ORACLE

ORACLE DOJO NR. 2

CARSTEN CZARSKI

Big Data: Eine Einführung

Oracle NoSQL Database, Hadoop MapReduce, Oracle Big Data Connectors

Oracle Dojo ist eine Serie von Heften, die Oracle Deutschland B.V. zu unterschiedlichsten Themen aus der Oracle-Welt herausgibt.

Der Begriff Dojo [‘do:dʒo] kommt aus dem japanischen Kampfsport und bedeutet Übungshalle oder Trainingsraum. Als „Trainingseinheiten“, die unseren Anwendern helfen, ihre Arbeit mit Oracle zu perfektionieren, sollen auch die Oracle Dojos verstanden werden. Ziel ist es, Oracle-Anwendern mit jedem Heft einen schnellen und fundierten Überblick zu einem abgeschlossenen Themengebiet zu bieten.

Im *Oracle Dojo Nr. 2* beschäftigt sich Carsten Czarski, Leitender Systemberater bei Oracle Deutschland B.V., mit der neuen Oracle NoSQL Database, Hadoop und dem Oracle Loader for Hadoop und führt Sie damit in eines der zurzeit meistdiskutierten IT-Themen ein: Big Data.

ORACLE

Inhalt

ORACLE

ORACLE DOJO NR. 2

CARSTEN CZARSKI

Big Data: Eine Einführung

Oracle NoSQL Database, Hadoop, MapReduce und der Oracle Loader for Hadoop im Zusammenspiel

VORWORT DES HERAUSGEBERS

Die IT-Welt ist nicht arm an neuen Themen. Ganz im Gegenteil. Die Frequenz und Intensität, mit der neue Themen diskutiert werden, ist wohl in keiner Branche so hoch wie im IT-Bereich.

Auf der „Hype-Skala“ ganz oben befindet sich aktuell das Thema „Big Data“. Mit Big Data können enorme Wettbewerbsvorteile erreicht werden, so die übereinstimmenden Aussagen der Analysten, und Big Data ist, wenn man den Ausführungen Glauben schenken will, die neue Art IT zu betreiben, um in völlig neue Dimensionen vorzustoßen.

So weit, so gut, so weit, so unklar.

Fakt ist, Big Data als Begriff ist neu. Noch vor zwölf Monaten (Anfang 2011) haben die meisten damit nichts Neues, nichts Spezielles und vor allem nicht die neue Heilslehre der IT verbunden. Heute sind Fachzeitschriften voll mit Artikeln zu diesem Thema. Konferenzen dazu sind überfüllt. Jeder versucht für sich zu klären und zu verstehen, ob und wie diese neuen Technologien mit Nutzen eingesetzt werden können.

Bei Oracle hat auf der Oracle OpenWorld 2011 offiziell das Big-Data-Zeitalter begonnen. Wir haben mehrere neue Softwareprodukte im Kontext von Big Data und eine neue

Big Data Appliance angekündigt. Diese Ankündigungen und auch die in der Folge zur Verfügung gestellten Produkte haben die Diskussionen mit unseren Kunden und vielen Interessenten weiter befeuert. Der Bedarf an fundierten Informationen zu diesem Thema ist enorm, und deshalb war es naheliegend, das *Dojo Nr. 2* dem Thema Big Data zu widmen.

Ich freue mich sehr, dass Carsten Czarski die nicht ganz einfache Aufgabe übernommen hat, in dieses neue Thema einzuführen. Erfahren Sie alles Wissenswerte über die neue Oracle NoSQL Database und, wenn Sie wollen, installieren Sie diese Datenbank nebenher – steigen Sie ein in die Welt von Hadoop und MapReduce und lernen Sie den Oracle Loader for Hadoop kennen. Machen Sie sich ein Bild davon, was diese neuen Technologien zu leisten imstande sind.

Ich bin mir sicher, dass dies der Beginn des ein oder anderen interessanten Projektes sein wird.

Ihr Günther Stürner
Vice President Sales Consulting

PS: Wir sind sehr an Ihrer Meinung interessiert. Anregungen, Lob oder Kritik gerne an barbara.frank@oracle.com. Vielen Dank!

1 Einführung

Big Data ist eines der derzeit meistdiskutierten IT-Themen. Immer mehr Unternehmen beschäftigen sich mit NoSQL-Datenbanken, Hadoop, MapReduce-Jobs und der Auswertung und Verwaltung immer größerer Datenmengen. Auch Oracle bietet Produkte zum Umgang mit Big Data an: Mit der **Oracle NoSQL Database** und dem **Oracle Loader for Hadoop** (letzterer als Teil der **Big Data Connectors**) seien zwei Beispiele genannt. Und mit der **Big Data Appliance** wird ein *Engineered System* speziell für Anforderungen im Big-Data-Umfeld angeboten.

Dieses Dojo gibt einen Einblick in den Themenkomplex Big Data und zeigt die Nutzung der relevanten Oracle Produkte im Zusammenspiel mit der klassischen Unternehmens-IT. Anhand eines Beispiels werden Setup und Umgang mit der Oracle NoSQL Database vorgestellt. Danach wird gezeigt, wie die Daten der NoSQL-Datenbank mit einem Map-Reduce-Job im Hadoop-Cluster ausgelesen und veredelt werden können. Schließlich wird das Ergebnis mit dem Oracle Loader for Hadoop in das bekannte RDBMS (Relational Database Management System) Oracle geladen.

Die Java-Codebeispiele dieses Dojo stehen übrigens zum Download auf **apex.oracle.com/folien** bereit. Geben Sie **bigdata-Dojo** als Schlüsselwort ein.

1.1 BIG DATA, NOSQL-DATENBANKEN UND HADOOP: WORUM GEHT ES?

Wenn man von Big Data spricht, ist mehr gemeint, als die wörtliche Übersetzung „große Datenmengen". Es geht hier um ganz spezielle große Datenmengen, nämlich solche, die in einem klassischen Data Warehouse keinen Platz finden würden: und das vor allem, weil die als „Big Data" bezeichneten Datenbestände bei weitem nicht die Strukturierung, Dichtheit und Qualität aufweisen, wie man sie in einem Data Warehouse gewohnt ist. Zur Verdeutlichung seien einige Beispiele genannt:

- **Sensordaten** – GPS-Geräte ermitteln die Position typischerweise im Sekundentakt. Obgleich mit Längen- und Breitengrad sehr wohl eine einfache Struktur vorliegt, sind die Rohdaten für eine Verarbeitung im Data Warehouse nicht „dicht" genug – so viele Einzelpositionen müssen zunächst zu sinnvollen Einheiten (beispielsweise Linienzüge) zusammengefasst werden. Neben GPS ist natürlich noch eine Fülle anderer Sensordaten denkbar.

- **Webserver-Logdateien** – Wenn stark frequentierte Seiten jeden Klick des Nutzers aufzeichnen, entsteht auch hier eine Fülle an Daten, die man in Rohform nicht in einer relationalen Datenbank oder einem Data Warehouse haben möchte. Bevor man Auswertungen machen oder die Daten analysieren kann, ist eine „Veredelung" nötig.

- **Nutzerkommentare und Social Networks** – Solche Kommentare werden häufig gemeinsam mit Clickstream-Daten erfasst und sind in Rohform ebenfalls nur schwer auszuwerten.

Das besondere Merkmal dieser Daten ist die geringe Qualität der Rohdaten bei gleichzeitigem Anfall enormer Datenmengen. Das System, in das diese Daten gelegt werden, muss auch bei hoher Last alle Daten mit kurzen Antwortzeiten aufnehmen können.

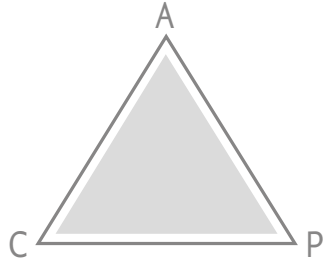
Um dies leisten zu können, wird – im Gegensatz zu einem RDBMS – auf massiv verteilte Verarbeitung gesetzt. Un- oder nur schwach strukturierte Daten werden entweder als Dateien ins HDFS (Hadoop Distributed Filesystem) oder als Key-Value-Paare in eine NoSQL-Datenbank abgelegt. Die verteilte Architektur dieser Systeme erlaubt die einfache, horizontale Skalierung und damit die Möglichkeit, ständig nach Anforderung zu wachsen.

Geht es an die Auswertung dieser Daten, so kommt Hadoop MapReduce zum Einsatz: Die gewünschten Auswertungen beziehungsweise Aggregationen werden als MapReduce-Jobs implementiert und im Hadoop-Cluster parallel ausgeführt. Abschnitt 3 dieses Dojos beschäftigt sich näher damit. Doch zunächst wird auf das Speichern von Key-Value-Paaren, NoSQL-Datenbanken und deren Unterschiede zum wohlbekannten RDBMS eingegangen.

1.2 CAP-THEOREM

Wenn es um Datenbanken und verteilte Systeme geht, wird das CAP-Theorem bedeutsam. Kurz zusammengefasst besagt das CAP-Theorem, dass es einem verteilten System nicht möglich ist, gleichzeitig die drei Anforderungen Konsistenz (C), Verfügbarkeit (A) und Partitionstoleranz (P) zu gewährleisten, es kann maximal zwei dieser Anforderungen zur gleichen Zeit erfüllen.

Abb.1: CAP-Theorem



Konsistenz (C) bedeutet dabei, dass alle Teile des Gesamtsystems stets die gleichen Daten sehen, Verfügbarkeit (A) besagt, dass alle Anfragen stets innerhalb einer geforderten maximalen Zeit beantwortet werden und Partitionstoleranz (P) bedeutet, dass das System auch bei Ausfall einzelner Teile (Partitionen) weiterhin arbeitet und antwortet.

Relationale Datenbanksysteme wie das RDBMS Oracle erfüllen in der Tat auch nur zwei dieser Anforderungen: Konsistenz und Verfügbarkeit (CA). Sobald man damit beginnt, ein solches System mit Replikationsmechanismen zu verteilen, sinkt die „Verfügbarkeit“, da eine Transaktion ja dann über alle beteiligten Systeme „committed“ werden muss. „Spielt“ man dagegen mit dem einen oder anderen Schalter zum Beeinflussen dieses Commit-Verhaltens, so schränkt man die Konsistenz über alle Teile des verteilten Systems ein.

Eine NoSQL-Datenbank legt höchsten Wert auf die verteilte Datenhaltung und damit auf die Partitionstoleranz (P). Inwieweit nun mehr Wert auf Verfügbarkeit (A) oder Konsistenz (C) gelegt wird, ist von Produkt zu Produkt verschieden, oder der Entwickler kann es, wie bei der Oracle NoSQL Database, selbst entscheiden. Es hängt also von den Anforderungen des konkreten Projektes ab, ob eine Oracle NoSQL DB als CP- oder AP-System eingesetzt wird.

1.3 NOSQL-UNTERSCHIEDE ZUM RDBMS: ACID VS. BASE

Ein klassisches RDBMS (CA) unterstützt ACID-Transaktionen, das bedeutet:

- **Atomicity** – Eine Transaktion wird ganz oder gar nicht ausgeführt.
- **Consistency** – Das RDBMS legt höchste Priorität auf die Datenkonsistenz; es werden immer konsistente Daten ausgeliefert.
- **Isolation** – Transaktionen sind voneinander unabhängig und das Datenbanksystem stellt deren gegenseitige Isolation voneinander sicher.
- **Durability** – Eine mit `Commit` bestätigte Änderung kann immer wiederhergestellt werden.

Die Tatsache, dass eine NoSQL-Datenbank ihre Priorität auf die Partitionstoleranz legt und dafür (je nach Projekt) entweder die Datenkonsistenz oder die Verfügbarkeit beziehungsweise Antwortzeit zurückstellt, hat Konsequenzen auf das Transaktionsverhalten. Es wäre ein Widerspruch, die Prioritäten auf Verfügbarkeit und Partitionstoleranz zu legen und gleichzeitig ACID zu unterstützen. Die Datenkonsistenz wird hier gerade *nicht* mehr garantiert. Also folgen NoSQL-Datenbanken nicht dem ACID, sondern dem diesem diametral gegenüberstehenden BASE-Konzept:

- **Basically Available (BA)** – Das System ist prinzipiell verfügbar, einzelne Teile können jedoch ausfallen.

- **Soft State (S), Eventually consistent (E)** –„Am Ende“ enthält das Gesamtsystem konsistente Daten; zwischenzeitlich können jedoch inkonsistente Zustände auftreten und auch an eine Applikation ausgeliefert werden.

Die wesentliche Eigenschaft eines BASE-Systems ist, dass die jederzeitige Datenkonsistenz nicht mehr das allerhöchste Gut ist. Inkonsistente Zustände können vorübergehend akzeptiert werden – und das muss man bei der Implementierung der Anwendung im Hinterkopf behalten.

1.4 KEY-VALUE-SPEICHERTECHNOLOGIE

NoSQL-Datenbanken arbeiten im Gegensatz zu RDBMS ohne feste Datenstrukturen – demnach gibt es keine Tabellen und kein Datenbankschema. Die mesiten der derzeit verfügbaren NoSQL-Datenbanken verwenden eine der folgenden Speichertechnologien:

- **Key-Value** – Alle Daten werden als Key-Value-Paare abgelegt. Ein Wert wird gemeinsam mit einem Schlüssel gespeichert, und nur mit dem Schlüssel kann man ihn wieder abrufen.

- **Document** – Alle Daten werden in Form von Dokumenten (beispielsweise XML) abgelegt. Ein Dokument wird unter einem Pfad (auch ein „Schlüssel“) abgelegt und kann mit diesem wieder abgerufen werden.

- **Graph** – Dies ist ein netzwerkorientierter Ansatz und wird demzufolge auch gerne von sozialen Netzwerken verwendet. Daten werden mitsamt Verknüpfungen zu anderen Daten gespeichert. Den Verknüpfungen kann man folgen, sodass es recht einfach ist, alle Kontakte einer Person X zu ermitteln.

Key-Value ist das generischste Verfahren; alle anderen Verfahren lassen sich auf Basis von Key-Value-Paaren aufbauen. Daher wird dieses Verfahren von der Oracle NoSQL Database verwendet. Der wesentliche Unterschied zwischen einer NoSQL-Datenbank mit Key-Value-Datenhaltung und einem RDBMS sei nochmals deutlich gemacht: **Ein Wert kann nur anhand des Schlüssels abgerufen werden – eine Abfragesprache mit Filtermöglichkeit, wie SQL, gibt es nicht.**

1.5 NOSQL-DATENBANKTECHNOLOGIE UND RDBMS IM ZUSAMMENSPIEL

Wenn man im Rahmen einer IT-Landschaft im Unternehmen über Big Data und NoSQL-Datenbanktechnologie

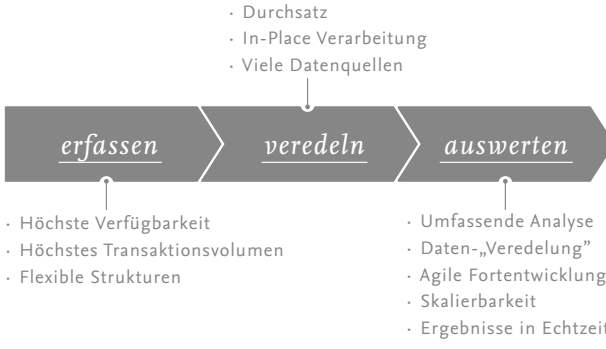
nachdenkt, so befindet man sich eigentlich niemals „auf der grünen Wiese“ – denn es ist typischerweise ein Data Warehouse, ein Reporting-System und gegebenenfalls auch eine Data-Mining-Software im Einsatz. Es muss die Frage beantwortet werden, wie die Information, die in den immensen Datenbeständen einer NoSQL-Datenbank enthalten ist, für die bereits vorhandenen Systeme und damit auch Prozesse verfügbar gemacht wird. Es geht darum, die Daten in der NoSQL-Datenbank schrittweise zu aggregieren, zu veredeln und das Ergebnis dieses Prozesses dann im „klassischen“ Data Warehouse beziehungsweise dem RDBMS zu speichern und von dort aus weiterzuverarbeiten.

1.6 DER BIG-DATA-PROZESS: ERFASSEN, VEREDELN, AUSWERTEN

Damit ist der Big-Data-Prozess definiert: Big Data werden zunächst (als Sensor- oder Clickstream-Daten) von den Quellsystemen ins HDFS oder eine NoSQL-Datenbank gespeichert. Letztere enthält alle anfallenden Daten als Key-Value-Paare. Wenn es darum geht, diese Daten zu aggregieren und zu veredeln, muss der gesamte Datenbestand durchgearbeitet werden. Daher kommen, für diesen Prozess, ebenfalls parallel ausgeführte MapReduce-Jobs in einem Hadoop-Cluster zum Einsatz. Das Ergebnis wird schließlich in strukturierter Form in ein RDBMS gela-

den, wo es als Teil des Data Warehouse mit „klassischen“ Methoden weiterverarbeitet wird.

Abb. 2: Der Big-Data-Prozess: erfassen, veredeln, auswerten



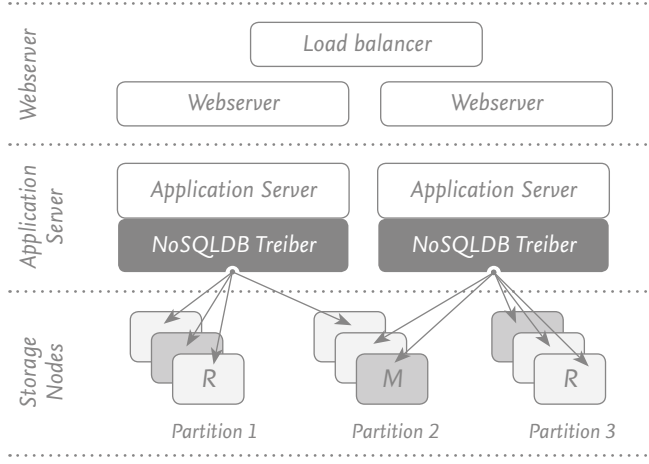
2 Oracle NoSQL Database

Die Oracle NoSQL Database wurde auf Basis der schon vor längerer Zeit von Oracle akquirierten BerkeleyDB implementiert. Neben der reinen Speicherung der Key-Value-Paare, wozu die BerkeleyDB ohne Weiteres in der Lage wäre, enthält die Oracle NoSQL DB zusätzliche Mechanismen zur verteilten Datenhaltung: Die gespeicherten Daten müssen auf verschiedene Rechnerknoten verteilt werden, es muss eine Replikation stattfinden, und das System muss mit dem Ausfall einzelner Teile fertig werden.

2.1 ARCHITEKTUR DER ORACLE NOSQL DATABASE

Da es keine Abfragesprache für die NoSQL-Datenbank gibt, greift die Java-Anwendung mit Programmier-Aufrufen (API) zu. Die Kommunikation übernimmt dabei der Oracle NoSQL-Datenbanktreiber.

Abb. 3: Topologie der Oracle NoSQL Database



Die NoSQL-Datenbank wird auf einem Cluster mit Storage Nodes installiert. Ein Storage Node ist ein physikalischer Rechnerknoten mit lokalen Plattensystemen, auf dem die

NoSQL-Datenbanksoftware läuft. Im Gegensatz zum Real Application Cluster des RDBMS Oracle, bei dem ein Shared Storage zum Einsatz kommt, verwendet die NoSQL-Datenbank ein Shared-Nothing-Konzept. Alle Storage Nodes speichern ihre Daten lokal.

Zur Verteilung der Daten auf die Storage Nodes werden die Schlüssel auf Partitionen abgebildet. Beim Aufsetzen der NoSQL-Datenbanktopologie entscheidet man sich für die Anzahl Partitionen, die man verwenden möchte. Die Key-Value-Paare werden beim Schreiben per Hash-Algorithmus gleichmäßig über alle Partitionen verteilt. Der Hash-Algorithmus ist für die gesamte Lebensdauer der NoSQL-Datenbank stabil. Die Partitionen selbst werden auf verschiedene Storage Nodes verteilt, wodurch sich das verteilte System ergibt.

Damit das Gesamtsystem auch bei Ausfall eines Storage Nodes verfügbar bleibt, müssen die Daten repliziert werden. Eine Replikationsgruppe besteht aus einem Master und mindestens einer Replika und ist für die redundante Speicherung einer bis mehrerer Partitionen verantwortlich – empfohlen ist allerdings genau eine Partition pro Replikationsgruppe. Der Replikationsfaktor ist definiert als „Master + Anzahl Replikas“. Ein Master und zwei Replikas bedeuten also einen Replikationsfaktor von 3. Werden Daten in der

NoSQL-DB gespeichert, so geschieht dies immer zunächst auf dem Master – anschließend wird repliziert. Leseanfragen können, je nach der Einstellung zur Lesekonsistenz, entweder vom Master oder von einer der Replikas bedient werden.

Fällt ein Master aus, so wählen die verbliebenen Knoten der Replikationsgruppe einen neuen – aus diesem Grund ist ein Replikationsfaktor von wenigstens 3 zu empfehlen, nur dann sind bei Ausfall des Masters noch zwei Knoten übrig, von denen einer der neue Master werden kann.

Als Beispiel sei angenommen, dass die zu speichernden Key-Value-Paare auf 50 Partitionen verteilt werden sollen – eine Replikationsgruppe soll (wie empfohlen) auch nur eine Partition aufnehmen. Als Replikationsfaktor wird ebenfalls die minimale Empfehlung von 3 angenommen. Das bedeutet im Einzelnen:

- Es werden 50 Replikationsgruppen benötigt – eine pro Partition.
- Der Replikationsfaktor ist 3 – es werden also 50 Master und 100 Replikas gebraucht.
- Im Ergebnis werden 150 Storage Nodes benötigt.

Oder, anders herum gedacht:

- Es stehen 100 Storage Nodes zur Verfügung.
- Es wird wieder ein Replikationsfaktor von 3 angestrebt.
- Möchte man wiederum eine Partition pro Replikationsgruppe einrichten, so können die Daten auf 33 Partitionen verteilt werden.

Eine einmal eingerichtete Topologie ist, im zur Zeit der Drucklegung dieses DOJOs verfügbaren Release 1 der Oracle NoSQL Database, statisch und kann nicht im laufenden Betrieb geändert werden.

2.2 INSTALLATION DER ORACLE NOSQL DATABASE

2.2.1 DOWNLOAD UND AUSPACKEN DES ARCHIVS

Die Oracle NoSQL DB benötigt mindestens JavaSE6 (JDK 1.6.0 u25); empfohlen ist JavaSE7. Die aktuell installierte Version kann mit dem Kommando `java -version` herausgefunden werden:

```
$ java -version
java version "1.7.0_01"
Java(TM) SE Runtime Environment (build 1.7.0_01-b08)
Java HotSpot(TM) Client VM (build 21.1-b02, mixed mode)
```

Nach dem Download der ca. 15 MB aus dem Oracle Technology Network (OTN) und dem Auspacken des ZIP-Archivs stellt sich die Verzeichnisstruktur in etwa wie folgt dar:

```
|-- bin
|-- doc
|   |-- AdminGuide
|   |-- GettingStartedGuide
|   |-- examples
|   |-- javadoc
|   `-- misc
|-- examples
|   |-- hadoop
|   |-- hello
|   `-- schema
`-- lib
```

Natürlich muss das ZIP-Archiv auf allen Rechnerknoten, auf denen die NoSQL-DB laufen soll, ausgepackt werden – dabei ist es empfehlenswert, die Verzeichnisnamen auf allen beteiligten Knoten gleich zu halten. Künftig wird dieses Verzeichnis als `NOSQL_HOME` bezeichnet – für Skripte beziehungsweise Batchdateien ist es empfehlenswert, sich eine entsprechende Umgebungsvariable anzulegen.

Damit die spätere Arbeit etwas einfacher wird, empfiehlt es sich, die Dateien `kvstore-{version}.jar` und `kvclient-`

{version.jar} in kvstore.jar beziehungsweise kvclient.jar zu kopieren oder umzubenennen. Auf Linux- und UNIX-Systemen ist das Erzeugen eines symbolischen Links am einfachsten:

```
$ cd ${NOSQL_HOME}/lib
$ ln -s kvclient-1.2.123.jar kvclient.jar
$ ln -s kvstore-1.2.123.jar kvstore.jar
```

Danach kann ein kurzer Selbsttest gemacht werden:

```
$ java -jar ${NOSQL_HOME}/lib/kvclient.jar
```

Die Version 1.2.123 muss gegebenenfalls durch die tatsächlich heruntergeladene ersetzt werden. Die Ausgabe sollte in etwa so aussehen:

```
11gR2.1.2.123
```

2.2.2 FESTLEGUNG DER VERZEICHNISSE UND TCP/IP-PORTS

Dann kann es mit dem Einrichten der NoSQL-DB-Topologie losgehen. Da die einzelnen Knoten des NoSQL-Datenbankclusters über das Netzwerk miteinander kommunizieren, sollten zu Beginn noch die TCP/IP-Ports festgelegt werden. Folgende werden benötigt:

- Der Port, über den die Anwendung die NoSQL-DB anspricht; in diesem Dojo wird stets 5000 verwendet.
- Der Port, auf dem die webbasierte Admin-Konsole läuft; im Dojo wird 5001 verwendet.
- Außerdem wird noch ein Bereich freier Ports, über welche die einzelnen Knoten miteinander kommunizieren, benötigt; im Dojo wird 5010, 5020 – also 5010 bis 5020 – verwendet.

Schließlich ist noch ein Verzeichnis für die Ablage der Daten selbst erforderlich – dies wird im Dojo als `KVROOT` bezeichnet. Wieder sollte der Verzeichnispfad auf allen Knoten gleich sein – es sollte aber *kein* gemeinsames (NFS-) Verzeichnis gewählt werden: Die NoSQL-Datenbank ist ein Shared-Nothing-System.

2.2.3 STARTEN DER NOSQL-DATENBANK AUF DEN STORAGE NODES

Legen Sie nun das Verzeichnis `KVROOT` auf jedem beteiligten Rechnerknoten an und erzeugen Sie die Speicherstrukturen:

```
$ mkdir -p ${KVROOT}

$ java -jar ${NOSQL_HOME}/lib/kvstore.jar
makebootconfig /
```

```
    -root ${KVROOT} \
    -port 5000 \
    -admin 5001 \
    -host <hostname> \
    -harange 5010,5020
```

Als nächstes kann die NoSQL-Datenbanksoftware auf den Knoten gestartet werden:

```
$ nohup java -jar ${NOSQL_HOME}/lib/kvstore.jar \
    start -root ${KVROOT} &
```

Mit dem Kommando `jps` lässt sich prüfen, ob die Prozesse laufen:

```
$ jps -m
4811 kvstore.jar start -root KVROOT1
4857 ManagedService -root KVROOT1 -class Admin -service
BootstrapAdmin.10000 -config config.xml
```

Wenn die NoSQL-DB-Software auf allen Rechnerknoten läuft, kann als nächstes die konkrete NoSQL-Datenbank mitsamt ihrer Topologie eingerichtet werden – also die vorher geplante Anzahl Partitionen und Replikationsgruppen sowie die Zuordnung der Storage Nodes.

2.2.4 STARTEN DES KOMMANDOZEILENWERKZEUGS DER NOSQL-DB

Am besten ist es, die Topologie mit dem Kommandozeilenwerkzeug (Command Line Interface, CLI) einzurichten: Man kann die Kommandos in einer Skriptdatei sammeln und danach ähnliche Topologien automatisiert erzeugen. Das Werkzeug kann auf irgendeinem der bereits mit der NoSQL-DB-Software ausgestatteten Rechnerknoten gestartet werden:

```
$ java -jar ${NOSQL_HOME}/lib/kvstore.jar runadmin \
   -port 5000 \
   -host <hostname>
kv->
```

Das Werkzeug meldet sich mit dem Prompt `kv->`. Das Kommando `help` gibt Ihnen einen Überblick über die verfügbaren Befehle. Diese können entweder einzeln und sofort oder per Stapelverarbeitung ausgeführt werden.

2.2.5 EINRICHTEN DER NOSQL-DATENBANKTOPOLOGIE

Die Einrichtung einer NoSQL-Datenbanktopologie beginnt mit der Festlegung des „Datenbanknamens“ – unter diesem *KVStore Name* spricht die Applikation später eine bestimmte NoSQL-Datenbank an:

```
kv-> configure Dojostore
Lost connection to AdminService.
Reconnecting....
kv->
```

Die Meldung `Lost connection ...` muss nicht zwingend erscheinen.

Nun wird ein *Data Center* eingerichtet. Ein NoSQL-DB-Store mitsamt seinen Storage Nodes und Replikationsgruppen wird einem Data Center zugeordnet. In der aktuellen Version haben Data Center zwar keine funktionale Bedeutung (sie sind rein deklarativ), müssen aber dennoch konfiguriert werden:

```
kv-> plan -execute deploy-datacenter "DC_Munich"
"Kommentar zu Munich"
1
kv-> show topology
dc=[dc1] name=DC_Munich
```

Das Kommando `plan` nimmt einen Konfigurationsbefehl entgegen. Wird zusätzlich das Flag `-execute` angegeben, so wird der Befehl sofort ausgeführt, andernfalls landet er auf dem Stapel für die spätere Verarbeitung. Die zurückgegebene „1“ ist eine hochgezählte Nummer für jeden mit `plan` abgesetzten Befehl. Das Kommando `cancel` entfernt ein auf den Stapel gelegtes Kommando wieder.

Mit `show topology` kann man sich jederzeit das Ergebnis seiner Arbeit ansehen. Das Data Center trägt die ID „1“ (`dc1`).

Als nächstes wird dem Data Center der erste Storage Node zugeordnet. Sie können zu diesem Zeitpunkt nur einen Storage Node einrichten – bevor die anderen zugeordnet werden, muss ein Administrationsprozess konfiguriert werden:

```
kv-> plan -execute deploy-sn 1 <hostname-storagenode-1> 5000
2
kv-> show topology
dc=[dc1] name=DC_Munich
   sn sn=[sn1]  dc=dc1 storagenode001:5000 status=RUNNING
```

Die Ausgabe von `show topology` zeigt Ihnen die ID des Storage Nodes (`sn1`). Diese brauchen Sie nun, um auf diesem Knoten den Administrationsprozess zu starten:

```
kv-> plan -execute deploy-admin 1 5001
```

Als nächstes müssen Sie mindestens einen Storage-Node-Pool einrichten. Auch diese sind, wie die Data Center, für spätere Versionen vorgesehen und haben in der aktuellen Version der Oracle NoSQL DB keine weitere Bedeutung. Konfiguriert werden müssen sie dennoch. Für die Zuordnung der Storage Nodes zum Pool benötigen Sie wiederum die IDs der Storage Nodes:

```
kv-> addpool DojoPool
kv-> joinpool DojoPool 1
AllStorageNodes: sn1
DojoPool: sn1
kv-> show topology
```

Nun können Sie Ihre übrigen Storage Nodes zunächst dem Data Center und dann dem Storage-Node-Pool hinzufügen:

```
kv-> plan -execute deploy-sn 1 <hostname-storagenode-2> 5000
kv-> :
kv-> plan -execute deploy-sn 1 <hostname-storagenode-n> 5000
kv-> show topology
dc=[dc1] name=DC_Munich
   sn=[sn1]  dc=dc1 storagenode-001:5000 status=RUNNING
   :
   sn=[sn6]  dc=dc1 storagenode-006:5000 status=RUNNING
kv-> joinpool DojoPool 2 3 4 5 6
AllStorageNodes: sn1 sn2 sn3 sn4 sn5 sn6
DojoPool: sn1 sn2 sn3 sn4 sn5 sn6
```

Nun folgt als letzter Schritt die finale Einrichtung Ihrer NoSQL-Datenbank. Festgelegt werden die Anzahl der Partitionen und der Replikationsfaktor. Daraus bildet die NoSQL-Datenbank die Replikationsgruppen. Im Beispiel liegen 6 Storage Nodes vor. Es sollen 10 Partitionen bei einem Replikationsfaktor von 3 erstellt werden. Also werden pro

Replikationsgruppe 3 Storage Nodes benötigt. Da 6 Storage Nodes vorhanden sind, wird eine Replikationsgruppe für 5 Partitionen zuständig sein:

```
kv-> plan -execute deploy-store DojoPool 3 10
11
kv->
```

An der mit `show topology` zurückgegebenen Topologie erkennt man sehr gut, dass zwei Replikationsgruppen gebildet wurden (`rg1,rg2`). Einer der drei Storage Nodes fungiert als Master für Schreibzugriffe, die anderen beiden als Replika:

```
kv-> show topology
dc=[dc1] name=DC_Munich
   sn=[sn1]  dc=dc1 storagenode001:5000 status=RUNNING
      rn=[rg1-rn1] sn=sn1 status=RUNNING
   sn=[sn2]  dc=dc1 storagenode002:5000 status=RUNNING
      rn=[rg1-rn2] sn=sn2 status=RUNNING
   sn=[sn3]  dc=dc1 storagenode003:5000 status=RUNNING
      rn=[rg1-rn3] sn=sn3 status=RUNNING
   sn=[sn4]  dc=dc1 storagenode004:5000 status=RUNNING
      rn=[rg2-rn1] sn=sn4 status=RUNNING
   sn=[sn5]  dc=dc1 storagenode005:5000 status=RUNNING
      rn=[rg2-rn2] sn=sn5 status=RUNNING
   sn=[sn6]  dc=dc1 storagenode006:5000 status=RUNNING
      rn=[rg2-rn3] sn=sn6 status=RUNNING
```

Damit ist die Konfiguration der NoSQL-Datenbank abgeschlossen. Mit `quit` können Sie das Administrationswerkzeug nun verlassen.

2.3 NUTZUNG DER ORACLE NOSQL DATABASE IN EINER JAVA-ANWENDUNG

2.3.1 KEY-VALUE-PAARE IN DER NOSQL-DB

Die Oracle NoSQL DB ist derzeit nur für die Programmiersprache Java verfügbar. Der Zugriff erfolgt allein durch API-Aufrufe – eine Abfragesprache wie SQL gibt es nicht. Da die Oracle NoSQL DB die Key-Value-Datenhaltung verwendet, ergeben sich drei grundlegende Operationen:

- `PUT` – Speichern eines Key-Value-Paares
- `GET` – Abrufen eines Key-Value-Paares
- `DELETE` – Löschen eines Key-Value-Paares

Der Schlüssel wird durch die Java-Klasse `oracle.kv.Key` repräsentiert. Meist werden Schlüssel aus einem String erzeugt, die Oracle NoSQL DB erlaubt jedoch auch das Erstellen aus einem Byte-Array.

Der Schlüssel eines Key-Value-Paares in der NoSQL-DB ist in einen „Major-“ und einen „Minor-“Teil zweigeteilt.

Zum Zugriff auf einen bestimmten Wert ist der komplette Schlüssel erforderlich. Allerdings bestimmt nur der Major-Teil die Partition und damit den Storage Node, auf dem das Key-Value-Paar abgelegt wird. Key-Value-Paare mit gleichem Major-Key werden damit auf dem gleichen Storage Node abgelegt und können mit speziellen Methoden wie `multiGet()` auch auf einmal abgerufen werden.

2.3.2 SPEICHERN EINES KEY-VALUE-PAARES

Zum Speichern eines konkreten Key-Value-Paares muss sich ein Java-Programm zuerst mit der NoSQL-Datenbank verbinden, dann eine Instanz der Klasse `oracle.kv.Key` für den Schlüssel und eine Instanz der Klasse `oracle.kv.Value` für den Wert erzeugen und diese mit einem Aufruf von `oracle.kv.KVStore.Put()` abspeichern:

```
import oracle.kv.*;

public class NoSQLDB_Store {
   public static void main(String args[]) {
      KVStore store = KVStoreFactory.getStore (
         new KVStoreConfig(
            "Dojostore",
            new String[] {"storagenode-001:5000",
            "storagenode-002:5000"}
         )
```

```
        );
        store.put(
           Key.createKey("DiesIstEinKey"),
           Value.createValue(new String("Dies ist ein
           Wert").getBytes())
        );

        store.close();
        }
    }
```

Das Programm wird wie jedes Java-Programm mit `javac` kompiliert. Achten Sie darauf, dass sich das Java-Archiv `kvclient.jar` aus dem Verzeichnis `NOSQL_HOME/lib` im `CLASSPATH` befindet. Kompilieren und starten Sie das Programm dann wie folgt:

```
$ export CLASSPATH=.:${NoSQL_HOME}/lib/kvclient.
jar:${CLASSPATH}
$ javac NoSQLDB_Store.java
$ java NoSQLDB_Store
```

Zuerst verbindet sich das Java-Programm mithilfe von `KVStoreFactory.getStore()` mit der NoSQL-Datenbank. Dazu wird der, bei Einrichtung der NoSQL-Datenbank festgelegte, KVStore Name und wenigstens ein beteiligter Rechnerknoten benötigt. Besser ist es allerdings, hier mehrere Rechnerknoten anzugeben, denn es könnte ja

sein, dass einer der hier angegebenen Knoten gerade nicht erreichbar ist. Der NoSQL-DB-Treiber arbeitet die Liste ab und verbindet sich auf den ersten erreichbaren Server.

Von dort holt sich der Treiber die Informationen zur Topologie der NoSQL-Datenbank – er erfährt also nach dem Verbinden, welche Storage Nodes verfügbar sind, wie viele Partitionen vorhanden sind und wie die Replikation eingerichtet wurde. Es ist also nicht zwingend nötig, alle Rechnerknoten anzugeben.

Das zweite Kommando ruft bereits die `Put()` Methode auf. Aus dem Text „`DiesIstEinKey`“ wird mit `Key.createKey` eine Instanz der Klasse `oracle.kv.Key` erzeugt. Als Wert dient in diesem Beispiel ebenfalls ein Java-String, mit dem eine Instanz von `oracle.kv.Value` generiert wird. Der Wert muss allerdings zuerst in ein Byte-Array umgewandelt werden – dafür ist der Aufruf von `getBytes()` verantwortlich.

Zum Abschluss wird die Verbindung zur NoSQL-Datenbank geschlossen.

2.3.3 ABRUFEN DES KEY-VALUE-PAARES

Das Abrufen funktioniert ganz ähnlich wie das Speichern. Zunächst muss eine Verbindung zur NoSQL-Datenbank erfolgen, dann wird wiederum eine Instanz von `oracle.kv.Key`

erzeugt, damit wird die Methode `oracle.kv.KVStore.get()` aufgerufen:

```
import oracle.kv.*;

public class NoSQLDB_Get {
   public static void main(String args[]) {
      KVStore store = KVStoreFactory.getStore (
         new KVStoreConfig(
            "Dojostore",
            new String[] {"storagenode001:5000",
                          "storagenode002:5000"}
         )
      );

      ValueVersion vv = store.get(
         Key.createKey("DiesIstEinKey")
      );

      Value v = vv.getValue();

      System.out.println(new String(v.getValue()));

      store.close();
      }
   }
```

Das Programm verbindet sich zunächst zur NoSQL-Datenbank, erzeugt dann, wie beim Speichern des Key-Value-Paares, einen Schlüssel als Instanz der Klasse `oracle.kv.Key` und ruft anschließend das Key-Value-Paar ab.

Allerdings liefert die Methode `get()` eine Instanz von `oracle.kv.ValueVersion` zurück: nämlich den Wert selbst und zusätzlich Informationen zur Version des jeweiligen Wertes. Das ist wichtig, wenn ein gegebenenfalls veralteter Wert von einer Replika gelesen wurde.

Mit der Methode `getValue()` wird der konkrete Wert aus dem `ValueVersion`-Objekt extrahiert, in einen Java-String gewandelt und schließlich ausgegeben. Das letzte Kommando schließt die Verbindung zur NoSQL-Datenbank.

2.4 STEUERUNG DER KONSISTENZ UND VERFÜGBARKEIT

Wie eingangs im Abschnitt 1.2 erläutert, kann ein verteiltes System, bei dem die Partitionstoleranz (P) bereits als Ziel festgelegt wurde, noch zwischen höchster Verfügbarkeit und höchster Datenkonsistenz entscheiden. Die Oracle NoSQL Database erlaubt dem Entwickler, in seiner Anwendung entweder Verfügbarkeit (A) oder Datenkonsistenz (C) zu priorisieren. Die folgenden Abschnitte beschreiben, wie das geht.

2.4.1 SCHREIBKONSISTENZ FESTLEGEN

Die Schreibkonsistenz einer Transaktion wird mit einer Instanz von `oracle.kv.Durability` festgelegt. Diese kann entweder beim Aufruf von KVStore.put für ein einzelnes Key-Value-Paar oder beim Aufbau der Verbindung zur NoSQL-DB als Default festgelegt werden. Im `Durability` Objekt werden festgelegt:

- die Commit-Policy für den Master
- die Commit-Policy für die Replikas
- die Replica Acknowledgement-Policy

Mit der Commit-Policy wird sowohl für den Master als auch für die Replikas festgelegt, ob die Anwendung:

- warten soll, bis der Storage Node die Transaktion in die Logdatei geschrieben und diese auf Platte synchronisiert hat (`SYNC`),
- warten soll, bis der Storage Node die Transaktion in die Logdatei geschrieben hat – auf das Synchronisieren auf Platte wird aber nicht mehr gewartet (`WRITE_NO_SYNC`)
- oder überhaupt nicht auf den Storage Node warten soll (`NO_SYNC`).

Mit der Replica Acknowledgement-Policy entscheidet der Entwickler, ob die Anwendung:

- warten soll, bis alle Replikas bestätigt haben, dass die Transaktion geschrieben wurde (`ReplicaAckPolicy.ALL`),
- warten soll, bis die einfache Mehrheit der Replikas die Transaktion bestätigt hat (`ReplicaAckPolicy.SIMPLE_MAJORITY`)
- oder gar nicht auf die Replikas warten soll (`ReplicaAckPolicy.NONE`).

Im Java-Code kann das nun wie folgt aussehen. Im Beispiel soll die allerhöchste Verfügbarkeit erreicht werden; Datenkonsistenz wird nicht als wichtig angesehen:

```
:
KVStoreConfig storeConf = new KVStoreConfig(
   "Dojostore",
   new String[] {"storagenode001:5000",
   "storagenode002:5000"}
);
storeConf.setDurability(
 new Durability(
   Durability.SyncPolicy.NO_SYNC,
   Durability.SyncPolicy.NO_SYNC,
   Durability.ReplicaAckPolicy.NONE
   )
);
```

```
   KVStore store = KVStoreFactory.getStore (storeConf);
   store.put(
      Key.createKey("DiesIstEinKey"),
      Value.createValue(new String("Dies ist ein Wert").
getBytes())
   );
:
```

Das umgekehrte Beispiel legt höchsten Wert auf Datenkonsistenz – durch Probleme im Netzwerk oder auf den einzelnen Storage Nodes kann sich die Antwortzeit nun aber verlängern:

```
:
storeConf.setDurability(
   new Durability(
      Durability.SyncPolicy.SYNC,
      Durability.SyncPolicy.SYNC,
      Durability.ReplicaAckPolicy.ALL
   )
);
:
```

2.4.2 LESEKONSISTENZ FESTLEGEN

Analog zum schreibenden Zugriff kann der Entwickler auch für lesende Zugriffe festlegen, was ihm wichtiger ist: konsistente Daten oder maximale Performance. Dazu dient die Klasse `oracle.kv.Consistency`:

- Der Entwickler kann mit `Consistency.ABSOLUTE` festlegen, dass nur vom Master gelesen werden darf. Damit ist sichergestellt, dass stets der aktuelle Inhalt zurückgeliefert wird.
- Mit `Consistency.Time` kann der Entwickler auch das Lesen von einer Replika erlauben; die Parameter legen fest, wie „veraltet" das zurückgelieferte Ergebnis sein darf.
- `Consistency.Version` arbeitet wie `Consistency.Time`, allerdings wird hier nicht mit Zeiteinheiten wie Sekunden gearbeitet, sondern mit Versionsinformationen.
- Schließlich akzeptiert der Entwickler mit `Consistency.NONE_REQUIRED` jedes Ergebnis – es kann von jeder Replika gelesen werden und darf beliebig veraltet sein.

Im Java-Code sieht das wie folgt aus. Wiederum wird die Policy für die Lesekonsistenz als Default für die NoSQL-DB-Verbindung gesetzt. Zunächst wird das Lesen vom Master erzwungen:

```
:
storeConf.setConsistency(Consistency.ABSOLUTE);
:
```

Das Gegenbeispiel akzeptiert auch beliebig veraltete Ergebnisse:

```
:
storeConf.setConsistency(Consistency.NONE_REQUIRED);
:
```

Darüber hinaus kann auch festgelegt werden, dass Ergebnisse maximal 5 Sekunden hinter dem Zeitstempel des Masters zurückliegen dürfen. Auf die Erfüllung dieser Bedingung wird maximal 10 Sekunden gewartet:

```
:
storeConf.setConsistency(
      new Consistency.Time(
         5, // Daten dürfen 5 Sekunden „zurückliegen“
         java.util.concurrent.TimeUnit.SECOND,
         10, // Timeout nach 10 Sekunden
         java.util.concurrent.TimeUnit.SECOND
   )
);
:
```

2.5 WAS PASSIERT BEIM AUSFALL EINES STORAGE NODES?

Wie sich der Ausfall eines Rechnerknotens auf die NoSQL-Datenbank als Ganzes auswirkt, hängt zum einen davon ab, ob dieser innerhalb einer Replikationsgruppe als Master oder Replika fungiert, und zum anderen, mit welchen Konsistenzeinstellungen die Anwendungen arbeiten.

2.5.1 AUSFALL EINES MASTER-KNOTENS

Fällt der Master innerhalb einer Replikationsgruppe aus, wären prinzipiell keinerlei Schreibzugriffe mehr möglich, da diese immer über den Master laufen. Sobald der oder die Admin-Prozesse, welche die Verfügbarkeit der Storage Nodes laufend überwachen, feststellen, dass ein Master-Knoten ausgefallen ist, wird unter den verbliebenen Replikas ein neuer Master gewählt. Genau aus diesem Grund ergibt sich die Empfehlung eines Replikationsfaktors von mindestens 3 – denn nur dann sind noch mindestens 2 Knoten vorhanden, von denen einer der neue Master werden kann. In der Logdatei des Administrationsprozesses sieht der Vorfall dann wie folgt aus:

```
16:06:52:37 UTC+1 INFO [admin1] [admin1] sn6: Service
status: UNREACHABLE 16:07:33:26 UTC+1 INFO [rg2-rn1]
JE: Replica IO exception: Expected bytes:
```

```
16:07:33:26 UTC+1 INFO [rg2-rn1] JE: Exiting inner Replica
loop.
16:07:33:26 UTC+1 INFO [rg2-rn1] JE: Replica stats - Lag
waits: 0 Lag wait
time: 0ms. VLSN waits: 0 Lag wait time: 0ms.
16:07:33:26 UTC+1 INFO [rg2-rn1] State: UNKNOWN, Master:
none
16:07:33:26 UTC+1 INFO [rg2-rn1] JE: Election initiated;
election #1
16:07:33:26 UTC+1 INFO [rg2-rn1] JE: Started election
thread
16:07:35:30 UTC+1 INFO [rg2-rn1] JE: Master changed to rg2-
rn2
```

Der Administrator der NoSQL-Datenbank sollte den ausgefallenen Knoten nun recht schnell wiederherstellen und, wie in 2.5.3 beschrieben, wieder einbinden. Nach dem erneuten Einbinden wird er als Replika fungieren – denn die Aufgabe des Masters hat ja nun ein anderer Knoten übernommen.

2.5.2 AUSFALL EINES REPLIKA-KNOTENS

Bei Ausfall eines Knotens, der als Replika fungiert, findet natürlich keine Neuwahl des Masters statt – denn der läuft ja noch. Ob sich dies überhaupt auf das Gesamtsystem auswirkt, hängt von den Einstellungen zur Lese- beziehungsweise Schreibkonsistenz in der jeweiligen Anwendung ab.

Wurde die Replica Acknowledgement-Policy auf „ALL“ eingestellt, wird also eine Bestätigung von allen Replikas erwartet, dann wird die Anwendung zunächst warten und nach dem Time-out eine Fehlermeldung bekommen – denn die Replika läuft ja nicht. Steht die Policy dagegen auf „SIMPLE_MAJORITY“ oder gar „NONE“, so kann das System weiterlaufen. Auch hier gilt natürlich, dass der Administrator aktiv werden sollte.

2.5.3 WIEDERHERSTELLEN EINES KNOTENS

Im einfachsten Fall reicht natürlich ein Reboot der ausgefallenen Hardware oder ein Neustart der NoSQL-DB-Software auf besagtem Rechnerknoten aus, um den Storage Node wiederherzustellen. Muss die Hardware jedoch ausgetauscht werden, ist ein wenig mehr zu tun. Es sei angenommen, dass der ausgefallene Knoten nicht wieder hochgefahren werden kann – vielmehr wird ein neuer Storage Node bereitgestellt. Wie beim initialen Erstellen der NoSQL-Datenbank müssen nun auch auf der Ersatzhardware die Speicherstrukturen erstellt werden:

```
$ mkdir -p ${KVROOT}

$ java -jar ${NOSQL_HOME}/lib/kvstore.jar makebootconfig \
      -root ${KVROOT} \
```

```
    -port 5000 \
    -admin 5001 \
    -host storagenode-replace001 \
    -harange 5010,5020
```

Anschließend wird die NoSQL-DB-Software auf diesem Knoten gestartet:

```
$ nohup java -jar ${NOSQL_HOME}/lib/kvstore.jar \
    start -root ${KVROOT} &
```

Mit dem Kommandozeilenwerkzeug (Abschnitt 2.2.4) wird der neue Rechnerknoten der Topologie hinzugefügt. Fügen Sie ihn auch dem Storage-Node-Pool hinzu:

```
kv-> plan -execute deploy-sn 1 storagenode-replace001 5000
kv-> joinpool DojoPool 7
AllStorageNodes: sn1 sn2 sn3 sn4 sn5 sn6 sn7
DojoPool: sn1 sn2 sn3 sn4 sn5 sn6 sn7

kv-> show topology
dc=[dc1] name=DC_Munich
   sn=[sn1]  dc=dc1 storagenode001:5000 status=RUNNING
      rn=[rg1-rn1] sn=sn1 status=RUNNING
   :
   sn=[sn6]  dc=dc1 storagenode020:5000 status=UNREACHABLE
      rn=[rg2-rn3] sn=sn6 status=UNREACHABLE   single-op avg
latency=8.029667 ms   multi-op avg latency=0.0 ms
```

```
sn=[sn7]  dc=dc1 storagenode-replace-001:5000 status=RUNNING
```

Nun wird der ausgefallene Knoten in der Topologie durch den neuen Knoten ersetzt:

```
kv-> plan -execute migrate-storagenode 6 7 5000
kv-> show topology
dc=[dc1] name=DC_Munich
   sn=[sn1]  dc=dc1 storagenode001:5000 status=RUNNING
      rn=[rg1-rn1] sn=sn1 status=RUNNING
   :
   sn=[sn6]  dc=dc1 storagenode020:5000 status=UNREACHABLE
   sn=[sn7]  dc=dc1 storagenode-replace-001:5000
status=RUNNING
      rn=[rg2-rn3] sn=sn7 status=RUNNING   single-op
avg
latency=8.029667 ms
multi-op avg latency=0.0 ms
```

Anschließend ist der neu hinzugefügte Knoten aktiv, der alte sollte nun nicht mehr hochgefahren werden. In der Ausgabe von `show topology` bleibt der alte Knoten im Release 1 mit dem Status `UNREACHABLE` stehen – in späteren Versionen wird man ihn aus der Topologie entfernen können.

2.6 ADMINISTRATION DER NOSQL-DATENBANK

Auch eine NoSQL-Datenbank muss administriert werden – die Aufgaben eines NoSQL-Datenbankadministrators sind jedoch nur teilweise mit denen eines RDBMS-DBA vergleichbar. So fallen beispielsweise alle Aufgaben hinsichtlich Sicherheit, Nutzerkonten und Zugriffskontrolle weg – denn die NoSQL-Datenbank kennt schlicht kein Nutzerkonzept. Andere Aufgaben, wie Backup und Recovery, haben bei einer NoSQL-Datenbank einen anderen Fokus: Schließlich sind alle Daten durch die Replikation ohnehin mehrfach gespeichert. In diesem Dojo seien drei typische Administrationsaufgaben vorgestellt: Backup, Recovery und Upgrade.

2.6.1 BACKUP

Ein Backup der NoSQL-Datenbank wird gemacht, indem ein Snapshot gezogen wird. Das geschieht mit dem Kommandozeilenwerkzeug der NoSQL-DB.

```
$ java -jar ${NOSQL_HOME}/lib/kvstore.jar runadmin \
       -port 5000 \
       -host <hostname>
kv-> snapshot create sn_1_Dojo
Created snapshot named 120125-113514-sn_1_Dojo
```

Die Snapshots werden, wie die NoSQL-Datenbank selbst, über die Knoten verteilt gespeichert. Jeder Knoten enthält in einem speziellen Verzeichnis die Snapshot-Dateien. Auf dem Storage Node `storagenode001` unserer NoSQL-Datenbank `Dojostore` könnte das Verzeichnis wie folgt aussehen:

```
$ cd ${KVROOT}
$ cd Dojostore
$ cd storagenode001
$ cd rg1-rn1
$ cd snapshots
$ cd 120125-113514-sn_1_Dojo/files
$ ls
00000000.jdb 00000002.jdb 00000004.jdb 00000006.jdb
00000001.jdb 00000003.jdb 00000005.jdb 00000007.jdb
```

Diese Dateien müssen nun von allen Storage Nodes auf das Backup-Medium kopiert werden. Alle Dateien zusammen ergeben dann das Backup der gesamten NoSQL-Datenbank (welches, wie man sich denken kann, sehr groß werden kann). Snapshots werden auf *allen* Storage Nodes generiert – also sowohl auf den Master- als auch auf den Replika-Knoten.

Wichtig zu wissen ist, dass die NoSQL-Datenbank die Konsistenz eines Backups nur über die Partitionen einer Replikationsgruppe hinweg garantiert. In unserem im Abschnitt

2.2 eingerichteten Beispiel sind 2 Replikationsgruppen mit je 5 Partitionen vorhanden. Die Snapshots einer Replikationsgruppe sind in sich konsistent. Über die beiden Replikationsgruppen hinweg können sich aber Inkonsistenzen ergeben. Auch wenn diese durch parallele Ausführung der Snapshots minimiert werden – gänzlich ausschließen kann man sie nicht.

Natürlich können Snapshots auch gelöscht werden. Hierfür ist wiederum das Kommandozeilenwerkzeug zuständig:

```
kv-> snapshot remove 120125-113514-sn_1_Dojo
Removed snapshot 120125-113514-sn_1_Dojo
```

Vorhandene Snapshots werden mit dem `snapshot-list` Kommando angezeigt:

```
kv-> snapshot list
120125-113514-sn_1_Dojo
120126-113514-sn_2_Dojo
120127-113514-sn_3_Dojo
```

2.6.2 RECOVERY

Wurde eine NoSQL-Datenbank zerstört oder ist ein Recovery aus anderen Gründen nötig, so bieten sich zwei Wege an.

Eine Möglichkeit ist das Load-Werkzeug `oracle.kv.util.Load`. Es arbeitet sich schrittweise durch alle Snapshots

durch, liest die Key-Value-Paare aus und speichert sie in eine neue NoSQL-DB ab. Die Topologie der neuen NoSQL-DB kann auch eine völlig andere sein – dieses Verfahren ist am ehesten mit einem Import des klassischen RDBMS vergleichbar. Angenommen, die Snapshots wären auf das Verzeichnis `/var/backups/nosqldb/snapshots` gesichert und wir hätten bereits eine neue NoSQL-Datenbank mit Namen `new_Dojostore` mit neuer Topologie erzeugt, dann würde der folgende Aufruf alle Inhalte des Snapshots in diese neue NoSQL-Datenbank laden:

```
$ java -jar KVHOME/lib/kvstore.jar load \
       -source /var/backups/nosqldb/
       snapshots/120125-113514
sn_1_Dojo \
       -store  new_Dojostore \
       -host   new_storagenode100 \
       -port   5000 \
       -status my_statusfile
```

Während des Ladevorgangs wird der Status (was bereits geladen wurde) in der Datei `my_statusfile` festgehalten. Im Release 1 merkt sich das Load-Werkzeug alle fertig geladenen Partitionen. Bricht der Vorgang ab, so müssen nur die noch nicht fertig geladenen Partitionen erneut geladen werden.

Da das Load-Werkzeug, wie beschrieben, ein Backup in jede beliebige NoSQL-Datenbank zurückspielen kann, können mit dessen Hilfe auch Topologieänderungen vorgenommen werden. Im aktuellen Release kann eine einmal festgelegte NoSQL-Datenbanktopologie – also die Anzahl der Partitionen, der Replikationsfaktor und damit die Anzahl der verwendeten Storage Nodes – nicht mehr geändert werden. Ist eine Änderung dennoch nötig, so kann das mit dem Load-Werkzeug erreicht werden: Zunächst richtet man die neue Topologie ein, dann erzeugt man einen Snapshot der alten NoSQL-Datenbank und nutzt das Load-Werkzeug zur Übertragung der Daten von der alten in die neue Datenbank.

Wenn die Topologie der NoSQL-Datenbank sich durch das Recovery nicht ändern soll, gibt es einen schnelleren Weg: Eine neue NoSQL-Datenbank kann direkt auf Basis der Snapshot-Dateien erstellt werden. Logischerweise muss diese dann die gleiche Topologie wie die alte haben. Und das gilt nicht nur für die Anzahl der Storage Nodes und den Replikationsfaktor, sondern für alle Details, auch Rechnernamen und TCP/IP-Ports.

Dazu muss nun jede Snapshot-Datei auf den Storage Node kopiert werden, für den sie gedacht ist. Dort muss sie in ein spezielles Verzeichnis namens `recovery` unterhalb von `KVROOT` abgelegt werden:

```
$ cd ${KVROOT}/Dojostore/sn1/rg1-sn1
$ mkdir recovery
$ cd recovery

$ mv /backups/Dojostore/sn1/rg1-rn1/snapshots/120125-
113514-sn_1_Dojo .
```

Dieser Schritt muss auf jedem Storage Node durchgeführt werden. Somit enthält das Verzeichnis `KVROOT` auf jedem Storage Node ein zusätzliches Verzeichnis `recovery` mit dem Snapshot für den jeweiligen Knoten. Nun wird die NoSQL-Datenbank auf allen Storage Nodes durchgestartet:

```
$ java -jar KVHOME/lib/kvstore.jar stop -root ${KVROOT}
$ nohup java -jar ${NOSQL_HOME}/lib/kvstore.jar \
   start -root ${KVROOT} &
```

Wenn ein Verzeichnis `recovery` vorhanden ist, wird es von der NoSQL-Datenbanksoftware erkannt, dann automatisch umbenannt und von da an genutzt.

Alle Recovery-Verfahren können die Datenbank nur bis zum Zeitpunkt, zu dem der Snapshot gezogen wurde, wiederherstellen. Änderungen, die seitdem gemacht wurden, gehen verloren. Das ist ein weiterer wesentlicher Unterschied zwischen einer NoSQL-Datenbank und einem RDBMS.

2.6.3 UPGRADE

Ein Upgrade der Oracle NoSQL Database ist denkbar einfach. Da die NoSQL-Datenbank verteilt ist, bleibt das Gesamtsystem auch während des Upgrades verfügbar. Nötig sind auf allen Storage Nodes „nur“ drei Schritte:

1. Installation der neuen Version der Oracle NoSQL Database in ein neues Verzeichnis: `${NOSQLDB_NEWHOME}`
2. Stoppen der NoSQL-Datenbank
3. Starten der NoSQL-Datenbank mit der neuen Software

Diese Schritte sollten seriell erfolgen, also ein Knoten nach dem anderen. Damit nicht ständig neue Master gewählt werden müssen (weil ein Master gerade ein Upgrade erfährt und durchgestartet wird), sollte das Upgrade zuerst für alle Replikas durchgeführt werden und erst danach für die Master-Knoten.

3 Parallele Verarbeitung mit Hadoop und MapReduce

3.1 HADOOP: WAS IST DAS?

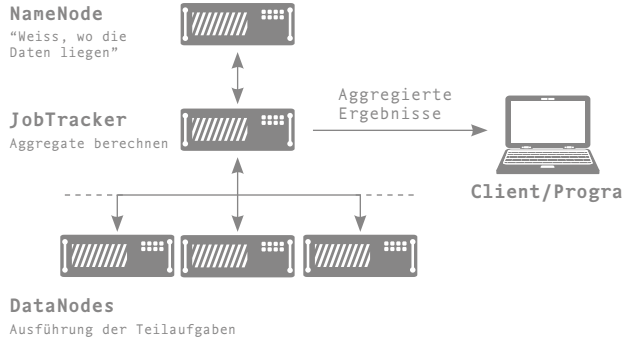
Im Abschnitt 2 haben Sie erfahren, wie eine NoSQL-Datenbank im Allgemeinen und die Oracle NoSQL Database im Besonderen funktioniert. Der wesentliche Unterschied zum RDBMS ist, dass freie Abfragen und Filter nicht möglich sind. Ein gespeicherter Wert (*Value*) kann nur anhand des Schlüssels (*Key*) abgerufen werden. Hat man zum Beispiel Daten einer Webseiten-Personalisierung in einer NoSQL-Datenbank gespeichert und möchte man wissen, welche Nutzer Interesse an „Filmen“ oder „Computerspielen“ eingetragen haben, so muss man alle Key-Value-Paare der Reihe nach durcharbeiten, diejenigen von Interesse selektieren und die anderen verwerfen.

Auf entsprechend großen Datenbeständen ist ein solches Vorgehen wiederum nur massiv parallel sinnvoll. So, wie die NoSQL-Datenbank selbst auf viele Rechnerknoten im Cluster verteilt ist, werden auch die Jobs, welche die Daten der NoSQL-Datenbank durcharbeiten, verdichten und aggregieren, massiv parallel im Cluster ausgeführt.

Die Ausführung solch paralleler Jobs muss man jedoch nicht selbst koordinieren und steuern – dazu kommt das Open-Source-Framework Hadoop zur Anwendung. Ein Hadoop-Cluster ist eine *Shared Nothing Compute Architecture* – also ein Cluster, in dem Daten verteilt verarbeitet (nicht gespeichert) werden. Natürlich müssen sich die einzelnen Rechnerknoten miteinander synchronisieren und Daten austauschen: Das geschieht über ein verteiltes Dateisystem – dem Hadoop Distributed Filesystem (HDFS). Dieses ist, wie schon gesagt, *shared nothing*, das heißt, die einzelnen Knoten halten jeweils einen Teil der Daten und tauschen diese aus – ein Shared Storage kommt nicht zum Einsatz. Darüber hinaus sind folgende Termini für das Verständnis des Hadoop-Clusters wichtig:

- **Hadoop-Client** – ist die Schnittstelle zum Cluster; startet die Verarbeitung, tut selbst aber nichts.
- **Name Node** – verwaltet Informationen über das HDFS (welcher Knoten hält welche Daten) und den Cluster.
- **Job Tracker** – ordnet die Teilaufgaben auf die einzelnen Knoten zu (Query Coordinator) und koordiniert die Ausführung beziehungsweise startet fehlgeschlagene Jobs neu.
- **Data Nodes** – diese enthalten das HDFS und damit Teile der Daten und führen die Aufgaben, also die MapReduce-Jobs, aus.

Abb. 4: Schematische Darstellung eines Hadoop-Clusters



Hadoop läuft auf Linux- und Windows-Systemen, allerdings ist der Betrieb der verteilten Installation auf Windows bislang nur wenig bis gar nicht getestet, sodass Windows nur als Entwicklungsplattform zu empfehlen ist.

Hadoop kann grundsätzlich in zwei Varianten heruntergeladen werden: Die Originalpakete stehen auf der Webseite der Apache Foundation (**hadoop.apache.org**) bereit. Allerdings wird in der Praxis meist mehr als die reine Hadoop-Engine benötigt – zusätzlich braucht man diverse Werkzeuge und Hilfspakete. Lädt man diese ebenfalls einzeln von den jeweiligen Webseiten herunter, so muss man sehr stark auf deren Kompatibilität zueinander achten.

Als Alternative stehen fertige Distributionen bereit. Diese Distributionen sind getestete, aufeinander abgestimmte Pakete, die, neben der Hadoop-Engine selbst, auch besagte Werkzeuge und Hilfspakete enthalten. Ein Vertreter dieser Distributionen ist die **Cloudera Distribution for Hadoop (CdH)**, die auch auf der Oracle Big Data Appliance enthalten ist. Eine freie Version kann von den Webseiten von Cloudera (**www.cloudera.com**) heruntergeladen werden. Die Beispiele in diesem Dojo basieren auf **CdH Version 3**.

3.2 AUFSETZEN EINES „HADOOP-PSEUDO-CLUSTERS“

Ein Hadoop-Cluster kann entweder als Einzelplatzinstallation, im pseudo-verteilten oder im „echt-“ verteilten Betrieb installiert werden. In diesem Dojo wählen wir den pseudo-verteilten Betrieb; man bekommt damit schon einen Eindruck von der verteilten Verarbeitung, muss jedoch nicht mehrere Knoten einzeln konfigurieren. Der Schritt von der pseudo-verteilten zur tatsächlich verteilten Installation ist dann nicht mehr besonders groß.

Nach dem Download des ca. 70 MB großen ZIP-Archivs packen Sie es in ein Verzeichnis Ihrer Wahl auf dem Linux-Server aus:

```
$ tar -xzf hadoop-0.20.2-cdh3u3.tar.gz
```

Die Verzeichnisstruktur sieht danach wie folgt aus:

```
|-- CHANGES.txt
|-- LICENSE.txt
|-- NOTICE.txt
|-- README.txt
|-- bin
|-- etc
|-- hadoop-sample.zip
|-- include
|-- lib
|-- :
```

Hadoop setzt eine vorhandene Java-Umgebung voraus. Stellen Sie also sicher, dass (wie für die Oracle NoSQL DB) eine solche auf Ihrem System installiert ist. Navigieren Sie dann zur Datei `${HADOOP_HOME}/conf/hadoop-env.sh` und tragen Sie dort den Pfad zu Ihrer Java-Installation (`JAVA_HOME`) wie folgt ein:

```
# Set Hadoop-specific environment variables here.
# The only required environment variable is JAVA_HOME.
All others are
# optional.  When running a distributed configuration
it is best to
# set JAVA_HOME in this file, so that it is correctly
defined on
```

```
# remote nodes.

# The java implementation to use.  Required.
export JAVA_HOME=/opt/jdk1.7.0_02/

# Extra Java CLASSPATH elements.  Optional.
:
```

Alle Zeilen, die mit einem „#“ beginnen, sind Kommentare. Navigieren Sie dann ins Verzeichnis `${HADOOP_HOME}/conf` und stellen Sie sicher, dass für den pseudo-verteilten Modus folgende Konfigurationsdateien und Inhalte vorhanden sind:

core-site.xml:

```
<configuration>
   <property>
      <name>fs.default.name</name>
      <value>hdfs://localhost:9000</value>
   </property>
</configuration>
```

hdfs-site.xml:

```
<configuration>
   <property>
      <name>dfs.replication</name>
```

```
      <value>1</value>
   </property>
</configuration>
```

mapred-site.xml:

```
<configuration>
   <property>
      <name>mapred.job.tracker</name>
      <value>localhost:9001</value>
   </property>
</configuration>
```

Das ist die absolute Minimalkonfiguration; für den produktiven Betrieb eines Hadoop-Clusters müssen wesentlich mehr Einstellungen gemacht werden. Für die verteilte (oder in unserem Fall pseudo-verteilte) Verarbeitung müssen die Knoten untereinander per *Secure Shell* (SSH) ohne Eingabe von Passwörtern kommunizieren können. Dazu müssen Schlüssel ausgetauscht werden – für den pseudo-verteilten Betrieb reichen nachstehende Kommandos aus:

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Machen Sie danach einen einfachen Test. Das folgende Kommando muss Ihnen sofort – ohne ein Passwort zu verlangen – eine Eingabeaufforderung bereitstellen:

```
$ ssh localhost
```

Nun kann der Hadoop-Pseudo-Cluster eingerichtet werden. Stellen Sie zunächst das verteilte Dateisystem, das HDFS, bereit.

```
$ bin/hadoop namenode -format
```

Starten Sie schließlich den Hadoop-Pseudo-Cluster:

```
$ bin/start-all.sh
```

3.3 ERSTE SCHRITTE MIT DEM HDFS

Wenn ein Knoten im Hadoop-Cluster eine Aufgabe erledigt hat, werden die Ergebnisse meist ins HDFS geschrieben. Allerdings kann das HDFS nicht ohne weiteres, wie ein „richtiges" Dateisystem, als Verzeichnis gemountet und dann angesprochen werden – der Zugang zum HDFS erfolgt mit den Kommandozeilenwerkzeugen des Hadoop-Frameworks. Das nachstehende Kommando listet die verfügbaren HDFS-Kommandos auf:

```
$ cd ${HADOOP_HOME}
$ bin/hadoop dfs
Usage: java FsShell
           [-ls <path>]
           [-lsr <path>]
           [-du <path>]
```

```
        [-dus <path>]
        :
```

Das folgende Kommando erzeugt ein Listing des HDFS-Wurzelverzeichnisses:

```
$ cd ${HADOOP_HOME}
$ bin/hadoop dfs -ls /
Found 2 items
drwxr-xr-x  - root supergroup      0 2012-01-12 13:36 /tmp
drwxr-xr-x  - root supergroup      0 2012-01-12 13:37 /user
```

Mit dem Kommando `put` kopiert man eine Datei vom „normalen“ Linux-Dateisystem ins HDFS:

```
$ cd ${HADOOP_HOME}
$ echo "Dojo: Dies ist ein Test" > mylocalfile.txt
$ bin/hadoop dfs -put mylocalfile.txt myhdfsfile.txt
$ bin/hadoop dfs -ls
Found 1 item
-rw-r--r-- 1 root supergroup 24 2012-01-30 16:38
/user/root/myhdfsfile.txt
```

Analog dazu können Dateien aus dem HDFS herauskopiert (`get`), Verzeichnisse erstellt (`mkdir`), gelöscht (`rm`, `rmr`) oder umbenannt (`mv`) werden. Der Umgang mit dem HDFS ist recht wichtig für die Arbeit mit dem Hadoop-Cluster, da, wie schon erwähnt, Ergebnisse eines MapReduce-Jobs meist aus diesem gelesen und in dieses geschrieben werden.

Damit sind alle Voraussetzungen gegeben, sodass ein erster einfacher MapReduce-Job ausprobiert werden kann. Während viele Beispiele im Internet einen MapReduce-Job beschreiben, der mit Dateien im HDFS arbeitet, wird das Beispiel in diesem Dojo auf der Oracle NoSQL DB arbeiten. Wer darüber hinaus nach weiteren Beispielen für MapReduce sucht, wird auf der Hadoop-Website (**hadoop.apache.org**) fündig.

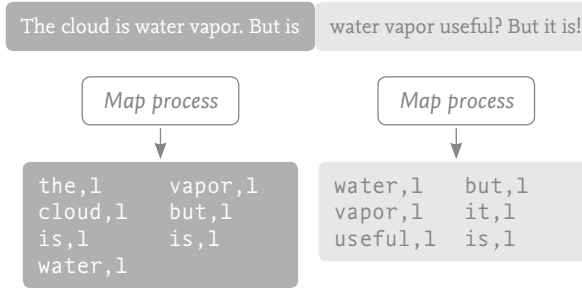
3.4 MAPREDUCE: WAS IST DAS?

Ein Hadoop-Cluster führt MapReduce-Jobs aus. In einem MapReduce-Job ist die zu erledigende Aufgabe nicht „einfach herunterkodiert", vielmehr hält sich der Job an ein bestimmtes Schema, eben MapReduce. Wie der Name schon nahelegt, besteht ein MapReduce-Job aus zwei Komponenten.

3.4.1 MAPPER

Der Mapper erstellt aus den Daten, die als Input in den MapReduce-Job gegeben werden, Key-Value-Paare – anders formuliert, er „mappt" die Daten auf Key-Value-Paare. Soll ein MapReduce-Job, wie im Hadoop-Einsteigertutorial, Wörter eines Fließtexts zählen, so wird jedes Wort auf genau ein Key-Value-Paar abgebildet. Das Wort ist der Schlüssel, der Wert ist stets „1", denn das Wort ist einmal vorgekommen. Die so

Abb. 5: Der Mapper bildet die Eingabedaten auf Key-Value-Paare ab



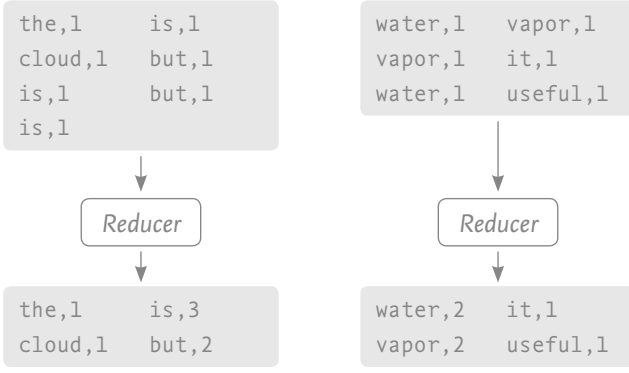
erzeugten Key-Value-Paare werden dann vom MapReduce-Framework nach Schlüsseln sortiert, zusammengefasst (das muss man nicht selbst machen) und „schlüsselweise“ an den Reducer übergeben.

3.4.2 REDUCER

Der Reducer bekommt nun pro Aufruf einen Schlüssel und alle Werte, die diesen Schlüssel tragen, als Array übergeben. Diese müssen nun verarbeitet und die Ergebnisse der Verarbeitung wiederum als ein oder mehrere Key-Value-Paare ausgegeben werden. Damit ist klar, woher der Name Reducer kommt. Die vielen Werte zu einem Schlüssel werden auf ein oder wenige Key-Value-Paare *reduziert*.

Natürlich kann es auch vorkommen, dass ein Reducer genauso viele oder gar noch mehr Key-Value-Paare erzeugt, als hineingegeben wurden; das sind jedoch Spezialfälle, auf die in diesem Dojo nicht eingegangen werden kann.

Abb. 6: Der Reducer aggregiert die Key-Value-Paare des Mappers



Zusätzlich können noch Partitionierer zum Einsatz kommen, welche die Key-Value-Paare, die vom Mapper ausgegeben werden, anhand eigener Kriterien partitionieren – diese sollen hier jedoch nicht betrachtet werden. Die Ergebnisse des Reducers sind auch die Ergebnisse des MapReduce-Jobs – je nach Jobkonfiguration werden sie entweder ins

HDFS oder in ein anderes Zielsystem geschrieben. Auch eine NoSQL-Datenbank oder, wie sich zum Ende des Dojo zeigen wird, das RDBMS Oracle sind denkbar.

3.5 IMPLEMENTIERUNG EINES EINFACHEN MAPREDUCE-JOBS

Im Folgenden wird ein einfaches Beispiel für einen MapReduce-Job vorgestellt: Zunächst füllen wir die Oracle NoSQL Database mit Daten. Ein Java-Programm (noch nicht Teil des MapReduce-Jobs) speichert 100.000 Key-Value-Paare in die NoSQL-DB ab. Als Schlüssel dienen die Zahlen von 1 bis 100.000, als Wert dient eine Zufallszahl zwischen 1 und 100 ohne Nachkommastellen. Legen Sie mit folgendem Code die Datei `NoSQLDB_Fill.java` an, kompilieren Sie diese mit `javac` und starten Sie das Programm danach.

```
import oracle.kv.*;

public class NoSQLDB_Fill {
   public static void main(String args[]) {
      KVStoreConfig storeConf =  new KVStoreConfig(
         "Dojostore",
         new String[] {"storagenode001:5000",
         "storagenode002:5000"}
      );
```

```
        KVStore store = KVStoreFactory.getStore (storeConf);

        for (int i=0;i<100000;i++) {
            store.put(
                Key.createKey(String.valueOf(i)),
                Value.createValue(
                    String.valueOf(Math.round(Math.random() *
                    100)).getBytes()
                )
            );
        }
        store.close();
    }
}
```

Im „echten“ Leben sieht eine NoSQL-Datenbank ganz ähnlich aus – natürlich stehen sinnvollere und vor allem mehr Daten in den Key-Value-Paaren. Als Dojo-Beispiel soll dies aber genügen. Nun werden die Daten mit einem MapReduce-Job verdichtet: Es soll gezählt werden, wie oft jede der Zufallszahlen zwischen 1 und 100 vorkommt. Das Ergebnis des MapReduce-Jobs sind demnach wiederum Key-Value-Paare, die wie folgt aussehen:

- Als Schlüssel dient die Zufallszahl zwischen 1 und 100.
- Wie oft diese Zufallszahl in den Ursprungsdaten vorkam, soll der Wert des Key-Value-Paares sein.

Die Oracle NoSQL DB bringt die Hadoop-Integration bereits „out-of-the-box“ mit. In der API enthalten ist die Klasse `oracle.kv.hadoop.KVInputFormat` – damit kann dann die Oracle NoSQL DB direkt als Datenquelle für einen MapReduce-Job eingerichtet werden. Die Mapper-Klasse des MapReduce-Jobs sieht so aus:

```
public static class Map
extends Mapper <Text, Text, Text, IntWritable> {
   private final static IntWritable value = new
IntWritable(1);
   Text key = new Text();

   @Override
   public void map(Text keyArg, Text valueArg, Context
context)
   throws IOException, InterruptedException {
      key.set(valueArg.toString());
      context.write(key, value);
   }
}
```

Aus der NoSQL-Datenbank werden sowohl die Schlüssel als auch die Werte als Datentyp `Text` angeliefert. Der Mapper-Prozess soll diese auf neue Key-Value-Paare mit den Datentypen `Text` (für den Schlüssel) und `IntWritable` (für

den Wert) abbilden. Zu erwähnen ist übrigens noch, dass hier nicht mit den normalen Java-Datentypen, sondern mit eigenen Hadoop-Klassen gearbeitet wird – letztere sind für die verteilte Verarbeitung geeignet.

Die Logik selbst ist in diesem Beispiel einfach: Der Wert (also die Zufallszahl) aus der NoSQL-Datenbank wird als neuer Schlüssel gesetzt und der neue Wert ist die Zahl „1" – für *ein* Vorkommen dieser Zahl. Der Mapper-Prozess verarbeitet jedes einzelne Key-Value-Paar aus der NoSQL-Datenbank – dessen Ausgabe sind also 100.000 Key-Value-Paare folgender Struktur:

```
Input                       Output
Key          Value          Key          Value
------------ ------------   ------------ ----------
1              67             67           1
2              12             12           1
3              1              1            1
:              :              :            :
99999          12             12           1
100000         56             56           1
```

Man sieht, dass die ursprünglichen Schlüssel „2" und „99999" den Wert „12" haben – der Mapper-Prozess bildet für beide das neue Key-Value-Paar „12, 1". Das Hadoop-Framework fasst die Ausgabe des Mapper-Prozesses nach

Schlüsseln zusammen und übergibt diese neuen Key-Value-Paare nach Schlüsseln sortiert an den Reducer, welcher mit folgendem Code implementiert ist:

```
public static class
Reduce extends Reducer  <Text, IntWritable, Text,
IntWritable>{
   private IntWritable result = new IntWritable();

   public void reduce(
      Text key, Iterable<IntWritable> values, Context
context
   )  throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {sum += val.
get();}
      result.set(sum);
      context.write(key, result);
   }
}
```

Der Reducer bekommt als Eingabe einen Schlüssel (Variable *key*) und ein Array mit allen Werten aus dem Mapper-Prozess, die diesen Schlüssel haben (Variable *values*).

Der Code ist wiederum einfach: Das Array wird in einer Schleife durchgearbeitet, die Werte (die immer gleich „1“ sind)

werden aufsummiert (also gezählt) und als Ausgabe wird ein Key-Value-Paar zurückgegeben; der Schlüssel ist wiederum die Zufallszahl und der „Wert" ist das Ergebnis der Zählung.

```
Input                          Output
Key             Value [ ]       Key          Value
--------------- -------------  ----------- -----------
67              {1,1,1,1,...}  67          617
12              {1,1,1}        12          3
1               {1,1,1,1,...}  1           132
:               :              :           :
56              {1,1,1,1...}   56          872
```

Als Ergebnis liefert der Reduce-Job 100 Key-Value-Paare mit dem Ergebnis der Zählung zurück. Der folgende Code macht aus den beiden Java-Klassen einen MapReduce-Job und bindet ihn ins Hadoop-Framework ein:

```
@Override
public int run(String[] args) throws Exception {
  Job job = new Job(getConf());
  job.setJarByClass(DojoHadoopJob.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  job.setMapperClass(Map.class);
  job.setReducerClass(Reduce.class);
```

```
    job.setInputFormatClass(KVInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.
    class);
    TextOutputFormat.setOutputPath(job, newPath(
    "hdfs-output"));
    KVInputFormat.setKVStoreName("Dojostore");
    KVInputFormat.setKVHelperHosts(
      new String[]{"storagenode001:5000",
      "storagenode002:5000"}
    );

    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
}
```

Ein MapReduce-Job wird in der Hadoop-API als Instanz der Klasse `Job` repräsentiert. Die Java-Klasse, welche das MapReduce-Interface implementieren muss, und damit der konkret auszuführende Code, wird dem Job durch die Aufrufe von `setJarByClass`, `setMapperClass` und `setReducerClass` mitgeteilt – man sieht, dass das Hadoop-Framework hier völlig dynamisch arbeitet.

Auch das Eingabeformat in den MapReduce-Job kann hier festgelegt werden. Im Beispiel wird das `KVInputFormat` festgelegt; diese Java-Klasse ist Teil der Oracle NoSQL Database und besagt, dass die Key-Value-Paare aus der

NoSQL-DB die Eingabe (der Input) in den Hadoop-Job sein sollen. Natürlich braucht es Verbindungsdaten zur laufenden NoSQL-Datenbank: Die Aufrufe von `setKVStoreName` und `setKVHelperHosts` erledigen das. Als Output wird die Klasse `TextOutputFormat` verwendet, das bedeutet, dass das Ergebnis in einfache Textdateien geschrieben wird, die im HDFS abgelegt werden. Man sieht hier sehr schön die Trennung zwischen Job-Konfiguration und der MapReduce-Implementierung: Dass die Ergebnisse unter `hdfs-output` ins HDFS geschrieben werden, wird nicht in den MapReduce-Job programmiert, sondern ist Teil der Job-Konfiguration.

Fasst man alles zusammen, so sieht der Java-Code (`DojoHadoopJob.java`) wie folgt aus:

```
import java.io.IOException;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.mapreduce.lib.reduce.*;
import org.apache.hadoop.util.*;

import oracle.kv.hadoop.KVInputFormat;
```

```
public class DojoHadoopJob extends Configured
implements Tool {
   // Mapper Klasse - siehe oben

   // Reducer Klasse - siehe oben

   // Job-Konfiguration - siehe oben

   public static void main(String[] args) throws
Exception {
      int ret = ToolRunner.run(new DojoHadoopJob(),
args);
      System.exit(ret);
   }
}
```

Vor dem Kompilieren muss die Client-Bibliothek der NoSQL-Datenbank der Hadoop-Installation zugänglich gemacht werden. Am einfachsten ist es, wenn Sie die Datei `kvclient-{version}.jar` aus dem Verzeichnis `{NOSQL_HOME}/lib` in das Verzeichnis `{HADOOP_HOME}/lib` kopieren und dort in `kvclient.jar` umbenennen. Dann wird der Code des MapReduce-Jobs wie folgt kompiliert – dabei sind zwei Dinge wichtig. Erstens muss sowohl die verwendete Schnittstelle zur NoSQL-DB als auch die Hadoop-API Teil des Java `CLASSPATH` sein und zweitens muss der kompilierte Code in ein Unterverzeichnis (hier: `compiled_output`) gelegt werden.

```
$ export HADOOP_VERSION=0.20.2-cdh3u3
$ export CLASSPATH=.:${HADOOP_HOME}/lib/kvclient.
jar:${HADOOP_HOME}
/hadoop-core-${HADOOP_VERSION}.jar:${CLASSPATH}
$ mkdir compiled_output
$ javac -d compiled_output DojoHadoopJob.java
```

Als nächstes wird der kompilierte Code in ein JAR-Archiv verpackt:

```
$ jar -cvf DojoHadoopJob.jar -C compiled_output/ .
added manifest
adding: DojoHadoopJob.class(in = 1868) (out= 986)
(deflated 47%)
adding: DojoHadoopJob$Map.class(in = 1595) (out= 640)
(deflated 59%)
:
```

Dann ist alles fertig. Nun kann der Hadoop-Job gestartet werden:

```
$ ${HADOOP_HOME}/bin/hadoop jar DojoHadoopJob.jar
DojoHadoopJob \
                                -libjars ${HADOOP_HOME}/
lib/kvclient.jar
```

Während der Ausführung sind folgende Ausgaben auf dem Bildschirm zu erkennen:

```
12/01/30 17:01:40 INFO mapred.JobClient:
Running job: job_201201121336_0039
12/01/30 17:01:41 INFO mapred.JobClient:
map 0% reduce 0%
12/01/30 17:02:14 INFO mapred.JobClient:
map 50% reduce 0%
12/01/30 17:02:35 INFO mapred.JobClient:
map 100% reduce 100%
12/01/30 17:02:43 INFO mapred.JobClient:
Job complete:
job_201201121336_0039
12/01/30 17:02:43 INFO mapred.JobClient:
Counters: 29
12/01/30 17:02:43 INFO mapred.JobClient:
Job Counters
12/01/30 17:02:43 INFO mapred.JobClient:
Launched reduce tasks=1
12/01/30 17:02:43 INFO mapred.JobClient:
SLOTS_MILLIS_MAPS=54726
:
```

Das Ergebnis wird ins Verzeichnis `hdfs-output` im HDFS geschrieben. Das schauen Sie sich am besten zunächst als Listing an:

```
$ bin/hadoop dfs ls hdfs-output
Found 3 items
```

```
-rw-r--r--   1 root supergroup    0 2012-01-30 17:02
wcout1/_SUCCESS
drwxr-xr-x   - root supergroup    0 2012-01-30 17:01
wcout1/_logs
-rw-r--r--   1 root supergroup 1608 2012-01-30 17:02
/part-r-00000
```

Holen Sie dann die Datei `part_r_00000` (das sind die Ergebnisse) ins „normale“ Dateisystem:

```
$ bin/hadoop dfs -get hdfs-output/part-r-00000 .
```

Danach können Sie das Ergebnis betrachten:

```
$ cat part-r-00000
0       506
1       1009
10      953
100     494
11      926
12      1027
13      981
14      1040
:
```

Im HDFS steht nun also das erste Ergebnis einer mitunter ganzen Kette von MapReduce-Jobs. Dieses Ergebnis lässt sich nun wiederum als Input für den nächsten MapReduce-Job festlegen, der diese Daten dann nochmals weiterverar-

beitet. Wenn schon feststeht, dass die Ergebnisse eines Jobs an einen anderen weitergereicht werden, sollte man jedoch nicht das `TextOutputFormat` verwenden; das `SequenceFileOutputFormat` ist wesentlich besser geeignet, da die Key-Value-Paare in kompakter Binärform im HDFS abgelegt werden. Natürlich muss der nachfolgende Job folgerichtig das `SequenceFileInputFormat` verwenden.

Am Ende der Verarbeitungskette stehen Daten im Zielformat – aus den Unmengen an Daten in der NoSQL-Datenbank wurde dann eine Untermenge extrahiert, die nun im Unternehmen weiterverarbeitet werden kann.

Und an diesem Punkt wird sich der Kreis zur „klassischen" Technologie schließen: Denn das Ergebnis einer mitunter komplexen MapReduce-Verarbeitungskette ist ja nicht dazu gedacht, als Textdatei stehen zu bleiben – vielmehr soll es den Weg ins „klassische" Data Warehouse finden, um dort Teil des normalen Reportings beziehungsweise Data Minings zu werden. An dieser Stelle kommen die Oracle Big Data Connectors zum Einsatz – und hier besonders der Oracle Loader for Hadoop.

3.6 ORACLE LOADER FOR HADOOP

Der Oracle Loader for Hadoop ist ein Ladewerkzeug, welches als MapReduce-Job für den Hadoop-Cluster implementiert ist. Es wird typischerweise als letzte Stufe einer MapReduce-Verarbeitungskette gestartet und lädt die berechneten, verdichteten oder aggregierten Daten in eine Tabelle im RDBMS Oracle. Wie alle MapReduce-Jobs hat auch der Oracle Loader for Hadoop eine Eingabe- und eine Ausgabeschnittstelle.

Folgende Eingabeformate werden unterstützt:

- Separierte Textdateien (beispielsweise komma- oder tabulatorsepariert) im HDFS
- AVRO(Apache AVRO Projekt)-Binärdateien im HDFS
- Hive-Tabellen im HDFS

Die „Ausgabe“ des Jobs ist das eigentliche Laden der Daten ins RDBMS Oracle. Auch hier wird mehr als eine Variante unterstützt:

- SQL-Insert-Kommandos in die Oracle-Tabelle per JDBC
- Erzeugen einer externen Tabelle im SQL*-Loader-Format (Dateien im HDFS)

- Erzeugen einer externen Tabelle im Data-Pump-Format (Dateien im HDFS)

Für dieses Dojo sei angenommen, dass als Ergebnis einer MapReduce-Verarbeitungskette folgende tabulatorseparierte Datei im HDFS steht:

```
$ bin/hadoop dfs -cat mapreduce-jobs-output/
part-r-00000
0     506    1009   1038   1023   1054   949    998    1020
10    953    926    1027   981    1040   1022   1028   1017
20    990    981    1010   1059   990    1004   1004   1064
30    992    996    1028   1026   1027   944    978    982
40    979    1002   957    950    956    973    1038   988
50    992    954    996    943    991    1013   989    994
60    979    1027   984    962    976    1006   1038   995
:
```

Dieses Ergebnis soll in eine Tabelle im RDBMS Oracle geladen werden. Im Schema SCOTT befindet sich die Tabelle `MATRIX` wie folgt:

```
SQL> desc matrix
 Name                  Null?    Typ
 --------------------- -------- ----------------
 C                              NUMBER
 C0                             NUMBER
 C1                             NUMBER
```

```
 C2                                NUMBER
 :                                 :
 C6                                NUMBER
 C7                                NUMBER
SQL>
```

Beachten Sie, dass die Tabelle im RDBMS zu den Daten, welche der Oracle Loader for Hadoop laden soll, passen muss. Insbesondere die Datentypen müssen zueinander passen.

3.6.1 DOWNLOAD UND INSTALLATION

Laden Sie den Oracle Loader for Hadoop aus dem Oracle Technology Network herunter und packen Sie das ZIP-Archiv auf dem Rechner oder dem Rechnerknoten aus, auf dem das Hadoop-Framework installiert ist. Die Verzeichnisstruktur sieht in etwa so aus:

```
|--examples
|--jlib
|   |-- avro-1.5.4.jar
|   |-- avro-mapred-1.5.4.jar
|   |-- commons-math-2.2.jar
|   |-- jackson-core-asl-1.5.2.jar
```

```
|   |-- oraloader-examples.jar
|   |-- oraloader.jar
|   |-- osdt_cert.jar
|   `-- osdt_core.jar
`--lib
    |-- libclntsh.so.11.1
    |-- libnnz11.so
    |-- libociei.so
    `-- libolhl1.so
```

Navigieren Sie dann nochmals zur Datei `${HADOOP_HOME}/conf/hadoop-env.sh` und fügen Sie der Umgebungsvariable `HADOOP_CLASSPATH` wie folgt das Verzeichnis `${ORACLELOADER_HOME}/jlib/*` an. Trennen Sie mehrere Einträge gegebenenfalls durch einen Doppelpunkt. Wenn Sie einen „echten“ Hadoop-Cluster verwenden, muss dies natürlich auf allen Cluster-Knoten geschehen:

```
:
# Extra Java CLASSPATH elements.  Optional.
export HADOOP_CLASSPATH=/opt/hadoop/olh/jlib/*
:
```

3.6.2 KONFIGURATION UND START DES ORACLE LOADER FOR HADOOP

Der Oracle Loader for Hadoop wird nun mit einer XML-Datei konfiguriert. Die Struktur der XML-Konfigurationsdatei ist sehr einfach:

```
<property>
   <name> {Name der Einstellung} </name>
   <value> {Wert der Einstellung} </value>
</property>
<property>
:
```

Es müssen Angaben zur Datenbankverbindung, zur Tabelle, zu den Ein- und Ausgabeformaten und mehr gemacht werden. Die folgende Tabelle enthält die für einen ersten Test nötigen Einstellungen:

oracle.hadoop.loader.olh_home: **/opt/oracle/olh**	Verzeichnis, in das Sie den Oracle Loader ausgepackt haben
mapred.input.dir: **mapreduce-jobs-output**	HDFS-Verzeichnis, in dem die zu ladenden Daten liegen
mapreduce.inputformat.class: **oracle.hadoop.loader.lib.input.DelimitedTextInputFormat**	Format der zu ladenden Daten

oracle.hadoop.loader.input. fieldTerminator: **\u0009**	Feldtrenn-Zeichen (\u0009 für Tabulator)
oracle.hadoop.loader.input. initialFieldEncloser: *-für dieses Beispiel leer lassen-*	optional: Zeichen, mit dem ein Feld anfängt
oracle.hadoop.loader.input. initialFieldEncloser: *-für dieses Beispiel leer lassen-*	optional: Zeichen, mit dem ein Feld endet
oracle.hadoop.loader.input.fieldNames **C,C0,C1,C2,C3,C4,C5,C6,C7**	die zu ladenden Spaltennamen der Tabelle
oracle.hadoop.loader.input. fieldNames.hadoop.loader.targetTable: **SCOTT.MATRIX**	Schema und Name der Tabelle
oracle.hadoop.loader.connection.url **jdbc:oracle:thin:@ databaseHost:1521:databaseSID**	JDBC-Connection-String für Zieldatenbank
oracle.hadoop.loader.connection.user **SCOTT**	Datenbank-User
oracle.hadoop.loader.connection.password **tiger**	Datenbankpasswort
mapred.output.dir: **hdfs-oraloader-output**	HDFS-Verzeichnis, in das der Oracle Loader seine Dateien ablegt
mapreduce.outputformat.class: **oracle.hadoop.loader.lib.output. DataPumpOutputFormat**	Lademethode (hier: External-Table-Data-Pump-Format)

Mit diesen Informationen können Sie die XML-Datei zusammenstellen und unter dem Namen `oraload-conf.xml` speichern:

```
<configuration>
   <property>
      <name>mapred.job.tracker</name>
      <value>localhost:9001</value>
   </property>
   <property>
      <name>oracle.hadoop.loader.olh_home</name>
      <value>/opt/hadoop/olh</value>
   </property>
   <property>
      <name>mapred.input.dir</name>
      <value>mapreduce-jobs-output</value>
   </property>
   :
   <!-- weitere Angaben hier ... -->
</configuration>
```

Das folgende Kommando schließlich startet den Oracle Loader for Hadoop und lädt die Daten aus dem HDFS in die Tabelle in der Oracle Datenbank. Man sieht sehr schön, dass der Oracle Loader nicht per Java-Code, sondern als fertiger MapReduce-Job angesprochen wird – seine Konfiguration holt er aus der XML-Datei:

```
$ bin/hadoop jar /opt/hadoop/olh/jlib/oraloader.jar \
```

```
        oracle.hadoop.loader.OraLoader \
        -conf oraload-conf.xml
```

Wiederum sehen Sie die für Hadoop typischen Bildschirmausgaben:

```
Oracle Loader for Hadoop Release 1.1.0.0.0 - Production
Copyright (c) 2011, Oracle and/or its affiliates. All
rights reserved.

18:02:17 INFO loader.OraLoader: Sampling disabled,
table: MATRIX is not partitioned
:
18:02:20 INFO input.FileInputFormat: Total input paths
to process : 1
18:02:21 INFO mapred.JobClient: Running job:
job_201201121336_0050
18:02:22 INFO mapred.JobClient:  map 0% reduce 0%
18:02:50 INFO mapred.JobClient:  map 100% reduce 0%
:
```

Nach Abschluss des Jobs liegen die Ergebnisdateien im HDFS im Verzeichnis `hdfs-oraloader-output`. Der nächste Schritt ist folgerichtig ein HDFS-Verzeichnislisting:

```
$ bin/hadoop dfs -ls hdfs-oraloader-output/*
Found 5 items
-rw-r--r--   1     0 2012-01-30 18:02 _SUCCESS
```

```
drwxr-xr-x   -     0 2012-01-30 18:02 _logs
-rw-r--r--   1 16384 2012-01-30 18:02  oraloader-
00000-dp-0.dat
-rw-r--r--   1  1049 2012-01-30 18:02  oraloader-dp.sql
-rw-r--r--   1  2524 2012-01-30 18:02  oraloader-
report.txt
```

Interessant sind die Dateien `oraloader-dp.sql` und `oraloader-00000-dp-0.dat`. Das müsste folgerichtig ein SQL-Skript zum Erstellen der externen Tabelle und eine Data-Pump-Exportdatei sein. Zuerst ein Blick in das SQL-Skript:

```
$ bin/hadoop dfs -cat hdfs-oraloader-output/oraloader-
dp.sql
--Oracle Loader for Hadoop Release 1.1.0.0.0 -
Production
--Copyright (c) 2011, Oracle and/or its affiliates.
All rights reserved.
--
--Generated by DataPumpOutputFormat
--
--CREATE OR REPLACE DIRECTORY OLH_EXTTAB_DIR AS '...';
--GRANT READ, WRITE ON DIRECTORY OLH_EXTTAB_DIR TO
SCOTT;
--
--ALTER SESSION ENABLE PARALLEL DML;
```

```
--INSERT INTO "SCOTT"."MATRIX" SELECT * FROM
"SCOTT"."EXT_MATRIX";
--
CREATE TABLE  "SCOTT"."EXT_MATRIX"
(
   "C"                                NUMBER,
   "C0"                               NUMBER,
   "C1"                               NUMBER,
  :
   "C8"                               NUMBER,
   "C9"                               NUMBER
)
ORGANIZATION EXTERNAL
  (TYPE ORACLE_DATAPUMP
   DEFAULT DIRECTORY OLH_EXTTAB_DIR
   LOCATION ('oraloader-00000-dp-0.dat')
  );
```

Voilà! Die Datei `oraloader-00000-dp-0.dat` ist demnach eine binäre und im Texteditor nicht lesbare Data-Pump-Datei. Diese Dateien können nun aus dem HDFS geholt und als externe Tabellen in die Oracle Datenbank geladen werden. Der **Oracle Connector for HDFS**, welcher ebenfalls Teil der Oracle Big Data Connectors ist, erlaubt auch den direkten Zugriff auf das HDFS, sodass das Kopieren ins „normale“ Dateisystem entfallen kann.

4 Fazit und Ausblick

Schließlich hat sich der Kreis geschlossen: Die Idee „Big Data“ sieht zunächst vor, dass Unmengen unstrukturierter oder nur schwach strukturierter Daten als Daten ins HDFS oder als Key-Value-Paare in die **Oracle NoSQL Database** gespeichert werden. Diese verteilten Systeme sind in der Lage, auch größte Datenmengen mit kurzen Antwortzeiten aufzunehmen und ständig zu wachsen.

Da hier keine Abfragesprache existiert, muss für jede Auswertung der gesamte Datenbestand durchgearbeitet werden. Diese Aufgabe übernimmt – ebenfalls massiv parallel – MapReduce. Die gewünschten Auswertungen beziehungsweise Aggregationen werden als MapReduce-Jobs implementiert und im Hadoop-Cluster ausgeführt. Als letzter Schritt einer solchen Jobkette kommt schließlich der **Oracle Loader for Hadoop** ins Spiel, der die gefundenen Aggregate ins **RDBMS Oracle** lädt, wo sie Teil des Data Warehouse und damit zur Basis für Reporting, Business Intelligence und weitere Analyse werden können.

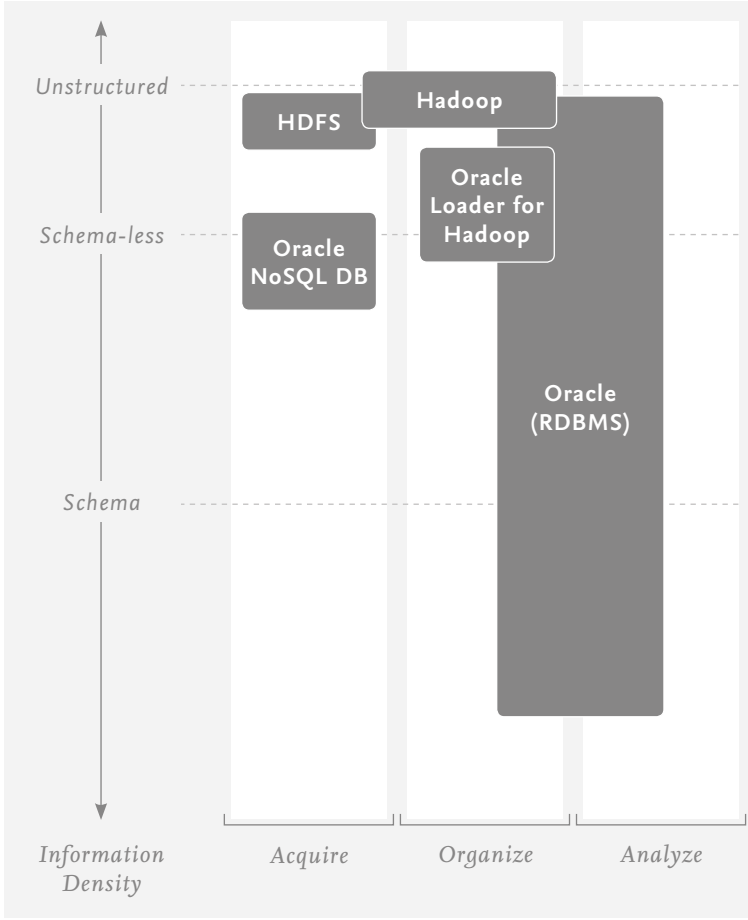


Abb. 7: Der Big-Data-Prozess: erfassen, veredeln, auswerten

5 Weitere Informationen

- Codebeispiel dieses Dojos:
apex.oracle.com/folien
Schlüsselwort: **bigdata-Dojo**

- Oracle White Paper: Big Data Overview
www.oracle.com/ocom/groups/public/@otn/documents/webcontent/1453236.pdf

- Oracle NoSQL DB im OTN:
www.oracle.com/technetwork/database/nosqldb/overview/index.html

- Apache Hadoop
hadoop.apache.org

- Oracle Big Data Connectors
www.oracle.com/technetwork/bdc/big-data-connectors/index.html

Herausgeber: Günther Stürner, Oracle Deutschland B.V.
Design: volkerstegmaier.de // Druck: Stober GmbH, Eggenstein

ORACLE®