ORACLE

ORACLE DOJO NR. 2

CARSTEN CZARSKI

Big Data: Eine Einführung
Oracle NoSQL Database, Hadoop, MapReduce und der Oracle Loader for Hadoop im Zusammenspiel

ORACLE DOJO NR. 2

2. AUFLAGE

ÜBERARBEITET & ERWEITERT

2013

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

1 Einführung 5

1.1 Big Data, NoSQL-Datenbanken und Hadoop: Worum geht es? *6*

2 Big Data: Erfassen und Speichern 9

2.1 Grundlagen *10*

2.1.1 Besonderheiten verteilter Datenhaltung: CAP-Theorem und „ACID vs. BASE“ *10*

2.1.2 Flexible Datenbankmodelle anstelle fester Datenbankschemas *12*

2.2 Speichern im Hadoop Distributed Filesystem *13*

2.2.1 Aufsetzen eines „Hadoop-Pseudo-Clusters“ *15*

2.2.2 Erste Schritte mit dem HDFS *20*

2.3 Speichern in der Oracle NoSQL Database *22*

2.3.1 Architektur der Oracle NoSQL Database *23*

2.3.2 Installation der Oracle NoSQL DB Software *26*

2.3.3 Planung von TCP/IP-Ports und Datendatei-Verzeichnis *27*

2.3.4 Einrichten der NoSQL-Datenbank *28*

2.3.4.1 Starten der NoSQL-Datenbanksoftware auf den Storage Nodes *28*

2.3.4.2 Starten des Kommandozeilenwerkzeugs der NOSQL-DB *30*

2.3.4.3 Einrichten der NoSQL-Datenbanktopologie *31*

2.3.5 Nutzung der Oracle NoSQL Database in einer Java-Anwendung *36*

2.3.5.1 Key-Value-Paare in der NoSQL-DB *36*

2.3.5.2 Speichern eines Key-Value-Paares *37*

2.3.5.3 Abrufen des Key-Value-Paares *40*

2.3.6 Steuerung der Konsistenz und Verfügbarkeit *41*

2.3.6.1 Schreibkonsistenz festlegen *42*

2.3.6.2 Lesekonsistenz festlegen *44*

2.3.7 Was passiert beim Ausfall eines Storage Nodes? *46*

2.3.8 Daten der NoSQL-DB als externe Tabelle im RDBMS Oracle bereitstellen *48*

2.3.8.1 Formatter-Programm in Java schreiben *50*

2.3.8.2 Datenbankuser anlegen, Rechte vergeben und externe Tabelle anlegen *52*

2.3.8.3 Konfigurationsdatei für die externe Tabelle erstellen *54*

2.3.8.4 NoSQL-DB-Bibliotheken und Executables zum RDBMS-Server bringen *57*

2.3.8.5 "nosql_stream" auf dem Datenbankserver einrichten und testen *58*

3 Big Data verarbeiten: Hadoop und MapReduce 60

3.1 Map Reduce: Was ist das? *60*

3.1.1 Mapper *60*

3.1.2 Reducer *61*

3.2 Implementierung eines einfachen MapReduce-Jobs *63*

3.3 Kompilieren und Starten des MapReduce-Jobs *70*

3.4 MapReduce ohne Java-Programmierung: Hive *73*

3.5 Oracle Loader for Hadoop *79*

3.5.1 Download und Installation *82*

3.5.2 Konfiguration und Start des Oracle Loader for Hadoop *83*

4 Fazit und Ausblick 90

5 Weitere Informationen 92

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 den **Oracle Big Data Connectors** seien zwei Beispiele genannt. Und mit der **Big Data Appliance** wird ein *Engineered System* speziell für An forderungen 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. Die Themenkomplexe Auswertung, Reporting und Analyse können aus Platzgründen hier nicht abgedeckt werden.

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 oder noch schneller. 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 großer Datenmengen aus unterschiedlichen Quellen. Das System, in das diese Daten gelegt werden, muss kostengünstigen Plattenplatz anbieten, aber 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.

Diese Systeme versetzen überhaupt erst in die Lage, die Rohdaten über einen längeren Zeitraum aufzunehmen und zu speichern – mit einem RDBMS wäre das sehr aufwändig und vielfach gar nicht möglich. Typischerweise wird hier nichts mehr gelöscht – denn das Ziel der Big-Data-Plattform ist es, all diese Daten langfristig so vorzuhalten, dass künftig alle erdenklichen Analysen – auch die, an die heute noch niemand denkt – möglich sind.

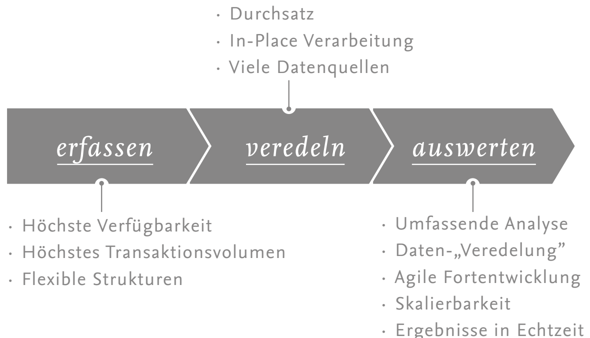
Endanwender greifen typischerweise nicht direkt auf das Hadoop Distributed Filesystem (HDFS) oder eine NoSQL-Datenbank zu. Die enthaltenen Daten haben keine Struktur; ein Fachanwender findet sich in diesen Rohdaten wahrscheinlich nicht zurecht. Die Daten müssen also zunächst aufbereitet und veredelt werden, damit sie anschließend, über die „normale“ Datawarehouse- und BI-Plattform, den Anwendern zur Verfügung gestellt werden können.

Für diesen Prozess kommt MapReduce zum Einsatz: Die gewünschten Veredelungen beziehungsweise Aggregationen werden als MapReduce-Jobs implementiert und im Hadoop Cluster parallel ausgeführt.

MapReduce-Jobs können zum einen from Scratch in Java geschrieben werden; damit ist die Umsetzung auch individueller Anforderungen möglich. Zum anderen gibt es fertige MapReduce-Jobs für standardisierte Aufgaben. Hive ist ein solches Beispiel: Hive übersetzt SQL-ähnliche Abfragen in einen MapReduce-Job und führt diesen anschließend im Hadoop Cluster aus. Abschnitt 3 dieses Dojos beschäftigt sich näher damit. Die Rohdaten im HDFS beziehungsweise der NoSQL-Datenbank bleiben dabei stets erhalten – können also immer wieder neu bearbeitet werden.

Das Ergebnis der MapReduce-Jobs wird schließlich in strukturierter Form in ein RDBMS geladen, wo es als Teil des Data Warehouse mit „klassischen“ Methoden weiterverarbeitet wird.

Abb.1: Der Big-Data-Prozess: erfassen, veredeln, auswerten



2 Big Data: Erfassen und Speichern

Dieser Abschnitt beschäftigt sich mit den grundlegenden Systemen zur Speicherung großer Datenmengen mit geringer Informationsdichte: Das Hadoop Distributed Filesystem (HDFS) bietet eine dateiorientierte Ablagemöglichkeit an; NoSQL-Datenbanken arbeiten auf Ebene des einzelnen Datensatzes. Beide sind verteilte Systeme und daher ist das CAP-Theorem für beide von Bedeutung.

2.1 GRUNDLAGEN

2.1.1 BESONDERHEITEN VERTEILTER DATENHALTUNG: CAP-THEOREM UND „ACID VS. BASE“

Das Cap-Theorem besagt, dass es einem verteilten System nicht möglich ist, gleichzeitig die drei Anforderungen Konsistenz, Verfügbarkeit und Partitionstoleranz zu gewährleisten, es kann maximal zwei dieser Anforderungen zur gleichen Zeit erfüllen.

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.

Hinsichtlich der Transaktionen folgt ein RDBMS dem ACID Konzept:

- **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.

Dagegen legen sowohl das Hadoop Distributed Filesystem (HDFS) als auch NoSQL-Datenbanken 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 bei NoSQL Datenbanken von Produkt zu Produkt verschieden: Manche sind AP-Systeme, andere sind CP-Systeme und wieder andere überlassen dem Entwickler die Entscheidung. Die Oracle NoSQL Datenbank gehört zu letzteren Systemen. Das 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.

2.1.2 FLEXIBLE DATENBANKMODELLE ANSTELLE FESTER DATENBANKSCHEMAS

Sowohl das HDFS als auch NoSQL-Datenbanken arbeiten im Gegensatz zu RDBMS ohne feste Datenstrukturen – demnach gibt es keine Tabellen und kein Datenbankschema. Während das HDFS – ebenso wie eine PC-Festplatte – mit Dateien arbeitet, die als Ganzes gelesen oder geschrieben

werden, arbeiten NoSQL-Datenbanken auf der Ebene einzelner Datensätze oder Datenobjekte. Aber auch hier wird – im Gegensatz zu einer relationalen Datenbank – ohne oder nur mit minimalen Strukturen gearbeitet.

Die Speicherung in **Key-Value-Paaren** ist das generischste Konzept. Ein Wert wird gemeinsam mit einem Schlüssel gespeichert, und nur mit diesem kann man ihn wieder abrufen. Der Inhalt des „Wertes" ist für die NoSQL-Datenbank eine „Black Box"; von skalaren Informationen bis hin zu komplexen Strukturen ist alles denkbar. Die NoSQL-Datenbank kann daher auch nicht nach den Inhalten des Wertes filtern – sie kennt die Strukturen ja nicht. Insofern sieht das Arbeiten mit einer NoSQL-Datenbank ganz anders aus als mit einem RDBMS: **Ein Wert kann nur anhand des Schlüssels abgerufen werden – eine Abfragesprache mit Filtermöglichkeit, wie SQL, gibt es nicht.**

2.2 SPEICHERN IM HADOOP DISTRIBUTED FILESYSTEM

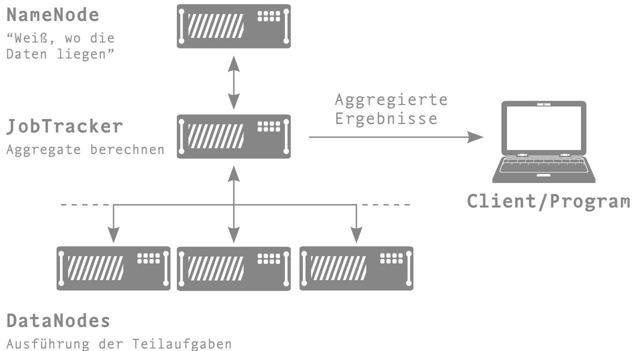
Das Hadoop Distributed Filesystem (HDFS) legt seine Dateien, verteilt über die Rechnerknoten des Hadoop Clusters ab. Prinzipiell kann man sich das HDFS recht einfach vorstellen:

- Dateien werden, wie beim Dateisystem eines PC, in Blöcke unterteilt. Da das HDFS für große Datenmengen ausgelegt ist, sind die Blöcke hier größer – der Default ist 64 MB.

- Das Dateisystem verwaltet die Zuordnung von Blöcken zu Dateien. Diese Aufgabe übernimmt beim HDFS einer der Rechnerknoten als **Name Node**. Der Name Node kennt also alle im HDFS enthaltenen Dateien und er „weiß", welche Blöcke auf welchen Knoten zu finden sind.

- Damit bei Ausfall eines der Rechnerknoten kein Datenverlust entsteht, werden alle Blöcke repliziert. Standardmäßig verwendet das HDFS einen Replikationsfaktor von „3"; das heißt jeder Dateiblock ist auf wenigstens drei verschiedenen Clusterknoten zu finden.

Abb. 2: Schematische Darstellung eines Hadoop Clusters



Hadoop kann grundsätzlich in zwei Varianten heruntergeladen werden: Die Originalpakete stehen auf der Website 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 Websites herunter, so muss man sehr 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 Oracles Big Data Appliance enthalten ist. Eine freie Version kann von der Website von Cloudera (**www.cloudera.com**) heruntergeladen werden. Die Beispiele in diesem Dojo basieren auf CdH Version 3.

2.2.1 AUFSETZEN EINES „HADOOP-PSEUDO-CLUSTERS“

Hadoop, und auch die danach beschriebene Oracle NoSQL DB, setzen eine vorhandene Java-Umgebung voraus. Stellen Sie also sicher, dass eine solche auf Ihrem System installiert ist. Verwenden Sie mindestens JavaSE 6, besser wäre JavaSE 7. Die aktuell installierte Version kann mit dem Kommando `java -version` herausgefunden werden:

```
$ java -version
java version "1.7.0_13"
Java(TM) SE Runtime Environment (build 1.7.0_13-b20)
:
```

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 circa 70 MB großen ZIP-Archivs packen Sie es in einem 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
|-- :
```

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
# 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>
   <property>
      <!-- In diesem Verzeichnis werden die Blöcke der
         HDFS-Dateien abgelegt -->
      <name>hadoop.tmp.dir</name>
      <value>/path/to/folder/for/hdfs/files</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. Im Hadoop Cluster müssen die Knoten untereinander per Secure Shell (SSH) und ohne Eingabe von Passwörtern kommunizieren können. Dazu müssen Schlüssel ausgetauscht werden – für den pseudoverteilten 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
```

Stellen Sie dann das verteilte Dateisystem, das HDFS, bereit:

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

Starten Sie schließlich den Hadoop-Pseudo-Cluster:

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

2.2.2 ERSTE SCHRITTE MIT DEM HDFS

Der Zugang zum HDFS erfolgt mit den Kommandozeilenwerkzeugen des Hadoop Frameworks. Darüber hinaus gibt es natürlich Programmierbibliotheken (APIs). 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   - oracle users       0 2012-01-12 13:36 /tmp
drwxr-xr-x   - oracle users       0 2012-01-12 13:37 /user
```

Mit dem `put` Kommando 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 oracle users      24 2012-01-30 16:38 /
user/oracle/myhdfsfile.txt
```

Analog dazu können Dateien aus dem HDFS herauskopiert (`get`), Verzeichnisse erstellt (`mkdir`), gelöscht (`rm`, `rmr`) oder umbenannt (mv) werden.

Beginnt eine HDFS-Pfadangabe mit einem Schrägstrich, so ist die Angabe (wie bei Unix und Linux) absolut. Ansonsten wird die Pfadangabe relativ zu `/user/{Linux-Userid}` interpretiert. Ist man also auf Linux als `oracle` angemeldet, so wird die relative Angabe `datei.txt` als `/user/oracle/datei.txt` aufgefasst.

Man kann sich nun leicht vorstellen, dass das HDFS nun, auf sehr einfache Art und Weise, sehr große Datenmengen aufnehmen kann. Die Dateien werden einfach in den Cluster gespeichert – und wenn das Dateisystem voll wird, fügt man entsprechend neue Clusterknoten hinzu. Allerdings sind ausschließlich Dateien gespeichert – und nur auf Ebene der Datei kann direkt zugegriffen werden. Möchte man die Daten nun auswerten, veredeln oder aggregieren,

so müssen die Dateien gelesen und verarbeitet werden – wie das mithilfe von MapReduce geht, ist ab Kapitel 3 in diesem Dojo beschrieben.

Benötigt man anstelle einer dateiorientierten eher eine objekt- oder satzorientierte Speicherung, so wäre eine NoSQL-Datenbank die bessere Alternative. Im nächsten Abschnitt ist daher der Umgang mit der Oracle NoSQL DB beschrieben.

2.3 SPEICHERN IN DER 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.

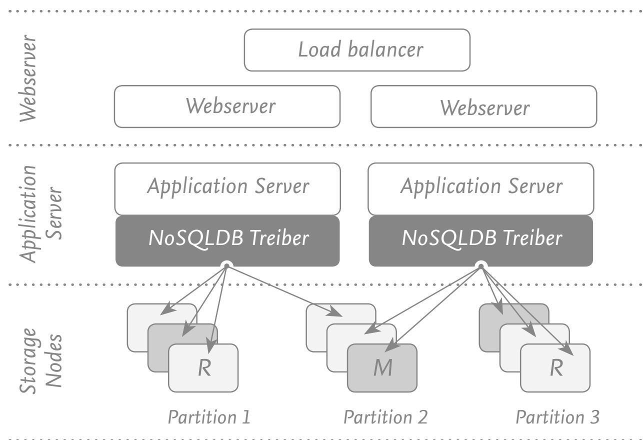
Die Oracle NoSQL DB ist als **Community Edition (CE)** oder als **Enterprise Edition (EE)** erhältlich. Die Community Edition ist kostenlos und eine OpenSource-Lizenz. Die Enterprise Edition ist eine kommerzielle Lizenz mit der Möglichkeit eines Wartungsvertrags. Features der Enterprise Edition beziehen sich zur Zeit der Drucklegung dieses

Dojos vor allem auf die Integration in ein Rechenzentrum: So gehört das JMX-Monitoring oder die Möglichkeit, Daten der NoSQL-DB als externe Tabelle im RDBMS verfügbar zu machen (siehe hierzu Abschnitt 2.3.8) zur Enterprise Edition.

2.3.1 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. 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. 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.

Ab Release 2 der Oracle NoSQL DB kann die eingerichtete Topologie auch im laufenden Betrieb verändert werden. Neben dem einfachen Hinzufügen oder Entfernen von

Storage Nodes ist auch das Hinterlegen einer komplett neuen Topologie möglich. Die NoSQL-DB verteilt die Daten dann entsprechend neu.

2.3.2 INSTALLATION DER ORACLE NOSQL DB SOFTWARE

Nach dem Download der ca. 24 MB aus dem Oracle Technology Network (OTN) und dem Auspacken des ZIP-Archivs stellt sich die Verzeichnisstruktur der Oracle NoSQL DB Enterprise Edition in etwa wie folgt dar:

```
|-- bin
|-- doc
|   |-- AdminGuide
|   |-- GettingStartedGuide
|   |-- examples
|   |-- javadoc
|   `-- misc
|-- exttab
     |-- bin
|-- 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-2.0.23.jar kvclient.jar
$ ln -s kvstore-2.0.23.jar kvstore.jar
```

2.3.3 PLANUNG VON TCP/IP-PORTS UND DATENDATEI-VERZEICHNIS

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.

2.3.4 EINRICHTEN DER NOSQL-DATENBANK

2.3.4.1 STARTEN DER NOSQL-DATENBANKSOFTWARE 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 \
   -memory mb 512
```

Der mit `port` angegebene TCP/IP-Port ist der „Listener-Port“ dieses Storage Node; Anwendungen nutzen diesen Port zum Verbinden mit der NoSQL-Datenbank. Auf dem mit `admin` festgelegten Port kann mit dem Browser die Administrationsoberfläche aufgerufen werden. Die mit `harange` bezeichneten Ports nutzt die NoSQL-DB intern. Der Parameter `memory_mb` legt fest, wie viel Hauptspeicher auf diesem Rechnerknoten der NoSQL-DB-Software bereitgestellt wird.

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.5000 -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.

2.3.4.2 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.

2.3.4.3 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 -name Dojostore
Store configured: Dojostore
kv->
```

Nun wird ein *Data Center* eingerichtet. Ein NoSQL-DB-Store mitsamt seinen Storage Nodes und Replikationsgruppen wird einem Data Center zugeordnet. An dieser Stelle legen Sie auch schon den Replikationsfaktor fest:

```
kv-> plan deploy-datacenter -name "DC Dojo" -rf 3 -wait
Executed plan 1, waiting for completion...
Plan 1 ended successfully
kv-> show topology
store=Dojostore numPartitions=0 sequence=1
 dc=[dc1] name=DC Dojo repFactor=3
```

Das Kommando `plan` nimmt einen Konfigurationsbefehl entgegen und legt ihn auf den Stapel. Die NoSQL-DB fängt sofort an, an diesem Stapel zu arbeiten. Wird zusätzlich das Flag `-wait` angegeben, so blockiert das Werkzeug bis der Befehl fertig ist.

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 deploy-sn -dc dc1 -host <hostname-sn-1> -port 5000 -wait
Executed plan 2, waiting for completion...
Plan 2 ended successfully

kv-> show topology
store=Dojostore numPartitions=0 sequence=1
   dc=[dc1] name=DC Dojo repFactor=3
   sn=[sn1]  dc=dc1 <hostname-sn-1>:5000 capacity=1 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 deploy-admin -sn sn1 -port 5001 -wait
```

Als nächstes müssen Sie einen Storage-Node-Pool einrichten. Für die Zuordnung der Storage Nodes zum Pool benötigen Sie wiederum die IDs der Storage Nodes:

```
kv-> pool create -name DojoPool
kv-> pool join -name DojoPool -sn sn1
Added Storage Node(s) [sn1] to pool DojoPool
kv->
```

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

```
kv-> plan deploy-sn -dc dc1 -host <hostname-sn-2> -port 5000 -wait
kv-> :
kv-> plan deploy-sn -dc dc1 -host <hostname-sn-n> -port 5000 -wait

kv-> show topology
store= Dojostore numPartitions=0 sequence=4
   dc=[dc1] name=DC Dojo repFactor=3
   :
   sn=[sn1]  dc=dc1 <hostname-sn-1>:5000 capacity=1 RUNNING
   sn=[sn2]  dc=dc1 <hostname-sn-2>:5000 capacity=1 RUNNING
   sn=[sn3]  dc=dc1 <hostname-sn-3>:5000 capacity=1 RUNNING

kv-> pool join -name DojoPool -sn sn2
```

```
Added Storage Node(s) [sn2] to pool DojoPool
kv-> pool join -name DojoPool -sn sn3
Added Storage Node(s) [sn3] to pool DojoPool
:
:
```

Als nächstes wird eine NoSQL-DB-Topologie erstellt. Dabei legen Sie die Anzahl der Partitionen und den Storage Node Pool, auf dem die Topologie erzeugt werden soll, fest. Diese Einstellung speichert nur Metadaten, tut aber noch nichts.

```
kv-> topology create -name DoJoTopo -pool DojoPool
-partitions 100
Created: DojoTopo
```

Das Kommando `topology preview` zeigt an, was die NoSQL-DB zu tun hat, um besagte Topologie umzusetzen. Das ist insbesondere wichtig, wenn die Topologie geändert werden soll. Aber auch jetzt kann das Kommando aufgerufen werden.

```
kv-> topology preview create -name DoJoTopo
Topology transformation from current deployed topology
to DojoTopo:
Create 2 shards
Create 6 RNs
Create 100 partitions
```

```
shard rg1
   3 new RNs : rg1-rn1 rg1-rn2 rg1-rn3
   50 new partitions
shard rg2
   3 new RNs : rg2-rn1 rg2-rn2 rg2-rn3
   50 new partitions
```

Das Kommando `topology deploy` legt die Topologie und damit den NoSQL-DB Store schließlich an.

```
kv-> plan deploy-topology -name DojoTopo -wait
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

An der mit `show topology` zurückgegebenen Topologie erkennt man sehr gut, dass nun Replikationsgruppen gebildet wurden (`rg1,rg2`). Ein Storage Node fungiert für jede Replikationsgruppe als Master für Schreibzugriffe, die anderen beiden als Replika:

```
kv-> show topology
store=Dojostore  numPartitions=100 sequence=108
   dc=[dc1] name=DC Dojo repFactor=3

   sn=[sn1]  dc=dc1 <hostname-sn-1>:5000 capacity=1 RUNNING
      [rg1-rn1] RUNNING
         No performance info available
   sn=[sn2]  dc=dc1 <hostname-sn-2>:5000 capacity=1 RUNNING
    [rg1-rn2] RUNNING
```

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

2.3.5 NUTZUNG DER ORACLE NOSQL DATABASE IN EINER JAVA-ANWENDUNG

2.3.5.1 KEY-VALUE-PAARE IN DER NOSQL-DB

Der Zugriff auf die Oracle NoSQL DB erfolgt allein durch API-Aufrufe – eine Abfragesprache wie SQL gibt es nicht. Programmier-APIs sind für Java und – ab Version 2 – für C verfügbar. 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.

2.3.5.2 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.

2.3.5.3 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.

2.3.6 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.

2.3.6.1 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())
   );
:
```

2.3.6.2 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:

```
:
storeConf.setConsistency(
      new Consistency.Time(
         5,
```

```
            java.util.concurrent.TimeUnit.SECOND,
            5,
            java.util.concurrent.TimeUnit.SECOND
    )
);
:
```

2.3.7 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.

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 wiederherstellen. Nach dem erneuten Einbinden wird er als Replika fungieren – denn die Aufgabe des Masters hat ja nun ein anderer Knoten übernommen.

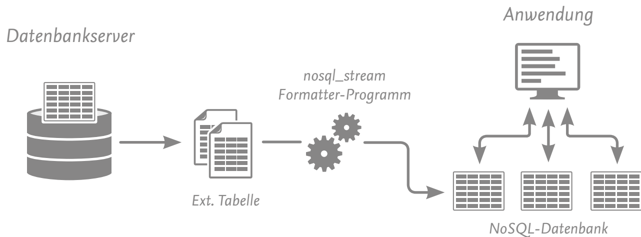
Bei Ausfall eines Knotens, der als Replika fungiert, findet natürlich keine Neuwahl des Masters statt – denn der läuft

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.

2.3.8 DATEN DER NOSQL-DB ALS EXTERNE TABELLE IM RDBMS ORACLE BEREITSTELLEN

Die Oracle NoSQL DB bietet in der Enterprise Edition ab Version 2 ein besonderes Feature an: Die Key-Value-Paare können so aufbereitet werden, dass sie im RDBMS Oracle-Datenbank als externe Tabelle bereitgestellt werden können. Die Community-Edition enthält diese Funktion nicht, sodass dieser Abschnitt nur mit der Enterprise Edition nachvollzogen werden kann. Das RDBMS Oracle muss mindestens in der Version 11.2 vorliegen.

Abb.4: Daten der NoSQL DB werden im RDBMS Oracle als externe Tabelle bereitgestellt



Zunächst füllen wir die Oracle NoSQL Database mit Daten. Ein Java-Programm 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();
    }
}
```

2.3.8.1 FORMATTER-PROGRAMM IN JAVA SCHREIBEN

Nun wird ein Formatter-Programm in Java erstellt, welches die Key-Value-Paare in der NoSQL-DB in ASCII-Text umwandelt. Für jedes Key-Value-Paar sollte eine Zeile entstehen. Die Java-Schnittstelle für das Formatter-Programm wird von der NoSQL-DB vorgegeben und ist im Java-Paket

`oracle.kv.exttab` enthalten. Schreiben und kompilieren Sie das Programm am besten auf einem der Server der Oracle NoSQL DB.

```
import java.util.List;
import oracle.kv.*;
import oracle.kv.exttab.*;

public class NoSQLDB_Formatter implements Formatter {
   public NoSQLDB_Formatter() {}

   @Override
   public String toOracleLoaderFormat(
      final KeyValueVersion kvv, final KVStore kvStore
   ) {
      final Key key = kvv.getKey();
      final Value value = kvv.getValue();
      final List<String> fullPath = key.getFullPath();
      final String dKey = fullPath.get(0);
      return dKey+"|"+new String(value.getValue());
   }
}
```

Kompilieren Sie das Java-Programm danach mit javac. Achten Sie dabei aber darauf, dass neben `kvclient-2.0.23.jar` und `kvstore-2.0.23.jar` auch das JAR-Archiv `${NOSQL_HOME}/lib/kvstore-ee-2.0.23.jar` im Java-CLASSPATH enthalten ist.

2.3.8.2 DATENBANKUSER ANLEGEN, RECHTE VERGEBEN UND EXTERNE TABELLE ANLEGEN

Als nächstes verbinden Sie sich auf die Oracle-Datenbank und legen sich einen neuen Datenbank-User namens NOSQLUSER an. Dieser braucht die folgenden Privilegien:

- CREATE SESSION
- CREATE TABLE
- EXECUTE on UTL_FILE

Das Privileg EXECUTE on UTL_FILE wird nur zur Einrichtung der externen Tabelle gebraucht; danach kann es entzogen werden.

Nun werden zwei Directory-Objekte benötigt. Legen Sie also auf dem Datenbankserver zwei neue Unix-Verzeichnisse an: `/exttab-data` und `/exttab-scripts`. Der Betriebssystem-User `oracle` muss beide lesen und schreiben können. Natürlich können Sie die Verzeichnisse auch anders nennen. Erzeugen Sie danach für jedes Verzeichnis ein Directory-Objekt in der Datenbank und geben Sie dem neuen Datenbankuser Privilegien darauf. Das SQL könnte so aussehen:

```
SQL> create directory DATADIR as '/exttab-data';

SQL> grant read, write on directory DATADIR to NOSQLUSER;
```

```
SQL> create directory SCRIPTDIR as '/exttab-scripts';

SQL> grant read, execute on directory SCRIPTDIR to
NOSQLUSER;
```

Wichtig ist, dass der Datenbank-User niemals sowohl Schreib- als auch Execute-Privilegien auf ein und dasselbe Directory-Objekt bekommt. Als nächstes legen Sie (als User NOSQLUSER) die externe Tabelle an.

```
CREATE TABLE NOSQL_DATA (
   KEY   NUMBER,
   VALUE NUMBER
) ORGANIZATION EXTERNAL (
   TYPE ORACLE_LOADER DEFAULT DIRECTORY "DATADIR"
   ACCESS PARAMETERS (
      records delimited by newline
      preprocessor scriptdir:'nosql_stream'
      fields terminated by '|'
   )
   LOCATION ('nosql.dat')
) PARALLEL;
```

Hierbei wird die in Oracle11*g* Release 2 neu eingeführte *Präprozessor-Klausel* einer externen Tabelle verwendet – es werden also nicht einfach nur Daten aus einer Datei gelesen; vielmehr werden diese mithilfe eines Executables (hier: `nosql_stream`) aufbereitet.

2.3.8.3 KONFIGURATIONSDATEI FÜR DIE EXTERNE TABELLE ERSTELLEN

Nun wird die Konfigurationsdatei für die externe Tabelle erstellt. Darin enthalten sind:

- Verbindungsinformationen zur Oracle NoSQL DB
- Verbindungsinformationen zur Oracle RDBMS
- Name des Formatter Programms
- Name der externen Tabelle

Diesen Schritt führen Sie wiederum auf einem der NoSQL-DB-Rechnerknoten durch. Eine Vorlage für die Datei `config.xml` befindet sich im Verzeichnis `examples/externaltables` unterhalb von `${NOSQL_HOME}`. Kopieren Sie sich diese am besten. Die Kopie können Sie danach anpassen. Achten Sie darauf, dass für das RDBMS Oracle folgende Konfiguration vorhanden ist.

```
<component name="publish" type="params">
   <property
      name="oracle.kv.exttab.connection.url"
      value="jdbc:oracle:thin:/@//{db-server}:1521/
      {oracle-service}"
      type="STRING"/>
   <property
```

```
      name="oracle.kv.exttab.connection.user"
      value="NOSQLUSER"
      type="STRING"/>
   <property
      name="oracle.kv.exttab.tableName"
      value="NOSQL_DATA"
      type="STRING"/>
</component>
```

Für die Oracle NoSQL DB sind folgende Einträge wichtig.

```
<component name="nosql_stream" type="params">
   <property name="oracle.kv.kvstore"
             value="dojostore"
             type="STRING"/>
   <property name="oracle.kv.hosts"
             value="<hostname-sn-1>:5000"
             type="STRING"/>
   <property name="oracle.kv.formatterClass"
             value="NoSQLDB_Formatter"
             type="STRING"/>
   <property name="oracle.kv.batchSize"
             value="100"
             type="INT"/>
   <property name="oracle.kv.depth"
             value="PARENT_AND_DESCENDANTS"
             type="STRING"/>
```

Stellen Sie sicher, dass folgender Eintrag *entfernt* wird.

```
<property name="oracle.kv.parentKey"
          value="/user"
          type="STRING"/>
```

Alle anderen Einträge dürften auskommentiert sein – und das können Sie auch so lassen. Speichern Sie die Datei danach ab. Als nächstes werden diese Informationen dem RDBMS bekannt gemacht. Dazu setzen Sie (immer noch auf dem Rechner der NoSQL-DB) folgendes Kommando ab:

```
$ java -classpath \
      ${NOSQL_HOME}/lib/kvstore-ee-2.0.23.jar: \
      ${NOSQL_HOME}/lib/kvstore-2.0.23.jar: \
      ${NOSQL_HOME}/lib/kvclient-2.0.23.jar: \
      ${ORACLE_HOME}/jdbc/lib/ojdbc6.jar \
   oracle.kv.exttab.Publish \
   -config config.xml \
   -publish
```

Aufmerksame Leser bemerken sofort, dass hier auch der Oracle JDBC-Treibers in den Java CLASSPATH eingebunden wurde. Und richtig: Diese Datei müssen Sie gegebenenfalls vom RDBMS-Server auf den NoSQL-DB-Server, mit dem Sie gerade arbeiten, kopieren. Wenn alles klappt, werden Sie nach einem **Database Password** gefragt – geben Sie das Passwort des Datenbankusers `NOSQLUSER` an.

2.3.8.4 NOSQL-DB-BIBLIOTHEKEN UND EXECUTABLES ZUM RDBMS-SERVER BRINGEN

Wenn die externe Tabelle im RDBMS angesprochen wird, müssen die Daten von der NoSQL-DB geholt werden; das RDBMS braucht also die Java-Bibliotheken der Oracle NoSQL DB, das in Abschnitt 2.3.8.1 geschriebene Formatter-Programm und außerdem das im CREATE-TABLE-Kommando verwendete Executable `nosql_stream`. Kopieren Sie diese von der NoSQL-DB-Softwareinstallation zum Datenbankserver und legen Sie die Dateien dort ins Verzeichnis `/exttab-scripts`.

```
$ scp ${NOSQL_HOME}/exttab/bin/nosql_stream \
      NoSQLDB_Formatter.class \
      ${NOSQL_HOME}/lib/kvclient-2.0.23.jar \
      ${NOSQL_HOME}/lib/kvstore-2.0.23.jar \
      ${NOSQL_HOME}/lib/kvstore-ee-2.0.23.jar \
      oracle@{db-server}:/exttab-scripts

oracle@{db-server}'s password: ________

nosql_stream              100%   483       0.5KB/s   00:00
NoSQLDB_Formatter.class   100%   921       0.9KB/s   00:00
kvclient-2.0.23.jar       100%   590KB   590.1KB/s   00:00
kvstore-2.0.23.jar        100%   10MB     10.4MB/s   00:00
kvstore-ee-2.0.23.jar     100%   78KB     78.4KB/s   00:00
```

2.3.8.5 "NOSQL_STREAM" AUF DEM DATENBANKSERVER EINRICHTEN UND TESTEN

Wechseln Sie nun zum Datenbankserver. Dort sollten sich in den Verzeichnissen `/exttab-scripts` und `/exttab-data` folgende Dateien befinden:

```
$ ls /exttab-scripts
kvclient-2.0.23.jar  kvstore-ee-2.0.23.jar  nosql_stream
kvstore-2.0.23.jar  NoSQLDB_Formatter.class

$ ls /exttab-data
nosql.dat
```

Die Datei `nosql.dat` wurde durch den „Publish"-Prozess angelegt. Öffnen Sie nun die Datei `/exttab-scripts/nosql_stream` mit einem Texteditor und stellen Sie sicher, dass sie wie folgt aussieht:

```
#!/bin/bash
# See the file LICENSE for redistribution information.
:
# in the CLASSPATH.

export PATH=/usr/local/jdkl.7.0_03/bin/
export CLASSPATH=/exttab-scripts/kvstore-ee-2.0.23.jar
export CLASSPATH=$CLASSPATH:/exttab-scripts/kvstore-2.0.23.jar
export CLASSPATH=$CLASSPATH:/exttab-scripts/
```

```
kvclient-2.0.23.jar
export CLASSPATH=$CLASSPATH:/exttab-scripts

java oracle.kv.exttab.Preproc $*
```

Wie schon beim Hadoop Cluster und der NoSQL-DB, benötigen Sie nun auch auf dem Datenbankserver mindestens JavaSE 6, besser ist JavaSE 7. Speichern Sie die Datei ab und führen Sie einen Test durch, indem Sie `nosql_stream` einfach von der Kommandozeile aus aufrufen.

```
$ /exttab-scripts/nosql_stream /exttab-data/nosql.dat
51908|52
51919|92
52126|96
52170|18
:
```

Wenn Sie diese Ausgabe sehen, funktioniert alles. Sie können sich danach am RDBMS als NOSQLUSER anmelden und die externe Tabelle verwenden.

```
SQL> select * from nosql_data where rownum <= 20;
       KEY      VALUE
---------- ----------
         0         21
         1         22
         2         66
         :          :
```

3 Big Data verarbeiten: Hadoop und MapReduce

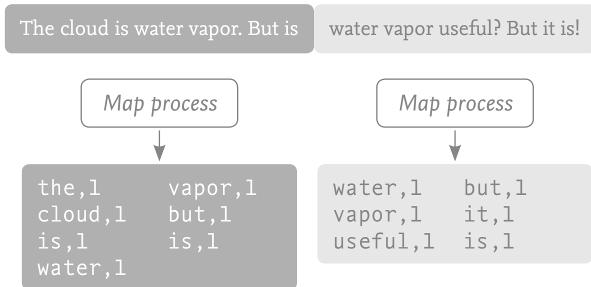
3.1 MAP REDUCE: WAS IST DAS?

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.

3.1.1 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 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.

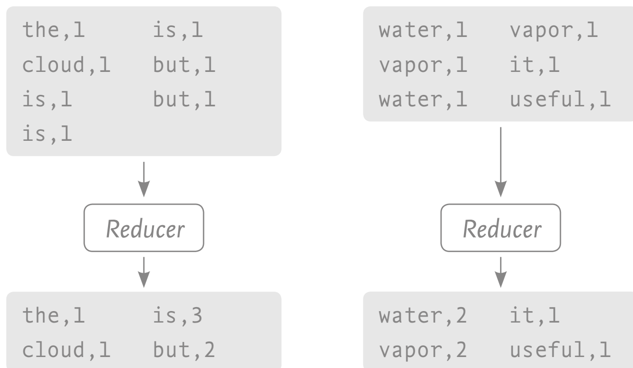
Abb.5: Der Mapper bildet die Eingabedaten auf Key-Value-Paare ab



3.1.2 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.

3.2 IMPLEMENTIERUNG EINES EINFACHEN MAPREDUCE-JOBS

Im Folgenden wird ein einfaches Beispiel für einen MapReduce-Job vorgestellt. Die 100.000 im Abschnitt 2.3.8 vorgestellten Key-Value-Paare sollen nun mit einem MapReduce-Job verdichtet werden:

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 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 hier 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);
   }
}
```

3.3 KOMPILIEREN UND STARTEN DES MAPREDUCE-JOBS

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/ .
```

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 oracle users    0 2012-01-30 17:02
wcout1/_SUCCESS
drwxr-xr-x   - oracle users    0 2012-01-30 17:01
wcout1/_logs
-rw-r--r--   1 oracle users 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 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 weiterverarbeitet. 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.

3.4 MAPREDUCE OHNE JAVA-PROGRAMMIERUNG: HIVE

MapReduce-Jobs müssen nicht zwingend in Java programmiert werden. Der folgende Abschnitt beschreibt den Umgang mit **Hive**, welches es erlaubt, SQL-ähnliche Abfragen als MapReduce-Job auszuführen. Laden Sie also zunächst Hive von der Cloudera-Website (**www.cloudera.com**) herunter. Nach dem Auspacken des ZIP-Archivs entsteht folgende Struktur.

```
.
|-- LICENSE
|-- NOTICE
|-- README.txt
|-- RELEASE_NOTES.txt
|-- bin
|-- conf
:
|-- metastore_db
|-- scripts
`-- src
```

Das Verzeichnis, in das Sie das Archiv ausgepackt haben, wird im Folgenden mit `HIVE_HOME` bezeichnet – setzen Sie sich gegebenenfalls eine Umgebungsvariable. Auf jeden Fall benötigt wird die Umgebungsvariable `HADOOP_HOME` – Sie muss auf das Verzeichnis zeigen, in das Sie das Hadoop-Archiv ausgepackt haben.

```
$ export HADOOP_HOME=/opt/cloudera/hadoop-0.20.2-cdh3u3
```

Erzeugen Sie danach im HDFS ein neues Verzeichnis namens `hivetest` und kopieren Sie die Ergebnisdatei des letzten MapReduce-Jobs (`part-r-00000`) als neue Datei in dieses Verzeichnis.

```
$ ${HADOOP_HOME}/bin/hadoop dfs -mkdir hivetest
$ ${HADOOP_HOME}/bin/hadoop dfs -cp hdfs-output/
```

```
part-r-00000 hivetest/daten.txt
```

Starten Sie anschließend hive mit `${HIVE_HOME}/bin/hive`.

```
$ cd ${HIVE_HOME}; bin/hive
Hive history file=/tmp/oracle/hive_job_log_
oracle_201302011409_2094161987.txt
hive>
```

Nun können Sie Hive-Kommandos absetzen. Wenn (wie im Beispiel) die im HDFS vorhandenen Dateien eine gewisse Struktur haben, kann man eine Hive-Tabellendefinition dafür erstellen. Das folgende Kommando erzeugt eine „externe Tabelle" für die Ausgabedatei des letzten MapReduce-Jobs. Der Parameter `location` gibt dabei an, in welchem HDFS-Verzeichnis die Dateien zu finden sind; er muss *absolut* angegeben werden.

```
   hive> create external table dojo_hivetab (
       >   zufallszahl int,
       >   vorkommen int
       > ) row format delimited fields terminated by '\t'
       > stored as textfile
       > location '/user/oracle/hivetest';
OK
Time taken: 0.534 seconds
hive>
```

Nun können Sie „SQL-Abfragen“ auf diese „Tabelle“ durchführen. Es soll herausgefunden werden, welche Zufallszahlen mehr als 1000 Mal vorkommen. Das ist in SQL eine ganz einfache Aufgabe

```
hive> select * from dojo_hivetab where vorkommen > 1000;
```

Wenn Sie diese Abfrage absetzen, sehen Sie bereits an der Ausgabe, dass auch Hive einen MapReduce-Job startet – das muss ja auch so sein, denn im HDFS liegt nur eine Textdatei. Der Unterschied zum vorigen Kapitel ist, dass der MapReduce-Job nicht mehr selbst geschrieben werden muss: Man kann mit einer „höheren“ Abfragesprache arbeiten und Hive kümmert sich im die MapReduce-Implementierung

```
hive> select * from dojo_hivetab where vorkommen > 1000;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no
reduce operator
Starting Job = job_201211051447_0021, Tracking URL =
http://sccloud038.de.oracle.com:50030/jobdetails.
jsp?jobid=job_20121
Kill Command = /opt/cloudera/hadoop-0.20.2-cdh3u3//
bin/hadoop job  -
Dmapred.job.tracker=sccloud038.de.oracle.com:9001
-kill job_201211051447_0021
```

```
2013-02-01 14:19:20,017 Stage-1 map =   0%,  reduce =   0%
2013-02-01 14:19:25,262 Stage-1 map = 100%,  reduce =   0%
2013-02-01 14:19:29,317 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201211051447_0021
OK
10      1061
12      1029
14      1027
:
```

Diese Aufgabe wurde, wie man erkennen kann, allein mit einem Mapper gelöst. Sobald man ein Aggregat einführt, wird auch mit einem Reducer gearbeitet. Wir wollen die Anzahl der Vorkommen nun für jeden 10er-Bereich bilden. Das geht einfach mit folgender Hive-Abfrage:

```
hive> select round(zufallszahl+4, -1), sum(vorkommen)
    > from dojo_hivetab
    > group by round(zufallszahl+4, -1)
    > having sum(vorkommen) > 10000;
Total MapReduce jobs = 1
:
2013-02-01 14:32:02,677 Stage-1 map = 0%,  reduce = 0%
2013-02-01 14:32:08,755 Stage-1 map = 100%,  reduce = 0%
2013-02-01 14:32:18,892 Stage-1 map = 100%,  reduce = 33%
2013-02-01 14:32:20,921 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201211051447_0026
```

```
OK
10.0    10055
50.0    10078
:
Time taken: 28.596 seconds
```

Soll das Ergebnis nicht nur ausgegeben, sondern auch abgespeichert werden, führt man einfach ein CREATE TABLE AS SELECT aus. Vielfach ist es also gar nicht nötig, MapReduce-Code selbst zu schreiben. Kennt man die Struktur der Daten im HDFS gut und lässt sich die Aufgabe „in SQL" ausdrücken, so ist Hive eine sehr gute Alternative. Dabei arbeitet Hive fehlertolerant: Passen Zeilen in den HDFS-Dateien nicht zur Tabellendefinition, überspringt Hive sie einfach.

Bei sehr großen Datenmengen im Tera oder gar im Petabyte-Bereich wird es eine Weile brauchen, bis der MapReduce-Job durch ist, aber schließlich wird er durchlaufen und ein Ergebnis bringen.

Am Ende der Verarbeitungskette, entweder mit Hive oder mit selbst implementierten MapReduce-Jobs, stehen Daten im Zielformat – aus den Unmengen der Rohdaten im HDFS oder 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, für sich 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.**

3.5 ORACLE LOADER FOR HADOOP

Der Oracle Loader for Hadoop (OLH) ist ein Ladewerkzeug, welches die mit Hive-Abfragen oder MapReduce-Jobs berechneten, verdichteten oder aggregierten Daten in eine Tabelle im RDBMS Oracle lädt. Der Fokus des Oracle Loader for Hadoop liegt dabei auf dem möglichst effizienten Laden der Daten ins RDBMS. Um dieses Ziel zu erreichen, wird bereits im Hadoop Cluster eine binäre Data Pump-Datei generiert, in der alle Daten bereits fix und fertig für die Oracle-Datenbank aufbereitet sind. Diese Datei kann dann in der Datenbank als externe Tabelle bereitgestellt werden, so dass erst beim direkten Tabellenzugriff überhaupt Daten bewegt werden. Der OLH selbst ist als MapReduce-Job implementiert, läuft als im Hadoop Cluster ab und hat, wie alle MapReduce-Jobs, 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

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 (in der Realität sind die Datenmengen sicher größer):

```
$ 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.

3.5.1 DOWNLOAD UND INSTALLATION

Laden Sie den Oracle Loader for Hadoop aus dem Oracle Technology Network herunter und packen Sie das etwa 120 MB große ZIP-Archiv auf dem Rechner oder dem Rechnerknoten aus, auf dem das Hadoop Framework installiert ist. Zunächst finden Sie wiederum zwei ZIP-Archive; die README-Datei gibt Auskunft darüber, welches Sie verwenden müssen. Nach Auspacken des „richtigen" Archivs sieht die Verzeichnisstruktur 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
    `-- libolh11.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/*
:
```

3.5.2 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 2.0.0 - Production
Copyright (c) 2011, 2012, Oracle and/or its affiliates.

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 4 items
-rw-r--r--   1      0 2013-01-30 18:02 _SUCCESS
drwxr-xr-x   -      0 2013-01-30 18:02 _logs
drwxr-xr-x   -      0 2013-01-30 18:02 _glh
-rw-r--r--   1 16384 2013-01-30 18:02  oraloader-
00000-dp-0.dat
```

Der Unterordner `olh_ enthält` wiederum einige Dateien:

```
$ bin/hadoop dfs -ls hdfs-oraloader-output/_olh/*
Found 4 items
-rw-r--r--   1   242 2013-01-30 18:02 oraloader-00000-r.xml
-rw-r--r--   -  1065 2013-01-30 18:02 oraloader-dp.sql
-rw-r--r--   -  4102 2013-01-30 18:02 oraloader-report.txt
-rw-r--r--   1 15495 2013-01-30 18:02 tableMetadata.xml
```

Die Datei `_olh/oraloader-dp.sql` ist folgerichtig ein SQL-Skript zum Erstellen der externen Tabelle; und die Datei `oraloader-00000-dp-0.dat` ist eine Data-Pump-Exportdatei. Werfen Sie einen Blick in das SQL-Skript:

```
$ bin/hadoop dfs -cat hdfs-oraloader-output/_glh/
oraloader-dp.sql
--Oracle Loader for Hadoop Release 2.0.0 - Production
--Copyright (c) 2011, 2012, Oracle and/or its affiliates.
--
--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 SQL 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 des Datenbankservers 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 Datenstrukturen vorhanden sind, muss für jede Auswertung der gesamte Datenbestand durchgearbeitet werden. Diese Aufgabe übernimmt – ebenfalls massiv parallel – MapReduce. Die gewünschten Auswertungen beziehungsweise Aggregationen werden entweder als selbst implementierte MapReduce-Jobs oder als Hive-Queries 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.

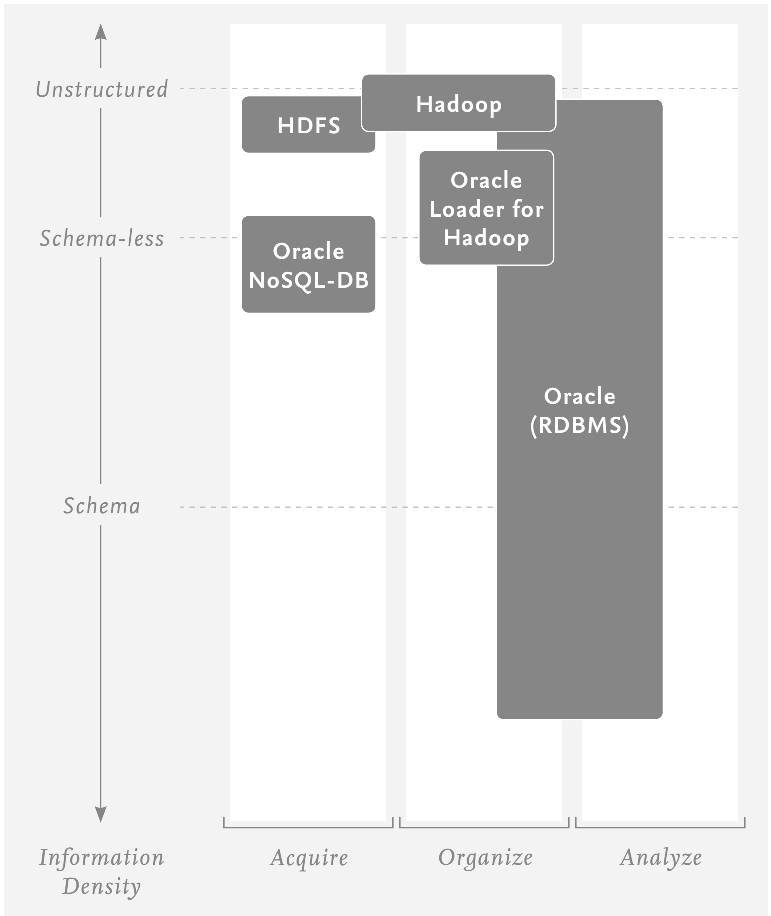


Abb. 7: Der Big-Data-Prozess: erfassen, veredeln, auswerten

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

Zugriff auf die komplette
Oracle Dojo-Bibliothek unter
http://tinyurl.com/dojoonline

ORACLE®

Copyright © 2012, 2013, Oracle. All rights reserved. Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

ORACLE®