptiert werden, dass nicht bei n zurückliegenden Versionen migriert wird, sondern etwa zu einer bestimmten Uhrzeit, zu der dem Advisor bekannt ist, dass die Datenbank für den Zeitraum der Migration nicht ausgelastet sein wird. Beispiel hierfür wäre der Einzelhandel, bei der eine komplexe Evolution und Migration mit möglicher Nichterreichbarkeit der Datenbank nach Ladenschluss durchgeführt werden kann.

Zu Optimierung in unserem Sinne sind kaum Vorarbeiten im RDBMS und NoSQL-Bereich bekannt. Es gibt Datenbankoptimierer bzw. -advisor, die jedoch anders vorgehen, indem sie versuchen, DML-Queries zu vereinfachen und einen entsprechenden Anfrageplan zu erstellen. In *Oracle Database 12c* wurde eine DDL-Optimierung eingeführt, sodass etwa beim Hinzufügen einer Spalte (analog zur Add-Operation) instantan ein Defaultwert ohne Sperren der Tabelle gesetzt werden kann (vgl.[7]). Die Optimierung für Multi-Entity-Operationen, die effiziente Migration nach dem Ausführen einer Evolutionsoperation und die dafür notwendige Strategieauswahl wurde aber noch nicht untersucht.

7. RAHMENPROJEKT DARWIN UND PROJEKTZIEL

Im DFG-Projekt *„NoSQL Schemaevolution und skalierbare Big Data Datenmigration“* wird die effiziente Evolution und Migration von Daten untersucht. Langfristiges Projektziel ist die Entwicklung des Tools Darwin, welches intelligent bei diesen Aufgaben unterstützt.

Arbeitet ein Anwender mit einer NoSQL-Datenbank und dem DARWIN-Framework[3], kann er über eine Weboberfläche eine Schemaevolution durchführen. In DARWIN ist dabei wählbar, wie die Daten migriert werden sollen (eager oder lazy, schrittweise oder konsolidiert, etc.). In Darwin soll ein Advisor implementiert werden, der auf Basis eines Kostenmodells mit relevanten Metriken einen Vorschlag zum Verfahren der Migration unterbreitet und automatisiert ausführt, damit der Schritt entfällt, bei dem der Anwender selbst eine Entscheidung über die Migrationsstrategie treffen muss.

Die Evaluierung der Schemaevolutions- und Migrationsalgorithmen soll mittels eines Benchmarks erfolgen. Da im Bereich der NoSQL Schema Evolution noch kein Benchmark bekannt ist, muss ein eigener Benchmark konzeptioniert und implementiert werden. Möglicherweise lassen sich Ansätze des Benchmarkmodells aus [3] übernehmen.

8. ZUSAMMENFASSUNG

Ändert sich das interne Schema von Entitäten einer NoSQL-Datenbank durch die Anwendung von Evolutionsoperationen, müssen dem Schema zugrunde liegende Daten migriert werden. Neben dem klassischen Weg der Eager Migration, bei dem alle Daten sofort in das neue Schema überführt werden und somit in der aktuellsten Version vorliegen, wurden vier weitere Migrationsvarianten vorgestellt, bei denen die Daten weiterhin vorübergehend in der alten Version vorliegen und zu einem späteren Zeitpunkt migriert werden. Ziel ist die Optimierung der Migrationsoperationen infolge häufiger Schemaänderungen durch Techniken wie der Konsolidierung von Operationen oder der unterschiedlichen Behandlung häufiger und selten genutzer Daten.

Für die Wahl einer Migrationsstrategie wurde die Nutzung eines Advisor motiviert, der auf Basis von Maßzahlen eine geeignete Strategie wählt. Je nach Ziel können monetäre Kosten oder die Migrationseffizienz optimiert werden.

Das Projekt DARWIN, das bei der Evolution und Migration von Daten unterstützen soll, wurde vorgestellt. Ziel ist die effiziente Implementierung der Migrationsstrategien in DARWIN unter Entwicklung und Nutzung eines Migrationsadvisors.

9. DANKSAGUNG

Das Projekt *„NoSQL Schemaevolution und skalierbare Big Data Datenmigration“*, in welchem die in diesem Artikel vorgestellte Teilthematik untersucht wird, wird von der *Deutschen Forschungsgemeinschaft (DFG)* unter der Projektnummer 385808805 gefördert.

Literatur

[1] Amazon. *Amazon DynamoDB – Preise*. online. Zugriff: 14.02.2018. Feb. 2018. URL: https://aws.amazon.com/de/dynamodb/pricing/.

[2] David Cohen, Mikael Lindvall und Patricia Costa. „An introduction to agile methods“. In: *Advances in Computers* 62 (2004), S. 1–66. DOI: 10.1016/S0065-2458(03)62001-2.

[3] Carlo Curino u. a. „Schema Evolution in Wikipedia - Toward a Web Information System Benchmark“. In: *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008*. Hrsg. von José Cordeiro und Joaquim Filipe. 2008, S. 323–332. ISBN: 978-989-8111-36-4.

[4] Google. *App Engine-Preise*. Zugriff: 14.02.2018. Jan. 2018. URL: https://cloud.google.com/appengine/pricing?hl=de.

[5] Google. *Google Cloud Datastore*. Zugriff: 14.02.2018. Nov. 2017. URL: https://cloud.google.com/datastore/docs/concepts/overview?hl=de.

[6] Robin Hecht. „Konzeptuelle und Methodische Aufarbeitung von NoSQL-Datenbanksystemen“. Diss. University of Bayreuth, 2015. URL: https://epub.uni-bayreuth.de/1847/.

[7] Mohamed Houri. Online. Zugriff: 23.02.2018. Okt. 2014. URL: http://www.oracle.com/technetwork/articles/database/ddl-optimizaton-in-odb12c-2331068.html.

[8] Meike Klettke, Uta Störl und Stefanie Scherzinger. „Herausforderungen bei der Anwendungsentwicklung mit schema-flexiblen NoSQL-Datenbanken“. In: *HMD Praxis der Wirtschaftsinformatik* 53.4 (2016), S. 428–442. URL: https://doi.org/10.1365/s40702-016-0234-9.

[9] Meike Klettke u. a. „NoSQL schema evolution and big data migration at scale DC, USA, December 05-08.12.2016“. In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, 05-08.12.2016*. Hrsg. von James Joshi u. a. IEEE, 2016, S. 2764–2774. URL: https://doi.org/10.1109/BigData.2016.7840924.

[10] Meike Klettke u. a. „Uncovering the evolution history of data lakes“. In: *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. Hrsg. von Jian-Yun Nie u. a. IEEE, 2017, S. 2462–2471. ISBN: 978-1-5386-2715-0. URL: https://doi.org/10.1109/BigData.2017.8258204.

[11] Donghee Lee u. a. „On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies“. In: *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Atlanta, Georgia, USA, May 1-4, 1999*. Hrsg. von Daniel A. Menascé und Carey Williamson. ACM, 1999, S. 134–143. URL: http://doi.acm.org/10.1145/301453.301487.

[12] Uta Störl u. a. „Enabling Efficient Agile Software Development of NoSQL-backed Applications 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings“. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. Hrsg. von Bernhard Mitschang u. a. LNI. GI, 2017, S. 611–614. ISBN: 978-3-88579-659-6.

[3] https://www.fbi.h-da.de/organisation/personen/stoerl-uta/projekte/darwin-schema-management-fuer-nosql-dbms.html