JSR Money und Currency ▸ 74

S&S

JAVA Mag

5.2016

Java™magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Apache Cassandra
Erfahrungen aus der Praxis ▸ 60

Designmuster in Java 8
Das Auge codet mit ▸ 12

MICRO-SERVICES

Cloud-Foundry-Architektur: So wird die App cloudy ▸ 92

Diskussion: „Java wird nie wirklich funktional“ ▸ 20

Neuerungen in HTML und CSS ▸ 106

Apache Cassandra im Projekteinsatz

So klappt's auch mit der NoSQL-Datenbank

Apache Cassandra hat sich in den letzten Jahren zur populärsten spaltenorientierten Datenbank entwickelt [1]. Bei der Implementierung einer Anwendung auf Basis von Cassandra ist von Entwicklern und Architekten allerdings an verschiedenen Stellen ein Umdenken erforderlich. Dies gilt insbesondere, wenn bisher hauptsächlich mit relationalen Datenbanken gearbeitet wurde.

von Martin Hermes und Philip Stroh

Cassandra ist eine spaltenorientierte NoSQL-Datenbank, im Englischen auch „Wide Column Store" genannt. Aufgrund der doppelten Begriffsbelegung sollte diese Art der Datenspeicher nicht mit spaltenorientierten RDBMS verwechselt werden, wie sie im Data-Warehouse-Bereich zum Einsatz kommen. Wide Column Stores speichern Daten in verteilten multidimensionalen Maps. Diese Maps werden bei Cassandra wie bei relationalen Datenbanken als Tabellen bezeichnet.

Cassandra wurde ursprünglich von Facebook entwickelt. Mittlerweile führt die Apache Software Foundation die Datenbank als Open-Source-Projekt. Die Firma DataStax bietet eine kommerzielle Enterprise-Distribution von Cassandra an und stellt gleichzeitig viele Committer im Apache-Projekt. Die Dokumentation, der Java-Treiber und ergänzende kostenfreie Tools sind dadurch beispielsweise direkt auf der Webseite von DataStax zu finden. Für ein tiefergehendes Studium verweisen wir auf die Dokumentation [2].

Artikelserie

Teil 1: Auswahl einer NoSQL-Lösung
Teil 2: Einführung einer NoSQL-Lösung

Architektur im Cluster

Cassandra-Datenbanken werden im Cluster betrieben. Im Cluster nimmt jeder Knoten die gleiche Rolle ein, das heißt, es gibt keine zentrale Steuerung, keinen Master-Knoten. Die Daten werden über das Cluster verteilt. Eine Verdopplung der Knotenanzahl hat daher beispielsweise zur Folge, dass sich die Datenmenge halbiert, für die ein einzelner Knoten verantwortlich ist. Cassandra-Systeme lassen sich somit linear horizontal skalieren. Die Verteilung der Datensätze auf die Knoten wird über den so genannten Partition Key gesteuert. Eine oder mehrere Spalten jeder Tabelle werden als Partition Key festgelegt. In einer Tabelle, die Benutzerdaten vorhält, wäre dies zum Beispiel die User-ID. Alle Datensätze zur gleichen User-ID werden in diesem Fall auf demselben Rechner gespeichert. Zur gleichmäßigen Verteilung der Daten wird der Key nicht direkt genutzt, sondern zuvor ein Hash-Wert gebildet.

Um den Zugriff auf das Cluster auch beim Ausfall einzelner Knoten sicherzustellen, lassen sich Replikationen von Datensätzen über mehrere Knoten verteilen. Die Anzahl der Replikationen wird über den Replication Factor konfiguriert. Ein Replication Factor von 3 bedeutet, dass jeder Datensatz auf drei Knoten gespeichert ist. Jede der Replikationen ist gleichwertig. Beim Zugriff auf Datensätze wird die Last unter den Replikationen aufgeteilt.

Um Performance und Ausfallsicherheit zu optimieren, lassen sich für einen Cassandra-Cluster Data Centers und Racks definieren. Jeder Knoten wird einem Rack und einem Data Center zugeordnet. Der Replication Factor lässt sich dabei für jedes Data Center einzeln konfigurieren. Die Verteilung der Daten auf verschiedene Racks innerhalb eines Data Centers erfolgt automatisch. So lässt es sich vermeiden, dass bei einem Rack-Ausfall mehrere Replikationen einer Partition gleichzeitig betroffen sind.

Cassandra arbeitet nach dem Prinzip von Eventual Consistency. Datensätze werden zwar über alle Replikationen verteilt, aber zum Zeitpunkt einer Abfrage ist die Replizierung des Datensatzes nicht zwangsläufig abgeschlossen. Je nachdem, von welchem Knoten die Daten gelesen werden, kann das Resultat daher unterschiedlich ausfallen. Über den konfigurierbaren Consistency Level kann der Anteil der Replikationen festgelegt werden, auf deren Antwort bei einer Abfrage gewartet wird. Beispielsweise wird beim Standardwert *ONE* bereits die erste erhaltene Antwort akzeptiert. Beim Consistency Level *QUORUM* wird hingegen gewartet, bis die Mehrheit der Knoten geantwortet hat. Bei lesendem Zugriff wird in diesem Fall die Antwort mit dem aktuellsten Zeitstempel an den Client weitergeleitet. Wenn ein Datensatz mit *QUORUM* in das Cluster geschrieben und anschließend mit *QUORUM* wieder gelesen wird, ist sichergestellt, dass der aktuellste Datensatz vorliegt, bevor er an den Client weitergeleitet wird. Allerdings verschlechtern sich hierdurch die Performance und auch die Ausfallsicherheit, da eine der benötigten Antworten

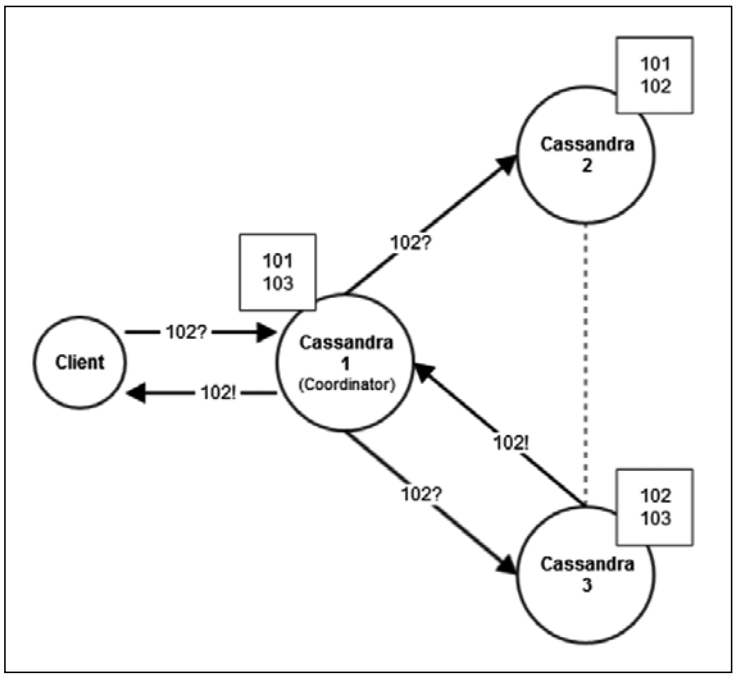

Abb. 1: Datenfluss einer Abfrage im Cassandra-Cluster: Fragezeichen kennzeichnen einen Request, Ausrufezeichen eine Antwort

eventuell länger braucht, oder die Anfrage aufgrund eines Ausfalls der Mehrheit der Knoten nicht beantwortet werden kann. Für eine Auflistung und Erläuterung aller möglichen Consistency Levels verweisen wir auf [3].

Ablauf der Abfragen

Der vollständige Ablauf einer Abfrage sieht wie folgt aus: Der Client schickt seine Abfrage an einen beliebigen Knoten des Clusters. Dieser Knoten übernimmt für diese Abfrage die Rolle des Coordinators. Der Coordinator ermittelt anhand des Partition Keys, auf welchen Knoten die Daten repliziert sind. Der Coordinator kann auch selbst einer dieser Knoten sein. Die Abfrage wird an diese Replikationen weitergeleitet. Der Coordinator wartet im Anschluss, bis die durch den Consistency Level vorgeschriebene Anzahl von Antworten eingetroffen ist. Anschließend sendet er die Antwort an den Client.

Den Ablauf von Abfragen soll ein Beispiel (**Abb. 1**) verdeutlichen. Ein Client fragt die Userdaten zu einem User mit der ID 102 ab. Zur besseren Veranschaulichung ist das Cluster mit drei Knoten und einem Replication Factor von 2 konfiguriert, obwohl in der Praxis der Replication Factor mindestens 3 betragen sollte. Die benötigten Daten werden daher auf zwei der drei Knoten vorgehalten. Als Partition Key wird die User-ID verwendet.

Der Client sendet die Abfrage an einen beliebigen der drei Knoten, in diesem Fall an Cassandra-1, der damit die Rolle des Coordinators übernimmt. Hier wird ermittelt, dass die benötigten Daten nur auf Cassandra-2 und -3 vorliegen. Die Anfrage wird entsprechend an diese beiden Knoten weitergeleitet. Die Antwort von Cassandra-3 kommt als Erstes bei Cassandra-1 an. Da der Consistency Level ONE konfiguriert ist, wird nicht auf die

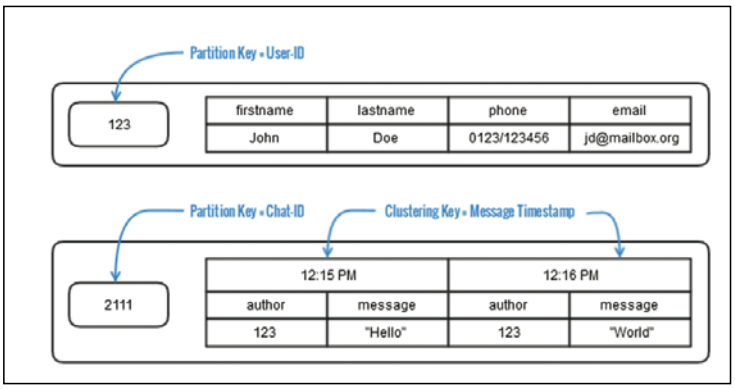

Abb. 2: Tabellenzeilen mit und ohne Clustering Key

Antwort von Cassandra-2 gewartet. Die Antwort wird umgehend an den Client weitergeleitet.

Die Persistierung von Datensätzen erfolgt in Cassandra-Datenbanken in mehreren Schritten. Um diese nachzuvollziehen, muss zuvor klar sein, dass bei Cassandra niemals bestehende Datensätze nachträglich manipuliert werden. Datenänderungen werden immer als zusätzlicher Datensatz angelegt. Beim Abruf der Daten stellt Cassandra unter Berücksichtigung der Zeitstempel aus den relevanten Datensätzen die Antwort zusammen. Datensätze werden auch nicht direkt gelöscht, sondern über ein Flag als gelöscht markiert. Schreibende Abfragen, die auf einem Knoten ausgeführt werden sollen, werden zunächst gesammelt. Erst wenn eine definierte Datenmenge erreicht wurde, werden die Änderungen sortiert und in dieser Reihenfolge erneut gespeichert. Die Sammlung von Änderungen startet daraufhin von vorne.

Man kann sich eine Tabelle in Cassandra als zwei- oder dreidimensionale Map vorstellen. Die erste Map enthält die Zeilen der Tabelle, auf deren Inhalt man über den Partition Key zugreift. Für Zeilen wird daher auch der Begriff Partitionen verwendet. Bei der zweidimensionalen Map stellt die zweite Ebene die Spalten der Tabelle dar, wobei der Schlüssel der inneren Map dem Spaltennamen entspricht. Bei der dreidimensionalen Variante kommt noch eine weitere Ebene hinzu, über die die Spalten gruppiert werden können. Hier erfolgt der Zugriff auf die gruppierten Spalten durch den so genannten Clustering Key. **Abbildung 2** zeigt jeweils eine Zeile aus zwei unterschiedlichen Tabellen. Die erste Tabelle enthält Userinformation, die ohne Clustering Key gespeichert sind. Die zweite Tabelle beinhaltet Chatnachrichten, die nach der Chat-ID partitioniert sind, und welche die Eingangszeit der Nachrichten als Clustering Key verwendet.

Cassandra ist in der Lage, auf der Nachrichtentabelle (**Abb. 2**) die Abfrage beispielsweise der letzten 100 Nachrichten eines Chats sehr performant zu beantworten. Zum einen liegen alle Nachrichten zu diesem Chat auf demselben Knoten, zum anderen lassen sie sich durch die Sortierung blockweise abrufen. Da der Partition Key für die Datenverteilung verwendet wird, muss er in allen Abfragen angegeben werden. Falls die Datensätze zu dieser Partition weiter gefiltert werden sollen, muss dies über den Clustering Key geschehen. Es ist daher in diesem Beispiel nicht ohne weiteres möglich, abzufragen, in welchen Chats der Autor mit der User-ID 123 Nachrichten geschrieben hat. Die hohe Zugriffsgeschwindigkeit und lineare Skalierung werden durch eine Einschränkung der Zugriffswege erkauft.

Aufgrund der zuvor beschriebenen Arbeitsweise müssen Cassandra-Knoten regelmäßig aufgeräumt werden. Dieser als Compaction bezeichnete Prozess führt die bestehenden Datensätze unterschiedlicher Zeitstempel zusammen und persistiert das Ergebnis erneut. Als gelöscht markierte Datensätze werden ausgelassen. Die alte Datenbasis wird nach dem Zusammenführen verworfen. Ein laufender Compaction-Prozess wirkt sich negativ auf die Performance des betroffenen Knotens aus.

Im laufenden Betrieb kommt es vor, dass ein Knoten temporär nicht erreichbar ist, weil er beispielsweise neu gestartet wird. In diesem Fall werden Schreibzugriffe an diesen Knoten auf den anderen Knoten als so genannte Hints zwischengespeichert. Wenn im Beispiel aus **Abbildung 1** der Knoten Cassandra-2 nicht antwortet, speichert der Coordinator (Cassandra-1) den Hint für Cassandra-2 zwischen. Sobald der Knoten wieder verfügbar ist, werden die in den Hints gespeicherten Abfragen ausgeführt. Dieses System wird allerdings nur für begrenzte Zeiträume angewendet. Fällt ein Knoten für längere Zeit aus, werden die Hints verworfen. Nachdem die Hints verworfen wurden, muss der ausgefallene Knoten wieder wie ein neuer Knoten hinzugefügt werden. Dazu wird ein so genannter Repair ausgeführt, was eine Umverteilung der Daten im Cluster zur Folge hat.

Listing 1

```
CREATE KEYSPACE chat_example WITH REPLICATION = {'class': 'NetworkTopologyStrategy',
                                                 'DataCenter_EU': 3, 'DataCenter_US': 2};

USE chat_example;

CREATE TABLE chatmessages_by_chat (
  chat_id UUID,
  message_timestamp TIMEUUID,
  author_user_id BIGINT,
  content TEXT,
  PRIMARY KEY (chat_id, message_timestamp)
) WITH CLUSTERING ORDER BY (message_timestamp DESC);

INSERT INTO chatmessages_by_chat (chat_id, message_timestamp, author_user_id, content)
VALUES (cfd66ccc-d857-4e90-b1e5-df98a3d40cd6, now(), 123, 'Sample message')
USING TTL 86400;

SELECT * FROM chatmessages_by_chat where chat_id = cfd66ccc-d857-4e90-b1e5-
                                                   df98a3d40cd6;
```

Über die Cassandra Query Language zugreifen

Der Zugriff vom Client auf die Daten erfolgt über CQL, die Cassandra Query Language. CQL ist stark an SQL

angelehnt, auch um den Einstieg in die Technologie zu vereinfachen. Allerdings kann diese Ähnlichkeit zu falschen Interpretationen führen, da an verschiedenen Stellen Einschränkungen im Vergleich zu bekannten SQL-Befehlen bestehen. CQL-Skripte lassen sich über das CQL-Kommandozeilentool oder komfortabler über das DataStax DevCenter ausführen [4].

Listing 1 zeigt CQL-Statements zum Anlegen eines Keyspace, einer Tabelle sowie zum Einfügen und Auslesen eines Datensatzes. Der Keyspace entspricht einem Datenbankschema. Am Keyspace wird der Replikationsfaktor gegebenenfalls pro Rechenzentrum konfiguriert.

Der bei der Tabellenerstellung angegebene Primary Key setzt sich aus zwei Teilen zusammen: dem Partition Key *chat_id* und dem Clustering Key *message_timestamp*. Wichtig ist, sich hier erneut klarzumachen, dass der Zugriff per Select auf einen Datensatz immer nur über den Partition Key und optional zusätzlich über den Clustering Key erfolgen kann. Es besteht zwar die Möglichkeit, diese Einschränkung über einen so genannten Secondary Index zu umgehen, dies ist aber nur für Spezialfälle geeignet.

Datenmodellierung

Allgemein stehen beim Entwurf des Datenmodells für NoSQL-Datenbanken die Abfragen stärker im Vordergrund als bei relationalen Systemen. Da meist keine *Join*-Operation unterstützt wird, werden die Daten denormalisiert bereitgestellt. Auch bei Cassandra gibt es keine tabellenübergreifenden Abfragen. Zudem erfolgen Schreibzugriffe bei Cassandra schneller als Lesezugriffe. Lesezugriffe werden auch performant umgesetzt, aber nur, wenn das Datenmodell korrekt entworfen ist. Bei der Datenmodellierung wird somit die Optimierung von Abfragen zugunsten der Einführung von Redundanzen in Kauf genommen. Mehr noch, beim Entwickeln und Entwerfen eines Modells für ein Cassandra-Projekt sollten Redundanzen in der Datenhaltung die Regel sein [5]. In englischen Beiträgen wird Entwicklern mit SQL-Vorkenntnissen an dieser Stelle gerne metaphorisch geraten: „embrace redundancy“. Bei der Datenmodellierung empfiehlt sich folgendes Vorgehen [6]:

- ER-Diagramme erstellen, um die fachlichen Entitäten, Attribute und deren Beziehungen zueinander zu verstehen (konzeptionelles Datenmodell)
- Alle Abfragen identifizieren, die von der Applikation gegen die Datenbank abgesetzt werden
- Ausgehend von dem ER-Diagramm: ein Datenmodell entwickeln, das die Zugriffspfade der jeweiligen Queries optimiert (logisches Datenmodell)
- Umwandlung in konkrete CQL-Tabellenstrukturen (physisches Datenmodell) unter Berücksichtigung von Überlegungen zu Partitionsgröße, Datenduplizierung, Konsistenz oder Auswirkungen paralleler Benutzerzugriffe

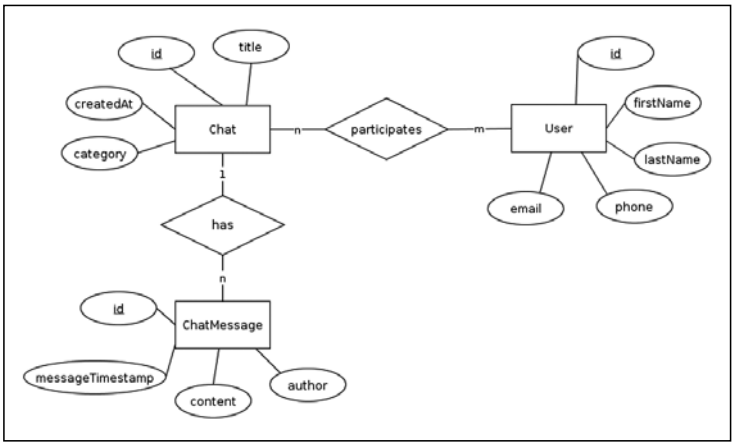

Abb. 3: Konzeptionelles Datenmodell (E/R-Diagramm nach Chen-Notation)

Unsere fiktive Beispieldomäne ist in **Abbildung 3** dargestellt. Es soll eine Chatapplikation entwickelt werden, die die folgenden Abfragen benötigt, um ihre Funktionalität umzusetzen. Es sollen alle Chats mit Titel, Kategorie und Erstellungsdatum des eingeloggten Users angezeigt werden, an denen er beteiligt ist.

- Query 1: Finde alle Chats inklusive aller Attribute zu einem User

Der User soll Chats nach bestimmten Kategorien einsehen können

- Query 2: Finde alle Chats inklusive aller Attribute zu einer bestimmten Kategorie

Bei der Auswahl eines Chats sollen alle Informationen zu einem Chat, alle Teilnehmer und die zugehörigen Nachrichten – die aktuellsten zuerst – angezeigt werden. Dies lässt sich auf folgende Abfragen herunterbrechen:

- Query 3a: Finde alle Daten zum jeweiligen Chat
- Query 3b: Finde alle Teilnehmer eines Chats inklusive Userdetails
- Query 3c: Finde die aktuellsten Nachrichten zu einem Chat

Beim nächsten Schritt liegt das Hauptaugenmerk auf der Optimierung der Abfrage. Es sollte pro Abfrage möglichst nur von einer Partition gelesen werden. Bei einer Abfrage, die Daten aus vielen Partitionen benötigt, wird im ungünstigsten Fall pro Datensatz von einem anderen Knoten gelesen. Dies führt zu vielen einzelnen Zugriffen innerhalb des Clusters und somit zu einer schlechten Performance.

Für das logische Datenmodell werden dementsprechend Tabellen modelliert, um die jeweiligen Abfragen ideal zu bedienen. Das Ergebnis kann in einem so genannten Chebotko-Diagramm dargestellt werden (**Abb. 4**). In dem Diagramm kennzeichnet K den Partition Key und C den Clustering Key der jeweiligen Tabelle. Mit etwas Übung erfolgt die Anlage dieser

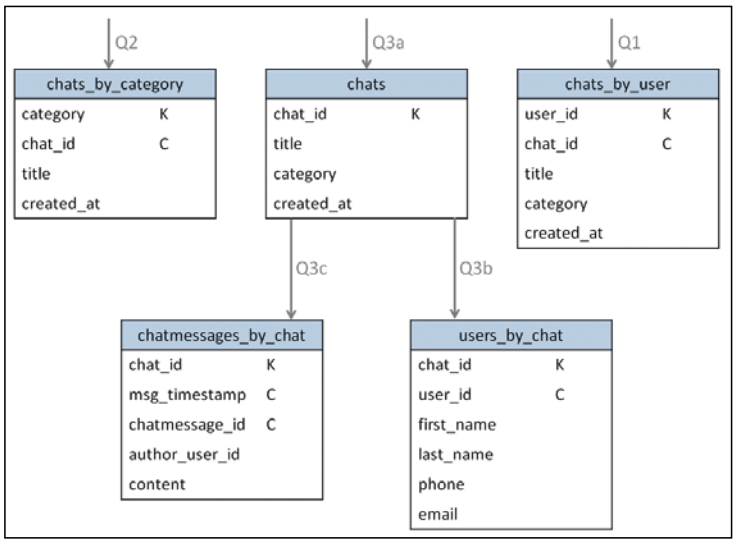

Abb. 4: Logisches Datenmodell (Chebotko-Diagramm): In dem Diagramm kennzeichnet K den Partition Key und C den Clustering Key der jeweiligen Tabelle

Tabellen sehr intuitiv. Wer einen strengeren Regeln folgenden akademischen Ansatz bevorzugt oder sich gerade anfangs damit leichter tut, kann dabei das ER-Diagramm nach den von Artem Chebotko, Andrey Kashlev und Shiyong Lu entwickelten Mapping-Regeln und -Patterns entsprechend in ein logisches Datenmodell umwandeln [7], [8]. In einer ausführlichen, kostenfreien Onlineschulung von DataStax wird dieses Vorgehen genauer erläutert [6].

Für das physische Datenmodell werden im Anschluss die Datentypen bestimmt. Hier kann beispielsweise für die ID der Chatnachricht der Cassandra-Datentyp *TimeUUID* verwendet werden. Dieser erstellt einen eindeutigen Schlüssel, in den ein Zeitstempel kodiert wird [9]. Die ID der Chatnachricht und deren Zeitstempel lassen sich somit zu einem Attribut zusammenführen.

Ein weiterer Aspekt ist die Prüfung, ob die Partitionsgröße ideal gewählt ist. Hier gibt es seitens Cassandra die Limitierung auf zwei Milliarden Einträge pro Partition. Zusätzlich muss eine Partition auf der Festplatte eines Knotens speicherbar sein. Tatsächlich empfiehlt es sich aber, bei der Partition weit unter diesem Wert zu bleiben, um effizient auf die Daten zugreifen zu können. Ein grober Richtwert sind hier 100 000 Einträge pro Partition und 100 MB benötigter Speicherplatz. Die Chatnachrichtentabelle wäre in unserem Beispiel ein Kandidat für die Überprüfung der Partitionsgröße. Eine Lösung, um die gewünschte Partitionsgröße zu erreichen, ist die Aufteilung der Partition in Untergruppen (Buckets). Dies kann durch eine Erweiterung des Partitionsschlüssels erfolgen. Die Datensätze werden dann pro Tag, Monat, Jahr oder nach einem anderen adäquaten Identifizierungsmerkmal gruppiert. Eine weitere Maßnahme, falls dies fachlich möglich ist, wäre es, die Speicherdauer für Nachrichten per TTL (Time-to-Live) zu begrenzen. Diese Lösung wurde in unserem Beispiel gewählt [10].

Durch das abfragengetriebene Vorgehen sind Tabellen entstanden, die Daten redundant speichern. Wichtig bei der Duplizierung ist es, zu prüfen, ob der Duplizierungsfaktor konstant ist. Kann der Duplizierungsfaktor unbegrenzt wachsen, muss das Datenmodell angepasst oder es müssen künstliche Limits eingeführt werden. Hierbei wird unterschieden zwischen Duplizierung in eine andere Tabelle, in eine andere Partition oder über den Clustering Key innerhalb der gleichen Partition. Die Duplizierung in eine andere Tabelle ist meist unproblematisch, in unserem Beispiel trifft dies auf die redundante Speicherung der Chatinformationen in der Tabelle *chats_by_category* zu. Hier liegt ein konstanter Faktor vor. Auch die Duplizierung in eine andere Partition (im Beispiel *chats_by_user*) ist meist akzeptabel. In dem Beispiel sollte allerdings geprüft werden, in wie vielen Chats ein User gleichzeitig aktiv sein kann. Ist der User in 100 Chats aktiv, sind seine Daten 100mal in der *users_by_chat*-Tabelle hinterlegt.

Für den Fall der Datenduplizierung in eine andere Tabelle wurde in der neuesten Cassandra-Version 3.0 das Feature Materialized Views eingeführt. Hierüber lassen sich andere Zugriffspfade auf eine Tabelle komfortabler umsetzen. Die so genannte View Table und die Base Table werden dabei von Cassandra synchron gehalten. Falls die Materialized View in einen bestehenden Keyspace hinzugefügt wird, erfolgt die Befüllung der View Table ebenfalls automatisch. Die Beispieltabelle *chats_by_category* ist ein idealer Anwendungsfall für dieses Feature [11].

Bezüglich des physischen Datenmodells sind außerdem der Aspekt Datenkonsistenz und das Verhalten bei gleichzeitigen Lese- und Schreibzugriffen zu validieren. Hier sind drei Mechanismen relevant: Batches, konfigurierbare Konsistenzlevel und die so genannten Lightweight Transactions.

Batches erlauben das Zusammenfassen mehrerer DML Statements zu einer atomaren Operation. Es werden alle oder keines der Statements ausgeführt. Zu beachten ist: Die Reihenfolge der Statements legt nicht die Ausführungsreihenfolge fest, und das Ergebnis der Manipulation kann bereits gelesen werden, bevor alle Statements des Batches auf allen Replikas ausgeführt sind. Batches sind vorgesehen, um bei redundanter Datenhaltung sicherzustellen, dass die Daten der Duplikate konsistent gehalten werden (im Beispiel die duplizierten Chat- und Userinformationen) [12].

Bei der konfigurierbaren Konsistenz kann der Konsistenzlevel bis auf Statementebene verändert werden. Das heißt, im Normalfall kann die Applikation mit Konsistenz *ONE* arbeiten, bei besonderen Abfrage- oder Schreibvorgängen kann ein höherer Konsistenzlevel gesetzt werden. Lightweight Transactions versprechen für Zugriffe auf eine Partition ein Isolation-Level ähnlich dem Serializable-Level bei ACID-Transaktionen (CQL: INSERT… IF EXISTS). Hierzu werden per Paxos-Protokoll insgesamt vier Anfragen an die entsprechenden Replikas geschickt. Diese Operation ist daher wesent-

lich teurer als normale Zugriffe und sollte nur in Ausnahmefällen zum Einsatz kommen [13].

Nach Prüfung und gegebenenfalls Anpassung des Models kann es in CQL-Statements umgesetzt werden. Hierbei ist ein Test des Models mit entsprechenden Dummydaten und Abfragen empfehlenswert. Listing 1 zeigt die Statements zur Tabellenerstellung sowie zur Anlage und zum Lesen der Chatnachrichten aus unserem Modellierungsbeispiel.

Zugriff aus Java-Anwendungen

Um aus einer Java-Anwendung auf den Cassandra-Cluster zuzugreifen, empfiehlt sich der Einsatz des DataStax-Java-Treibers [14]. Für alle Zugriffe gegen die Datenbank stellt der Treiber ein Sessionobjekt bereit. Dies wird über die *connect*-Methode an der Cluster-Klasse erzeugt. Über den *Cluster.Builder* können dabei diverse Verbindungseinstellungen angegeben werden, z. B. Username und Passwort. Das Zusammenspiel ist in Listing 2 dargestellt. Das verwendete Sessionobjekt ist threadsafe, und es wird nur eine Instanz pro Anwendung je Keyspace benötigt. In CDI-Anwendungen kann die Session beispielweiswe per Producer-Methode mit Application Scope bereitgestellt werden.

Mittels *PreparedStatements* können CQL-Statements statisch abgesetzt werden, der *QueryBuilder* erlaubt ein dynamisches Erstellen der Abfragen. Hier lässt sich jeweils auch das Konsistenzlevel setzen (Listing 3).

Der Treiber stellt über eine weitere JAR-Datei auch einen Objekt-Mapper bereit. Hier können per Annotation Tabellenstrukturen auf Java-Klassen abgebildet werden (Listing 4). Über den *MappingManager* lässt sich für die annotierte Klasse ein entsprechender Mapper erzeugen, der CRUD-Operationen bereitstellt (Listing 5). Der generierte Mapper wird im *MappingManager* für weitere Zugriffe vorgehalten.

Ein Accessor ermöglicht außerdem die Nutzung des Mappers in Verbindung mit selbstdefinierten Abfragen. Listing 6 zeigt die Definition eines Accessors per Interface. Um den jeweiligen Accessor zu nutzen, kommt wiederum der *MappingManager* ins Spiel (Listing 7). Im Hintergrund erzeugt der Treiber eine konkrete Accessor-Implementierung. Auch hier wird das erstellte Objekt, analog zum Vorgehen bei den Mappern, im *MappingManager* zur weiteren Verwendung gecacht. Der *MappingManger* sowie die erzeugten Mapper und Accessors sind threadsafe. Die Anwendung benötigt pro Session nur eine Instanz des *MappingManager*.

Über die beschriebenen Funktionen hinaus bietet der Treiber auch ein API für asynchrone, nicht-blockende Abfragen, ein erweitertes Logging für das Debugging langsamer Statements oder Zugriff auf Sessioninformationen. Hierüber lässt sich beispielweise feststellen, wie viele Abfragen gerade „in flight“ sind.

Stolperstein Datenmodellierung

Unsere Schwierigkeiten bei der Modellierung scheinen sich mit denen zu decken, die andere bei ihren ersten Cassandra-Projekten gemacht habe: Nach jahrelanger Arbeit mit relationalen Datenbanken ist es extrem schwierig, die SQL-Denke abzulegen und den Grad an Redundanzen zu akzeptieren, den diese Datenbank erfordert. Die mittlerweile existierenden Onlineschulungen und Blogartikel hierzu waren zum Start unseres

Listing 2

```
Cluster cluster = Cluster.builder()
  .addContactPoints("10.1.2.3", "10.1.2.4", "10.1.2.5")
  .withLoadBalancingPolicy(new TokenAwarePolicy(new RoundRobinPolicy()))
  .withCredentials("username", "password")
  .build();
cluster.getConfiguration()
  .getProtocolOptions()
  .setCompression(ProtocolOptions.Compression.LZ4);

session = cluster.connect("chat_example");

cluster.close(); // on shutdown
```

Listing 3

```
PreparedStatement preparedStatement = session.prepare("SELECT * FROM
                                                chatmessages_by_chat WHERE chat_id = ?");
preparedStatement.setConsistencyLevel(ConsistencyLevel.ONE);
ResultSet resultSet = session.execute(preparedStatement.bind(chatId));

Statement select = QueryBuilder.select().all().from("chatmessages_by_chat").
                                                        where(eq("chat_id", chatId));
select.setConsistencyLevel(ConsistencyLevel.ALL);
ResultSet resultSet = session.execute(select);
```

Listing 4

```
@Table(name = "chatmessages_by_chat", readConsistency = "ONE", writeConsistency = "ONE")
public class ChatMessage {

  @PartitionKey
  @Column(name = "chat_id")
  private UUID chatId;

  @ClusteringColumn
  @Column(name = "message_timestamp")
  private UUID messageTimestamp;

  @Column(name = "author_user_id")
  private long authorUserId;

  @Column
  private String content;

  // ... constructors (default constructor necessary) / getters / setters
}
```

Projekts noch nicht verfügbar. Zu diesem Zeitpunkt waren noch viele Beispiele mit dem alten Thrift-Interface zu finden, die für zusätzliche Verwirrung sorgten. Thrift wurde inzwischen durch CQL abgelöst. Seitens DataStax hat man diese Einstiegshürde aber anscheinend erkannt und mehr Lernmaterial, das meiste online und kostenlos, zum dem Thema zur Verfügung gestellt. Es ist sinnvoll, vor einem Projektstart Zeit zu investieren und sich intensiv mit dem Bereich Datenmodellierung auseinanderzusetzen.

Auswirkung der starken Abfrageorientierung

An unserem einfachen Modellierungsbeispiel wird deutlich, dass die Abfragen der Anwendung direkte Auswirkungen auf das logische und physische Datenmodell haben. Dies bedeutet aber auch, dass für neue Abfragen in vielen Fällen auch eine Anpassung an den entworfenen Tabellen notwendig ist. Gerade in agilen Projekten oder in Projekten in einem sehr frühen Entwicklungsstadium kann dies zum Problem werden. Eine Situation, in der man in jeder Entwicklungsiteration größere Änderungen am Datenmodell und der Zugriffsschicht vornehmen muss, sollte vermieden werden. Die Domäne der Anwendung muss also gut verstanden sein. Es bietet sich an, das erste Projekt mit dieser Technologie für eine Domäne zu entwickeln, die nicht zu groß ist. Der Anforderungssatz sollte zum Start des Projekts zudem möglichst gut ausgearbeitet sein, um möglichst viele Abfragen der Anwendung frühzeitig ableiten zu können.

Listing 5

```
UUID chatID = UUIDs.random();
UUID messageTimestamp = UUIDs.timeBased();
ChatMessage chatMessage = new ChatMessage(chatID, messageTimestamp, 123, "Sample
                                                                    message");

MappingManager mappingManager = new MappingManager(session);
Mapper<ChatMessage> mapper = mappingManager.mapper(ChatMessage.class);

mapper.save(chatMessage);
ChatMessage storedChatMessage = mapper.get(chatID, messageTimestamp);
mapper.delete(storedChatMessage);
```

Listing 6

```
@Accessor
public interface ChatMessageAccessor {

  @Query("SELECT * FROM chatmessages_by_chat WHERE chat_id = :chatId")
  Result<ChatMessage> findByChatId(@Param("chatId") UUID chatId);
}
```

Listing 7

```
ChatMessageAccessor chatMessageAccessor = mappingManager.createAccessor(
                                                    ChatMessageAccessor.class);
List<ChatMessage> chatMessages = chatMessageAccessor.findByChatId(chatId).all();
```

Auswirkung der eingeschränkten Konsistenz- und Transaktionsgarantien

Idealerweise sollte man die Projektentwicklung mit Konsistenzlevel *ONE* als Standard beginnen und danach die Spezialfälle hinsichtlich möglicher Probleme untersuchen. Die Auswirkungen paralleler Zugriffe und möglicher Race Conditions zu durchdenken und Lösungsmöglichkeiten zu entwickeln, benötigt Zeit. Aufwände, die hierbei zusätzlich anfallen, sind nicht zu unterschätzen. Dies trifft auch für die Analyse von Fehlern zu, die im Zuge paralleler Zugriffe auftreten können.

Im Folgenden ein Beispiel hierzu aus dem Bereich JUnit-Integrationstests. Diese sind meist wie folgt aufgebaut: Es wird eine bestimmte Datenkonstellation in eine Tabelle geschrieben, die zu testende Methode wird ausgeführt und danach wird geprüft, ob die Daten in der Tabelle korrekt verändert wurden. Dies wird bei einem Replikationsfaktor >1 und Konsistenzlevel *ONE* zu sporadischen Fehlern führen, da das Ergebnis unter Umständen von einem noch nicht aktualisierten Knoten gelesen wird. Die Aktualisierung der Knoten erfolgt innerhalb von Millisekunden. An dieser Stelle muss eventuell die Ergebnisauswertung verzögert werden, beispielsweise über [15].

Performance-, Lastmessungen und Ausfalltests unbedingt durchführen

Vor dem Produktivgang ist es wichtig, Erfahrungen auf der eigenen Infrastruktur insbesondere im Hochlastbereich zu sammeln und mögliche Ausfallszenarien durchzuspielen. In Zusammenarbeit mit dem Betrieb lässt sich hierdurch die Grenze des Systems hinsichtlich Durchsatz und Antwortzeiten bestimmen und auch die Reaktion der Anwendung auf Ausfälle des Systems absichern. Lasttests sollten dabei mit langen Laufzeiten ausgelegt werden – mindestens über mehrere Stunden –, um beispielsweise Auswirkungen der Compactions zu messen. Die Tests sollte man möglichst frühzeitig oder parallel zur Projektentwicklung aufsetzen, da dieses Thema zeitintensiv werden kann. Dies gilt insbesondere, wenn aufgrund der Ergebnisse Anpassungen in der Infrastruktur, der Cluster-Konfiguration oder dem Datenmodell notwendig sind und die Tests mehrfach wiederholt werden müssen. Für solche Stresstests enthält die Cassandra-Installation bereits das Kommandozeilentool *cassandra-stress*, mit dem sich verschiedene Workloads gegen ein beliebiges Schema simulieren lassen.

Einsatz der Monitoringfunktionen

Hand in Hand mit den Performance- und Lasttests geht das Thema Monitoring. Hier bietet das DataStax OpsCenter ein konfigurierbares Dashboard, um diverse Metriken des Clusters zu überwachen, darunter CPU- und Heap-Nutzung, Request pro Sekunde oder Laten-

Apache Cassandra entwickelt sich sehr schnell weiter, und einige der Punkte, die in unserem Projekt als Schwachstellen empfunden wurden, werden in neueren Versionen bereits adressiert.

zen in verschiedenen Perzentilen. Das OpsCenter lässt sich auch mit der Open-Source-Variante von Cassandra nutzen [16], [17]. Standardmäßig aktiviert ist im vorgestellten Java-Treiber das Reporting diverser Performancemetriken per JMX. Hierzu verwendet der Treiber intern die Metrics Library in Dropwizard. Der Einsatz dieser Monitoringtools ist sehr hilfreich und dringend anzuraten.

Aggregationen, Analysen und Datenmigration nur eingeschränkt möglich

In den vorherigen Abschnitten ist deutlich geworden, dass Cassandra nur Zugriffe über Partition und Clustering Key performant unterstützt. Analysen und Aggregationen auf einem bestehenden Datenmodell sind daher nur sehr eingeschränkt möglich. Bis Version 2.1 wurde nur *count* unterstützt. Mit Version 2.2. sind weitere Standardaggregationen sowie so genannte User-defined Aggregations hinzugekommen. Für weiterführende Analysen und Berechnungen ist der Einsatz anderer Werkzeuge empfohlen. Die DataStax Enterprise Distribution integriert daher Werkzeuge wie Solr und Spark. Auch für Datenmigrationen, die über ein einfaches Kopieren von einer Tabelle in eine andere hinausgehen, ist der Einsatz eines zusätzlichen ETL-Tools notwendig.

Fazit

Cassandra ist eine spannende, Enterprise-reife Technologie, die durch ihre Fähigkeit, linear zu skalieren, überzeugt. Sie eignet sich insbesondere für Einsatzszenarien mit großen Datenmengen, bei denen Hochverfügbarkeit und Performance gefragt sind. Gerade bei Entwicklern und Administratoren, die bisher hauptsächlich im SQL-Bereich tätig waren, gibt es allerdings aufgrund der vielen neuen Konzepte eine Einstiegshürde. Hier ist ein Umdenken beispielsweise beim Thema Modellierung und dem Umgang mit den fehlenden Transaktionsgarantien notwendig. Apache Cassandra entwickelt sich sehr schnell weiter, und einige der Punkte, die in unserem Projekt als Schwachstellen empfunden wurden, werden in neueren Versionen bereits adressiert. Mit den User-defined Aggregations/Functions sowie dem JSON-Support aus Version 2.2. und den Materialized Views in Version 3.0 sind im Juli und Dezember 2015 wichtige Funktionen hinzugekommen, die hoffentlich auch schon bald in der kommerziellen Version verfügbar sind.

Bei der TimoCom Soft- und Hardware GmbH wird Cassandra als Datenbank für eine Messaging-Applikation eingesetzt. Für diesen Anwendungsfall hat sie sich als die richtige Wahl erwiesen.

Martin Hermes studierte Software Systems Engineering und arbeitet als Softwareentwickler bei der TimoCom Soft- und Hardware GmbH. Er befasst sich unter anderem mit JavaScript-Entwicklung im Enterprise-Umfeld und plattformübergreifender App-Entwicklung.

Philip Stroh arbeitet als Softwarearchitekt bei der TimoCom Soft- und Hardware GmbH. Er hat langjährige Erfahrung in der Entwicklung und Konzeption von Java-basierten Webanwendungen.

Links & Literatur

[1] DB-engines Ranking von Wide Column Stores: http://bit.ly/1ZYCHD0

[2] DataStax Documentation: http://bit.ly/20xKPw9

[3] Configuring data consistency: http://bit.ly/20xKWrJ

[4] DevCenter: http://bit.ly/1PETqoW

[5] Basic Rules of Cassandra Data Modelling: http://bit.ly/1BGVnzQ

[6] DS220: Data Modeling: http://bit.ly/1nCUIuU

[7] Chebotko, Artem; Kaslev, Andrey; Lu, Shiyong: „A Big Data Modeling Methodology for Apache Cassandra“: http://bit.ly/1QxePDH

[8] KDM: An Automated Data Modeling Tool for Apache Cassandra, Pt. 2: http://bit.ly/1UvgECp

[9] CQL data types: http://bit.ly/20bgXZY

[10] Getting Started with Time Series Data Modeling: http://bit.ly/1nTmYZZ

[11] Cassandra Materialized Views: http://bit.ly/1JLYAU8

[12] Batch: http://bit.ly/23xWiyp

[13] Lightweight transactions in Cassandra 2.0: http://bit.ly/1tKkuMg

[14] Java-Treiber: http://bit.ly/1Sg4wri

[15] Awaitility: http://bit.ly/1Sg4AYI

[16] OpsCenter: http://bit.ly/1VvqqEP

[17] OpsCenter compatibility: http://bit.ly/1SO5t9Y