PL/SQL vs. Spark – Umsteigertipps für's DWH

Christopher Thomsen
OPITZ CONSULTING Deutschland GmbH
Hamburg

Jens Bleiholder
OPITZ CONSULTING Deutschland GmbH
Berlin

Schlüsselworte
Big Data, Spark, PL/SQL, SQL, ETL, Hadoop, DWH

Einleitung
Mit Hadoop 2.0 öffnete sich die Big Data-Plattform für neue Algorithmen und Technologien, um auch als Basis für In-memory Computing, Ad-Hoc Query und Streaming-Anwendungen nutzbar zu sein. Apache Spark etabliert sich hier derzeit als Vorreiter unter den Hadoop-Allzweckwaffen und kommt in immer mehr Produkten - unter anderem auch dem Big Data Connector des Oracle Data Integrators und Oracle Big Data Discovery - als Ausführungsframework zum Einsatz. Was Spark ist, wo es in den neuen Oracle-Produkten in welcher Form zum Einsatz kommt und wie sich ETL-Prozesse, -Werkzeuge und die Datenintegration im Data Warehouse dadurch verändern, soll in folgendem exemplarisch durch Gegenüberstellung von in PL/SQL und Spark implementierten Anwendungsbeispielen aufgezeigt werden.

Was ist Spark
MapReduce ist der zentrale, ursprünglich von Google veröffentlichte Algorithmus, der sich seit nun über 10 Jahren hinter skalierbaren Technologien des Hadoop Ökosystems verbirgt. Damit handelt es sich um ein bereits in die Jahre gekommenes Verfahren in dem State-of-the-Art Framework, welches sich Hadoop nennt, und heute durch den Big Data Hype immer häufiger zum Einsatz kommt. Derzeit gibt es deutliche Anzeichen für einen Technologieschwenk im Big Data-Bereich: Mehr und mehr OpenSource Projekte (z.B. Hive, Pig, Mahout) wie auch proprietäre Projekte der großen Mitbewerber tauschen das inzwischen in die Jahre gekommene MapReduce-Framework von Hadoop gegen das neuere Apache Spark-Framework aus und versprechen sich dadurch erhebliche Vorteile.

Hinter Spark verbirgt sich also ein Framework für verteiltes Rechnen, welches sich inzwischen auf zahlreichen Infrastrukturen (Hadoop, Cassandra, Mesos, Amazon EMR, HBase, Spark Cluster) ausführen lässt. Wie MapReduce ist Spark ursprünglich als Skalierungsframework für Batchjobs entwickelt worden, ließ im Gegensatz zu MapReduce jedoch auch um eine Streaming Komponente erweitern. Mittlerweile bietet Spark ebenfalls von Hause aus die Bibliotheken Spark SQL zur Interpretation und Ausführung von SQL in Spark Jobs, Spark Mlib für Machine Learning und Spark GraphX für Graphenoperationen.

Damit lässt sich Spark hervorragend für die Datentransformation einsetzen: Es ist auf Commodity-Hardware skalierbar, läuft nativ auf zahlreichen Infrastrukturen (inkl. Amazon EMR Cloud), kann sowohl tabellarisch strukturierte Daten (nativ oder via SQL) als auch mit semistrukturierten und unter Einbindung anderer Apache Projekte wie Lucene und OpenNLP auch mit unstrukturierten Daten verarbeiten und kann dies bei Bedarf auch Datenstrombasiert. Zudem zeigt sich, dass die API Umfangreich und in Kombination mit Scala oder Python als Programmiersprachen die Queries sehr kompakt sind.

Erste Schritte in Spark
Spark lässt sich zum Testen einfach von der gleichnamigen Seite der Apache Foundation herunterladen und als Standalone Cluster auf der eigenen Maschine ausführen. In vielen Hadoop Distributionen ist Spark bereits vorinstalliert, sodass sich der kompilierte Spark Bytecode dort direkt ausführen lässt und auch auf die verteilt gespeicherten Daten im HDFS zugreifen kann. Spark liefert neben der Runtime auch eine interaktive Shell auf welcher sich Programme interaktiv entwickeln lassen. Die Shell-Funktion ist ebenfalls im Clusterbetrieb verfügbar und nutzbar.

Das wichtigste Element in Spark ist das Resilient Distributed Dataset (RDD), welche die Implementierung einer verteilten Liste ist. Diese Liste kann alle möglichen Formen von Inhalten enthalten, von einfachen Datentypen, binären Daten, hierarchischen Daten bis hin zu Tabellen oder Sublisten. Alle Operationen, die auf RDDs ausgeführt werden, nutzen automatisch die Spark Runtime, um die Operationen auf der zugrunde liegenden Infrastruktur zu parallelisieren. Ein RDD lässt sich entweder aus jeder beliebigen Liste erstellen oder aber auch direkt aus Datenquellen, wie Datenbanken oder Dateien:

```
// RDD aus einer Liste fortlaufender Zahlen erstellen
val myRDD1 = sc parallelize (1 to 1000000)

// RDD aus einer CSV Datei erstellen
val myRDD2 = sc textFile "~/sometable.csv"
```

Verarbeitet man auf diese Art und Weise z.B. CSV Dateien, so muss der Inhalt noch über den Delimitter gesplittet werden und optional auch in ein Objekt mit dem Schema der CSV konvertiert werden:

```
// Spalten der CSV Datei am Delimitter "," trennen
val myRawData = myRDD2 map (_ split ",")

// Das Schema in ein Objekt mappen
case class Employee (name: String, age: Int, income: float,
                     job: String)
val myData = myRawData map (v => Employee(v(0), v(1).toInt, v(2).toFloat,
                           v(3)))
```

Mit diesen RDDs können nun allerhand Filterungen, Transformationen und Gruppierungen vorgenommen werden.

```
# Daten in SQL filtern
SELECT * FROM myData d WHERE d.age >= 18 AND d.name LIKE 'M%';

// Daten in Spark filtern
myData filter (d => d.age >= 18 && d.name startsWith "M")

# Transformation und Projektion in SQL
SELECT d.name, (d.age + 1) FROM myData d;

// Transformation und Projektion in Spark
myData map (d => d.name -> d.age + 1)

# Einfache Distinct Count Aggregation in SQL
SELECT COUNT(DISTINCT d.name) FROM myData d WHERE d.age < 18;
```

```
// Einfache Distinct Count Aggregation in Spark
(myData filter (_.age < 18)).distinct.count

# Gruppierung und Aggregation in SQL
SELECT d.age, SUM(d.income) FROM myData d GROUP BY d.age;

// Gruppierung und Aggregation in Spark
myData map (d => d.age -> d.income) reduceByKey (_ + _)
```

Diese grundlegenden Funktionen stellen in SQL und in Spark keine Herausforderung dar und lassen sich hier bequem abwickeln. Aber auch analytisches SQL lässt sich in Spark abbilden, wenn auch etwas komplizierter:

```
# Gruppenweise Aggregation in SQL
SELECT
  d.name,
  d.job,
  d.income,
  AVG(d.income) OVER (PARTITION BY d.job)
FROM myData d WHERE d.income < 4000 ORDER BY d.job;

// Gruppenweise Aggregation in Spark
val ds1 = myData filter (_.income < 4000)
val jobs = ds1 map (
  d => d.job -> (d.income, 1)
) reduceByKey (
  (a, b) => a._1 + b._1 -> a._2 + b._2
) map (
  d => d._1 -> d._2._1 / d._2._2)
ds1 map (
  d => d.job -> d
) join jobs map (
  d => (d._2._1.name, d._1, d._2._1.income, d._2._2)
) sortBy (_._2)

# Vergleich mit benachbarten Zeilen in SQL
SELECT
  s.name,
  s.age,
  CAST(CASE WHEN s.previousIncome <= s.income THEN 1 ELSE 0 AS bit)
FROM (
  SELECT *,
    LAG (d.income) OVER (ORDER BY s.name) AS previousIncome
  FROM myData d
) s ORDER BY s.name DESC

// Vergleich mit benachbarten Zeilen in Spark
myData sortBy (_.name, false) sliding 2 map (t =>
  (t._2.name, t._2.age, t._1.income <= t._2.income))
```

Jetzt wird's prozedural

Die zuvor gezeigten Aufgaben ließen sich vollständig in SQL abbilden und waren hier mindestens genauso einfach und kompakt wie das äquivalent in Spark. Interessant werden die Möglichkeiten der Spark Syntax dort, wo man gezwungen ist prozedural zu arbeiten oder erweiterte Funktionen einzubinden:

```
# Die obersten N Elemente in PL/SQL ausgeben
DECLARE
  CURSOR c1 IS
    SELECT d.name, d.job, d.income FROM myData d ORDER BY d.income DESC;
  myName CHAR(40);
  myJob    CHAR(40);
  myIncome FLOAT(4);
BEGIN
  OPEN c1;
  dbms_output.put_line('Top Verdiener:');
  LOOP
    FETCH c1 INTO myName, myJob, myIncome;
    EXIT WHEN (c1%ROWCOUNT > 5) OR (c1%NOTFOUND);
    dbms_output.put_line(myName || ' verdient ' || myIncome || ' als '
      || myJob);
  END LOOP;
  CLOSE c1;
END;

// Die obersten N Elemente in Spark ausgeben
val result = (myData sortBy (_.income, false) takeOrdered 5).collect
println("Top Verdiener:")
result foreach (d => println(String format ("%s verdient %s als %s",
  d.name, d.income.toString, d.job)
```

Spark kann SQL auch direkt interpretieren

Standard SQL kann durch das Spark SQL Modul auch direkt wie in PL/SQL innerhalb des Spark Codes verwendet und mit anderen Codebestandteilen gemixt werden. Hierfür wird der SQL Context verwendet:

```
myData registerTempTable "myTable"
sqlContext sql "SELECT d.age, AVG(d.income) FROM myData d GROUP BY d.age"
  filter (_(0) >= 18) map (
    d => String format ("%s years => %s€", d(0).toString, d(1).toString)
  ) foreach (println)
```

Auf diese Art und Weise lassen sich querybasierte Operationen in SQL abbilden und direkt mit kompakten prozeduralen Funktionen in Spark kombinieren.

Kontaktadresse:
Christopher Thomsen
OPITZ CONSULTING Deutschland GmbH
Butendeitchsweg 2
D-21129 Hamburg

Telefon: +49 (0) 173 72729604
E-Mail christopher.thomsen@opitz-consulting.com
Internet: www.opitz-consulting.de

Jens Bleiholder
OPITZ CONSULTING Deutschland GmbH
Tempelhofer Weg 64
D-12347 Berlin

Telefon: +49 (0) 173 7279426
E-Mail jens.bleiholder@opitz-consulting.com
Internet: www.opitz-consulting.de