Parallelisierte Datenanalyse im Computercluster

Zweckehe

Mit dem Framework und Programmiermodell MapReduce können riesige Datenmengen robust und schnell in einem Computercluster verarbeitet werden. In diesem Artikel wird MapReduce anhand eines Beispiels vorgestellt. Hadoop MapReduce arbeitet standardmäßig auf dem verteilten Dateisystem HDFS. Alternativ wird die NoSQL-Datenbank Cassandra verwendet, um von den Stärken beider Technologien zu profitieren.

von Kai Spichale

MapReduce [1] ist ein von Google entwickeltes Framework und Programmiermodell zur Verarbeitung großer Datenmengen in Computerclustern. Mit ihm können petabytegroße Datenmengen zuverlässig mit Commodity-Hardware verarbeitet werden. Eine Arbeitseinheit wird als „MapReduce Job" bezeichnet. Ein solcher Job könnte beispielsweise Texte für Suchanfragen indizieren oder Statistiken auf Basis von Server-Log-Dateien erstellen. Dazu werden die riesigen Datenmengen in kleinere Einheiten aufgeteilt und von unterschiedlichen Servern verarbeitet. So besteht ein einzelner Job aus vielen kleinen Tasks, die benutzerdefinierte Funktionen ausführen, um die gewünschte Analyse Stück für Stück durchzuführen.

Apache Hadoop [2] bietet eine Java-basierte Implementierung von MapReduce, dessen Funktionsweise im Mittelpunkt dieses Artikels steht. Doch Apache Hadoop ist mehr als nur MapReduce. Zu den Subprojekten gehören zum Beispiel das verteilte Dateisystem HDFS (Hadoop Distributed File System), die NoSQL-Datenbank HBase, die Datawarehouse-Infrastruktur Hive und die Datenverarbeitungsplattform Pig. Beim Entwurf von Hadoop wurden folgende Ideen, Anforderungen und Probleme berücksichtigt:

- *Die Datenmenge wächst unaufhörlich*: Lineare Skalierbarkeit ist eine der wichtigsten Eigenschaften von MapReduce. Verdoppelt sich die Menge der zu verarbeitenden Daten, kann die Verarbeitungszeit prinzipiell konstant gehalten werden, indem die Anzahl der Knoten im Computercluster verdoppelt wird.
- *Verteilung von Daten*: Die Kapazitäten, Transferraten und Zugriffszeiten von Festplatten wuchsen in den vergangen Jahrzehnten in ungleichen Verhältnissen. Deswegen benötigt man heute mehr Zeit zum Lesen des gesamten Festplatteninhalts als vor 20 Jahren [3]. Die Lesezeit kann durch Parallelisierung verringert werden. Das heißt, verteilt man beispielsweise die Daten einer Festplatte auf 100 Festplatten, dann müssen diese nur jeweils einen Prozent der Daten lesen.
- *Kolokation von Daten und Berechnung*: Damit die Netzwerkbandbreite nicht zum Flaschenhals wird, versucht MapReduce die Netzwerklast zu minimieren, indem es die Berechnung auf dem Knoten ausführt, auf dem auch die Daten gespeichert sind.
- *Fehler können immer auftreten*: Um Hochverfügbarkeit trotz Einsatz von kostengünstiger Commodity-Hardware garantieren zu können, muss die Fehlererkennung und -behandlung in die Anwendungsschicht gehoben werden. HDFS und Cassandra können den Ausfall einiger Knoten durch den Einsatz von Replikaten kompensieren. Auch das MapReduce-Framework kann mit Fehlern umgehen. Falls während eines MapReduce-Jobs ein Fehler auf einem Knoten auftritt, werden die Teilergebnisse dieses Knotens verworfen und die Aufgabe des Knotens wird von einem anderen wiederholt.

Wenn man für einen Moment von einigen technischen Details abstrahiert, dann ist die Grundidee hinter Hadoop MapReduce recht einfach: Sowohl die Speicherung der Daten als auch deren Verarbeitung wird auf ausreichend viele Knoten in einem Computercluster verteilt. Dauert die Verarbeitung zu lang oder wächst die Datenmenge, dann können weitere Knoten hinzugefügt und die Daten neu verteilt werden. Durch Parallelisierung wird die Zeit für das Lesen und die Verarbeitung verkürzt. Um die Netzwerklast zu minimieren und die Performance zu verbessern, werden die Daten von den

Knoten verarbeitet, auf denen sie auch gespeichert werden. Die Koordination der beteiligten Prozesse sowie die Fehlerbehandlung übernimmt das MapReduce-Framework. Benutzerdefinierte Funktionen werden eingesetzt, um große Datenmengen in kleinen Teilen zu verarbeiten. Wie im folgenden Abschnitt erläutert wird, sind diese benutzerdefinierten Funktionen nicht beliebig. Sie müssen zustandslos sein und bestimmte Interfaces implementieren. Auch der gesamte Datenfluss und die Phasen eines MapReduce-Jobs sind genau definiert. Die Daten werden in Form von Schlüssel-Wert-Paaren verarbeitet. Die Aufteilung der Daten in kleinere Einheiten, genannt Input Splits, muss zur beabsichtigten Verarbeitung passen. Eine Einheit muss unabhängig von den anderen Einheiten verarbeitet werden können. Soll beispielsweise das Vorkommen eines Wortes in einem ein Gigabyte großen Dokument gezählt werden, so könnte dieses Dokument in 16 Einheiten a 64 Megabyte gespeichert und verarbeitet werden. Die Speicherung und Verarbeitung dieser Einheiten kann auf verschiedenen Knoten erfolgen.

Phasen und Funktionen eines MapReduce-Jobs

Ein MapReduce-Job durchläuft drei Phasen. In der ersten Phase, der Map-Phase, verarbeitet eine Map-Funktion einzelne Schlüssel-Wert-Paare und erzeugt dabei eine Menge weiterer Schlüssel-Wert-Paare. Die so gebildeten Paare werden in der anschließenden Combine-Phase anhand ihrer Schlüssel gruppiert und zusammengefasst. In der optionalen Reduce-Phase berechnet die Reduce-Funktion das Endergebnis. Sowohl die Map- als auch die Reduce-Funktion sind zustandslos und können parallelisiert werden. Aus diesem Grund eignet sich MapReduce zur Verarbeitung großer Datenmengen in Computerclustern. Dieses Verarbeitungsmodell mag auf den ersten Blick starr und eingeschränkt erscheinen, doch es lassen sich viele Probleme auf MapReduce abbilden. Typischerweise wird es für Datamining, maschinelles Lernen, Tokenisierung und Indizierung von Texten oder zur verteilten Suche verwendet. Falls die drei Phasen doch nicht ausreichen, können mehrere MapReduce-Jobs miteinander kombiniert werden. Für komplexere Analysen werden Pig und Hive empfohlen [3]. Beide basieren auf MapReduce.

Das MapReduce-Beispiel

Die Funktionsweise von MapReduce soll mit dem in **Abbildung 1** dargestellten Beispiel genauer erläutert werden. Am Ende des Artikels wird dieses Beispiel noch einmal aufgegriffen und die Implementierung mit Quellcodeauszügen beschrieben.

Angenommen eine Anwendung verwaltet für jeden Benutzer eine Kontaktliste. Ein Kontakt ist eine bidirektionale Beziehung zwischen zwei Benutzern. Für eine Analyse der Benutzerdaten sollen die gemeinsamen Kontakte zweier Benutzer berechnet werden. Im dargestellten Beispiel heißen die Benutzer A, B und C. Zur Vereinfachung des Beispiels wurden die Kontaktlisten so

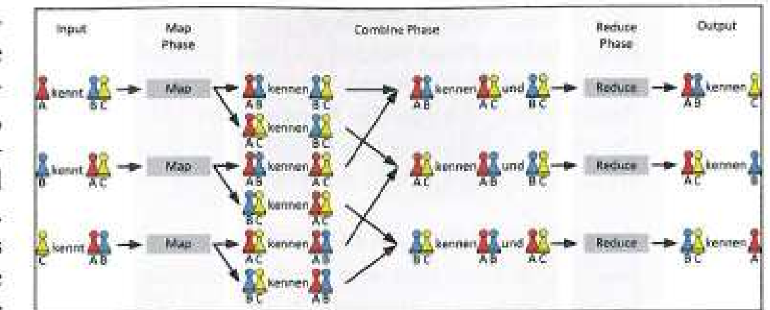

Abb. 1: Berechnung der gemeinsamen Bekannten

gewählt, dass alle drei Benutzer einander kennen. Jede Kontaktliste wird einzeln in der Map-Phase verarbeitet. Der Schlüssel eines gebildeten Schlüssel-Wert-Paares besteht aus dem Benutzer und je einem seiner Kontakte. Weil Benutzer A die beiden Benutzer B und C kennt, bildet die Map-Funktion zwei Schlüssel-Wert-Paare mit den Schlüsseln AB und AC. Der Wert dieser Schlüssel-Wert-Paare ist die Kontaktliste von Benutzer A. Nach Abschluss der Map-Phase werden in der Combine-Phase

Klassifizierung der Anwendungsgebiete und Beispiele

Die unterschiedlichsten Probleme können auf MapReduce abgebildet werden. Es gibt Probleme, die ohne Reduce-Phase gelöst werden können, und andere, die eine Reduce-Phase benötigen [4].

Problemklassen ohne Reduce-Phase

- *Suchen*: Die Map-Funktion emittiert die Input Splits, die das Suchkriterium erfüllen. Ein Beispiel hierfür ist das Durchsuchen von Texten nach einem bestimmten Suchbegriff. Nur wenn der Text den Suchbegriff enthält, wird der Text vom Mapper emittiert. Auf diese Weise werden alle Texte herausgefiltert, die den Suchbegriff nicht enthalten.
- *Massenkonvertierung*: Die Map-Funktion führt die Konvertierung in das Zielformat aus. Die Input Splits könnten zum Beispiel Bilder in unterschiedlichen Formaten sein. Die Map-Funktion konvertiert die Bilder in das gewünschte Ausgabeformat.
- *Sortieren*: Die Map-Funktion emittiert die Input Splits mit dem Sortierkriterium als Schlüssel. In der Combine-Phase erfolgt die eigentliche Sortierung durch das MapReduce-Framework. Zum Beispiel könnten Spieler eines Onlinecomputerspiels anhand ihrer Punktestände sortiert werden.

Problemklassen mit Reduce-Phase

Gruppieren und Aggregieren: Die Map-Funktion emittiert Datensätze mit dem Gruppenattribut als Schlüssel, und die Reduce-Funktion aggregiert alle Werte mit gleichem Gruppenattribut. Sollen zum Beispiel für einen Händler die Umsätze seiner Kunden ermittelt werden, so muss für jeden Kunden die Summe seiner Rechnungen gebildet werden. Die Map-Funktion emittiert Datensätze mit der Kunden-ID als Schlüssel und dem Rechnungsbetrag als Wert. In der Combine-Phase entsteht pro Kunden-ID eine Liste mit Rechnungsbeträgen. Die Reduce-Funktion muss dann nur noch die Summe der Rechnungsbeträge in den Listen bilden.

die Schlüssel-Wert-Paare sortiert und zusammengefasst. In der Reduce-Phase werden schließlich die Schnittmengen der Kontaktlisten pro Benutzerpaar gebildet. Das Benutzerpaar AB hat die Kontaktlisten AC und BC. Durch die Berechnung der Schnittmenge ergibt sich C als gemeinsamer Kontakt von A und B.

Hadoop MapReduce mit HDFS oder Cassandra

Die MapReduce-Jobs von Hadoop arbeiten normalerweise auf dem Hadoop Distributed File System (HDFS). HDFS ist kein Allzweckdateisystem für beliebige Anwendungen. Es ist geeignet für Applikationen, die Streamingzugriff [5] auf die gespeicherten Daten benötigen. Hoher Durchsatz ist wichtiger als geringe Latenz. Deswegen ist HDFS für langlaufende Batch-Verarbeitung ideal und vergleichsweise ungeeignet für interaktive Applikationen, die kurze Antwortzeiten voraussetzen.

Listing 1

```
public class MutualContactsMapper extends
  Mapper<ByteBuffer, SortedMap<ByteBuffer, IColumn>, Text, MapWritable> {

  @Override
  protected void setup(Context context) throws IOException, InterruptedException { }

  @Override
  public void map(ByteBuffer key, SortedMap<ByteBuffer, IColumn> columns,
                                               Context context) throws Exception {

    // Kontaktliste erstellen
    MapWritable contacts = new MapWritable();
    for (Map.Entry<ByteBuffer, IColumn> entry : columns.entrySet()) {
      IColumn column = entry.getValue();
      String columnValue = ByteBufferUtil.string(column.value());
      Text keyText = new Text(columnValue);
      contacts.put(keyText, keyText);
    }

    // User-Paare mit Kontaktliste als Zwischenergebnis speichern
    String userId = ByteBufferUtil.string(key);
    for (Map.Entry<ByteBuffer, IColumn> entry : columns.entrySet()) {
      IColumn column = entry.getValue();
      String contactUserId = ByteBufferUtil.string(column.value());
      String userPair = createMapOutputKey(userId, contactUserId);
      Text mapOutputKeyText = new Text(userPair);
      context.write(mapOutputKeyText, contacts);
    }
  }
  // Schlüssel für Zwischenergebnis berechnen
  private String createMapOutputKey(String userId1, String userId2) {
    if (userId1.compareTo(userId2) < 0) return userId1 + userId2;
    return userId2 + userId1;
  }
}
```

Apache Cassandra [6] ist ein Wide Column Datastore, der geeignet ist, riesige Datenmengen zu verwalten. Mit Techniken wie Replikation und Sharding kann Cassandra als skalierbarer, ausfallsicherer und Storage Layer verwendet werden. Cassandra überzeugt durch das Random-Access-Lesen und besonders schnelles Schreiben, doch leider sind die Abfragemöglichkeiten aufgrund der Architektur und des Datenmodells eingeschränkt.

Zur Umsetzung des in diesem Artikel beschriebenen Beispiels wird Hadoop MapReduce in Kombination mit Cassandra eingesetzt, sodass HDFS entfällt. Durch die Integration von Hadoop und Cassandra [7] können die Stärken beider Technologien ausgenutzt werden. Einerseits können zeitaufwändige und komplexe Datenanalysen mit MapReduce-Jobs durchgeführt werden und andererseits bietet Cassandra einen skalierbaren Storage Layer mit geringer Latenz. Die MapReduce-Implementierung von Hadoop ist die technologische Basis für Pig und Hive. Diese beiden Technologien können ebenfalls in Kombination mit Cassandra eingesetzt werden. Pig ist eine Plattform zur Analyse großer Datenmengen. Mit der Sprache Pig Latin können Sequenzen von Datentransformationen definiert werden. Hive unterstützt die SQL-ähnliche Sprache HiveQL.

Systemaufbau und Datenfluss

Bestandteil eines MapReduce-Jobs sind die Eingabedaten, das MapReduce-Programm und die Konfigurationsinformationen. Das Programm und die Konfigurationsinformationen werden automatisch im Cluster verteilt, wenn ein Job gestartet wird. Das Hadoop-Framework führt den Job aus und verarbeitet die Daten mit einer Vielzahl von Tasks. Hadoop unterscheidet Map-Tasks und Reduce-Tasks.

Hadoop arbeitet nach einem Master-Slave-Ansatz. Das heißt es gibt einen Master-Knoten und mehrere Slave-Knoten. Ersterer ist der Job-Tracker, die Slave-Knoten sind die Task Tracker. Der Job-Tracker ist das zentrale Steuerprogramm für die Ausführung der Jobs. Die Task Tracker führen die Map-, Combine- und Reduce-Funktionen aus. Der Job-Tracker überwacht den Fortschritt und reagiert auf Fehler. Falls ein Task Tracker ausfällt, wird die Arbeit mit einem anderen wiederholt, bis der gesamte MapReduce-Job vollständig abgearbeitet ist. Hadoop teilt die Eingabedaten in Input Splits. Die Task Tracker starten für jeden Input Split einen separaten Java-Prozess, um alle Datensätze im Input Split mit der benutzerdefinierten Map-Funktion zu verarbeiten. Der Job-Tracker wählt jeweils den Task Tracker aus, der sich am nächsten an den Daten befindet und berücksichtigt dabei nicht dessen aktuelle Last. Die Durchführung des gesamten Jobs kann wegen eines einzelnen langsamen Servers in die Länge gezogen werden. Für die meisten Jobs wird ein HDFS-Block mit 64 Megabyte als Split-Größe verwendet [3].

Die Map Tasks schreiben ihre Ergebnisse nicht in HDFS beziehungsweise Cassandra, sondern auf die lokale Festplatte. Dieses Zwischenergebnis kann gelöscht werden, sobald es von einem Reduce Task erfolgreich weiterverarbeitet wurde. Deswegen wäre das Speichern in HDFS beziehungsweise in Cassandra mit der Verwaltung von Replikaten zu aufwendig. Falls ein Knoten ausfällt, noch bevor die Zwischenergebnisse weiterverarbeitet werden konnten, wird der Map Task auf einem anderen Task Tracker wiederholt, um das Map-Ergebnis wiederherzustellen.

Die Reduce Tasks profitieren nicht von der Datenlokalität, denn ihre Eingabedaten können potenziell von allen Map Tasks stammen. Die Ausgabedaten der Map Tasks sind sortierte Maps, die über das Netzwerk zu dem Knoten transportiert werden, auf dem der Reduce Task durchgeführt wird. Dort werden die Ausgabedaten gruppiert und zur benutzerdefinierten Reduce-Funktion weitergeleitet. Das Ergebnis des Reduce Tasks wird in HDFS beziehungsweise Cassandra gespeichert, um die Ergebnisse vor Verlust zu schützen.

Um den Performancevorteil durch Kolokation von Datenspeicherung und -verarbeitung auszunutzen, sollten die Task Tracker und die Cassandra-Knoten auf den gleichen Maschinen laufen. Dieses Deployment-Szenario ist in **Abbildung 2** dargestellt. Die Daten müssen nicht über ein Netzwerk ausgetauscht werden, denn der Job-Tracker kann jeweils den Task Tracker beauftragen, auf dessen Rechner die Daten liegen.

Alternativ könnte für die Datenanalyse mit Hadoop eine dedizierte Replikatgruppe verwendet werden. Dieses Deployment-Szenario ist in **Abbildung 3** dargestellt. Cassandra arbeitet mit einer konfigurierbaren Replikatanzahl. Diese Replikate können gezielt auf Knoten in bestimmten Rechenzentren und Racks verteilt werden. Vordergründig sollen diese Mechanismen die Ausfallsicherheit und die Verfügbarkeit erhöhen. Doch mit diesen Mechanismen kann auch eine dedizierte Replikatgruppe für die Datenanalyse erzeugt werden. Die anderen Cassandra-Knoten können weiterhin als Backend mit geringer Latenz für Applikationen verwendet werden. Die Replikate werden bei Schreiboperationen automatisch durch Cassandra synchronisiert. Somit müssen die Daten für eine Analyse nicht erst durch einen ETL-Prozess in ein anderes System geladen, sondern können direkt verwendet werden.

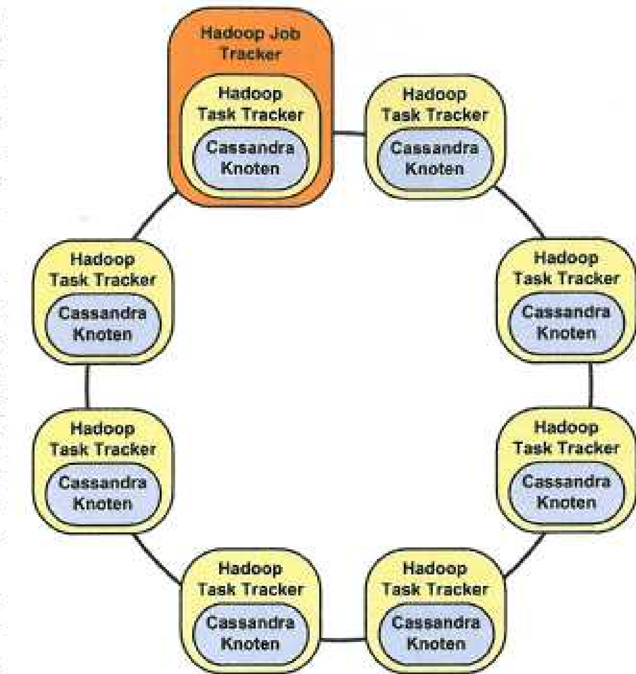

Abb. 2: Vollständige Zusammenlegung von Hadoop und Cassandra

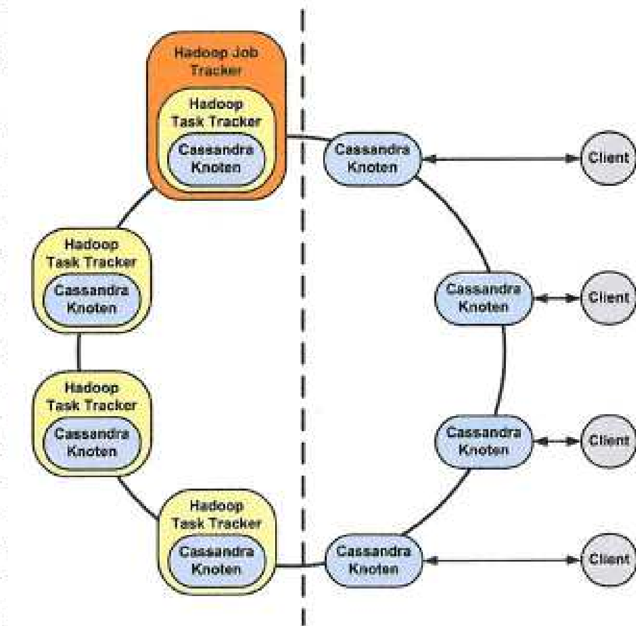

Abb. 3: Hadoop-Analysen in separater Replikatgruppe

Implementierung des Beispiels

Die Implementierung des MapReduce-Beispiels in **Abbildung 1** zur Berechnung der gemeinsamen Kontakte zweier Benutzer wird nun vorgestellt. Der vollständige Quellcode ist auf der Heft-CD enthalten. In der Datenbank befindet sich ein Keyspace mit drei Column Families: *Users*, *Contacts* und *MutualContacts*. Die Column Family *Contacts* speichert die Kontakte der Benutzer. Dafür enthält sie für jede Benutzer-ID eine Liste mit Benutzer-IDs. Die Column Family *Contacts* enthält demzufolge die Eingabedaten für den Mapper. In der Column Family *MutualContacts* werden die Ergebnisse gespeichert. Wie noch gezeigt wird, schreibt der Reducer seine Ergebnisse in diese Column Family. Da die Benutzer-IDs eindeutig sind, ergibt die Konkatenation von zwei Benutzer-IDs einen eindeutigen Schlüssel. Auf diese Weise kann ein eindeutiger Schlüssel für die gemeinsamen Kontakte zweier Benutzer gebildet werden. Die Kontakte werden, wie in *Contacts*, in Form einer Liste von Benutzer-IDs gespeichert.

Listing 1 zeigt die Implementierung des Mappers. Die Abarbeitung der Eingabedaten erfolgt nach dem Push-Prinzip. Das heißt, das MapReduce-Framework ruft die Methoden des Mappers auf. Die Methode *map* der Klasse *MutualContactsMapper* erhält jeweils eine Row aus der Column Family *Contacts*. Der Methodenparameter *key* ist der Zeilenschlüssel und entspricht einer

Benutzer-ID. Der Methodenparameter *columns* entspricht den Columns in der Row, also der Kontaktliste. Die Zwischenergebnisse speichert der Mapper im Methodenparameter *context*. In Listing 2 ist der Reducer in verkürzter Form abgebildet. Der Methodenparameter *word* enthält den Schlüssel eines Zwischenergebnisses. Die Kontaktlisten sind im Parameter *values* enthalten und das Objekt *context* wird wieder benutzt, um die Ergebnisse zu speichern. Die Konfiguration des MapReduce-Jobs ist in Listing 3 abgebildet. Mit einer Key Range wird ein Teil der Column Family selektiert. Auf diese Weise kann ein Input Split gebildet werden. Für jede Input Split wird die Instanz der Klasse *Job* erzeugt und gestartet.

Listing 2

```
public class MutualContactsReducer extends
  Reducer<Text, MapWritable, ByteBuffer, List<Mutation>> {

  @Override
  public void reduce(Text word, Iterable<MapWritable> values, Context context)
    throws Exception{

    // Gemeinsame Kontakte berechnen
    Set<Writable> mutualContacts = ...

    // Ergebnis speichern
    ByteBuffer outputKey = StringSerializer.get().toByteBuffer(word.toString());
    List<Mutation> updateOperations = createUpdateOperations(mutualContacts);
    context.write(outputKey, updateOperations);
  }

  // Noch mehr Code
}
```

Listing 3

```
public class MutualContactsJob extends Configured {

  public void run() throws Exception {
    super.setConf(new Configuration());
    Map<String, String> keyRangeMap = Maps.newHashMap();

    // Code fuer KeyRange-Definition

    for (Map.Entry<String, String> entry : keyRangeMap.entrySet()) {
      Job job = new Job(getConf(), JOB_NAME);
      job.setJarByClass(MutualContactsJob.class);
      job.setMapperClass(MutualContactsMapper.class);
      job.setReducerClass(MutualContactsReducer.class);

      // noch mehr Code für Konfiguration

      SlicePredicate predicate = new SlicePredicate().setSlice_range(range);
      ConfigHelper.setInputSlicePredicate(job.getConfiguration(), predicate);
      job.waitForCompletion(true);
    }
  }
}
```

Zusammenfassung

MapReduce ist ein einfaches, aber wirkungsvolles Programmiermodell zur Verarbeitung riesiger Datenmengen mit einer großen Anzahl von Computern. Die Speicherung der Daten und deren Verarbeitung wird auf viele Computer beziehungsweise viele Festplatten verteilt. Hadoop glänzt, wenn die Datenmenge und nicht die Komplexität der Verarbeitungsalgorithmen die MapReduce-Jobs bestimmen. Ein wichtiges Konzept ist die Kolokation von Daten und Verarbeitung, denn bei sehr großen Datenmengen ist es weniger aufwändig, das MapReduce-Programm zu den Daten zu bringen, als die Daten zum MapReduce-Programm. HDFS ist hervorragend für MapReduce- und Batch-Jobs geeignet, aber aufgrund seiner hohen Latenz eignet es sich nicht für Anwendungen, die kleine Datenmengen in möglichst schneller Zeit lesen und schreiben wollen. Die NoSQL-Datenbank Cassandra profitiert von der Hadoop-Integration. Denn mit MapReduce beziehungsweise Hive und Pig ist es möglich, den aktuellen Datenbestand zu analysieren, sodass produktive Datenbestände zeitnah ausgewertet werden können.

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

Links & Literatur

[1] http://labs.google.com/papers/mapreduce.html

[2] http://hadoop.apache.org/

[3] White, Tom: „Hadoop: The Definitive Guide", O'Reilly Media, 2009

[4] Masanori Fujita: „Hadoop – High Performance Batches in der Cloud", OOP 2011

[5] http://hadoop.apache.org/common/docs/current/streaming.html

[6] http://cassandra.apache.org/

[7] http://wiki.apache.org/cassandra/HadoopSupport