

# Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

## CD-INHALT



### IPHONE 4 JAVA DEVELOPERS

Video von der W-JAX 2010

### HIGHLIGHT

Eclipse Riena 3.0  
Hector 0.7  
Apache Cassandra

### WEITERE INHALTE

- Jackson 1.8.1
- Griffon 0.9.3 Beta 1
- Cassandra GUI 0.7.1

Alle CD-Infos ab Seite 3

### Griffon

Neuer Schwung  
für Swing »25

### OSGi

Architektur in  
OSGi umgesetzt »98



Brandneues Programm »51

# Tools

**JIRA & Co. erfolgreich einsetzen » S.52,58**

## RESTful JSF

Ein Widerspruch? »92

## Java Web Services

SOAP mit JAX-WS »69

## Cassandra und Hector

Griechische NoSQL-Mythologie »47



Datenträger enthält  
Info- und  
Lehrprogramme  
gemäß §14 JuSchG



Anwendungsentwicklung mit nichtrelationalem Datenmodell

# Cassandra und Hector

Hector [1] ist ein Java-Client für die verteilte „NoSQL-Datenbank“ Cassandra [2]. Hector wurde im Februar 2010 erstmals veröffentlicht und gilt inzwischen als De-facto-Java-Client. Dieser Artikel stellt Hectors API vor und beschreibt Cassandras nicht relationales Datenmodell sowie ausgewählte Aspekte der Verteilung und Datenkonsistenz.

von Kai Spichale

Die Namen Hector und Cassandra wurden nicht zufällig ausgewählt, denn in der griechischen Mythologie sind Hektor und Cassandra Geschwister. Hektor ist der größte trojanische Held und Cassandra „die, die Männer umwickelt“ [3]. Übrigens hat Cassandra hellseherische Fähigkeiten. Sie hat aber nichts mit dem Orakel von Delphi gemein, denn das ist der Namensgeber einer ganz anderen Datenbank.

Zum Lesen und Schreiben von Daten bietet Cassandra ein Low-Level-Thrift-API. Laut Apache gehören Entwickler eigener Java-Client-Bibliotheken zur Zielgruppe dieser vergleichsweise systemnahen API. Für alle anderen wird

der Einsatz eines High-Level-Clients, wie Hector, empfohlen. Hector ist ein Open-Source-Projekt, das von Ran Tavory gestartet wurde. Hector bietet eine einfache und typischere Programmierschnittstelle und versucht Low-Level-Themen, wie Cluster-Management, Connection Pooling und Fehlerbehandlung, besser vom eigentlichen Datenzugriff zu trennen als das Thrift-API. In diesem Artikel wird Version 0.7.0 vorgestellt. Sowohl Cassandra als auch Hector unterstützen mit JMX das Monitoring von Laufzeit- und Performanceinformationen.

## Datenmodell von Cassandra

Um Hector bzw. Cassandra einsetzen zu können, ist Grundwissen für das Datenmodell wichtig. Dieses Da-

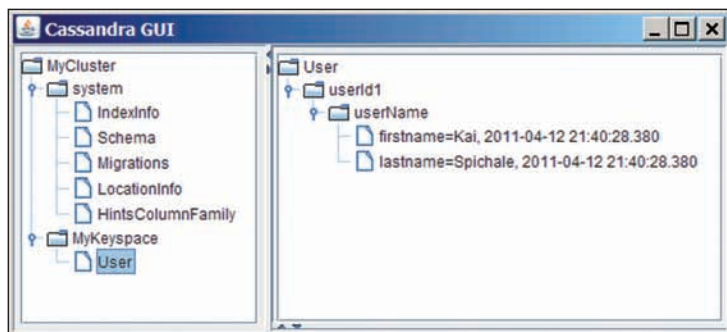


Abb. 1: Anzeige des Datenbankinhalts mit dem Cassandra GUI

tenmodell könnte man sich intuitiv als vier- bis fünfdimensionale Map vorstellen. Aber wer kann das schon? Deswegen werden nachfolgend die wichtigsten Konzepte vorgestellt:

- Cluster: Die Server (Knoten) einer logischen Cassandra-Instanz
- Keyspace: Der Namensraum für ColumnFamilies; in einem Cluster kann es mehrere Keyspaces geben, aber typischerweise gibt es nur einen Keyspace pro Applikation
- ColumnFamily: Sortierte Liste von Zeilen (Rows), die durch Schlüssel referenziert werden; die Zeilen bestehen entweder aus einer variablen Anzahl von Columns oder SuperColumns
- Column: Besteht aus einem Schlüssel-Wert-Paar mit Zeitstempel und wird durch einen Schlüssel innerhalb einer ColumnFamily referenziert

### Listing 1

```
Cluster myCluster = HFactory.getOrCreateCluster
    ("MyCluster", "host:9160");
Keyspace keyspace = HFactory.createKeyspace("MyKeyspace", myCluster);
StringSerializer se = StringSerializer.get();
Mutator<String> mutator = HFactory.createMutator(keyspace, se);

List<HColumn<String, String>> subColumns = Arrays.asList(
    HFactory.createColumn("firstname", "Kai", se, se),
    HFactory.createColumn("lastname", "Spichale", se, se));

HSuperColumn<String, String, String> superColumn = HFactory
    .createSuperColumn("userName", subColumns, se, se, se);

mutator.insert("userId1", "User", superColumn);

SuperColumnQuery<String, String, String, String> query = HFactory
    .createSuperColumnQuery(keyspace, se, se, se, se);

QueryResult<HSuperColumn<String, String, String>> queryResult = query
    .setColumnFamily("User").setKey("userId1")
    .setSuperName("userName").execute();

myCluster.getConnectionManager().shutdown();
```

- SuperColumn: Eine Column bestehend aus Columns (Sub-Columns)

Die eingeführten Konzepte wollen wir anhand eines Beispiels vertiefen. Als Beispiel betrachten wir die Entität User. Ein User hat einen Namen, der wiederum aus einem Vor- und Nachnamen besteht. Für den User verwenden wir eine gleichnamige ColumnFamily. Der Name des Users wird mithilfe einer SuperColumn umgesetzt. Diese SuperColumn hat für den Vor- und Nachnamen jeweils eine „Sub-Column“.

Die Columns bilden die kleinste Dateneinheit im Datenmodell von Cassandra. Sie bestehen aus einem binären Namen, einem binären Wert und Zeitstempel. Es folgt die Column für den Vornamen in JSON-Notation:

```
{// Das ist eine Column
  name: "firstname",
  value: "Kai",
  timestamp: 123456789
}
```

In Analogie zu relationalen Datenbanken entspricht eine Column einer einzelnen Tabellenzelle mit Spaltennamen und Zeitstempel. Der Zeitstempel hat eine wichtige Funktion, denn er wird zur Konfliktauflösung verwendet. In der Regel reicht es jedoch aus, sich eine Column als einfaches Schlüssel-Wert-Paar vorzustellen. In verkürzter Form sieht das folgendermaßen aus:

```
{ "firstname": "Kai" }
```

Eine ColumnFamily besteht aus Zeilen, die durch einen Zeilenschlüssel referenziert werden. Mithilfe des Zeilenschlüssels werden die Zeilen sortiert. In Analogie zu relationalen Datenbanken kann man sich eine ColumnFamily als Tabelle vorstellen. Eine Zeile besteht aus einer geordneten Liste von Columns bzw. SuperColumns, die innerhalb der Zeile ebenfalls über Schlüssel referenziert werden. Als Schlüssel werden typischerweise die Namen der Columns bzw. SuperColumns verwendet. Jede ColumnFamily wird in einer separaten Datei gespeichert, daher sollten zusammengehörige Columns bzw. SuperColumns in derselben ColumnFamily gebündelt werden. Im folgenden Beispiel zeigen wir eine SuperColumn, die innerhalb einer Zeile mit dem Schlüssel *userName* referenziert wird. Innerhalb dieser SuperColumn gibt es zwei Sub-Columns mit den Schlüssel *firstname* und *lastname*:

```
{// Das ist eine SuperColumn mit dem Schlüssel userName
  "userName": {
    // Die Namen der Columns werden als Schlüssel verwendet
    "firstname": { "firstname": "Kai" },
    "lastname": { "lastname": "Spichale" }
  }
}
```

Im folgenden Beispiel wird der Keyspace mit zwei Zeilen dargestellt, deren Zeilenschlüssel *userId1* und *userId2* sind. Jede Zeile enthält eine SuperColumn mit dem Schlüssel *userName*. Die Struktur der SuperColumn haben wir zuvor schon betrachtet:

```
{ // Das ist der Keyspace mit zwei Zeilen
  "userId1": {
    "userName": { .. }
  },
  "userId2": {
    "userName": { .. }
  }
}
```

### Hectors API

Nach der theoretischen Vorarbeit kommen wir nun endlich zum Coding mit Hector. Listing 1 zeigt, wie Hectors API verwendet werden kann, um die zuvor beschriebene Datenstruktur zu erzeugen und anschließend mit einer Query abzufragen.

Zu Beginn erzeugen wir ein Cluster-Objekt, das die Clientsicht auf den Cassandra-Cluster darstellt. Zu welchem Knoten des Clusters wir uns verbinden, ist egal. Ein Cassandra-Cluster wird durch seinen Namen identifiziert. Schon in der ersten Zeile unseres Beispiels begegnet uns die *HFactory*. Mithilfe dieser Klasse werden fast alle Objekte des API erzeugt.

Als Nächstes holen wir uns ein Keyspace-Objekt und einen StringSerializer. Hector bietet typsicheren Datenzugriff durch verschiedene Serializer, wie *BooleanSerializer* und *DateSerializer*. Benutzerdefinierte Serializer können durch Erweiterung der Klasse *AbstractSerializer* relativ einfach erstellt werden. Ohne diese Serializer wären die Werte, Namen und Schlüssel lediglich vom Typ

*byte[]*. In unserem Beispiel benutzen wir den *StringSerializer* für alle Serialisierungen.

Nun kommen wir zum Mutator. Mit dieser Klasse werden *insert*- und *delete*-Operationen durchgeführt. Mit der *insert*-Operation können sowohl neue Daten eingefügt als auch existierende aktualisiert werden. Mit dem Mutator lassen sich auch Batch-Operationen durchführen.

Nach dem Mutator erzeugen wir eine Liste mit je einer Column für *firstname* und *lastname*. Diese Columns nutzen wir als Sub-Columns für die SuperColumn *userName* und speichern sie mit dem Schlüssel *userId1* in der ColumnFamily *User*. Zum Schluss benutzen wir eine *SuperColumnQuery* zum Lesen der gerade geschriebenen SuperColumn.

Zum Betrachten des Datenbankinhalts empfiehlt sich das Cassandra GUI [4], ein kleines nützliches Werkzeug, das in **Abbildung 1** dargestellt ist.

### Queries

Hector bietet verschiedene Query-Klassen zur Datenabfrage. Eine Datenabfrage bezieht sich stets auf einen Keyspace und eine ColumnFamily, deren Namen bekannt sein müssen.

Die einfachsten Abfragen erfolgen durch eine *ColumnQuery* bzw. *SuperColumnQuery*. Mit einer *ColumnQuery* kann eine einzelne Column selektiert werden, sofern der Schlüssel der Zeile und der Schlüssel der Column bekannt sind. Analog dazu kann eine einzelne SuperColumn mit einer *SuperColumnQuery* selektiert werden.

Weiterhin gibt es eine Reihe von Query-Klassen, deren Namen auf *SliceQuery* enden. Auf alle wollen wir hier nicht im Detail eingehen. Wichtig zu wissen ist, dass mit ihnen eine oder mehrere Columns, SuperColumns oder

## Apache Cassandra

Die nichtrelationale Datenbank Cassandra wurde ursprünglich von Facebook entwickelt, gehört aber inzwischen zu den Top-Level-Projekten von Apache, sodass sie allen Interessierten kostenlos und quelloffen zur Verfügung steht. Apache beschreibt sie als hochskalierbarer, schlussendlich konsistenter, verteilter, strukturierter Schlüssel-Wert-Speicher. Ein Cassandra Cluster kann sukzessiv durch Hinzufügen weiterer Server linear horizontal skaliert werden. Durch das dezentrale, symmetrische Design können Petabytes an Daten hochverfügbar und fehlertolerant verwaltet werden. Aufgrund dieser Eigenschaften eignet sie sich für Internetplattformen mit sehr hohem Traffic, und dazu zählen Social-Media-Unternehmen wie Facebook, Twitter und Digg. Cassandra kombiniert eine durch Googles BigTable inspirierte Datenstruktur mit einer Verteilungsinfrastruktur, die an das verteilte Dateisystem Amazon Dynamo erinnert. BigTable wurde speziell für eine hochverteilte, Share-Nothing-Architektur entworfen, bei der die Daten mithilfe ihrer Schlüssel durch Partitionierung und Replizierung auf einer Vielzahl von Commo-

dity-Servern gespeichert werden können. BigTable ist gewissermaßen „schemalos“, denn die Anzahl der Columns innerhalb einer Column-Family ist variabel. Das macht die Datenstruktur im Vergleich zu relationalen Datenbankmodellen sehr flexibel. Wie Amazon Dynamo ordnet Cassandra alle Server eines Clusters ringförmig an. Alle Server sind gleich. Es gibt keine zentrale Steuerung. Jeder Knoten ist nur für einen Bereich der Daten verantwortlich. Wo die Daten letztendlich liegen, wird mithilfe der Zeilenschlüssel und einer konsistenten, verteilten Hash-Tabelle bestimmt. Falls ein Server ausfällt, sind dessen Daten nicht verloren, denn die Daten jedes Servers können auf dessen Nachfolgern im Ring gespiegelt werden.

Auf Basis von Apache Hadoop unterstützt Cassandra Map/Reduce zur nebenläufigen Verarbeitung riesiger Datenmengen in Clustern. Die Hadoop-nahen Technologien Hive und Pig werden ebenfalls unterstützt. Hive ist eine Data-Warehouse-Infrastruktur zum Zusammenführen großer Datenmengen und ermöglicht Ad-hoc-Anfragen mit Hive QL. Pig bietet zur Datenanalyse eine High-Level-Sprache und unterstützt die Parallelisierung von Tasks.

Sub-Columns aus einer oder mehreren Zeilen selektiert werden können. Die notwendigen Schlüssel werden entweder einzeln aufgezählt oder in Form einer Range angegeben.

Doch wie können Daten gelesen werden, wenn deren Zeilenschlüssel nicht bekannt sind? Um den Zeilenschlüssel herauszufinden, könnte ein sekundärer Index in Form einer ColumnFamily genutzt werden. Diese ColumnFamily könnte beispielsweise die E-Mail-Adresse des Users als Zeilenschlüssel verwenden:

```
{ // Zweiter Index
  "kspichale@gmail.com": {
    "userId": { "userId": "userId1" }
  }
}
```

Ab Version 0.7 unterstützt Cassandra die automatische Verwaltung von sekundären Indizes für ausgewählte Columns. Passend dazu bietet Hector mit der Klasse *IndexedSlicesQuery* die Möglichkeit, sekundäre Indizes zu nutzen, um Columns in einer oder mehreren Zeilen zu selektieren.

Weder referenzielle Integrität noch kaskadierendes Löschen oder Ad-hoc-Abfragen werden unterstützt. Auch

Join-Operationen und andere SQL-typische Klauseln, wie ORDER BY, GROUP BY und LIKE, sind nicht möglich.

Bereits beim Entwurf des Datenmodells sollten alle Abfragen identifiziert werden. Neue Anfragepfade können nachträglich nur mit sehr viel Aufwand hinzugefügt werden. Die Daten sollten in einer denormalisierten Struktur gespeichert werden, sodass jede Anfrage mit einer oder mehreren Zeilen einer ColumnFamily beantwortet werden kann.

An dieser Stelle sollte auf die Integration von Hadoop hingewiesen werden, denn durch Hadoop bietet Cassandra Map/Reduce und andere Formen der Datenverarbeitung mit Hadoop-Technologien.

### Hector Object Mapper (HOM)

Hector bietet einen einfachen Object Mapper, der Entitäten in Form von annotierten POJOs auf die spaltenorientierte Datenstruktur abbildet. Zum Laden und Speichern der Entitäten wird ein EntityManager verwendet. Bei der Gestaltung des API haben sich die Entwickler JPA 1.0 zum Vorbild genommen. Der Object Mapper befindet sich noch in einem frühen Entwicklungsstadium, doch er soll bald zu einem „JPA 1.0 Basic“ ausgebaut werden. Vererbung wird unterstützt, jedoch keine One to Many oder Many to One Relati-

## Warum ist starke Konsistenz nicht immer von Vorteil?

Eine wichtige Eigenschaft von klassischen relationalen Datenbanken ist Transaktionssicherheit, d. h. bei der Ausführung von Transaktionen werden die ACID-Eigenschaften garantiert. Doch es gibt Anwendungen mit sehr hohen Performance- und Verfügbarkeitsanforderungen, für die diese Eigenschaften nicht garantiert werden können.

### Horizontale Skalierung

Wer schreibintensive, hochperformante Systeme mit schnell wachsenden Datenmengen betreiben möchte, der wird mit einem einzelnen Server schnell an Grenzen stoßen. Selbst die beste und teuerste Hardware wird irgendwann den wachsenden Anforderungen nicht mehr gerecht werden. Die Lösung ist horizontale Skalierung. Dabei werden zusätzliche Server in das System eingefügt, um die Arbeit zu verteilen. Das hat auch einen positiven Effekt auf Zuverlässigkeit und Verfügbarkeit. Zwar können die ACID-Eigenschaften von Transaktionen auch von verteilten Systemen durch die Anwendung eines Zwei-Phasen-Commit-Protokolls garantiert werden, doch bei sehr großen verteilten Systemen tritt ein Problem auf, das bei kleineren Systemen in der Regel keine Bedeutung hat. Gemeint ist das CAP-Theorem, das bei der Architektur von hochskalierbaren Systemen zu berücksichtigen ist.

### CAP-Theorem

Die Abkürzung CAP steht für Strong Consistency, High Availability und Partition Tolerance. Das CAP-Theorem von Eric Brewer

besagt, dass ein verteiltes System nicht gleichzeitig alle drei Anforderungen erfüllen kann, sondern höchstens zwei:

- Starke Konsistenz bedeutet, dass alle Knoten zur selben Zeit die gleiche Sicht auf die Daten haben, auch wenn die Daten verändert wurden
- Hochverfügbarkeit bedeutet, dass der Ausfall von einigen Knoten durch die verbleibenden kompensiert werden kann
- Partitionstoleranz bedeutet, dass die Funktionsfähigkeit des Systems erhalten bleibt, auch wenn Nachrichten verloren gehen

Also muss ein Kompromiss gefunden werden, und der geht bei Cassandra und anderen NoSQL-Datenbanken auf Kosten der starken Konsistenz. Denn die riesigen Systeme von Amazon, EBay, Facebook und Twitter müssen ständig von ihren Usern erreichbar sein und aufgrund ihrer immensen Größe sind Partitionierungen kaum zu vermeiden. Grundvoraussetzung für hohe Benutzerakzeptanz ist gute Performance.

### Schlussendliche Konsistenz

Schlussendliche Konsistenz (Eventually Consistency) ist eine besondere Form der schwachen Konsistenz, bei der die Daten innerhalb eines Zeitfensters inkonsistent sein können. Diese Toleranz verbessert jedoch die Performance von Lese- und Schreiboperationen von hochverteilten Systemen erheblich. Die Partitionstoleranz wird ebenfalls verbessert, denn das System kann Anfragen bedienen, auch wenn die Mehrheit der Server nicht erreichbar ist.

onships. Würde der Object Mapper diese Relationships unterstützen, so würde er die Performance- und Skalierbarkeitseigenschaften der Datenbank neutralisieren. Ziel des Object Mappers ist es, den Entwicklungs- und Wartungsaufwand zu verringern.

### Transaktionen

Aufgrund der hochskalierbaren Architektur können Transaktionen im klassischen Sinne mit ACID-Eigenschaften nicht unterstützt werden. Mit einem konfigurierbaren Replikationsfaktor und auswählbaren Konsistenzstufen für Lese- und Schreiboperationen kann der Spagat zwischen Konsistenz und Performance gesteuert werden. Der Replikationsfaktor entspricht der Anzahl der Knoten, auf denen Daten repliziert werden. Die Konsistenzstufe bestimmt die Anzahl der Bestätigungen, die notwendig sind, bevor eine Lese- bzw. Schreiboperation gültig ist. Es gilt: Je größer die Anzahl der Bestätigungen, desto konsistenter die Lese- und Schreiboperationen, aber desto schlechter deren Performance:

- N = Anzahl der Knoten, auf denen Daten repliziert werden
- W = Anzahl der notwendigen Bestätigungen, um eine Schreiboperation als erfolgreich einzustufen
- R = Anzahl der Knoten, die bei einer Leseoperation aufgerufen werden

Cassandra garantiert (fast) starke Konsistenz (Strong Consistency), wenn  $W + R > N$ . In allen anderen Fällen arbeitet Cassandra mit schlussendlicher Konsistenz (Eventually Consistency).

Die performanteste Konsistenzstufe gewährleistet die geringste Konsistenz der Daten. Für Schreiboperationen reichen die Konsistenzstufen von ZERO bis ALL. ZERO ist der schnellste Weg, Daten zu schreiben, denn die Schreiboperation erfolgt asynchron im Hintergrund. Bei ALL wird auf die Bestätigung aller N-Knoten gewartet. Falls einer der N-Knoten die Schreiboperation nicht erfolgreich ausführen kann, scheitert die komplette Schreiboperation. Für Leseoperationen reichen die Konsistenzstufen von ONE bis ALL. Bei ONE wird der Wert des Knotens verwendet, der am schnellsten antwortet. Bei ALL wird der Wert von allen N-Knoten gelesen und der mit dem neuesten Zeitstempel ausgewählt. Falls ein Knoten nicht antwortet, scheitert die komplette Leseoperation.

### UUIDs

Als eindeutige Schlüssel können Universally Unique Identifiers (UUIDs) verwendet werden. Angenommen, zwei Clients schreiben simultan eine Column in die gleiche Zeile und verwenden einen Zeitstempel als Schlüssel, dann überschreibt der zweite Client die Column des ersten Clients, falls beide den gleichen Zeitstempel verwenden. Um diese Kollision zu vermeiden, aber trotzdem die Columns nach Zeit sortieren zu können, kann eine

UUID der Version 1 mittels Cassandras *TimeUUID-Type* verwendet werden. Für alle anderen UUID-Versionen dient Cassandras *LexicalUUIDType*.

### Fazit

Cassandra ist eine leistungsstarke Alternative zu konventionellen relationalen Datenbanken. Die Open-Source-Datenbank eignet sich insbesondere für Unternehmensanwendungen mit extrem großen Datenmengen und hoher Benutzeranzahl. Das beweisen Größen wie Twitter, Facebook, Rackspace und Reddit, die Cassandra erfolgreich einsetzen. Jedoch müssen die Anforderungen an Skalierbarkeit, Verfügbarkeit und Konsistenz im Einzelfall kritisch bewertet werden, denn schwache Konsistenz ist nicht für jede Unternehmensanwendungen akzeptabel. Mit Hector steht ein High-Level-Java-Client zur Verfügung, der den Umgang mit Cassandra aus Entwicklersicht vereinfacht. Doch die Vereinfachung geht nicht weit genug. Denn immer noch muss für das Lesen und Schreiben der Daten relativ viel Code geschrieben und später gewartet werden. Dieses Manko könnte der vielversprechende Object Mapper beseitigen.

Darüber hinaus wird Cassandra ab Version 0.8 die Cassandra Query Language (CQL) unterstützen. Mit dieser SQL-ähnlichen Abfragesprache könnte der Entwicklungs- und Wartungsaufwand nochmals verbessert werden.



**Kai Spichale** arbeitet als Senior Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG und entwirft und entwickelt Unternehmensanwendungen mit allem, was die Java-Plattform zu bieten hat.

### Links & Literatur

- [1] <http://cassandra.apache.org/>
- [2] <https://github.com/rantav/hector>
- [3] [http://de.wikipedia.org/wiki/Kassandra\\_\(Mythologie\)](http://de.wikipedia.org/wiki/Kassandra_(Mythologie))
- [4] <http://code.google.com/p/cassandra-gui/>
- [5] [https://github.com/rantav/hector/wiki/Hector-Object-Mapper-\(HOM\)](https://github.com/rantav/hector/wiki/Hector-Object-Mapper-(HOM))