Überblick

MapReduce Framework
- MapReduce
  - Einführung und Grundlagen
  - Ablauf eines MapReduce-Jobs
  - Aufgaben des Frameworks
- Aufgabe 3
  - Abstract Factory Entwurfsmuster
  - Vergleichen und Sortieren mit Java
  - Zusammenführung vorsortierter Listen
  - Futures
  - Daten finden und extrahieren

MapReduce Einführung

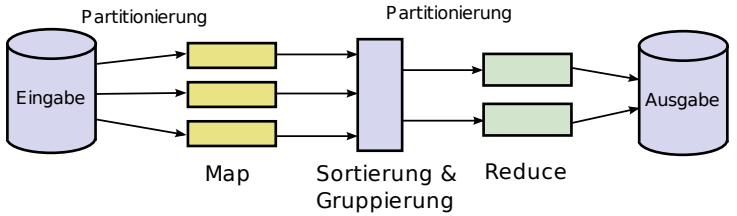
- MapReduce ist Modell zur **Strukturierung** von Programmen für **parallele**, **verteilte** Ausführung
- **Map** und **Reduce** ursprünglich Bausteine aus **funktionalen** Programmiersprachen (z.B. LISP).
  - **map**: Abbildung eines Eingabeelements auf ein Ausgabeelement
  - **reduce**: Zusammenfassung mehrerer gleichartiger Eingaben zu einer Ausgabe
- Programmierer konzentriert sich auf zu lösende **Aufgabe**:
  - Implementiert lediglich Map- und Reduce-Funktionen
  - MapReduce-**Framework** übernimmt Parallelisierung und Verteilung

MapReduce Einführung

- Popularität als Modell zur Parallelisierung durch Beitrag von Google zur OSDI 2004
- Implementierung von Google nicht öffentlich
- Zahlreiche Open-Source-Projekte:
  - Apache Hadoop
  - Twister
  - Phoenix
  - Disco
- → Ermöglicht Verarbeitung **riesiger** Datenmengen
- → **Vereinfachung** der Anwendungsentwicklung

Ablauf von MapReduce

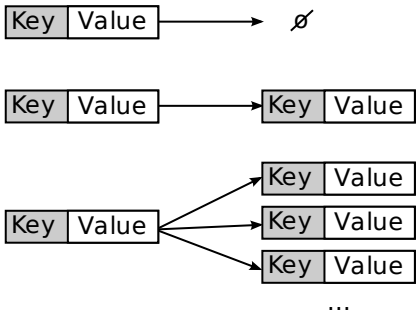
- Übersicht über den Ablauf eines MapReduce-Durchlaufs:



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

Map-Phase

- Abbildung in der Map-Phase
  - Parallele Verarbeitung verschiedener **Teilbereichen der Eingabedaten**
  - Eingabedaten in Form von **Schlüssel/Wert**-Paaren
  - Abbildung auf **variable** Anzahl von **anderen** Schlüssel/Wert-Paaren



Mapper-Schnittstelle

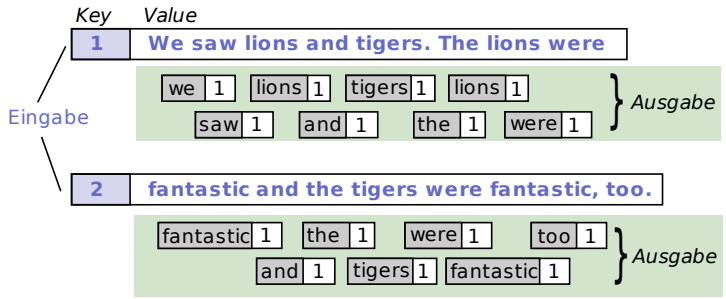
- Schnittstelle **Mapper** in Apache Hadoop:

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

- **key**: Schlüssel, z.B. Zeilennummer
- **value**: Wert, z.B. Inhalt der Zeile
- **context**: Ausführungskontext, enthält `write()`-Methode zur **Ausgabe** von Schlüssel/Wert-Paaren

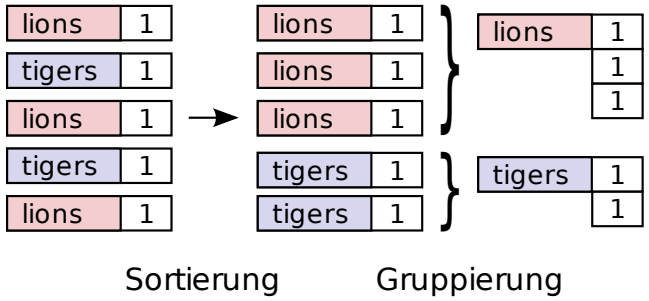
Mapper Beispiel

- **Beispiel**: Zählen von Wörtern
  - Eingabe: **Schlüssel**: Zeilennummer – **Wert**: Textzeile
  - Ausgaben: **Schlüssel**: Wort – **Wert**: Anzahl (hier: 1 für jedes Wort)



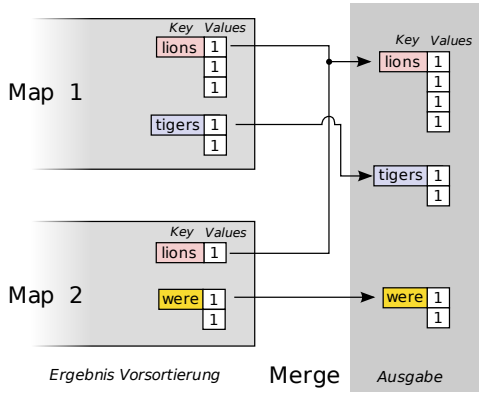
Sortierung und Gruppierung

- Sortierung und Gruppierung nach **Schlüssel**
  - Zusammenfassen **aller Werte** unter einem Schlüssel
  - Statt Schlüssel/Wert-Paar nun Schlüssel und **Liste von Werten**
  - **Vorsortierung** nach Verarbeitung der Daten durch Map
    → **Parallelität** nutzen



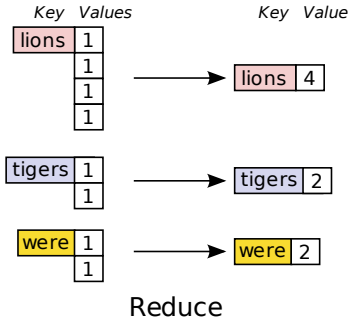
Sortierung und Gruppierung

- Zusammenfassung der sortierten **Teillisten** zu einer vollständig sortierten und gruppierten **Gesamtliste** (Merge)



Reduce-Phase

- **Zusammenführen** von Daten in der Reduce-Phase
  - Parallele Verarbeitung verschiedener **Teilbereiche der Werte**
  - Eingabedaten in Form von **Schlüssel** und **Liste von Werten**
  - Abbildung auf **variable** Anzahl von **anderen** 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

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** und **Bibliotheken** zur Anpassung von
  - Partitionierung
  - Dateneingabe (Deserialisierung)
  - Mapper
  - Sortierung/Gruppierung
  - Reducer
  - Datenausgabe (Serialisierung)

Aufgaben des Frameworks

- Schnittstelle zur **Kombination** der angepassten Komponenten zu **Lösungen** für bestimmte Aufgaben
  - Einfache **Wiederverwendung**
  - Abruf unter **gemeinsamer Bezeichnung**
  - Instanziierung von Einzelkomponenten typischerweise über **Factory**
- Verwaltet **Kontext** zum **Verbinden** beteiligter Objekte einzelner Schritte:
  - **Mapper-Kontext**: Dateneingabe, Mapper, Sortierung/Gruppierung
  - **Reducer-Kontext**: Einlesen Zwischenergebnisse, Reducer, Datenausgabe

Framework-Entwicklung

- Framework stellt **Rahmen** für Anwendungen auf
  - Lediglich **grundsätzlicher Ablauf** vorgegeben
  - Viele Details der Anwendung **nicht vorab bekannt**
    → Hohe **Flexibilität** und **Konfigurierbarkeit** notwendig
- Im Fall des MapReduce Frameworks aus Aufgabe 3:
  - Zu verwendenede Mapper, Reducer und Sortierung
- Hauptschwierigkeit: Auswahl der zu verwendenden Implementierungen

Factory Pattern

- **Problemstellung**: Es sollen Objekte **instanziiert** werden, welche eine **bestimmte Schnittstelle** zur verfügen 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**: `WordCountFactory` muss bekannt sein

Abstract Factory Pattern

- Lösung durch weitere **Abstraktionsschicht**

```
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**
- `equals()` vergleicht Äquivalenz verschiedener `Comparator`-Typen

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

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) {
    Collator strcoll = Collator.getInstance();
    return -strcoll.compare(o1, o2);
  }
}
```

Sortieren mittels Comparator-Objekten

- **Comparator** ermöglicht Änderung der Sortierreihenfolge **ohne Ableitung der zu sortierenden Objekte**

- Kann bei sortierenden Standard-Containern von Java angegeben werden

Beispiel: `TreeMap` (implementiert `SortedMap`)

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

- Iterieren über Map liefert Schlüssel in umgekehrter Reihenfolge

Zusammenführung mittels Priority-Queues

- **Problemstellung**: 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 allen **Listen befüllen**
2. **Entnahme** des Elements höchster Priorität ergibt **Liste**, welche das nächste auszugebende **Listenelement** enthält
3. **Ausgabe** und **Entfernen** des obersten Listenelements der eben entnommenen Liste
4. **Liste** mit neuem oberstem Element (=geänderte Priorität) erneut 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

- Oftmals synonym verwendet: *Promise*
- Schnittstelle

```
boolean poll();
<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
   → `poll()`
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`

- Umfang
  - Methoden der allgemeinen Future-Schnittstelle
  - Zusätzliche Methoden zum Abbrechen von Tasks
- Schnittstelle

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

  public boolean isDone();   // --> poll()

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

Futures in Java — java.util.concurrent.ExecutorService

Anwendungsbeispiel Executor-Service

- Interface `ExecutorService`
  - Erlaubt asynchrone Ausführung von Tasks
  - Task bei Executor-Service „abgeben", Ergebnis per Future
  - Zentrale Methode

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

- Interface `Callable`
  - Schnittstelle

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

  - Unterschiede zu `Runnable`
    - Rückgabewert
    - Exception

Futures in Java – Anwendungsbeispiel ExecutorService

Klasse `java.util.concurrent.Executors`

- Überblick
  - Hilfsmethoden zur Erzeugung von `Callable`-Objekten
  - Bereitstellung von `ExecutorService`-Implementierungen
- Wichtige Factory-Methoden für `ExecutorServices`
  - 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 ExecutorService

- Beispielklasse

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

  public FutureExample(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public Integer call() throws Exception {
    return a * b;
  }
}
```

- Aufruf

```
ExecutorService es = Executors.newSingleThreadExecutor();
FutureExample task = new FutureExample(4, 7);
Future<Integer> f = es.submit(task);
[...]
System.out.println("result: " + f.get());
```

Extrahieren von Daten

- **Extrahieren** von Daten typische MapReduce-Anwendung
  - 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 von Zeichenketten

- Einfache Methoden in `Java.lang.String`
- Finden **fester** 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);
```

- Operationen mit **regulären Ausdrücken**:
  - **Test**, ob regulärer Ausdruck anwendbar:

```
public boolean matches(String regex);
```

  - **Aufteilen in Array** anhand regulärem Ausdruck:

```
public String[] split(String regex, int limit);
```

Reguläre Ausdrücke in Java

- **Reguläre Ausdrücke** mit `java.util.regex.Pattern`
- Nützliche Teilausdrücke
  - Beliebiges Zeichen: `.`
  - Anfang des Strings: `^`
  - String-Ende: `$`
  - Wiederholung: `*` → beliebig oft, `+` → mindestens einmal
  - Zeichenauswahl: `[abc]` → a, b oder c
  - Zeichenklassen: `\s` Leerzeichen, `\d` Ziffern
- **Beispiele:**

```
^Hallo    // Hallo am Anfang des Strings
welt.$    // welt gefolgt von beliebigem Zeichen am Stringende
te[sx]+t  // te, mindestens einmal s oder x, t
```

Reguläre Ausdrücke in Java

- Vorkompilieren häufig benötigter Ausdrücke:

```
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(str);
```

- **Test**, ob Ausdruck passt:

```
public boolean matches();
```

- **Postion abfragen**, wo ein Treffer gefunden wurde:

```
public int start();
```

- **Beispiel**:

```
if (m.matches()) {
  int pos = m.start();
  ...
}
```

Teilstrings extrahieren

- Bei bekanntem Start- und End-**Index**:

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

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

```
String[] parts = input.split(regex, 2);

if (parts.length() < 2)
  System.err.println("Not found");

String left = parts[0], right = parts[1];
```

Weitere Informationen

- Google MapReduce
  `http://labs.google.com/papers/mapreduce.html`
- Apache Hadoop
  `http://hadoop.apache.org/`
- Hadoop SVN repository
  `http://svn.apache.org/viewvc/hadoop/common/`