MapReduce

Eine kleine Einführung

Simon Elsbrock
<simon@iodev.org>

RaumZeitLabor e.V.
Mannheim, 17.01.2012

Agenda

Einführung
- MapReduce: Das Konzept
- Jobs, Master, Worker
- Fehlertoleranz
- Taskgranularität
- Redundante Ausführung

Beispiele
- Wörter zählen
- Anagramme
- Reverse Web Graph

Praxis
- MapReduce mit AWS

Motivation

- (sehr) große Datenmengen (z.B. Petabytes)
- „konventionelle“ Ansätze nicht ausreichend
  - Beispiel: 20 Mrd. Dateien mit jeweils 20kb ($\approx$ 400 TB)
  - bei 50 MB/s dauert lesen ca. drei Monate
  - Verarbeitung der Daten noch länger!

Motivation

- (sehr) große Datenmengen (z.B. Petabytes)
- „konventionelle“ Ansätze nicht ausreichend
  - Beispiel: 20 Mrd. Dateien mit jeweils 20kb ($\approx$ 400 TB)
  - bei 50 MB/s dauert lesen ca. drei Monate
  - Verarbeitung der Daten noch länger!

- Lösungsansatz: Verteiltes Rechnen
- gibts natürlich nicht umsonst:
  - Kommunikation, Koordination
  - Recovery, Monitoring
  - Storage

Motivation

- (sehr) große Datenmengen (z.B. Petabytes)
- „konventionelle“ Ansätze nicht ausreichend
  - Beispiel: 20 Mrd. Dateien mit jeweils 20kb ($\approx$ 400 TB)
  - bei 50 MB/s dauert lesen ca. drei Monate
  - Verarbeitung der Daten noch länger!

- Lösungsansatz: Verteiltes Rechnen
- gibts natürlich nicht umsonst:
  - Kommunikation, Koordination
  - Recovery, Monitoring
  - Storage
- *Rinse, lather, repeat!*

Was ist MapReduce?

„MapReduce is a programming model and an associated implementation for processing and generating large data sets.“ [1]

- Abstraktionsschicht (Library) von Google, die sich um
  - Parallelisierung,
  - Fehlertoleranz,
  - Daten- und
  - Lastverteilung

  kümmert.
- *nicht* öffentlich verfügbar
- freie Implementierung: Apache Hadoop

map() & reduce()

- Ein Großteil aller Berechnungen lässt sich auf zwei Funktionen runterbrechen:
  - *map*(key, value): nimmt das Eingabepaar und erstellt eine neue Menge an „*Intermediate-Pairs*“
  - *reduce*(key, values): nimmt alle Intermediate-Pairs mit gleichem *key* und „aggregiert“ dessen Werte
- Aufgabe des Benutzers, diese Funktionen zu definieren!

- viele map und reduce Funktionen sind sehr simpel und lassen sich daher einfach wiederverwenden

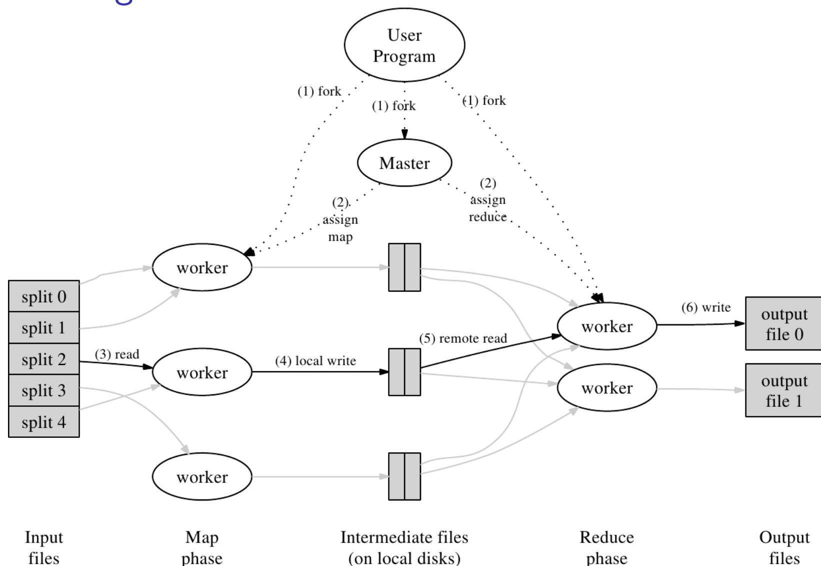
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Anzahl an map & reduce Funktionen bei Google im Jahre 2004 [1]

MapReduce Jobs & Ablauf

- Ein Job besteht aus
  - Referenz zu Eingabedaten / Ausgabeort
  - Map-Funktion
  - Reduce-Funktion
  - Partitioning-Funktion (optional)
  - Combiner-Funktion (optional)
- Jobs können „verkettet“ werden
- Scheduler (Master) verteilt Jobs an verfügbare Worker
  - Lokalität der Daten wird hierbei beachtet
- Worker führen map bzw. reduce aus
  - die notwendigen Daten werden lokal zwischengespeichert
  - beim reduce werden die Daten von den map-Workern gelesen
  - das Endergebnis wird wieder ins verteilte FS geschrieben

Ausführung



Quelle: [1]

Fehlertoleranz

- viele Rechner (Cluster); es gehen ständig welche kaputt
- Zuverlässige Berechnung absolut erforderlich!

- Worker Failure:
  - Failure wird über Heartbeat festgestellt (vom Master)
  - Erneutes Scheduling von fertigen und laufenden map-Tasks
  - Erneutes Scheduling von laufenden reduce-Tasks

- Master Failure:
  - da nur ein Rechner, eher unwahrscheinlich
  - aber: regelmäßiges Schreiben von Checkpoints möglich
  - bei Failure ab letztem Checkpoint fortsetzen

Taskgranularität

- M map-Tasks, R reduce-Tasks, n Rechner
- Es sollte gelten: $M + R \gg n$
  - verbessert die Lastverteilung
  - beschleunigt Recovery bei Failures

- M wird üblicherweise so gewählt, dass die Eingabedaten jedes Map-Tasks 16 – 64 MB groß sind
- R ist üblicherweise durch den User beschränkt, da jeder Reduce-Task eine Ausgabedatei erzeugt

- Beispiel: $M = 200000$, $R = 5000$, $n = 2000$

Redundante Ausführung

- Problem: *Straggler*: Rechner, der bei der Ausführung der Tasks eine ungewöhnlich lange Zeit benötigt
- Lösung: Master veranlasst bei Näherung der Fertigstellung die erneute Ausführung der verbleibenen Tasks, die momentan noch ausgeführt werden
- sobald der *Backup Task* von einem Rechner ausgeführt wurde, wird er als *completed* markiert
- beschleunigt die Fertigstellung des Jobs signifikant!

Beispiel: Wörter zählen

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts int result = 0;
  for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```

Beispiel: Anagramme finden

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(sortLetters(w), w);

reduce(String key, Iterator values):
  // key: a word with alphabetically sorted letters
  // values: anagram of the word
  for each v in values:
      Emit(AsString(key + " " v));
```

Beispiel: Reverse Web Graph

```
map(String key, String value):
  // key: document url
  // value: document contents
  for each link a in value:
    EmitIntermediate(a.href, key);

reduce(String key, Iterator values):
  // key: a link target
  // values: urls pointing to the target
  for each v in values:
      Emit(AsString(key + " " v));
```

Amazon Elastic MapReduce

Keinen eigenen Cluster, um das ganze mal auszuprobieren?

- Amazon Elastic MapReduce (EMR) = virtualisierter Cluster basierend auf Apache Hadoop
- startet vorkonfigurierte Amazon Elastic Computing Cloud (EC2) Instanzen
- liest Eingabedaten von / speichert Ausgabedaten auf Amazon Simple Storage Service (S3) oder Apache Hadoop FS (HDFS)

- „Metered Usage“: man bezahlt nur die Zeit, in der gerechnet wird (Instanzen werden automatisch heruntergefahren)
- Preis = Amazon EC2 + Amazon EMR + Amazon S3

Amazon EMR: Job Flows

- *Job flow*: „collection of processing steps that Amazon EMR runs on a specified dataset“
  - Anzahl und Art der Instanzen
  - Logdatei
- *Job Flow Step*: „user-defined unit of processing“
  - verarbeitende Anwendung
    - Hadoop App: Java
    - Streaming: Java, Ruby, Perl, Python, PHP, R oder C++
  - Ort der Eingabedaten (Amazon S3, HDFS)
  - Zielort der Ausgabedaten (Amazon S3, HDFS)
- können über die AWS Management Console, API oder CLI definiert & gestartet werden

Amazon EMR: Beispiel

```
$ ./elastic-mapreduce --create --stream \
    --num-instances COUNT --instance-type TYPE \
    --mapper  s3://fnord/wordcount/wordSplitter.py \
    --input   s3://fnord/wordcount/input \
    --output  s3n://rzl/experiment/ \
    --reducer aggregate            # built-in reducer
```

Ausprobieren!

- Amazon EMR
  - Kreditkarte benötigt
  - `http://aws.amazon.com/elasticmapreduce/`
- Anagramm-Beispiel (Streaming)
  - `git clone git://git.iodev.org/mapreduce`
  - `README` lesen

Ende

Danke für eure Aufmerksamkeit.

Literatur

Jeffrey Dean, Sanjay Ghemawat
MapReduce: Simplified Data Processing on Large Clusters, 2004
http://research.google.com/archive/mapreduce.html