Überblick

MapReduce
- Einführung und Grundlagen
- Ablauf eines MapReduce-Jobs
- Aufgaben des Frameworks

Aufgabe 5
- Abstract Factory Entwurfsmuster
- Vergleichen und Sortieren mit Java
- Zusammenführung vorsortierter Listen
- Futures
- Daten finden und extrahieren

MapReduce: Einführung

- MapReduce: Programmiermodell zur Strukturierung von Programmen für **parallele**, **verteilte** Ausführung
- Map und Reduce ursprünglich Bausteine aus funktionalen Programmiersprachen (z. B. LISP)
  - **Map**: Abbildung eines Eingabeelements auf ein Ausgabeelement
  - **Reduce**: Zusammenfassung mehrerer gleichartiger Eingaben zu einer einzelnen Ausgabe
- Formulierung zu lösender Aufgabe in MapReduce
  - Aufteilen in (potentiell mehrere) Map- und Reduce-Schritte
  - Implementierung der Map- und Reduce-Methoden (Entwickler)
  - Parallelisierung und Verteilung (MapReduce-**Framework**)

MapReduce: Einführung

- „*MapReduce: Simplified data processing on large clusters*" (OSDI'04)
- Implementierung von Google nicht öffentlich
- Zahlreiche Open-Source-Implementierungen (z. B. Disco, **Apache Hadoop**, Phoenix)

  → Ermöglicht Verarbeitung riesiger Datenmengen

  → Vereinfachung der Anwendungsentwicklung

- Literatur

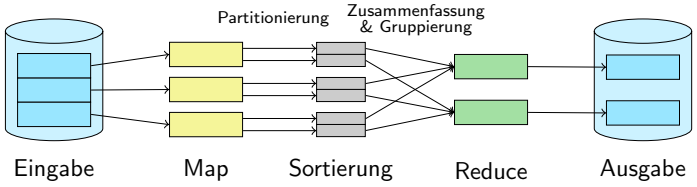
Jeffrey Dean and Sanjay Ghemawat
MapReduce: Simplified data processing on large clusters
Proceedings of the 6th Conference on Operating Systems Design and Implementation (OSDI '04), pages 137–150, 2004.

Hadoop-Framework (Komponenten)

Quelle der Illustration: https://blog.codecentric.de/2013/08/einfuhrung-in-hadoop-die-wichtigsten-komponenten-von-hadoop-teil-3-von-5/

Ablauf von MapReduce

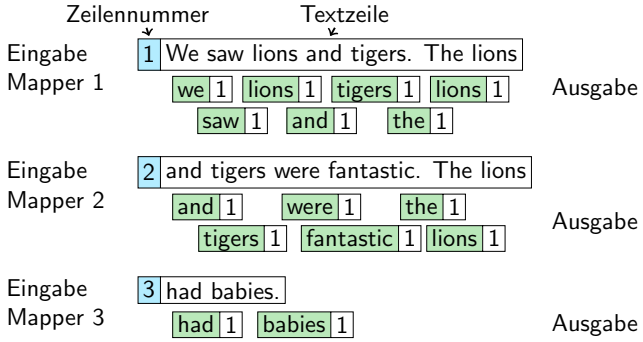
- Übersicht: Ablauf eines MapReduce-Durchlaufs



- Darstellung der Daten in Form von **Schlüssel-Wert-Paaren**

Map-Phase

- Abbildung in der Map-Phase
  - Parallele Verarbeitung verschiedener Teilbereiche der Eingabedaten
  - Eingabedaten in Form von Schlüssel-Wert-Paaren
  - Abbildung auf **variable Anzahl** von **neuen** Schlüssel-Wert-Paaren
- Beispiel: Zählen von Wörtern



Mapper-Schnittstelle

- Schnittstelle **Mapper** in Apache Hadoop

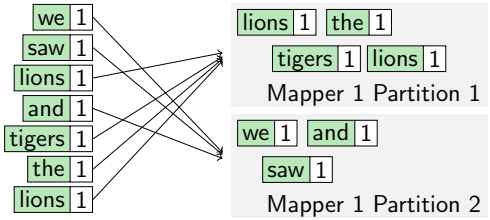
```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  void map(KEYIN key, VALUEIN value, Context context) {
    context.write((KEYOUT) key, (VALUEOUT) value);
  }
}
```

- Festlegen von Datentypen mittels „Generics“
- **key**: Schlüssel, z. B. Zeilennummer
- **value**: Wert, z. B. Inhalt der Zeile
- **context**: Ausführungskontext, enthält `write()`-Methode zur Ausgabe von Schlüssel-Wert-Paaren

Partitionierung

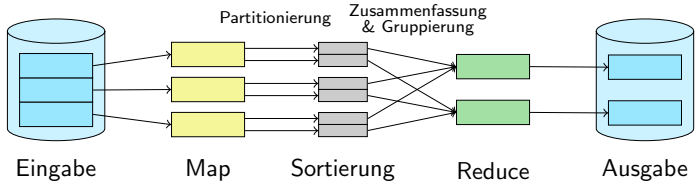
- Zuordnung der Mapper-Ausgabe zu späterem Reducer
  - Gleiche Schlüssel müssen zu gleichem Reducer
  - Eingaben der einzelnen Reducer sind unabhängig → parallelisierbar
- Schnittstelle **Partitioner** in Apache Hadoop

```
public class Partitioner<KEY, VALUE> {
  int getPartition(KEY key, VALUE value, int numPartitions) {
    return (key.hashCode() & Integer.MAX_VALUE) % numPartitions;
  }
}
```



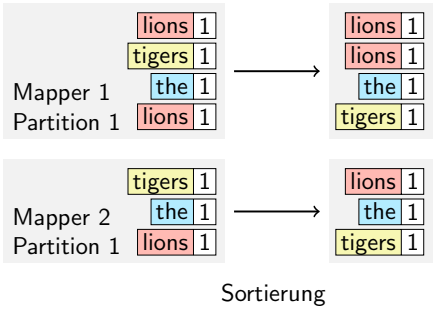
Ablauf von MapReduce

- Übersicht: Ablauf eines MapReduce-Durchlaufs



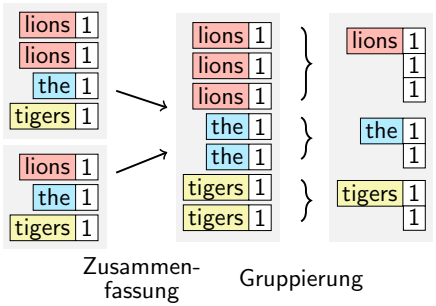
Sortierung

- Sortieren der Partitionen nach **Schlüssel**
  - Lokale Vorsortierung nach Verarbeitung der Daten durch Mapper
  - Jede Partition wird einzeln sortiert



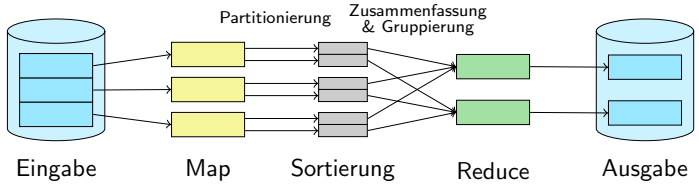
Zusammenfassung und Gruppierung

- Zusammenfassen und Gruppierung der Daten nach **Schlüssel**
  - Eingaben für Reducer befinden sich in (mehreren) Mapper-Ausgaben
  - **Zusammenfassung** der vorsortierten Partitionen zu einer vollständig sortierten Gesamtliste
  - **Gruppierung** aller Werte unter identischem Schlüssel
  - Statt Schlüssel-Wert-Paar nun Schlüssel und **Liste von Werten**



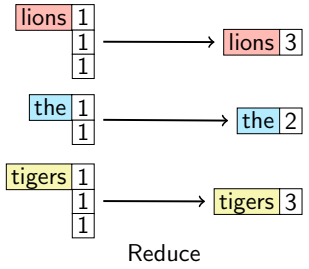
Ablauf von MapReduce

- Übersicht: Ablauf eines MapReduce-Durchlaufs



Reduce-Phase

- **Zusammenführen** von Daten in der Reduce-Phase
  - Eingabe in Form von Schlüssel und allen zugehörigen Werten aus Mapper
  - Parallele Verarbeitung verschiedener Teilbereiche von Schlüsseln
  - Abbildung auf **variable Anzahl** von **neuen Schlüssel-Wert-Paaren**



Reducer-Schnittstelle

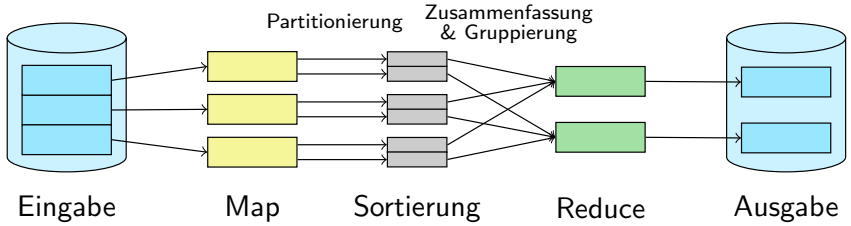
- Schnittstelle **Reducer** in Apache Hadoop:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  void reduce(KEYIN key, Iterable<VALUEIN> values, Context context) {
    for(VALUEIN value : values) {
      context.write((KEYOUT) key, (VALUEOUT) value);
    }
  }
}
```

- **key**: Schlüssel aus Sortierungsphase
- **values**: Liste von Werten, welche zu dem Schlüssel gruppiert wurden
- **context**: Ausführungskontext, enthält `write()`-Methode zur Ausgabe von Schlüssel-Wert-Paaren

Ablauf von MapReduce

- Übersicht: Ablauf eines MapReduce-Durchlaufs



Aufgaben des Frameworks

- Generelle **Steuerung** der MapReduce-Abläufe
  - Scheduling einzelner (Teil-)Aufgaben
  - Einhaltung der Reihenfolge bei Abhängigkeiten
  - Zwischenspeicherung der Daten
- Implementiert grundsätzliche **Algorithmen** (z. B. Sortierung)
- Bereitstellen von **Schnittstellen** zur Anpassung von
  - Dateneingabe (Deserialisierung)
  - Mapper
  - Partitionierung
  - Sortierung/Gruppierung
  - Reducer
  - Datenausgabe (Serialisierung)

Überblick

Framework-Entwicklung

- Framework stellt **Rahmen** für Anwendungen auf
  - Lediglich **grundsätzlicher Ablauf** vorgegeben
  - Details der Anwendung nicht vorab bekannt
    → Hohe Flexibilität und Konfigurierbarkeit notwendig
- Im Fall des MapReduce-Frameworks aus Aufgabe 5:
  - Deserialisierung
  - Mapper
  - Reducer
  - Sortierkriterium
- Auswählbare Implementierung für einzelne Schritte
  - Framework muss notwendige Objekte selbst instanziieren
  - Lösung mittels „Factory Pattern“

Factory Pattern

- Problemstellung: Es sollen Objekte instanziiert werden, welche eine bestimmte Schnittstelle zur Verfügung stellen, ohne dass der genaue Typ vorab bekannt ist.
  → **Kapselung der Instanziierung** in eigener Klasse
- Beispiel:

```
public class WordCountMapper implements Mapper {
  ...
}

public class WordCountFactory {
  public Mapper createMapper() {
    return new WordCountMapper();
  }
}
```

- **Allerdings**: Klasse `WordCountFactory` muss Framework bekannt sein

Abstract Factory Pattern

- Lösung durch weitere Abstraktionsschicht: Schnittstelle zur Instanziierung

```
public class WordCountMapper implements Mapper { ... }

public interface MapperFactory {
  public Mapper createMapper();
}

public class WordCountFactory implements MapperFactory {
  public Mapper createMapper() {
    return new WordCountMapper();
  }
}
```

- Verwendung:

```
void myMethod(MapperFactory mfact) {
  Mapper m = mfact.createMapper();
  ...
}
```

Sortieren mittels Comparator-Objekten

- Standardisierte Schnittstellen zum Vergleich von Objekten:

Comparable

- Vergleicht Objekt mit **anderem gegebenen Objekt**

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

Comparator

- Vergleicht **zwei gegebene Objekte miteinander**
- **Invariante:**
  `Object.equals(o1, o2) == true` ⇔ `Comparator.compare(o1, o2) == 0`

```
public abstract class Comparator<T> {
  public int compare(T o1, T o2);
}
```

Sortieren mittels Comparator-Objekten

- Verwendung:

```
int x = links.compareTo(rechts);
int y = comparator.compare(links, rechts);
```

- Methoden `compareTo()` und `compare()` liefern Integer zurück
  - **negativ**: Linker Wert **kleiner** als rechter Wert (kommt **vor**...)
  - **0**: Beide Werte sind **gleich** (**äquivalent**)
  - **positiv**: Linker Wert **größer** als rechter Wert (kommt **nach**...)

- Beispiel: Strings rückwärts sortieren

```
class RevStringComparator implements Comparator<String> {
  public int compare(String o1, String o2) {
    return -o1.compareTo(o2);
  }
}
```

Sortieren mittels Comparator-Objekten

- Comparator ermöglicht Änderung der Sortierreihenfolge
  ohne Ableiten der zu sortierenden Objekte

- Einstellung bei sortierenden Standard-Containern in Java

  Beispiel: `TreeMap` (implementiert `SortedMap`)

```
RevStringComparator revcmp = new RevStringComparator();
TreeMap<String,X> treemap = new TreeMap<String,X>(revcmp);
```

→ Iterieren über Map liefert Schlüssel in umgekehrter Reihenfolge

Zusammenführung mittels Priority-Queues

- Aufgabe: Zusammenführen bereits vorsortierter Listen
  - Vergleich des obersten Elements über alle Listen
  - Kleinstes Element bestimmt nächstes Ausgabeelement

- Datenstruktur **Priority-Queue**
  - Einfügen von Elementen mit zugeordneter Priorität
  - Entfernen entnimmt immer Element mit **höchster** Priorität
  - Üblicherweise als Heap-Datenstruktur implementiert

- Nutzung als Merge-Algorithmus
  - Priorität entspricht Wertigkeit des **obersten Elements** jeder **Liste**
  - Entnahme aus Priority-Queue liefert Liste mit nächstem Element

Zusammenführung mittels Priority-Queues

- Algorithmus:

1. Priority-Queue mit vorsortierten Listen befüllen
2. Entnahme des Elements höchster Priorität liefert Liste, welche das nächste auszugebende Listenelement an erster Stelle enthält
3. Ausgeben und Entfernen des obersten Listenelements aus der entnommenen Liste
4. Liste wieder in Priority-Queue einfügen
5. Wiederholen ab (2), bis alle Listen leer sind

Zusammenführung mittels Priority-Queues

- Priority-Queues in Java: `java.util.PriorityQueue`
  - **Höchste Priorität** entspricht **erster Stelle** nach Sortierung
  - Festlegen der Sortierung mittels `Comparator`:

```
public PriorityQueue(int capacity, Comparator c);
```

  - Einfügen eines Elements vom Typ `E`:

```
public boolean add(E item);
```

  - Abfrage des obersten Elements:

```
public E peek();
```

  - Entnahme des obersten Elements:

```
public E poll();
```

Futures

- Allgemeine Schnittstelle

```
boolean isDone();
<beliebiger Datentyp> get();
```

- Funktionsweise

1. Beim asynchronen Aufruf wird (statt dem eigentlichen Ergebnis) sofort ein Future-Objekt zurückgegeben
2. Das Future-Objekt lässt sich befragen, ob der tatsächliche Rückgabewert der Operation bereits vorliegt bzw. ob die Operation beendet ist → `isDone()`
3. Ein Aufruf von `get()`
   - liefert das Ergebnis der Operation sofort zurück, sofern es zu diesem Zeitpunkt bereits vorliegt **oder**
   - blockiert solange, bis das Ergebnis eingetroffen ist

Futures in Java

`java.util.concurrent.Future`

- Schnittstelle **Future**

```
public interface Future<V> {
  public V get()  throws InterruptedException, ExecutionException;
  public V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;

  public boolean isDone();

  public boolean cancel(boolean mayInterruptIfRunning);
  public boolean isCancelled();
}
```

- Umfang
  - Methoden der allgemeinen Future-Schnittstelle
  - Zusätzliche Methoden zum Abbrechen von Tasks
  - `get()` wirft ggf. von Operation geworfene `Exception`
    - Verpackt als `ExecutionException`
    - Zugriff auf ursprüngliche Exception:

```
excecutionException.getCause()
```

Futures in Java

java.util.concurrent.ExecutorService

- Interface `ExecutorService`
  - Erlaubt asynchrone Ausführung von Tasks
  - Task bei Executor-Service „abgeben“, Ergebnis per Future:

```
<T> Future<T> submit(Callable<T> task)
```

  - Auch für `Runnable` möglich:

```
Future<?> submit(Runnable task)
```

- Interface `Callable`

```
public interface Callable<V> {
  V call() throws Exception;
}
```

- Interface `Runnable`

```
public interface Runnable {
  void run();
}
```

  - Ein Future gibt für ein bearbeitetes Runnable immer `null` zurück

Futures in Java

java.util.concurrent.ExecutorService

- Bereitstellung von `ExecutorService`-Implementierungen
  - Ausführung in einem einzigen Thread

```
public static ExecutorService newSingleThreadExecutor();
```

  - Konstante Thread-Anzahl

```
public static ExecutorService newFixedThreadPool(int nThreads);
```

  - ...
- `ExecutorService` nach Verwendung wieder beenden:

```
public void shutdown();
```

Futures in Java – Anwendungsbeispiel

java.util.concurrent.ExecutorService

- Beispielklasse

```
public class FutureExample implements Callable<Integer> {
  private int a;

  public FutureExample(int a) {
    this.a = a;
  }
  public Integer call() throws Exception {
    return a * a;
  }
}
```

- Aufruf

```
ExecutorService es = Executors.newSingleThreadExecutor();
FutureExample task = new FutureExample(4);
Future<Integer> f = es.submit(task);
[...]
try {
  System.out.println("result: " + f.get());
} catch (InterruptedException | ExecutionException e) {
  // Fehlerbehandlung
}
```

Extrahieren von Daten

- Typische MapReduce-Anwendung: Extrahieren von Daten
  - Statistiken, Data Mining
  - Mustererkennung, Machine Learning
  - Graph-Algorithmen
- Eingabedaten häufig in Form von Textzeilen
- Partitionierung von Eingabedaten problematisch:

  Zusammengehörige Daten können in **unterschiedlichen** Worker-Threads verarbeitet werden
- Lösungsmöglichkeiten:
  - Beeinflussung der Partitionierung durch Eingabedaten
  - Verwerfen unvollständiger Datensätze, z. B. bei statistischen Auswertungen großer Datenmengen

Auffinden und Extrahieren von Zeichenketten

- Einfache Methoden in `Java.lang.String`
- Finden konstanter Zeichenketten
  - Vorwärts suchen ab bestimmter Position:

```
public int indexOf(String str, int start);
```

  - Rückwärts suchen ab bestimmter Position:

```
public int lastIndexOf(String str, int start);
```

- Teilstrings extrahieren:

```
public String substring(int start, int end);
```

  - Ausgabe des Strings ab `start` bis `end`, **ohne** `end` selbst
  - Tipp: Zum Testen ein Zeichen vor und nach dem gesuchten Teilbereich ausgeben lassen