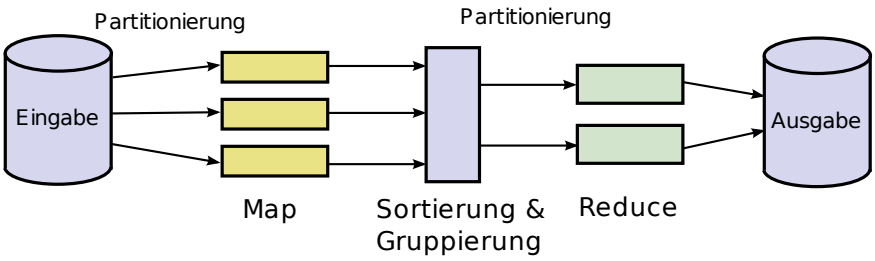
3 Übungsaufgabe #3: MapReduce-Framework

In dieser Aufgabe soll ein Framework für das *MapReduce*-Programmiermodell implementiert werden, das es erlaubt, beliebige MapReduce-Jobs mehrfädig auf dem lokalen Rechner auszuführen. Als generelles Vorbild dient dabei die MapReduce-Implementierung von *Hadoop*, die an einigen Stellen jedoch vereinfacht umgesetzt wird. Abschließend ist unter Verwendung des Frameworks eine Reihe von einfachen Anwendungen zu realisieren.

3.1 Implementierung des MapReduce-Frameworks (für alle)

Die Grundidee von MapReduce besteht darin, parallele Anwendungen durch Implementierung der beiden Methoden `map()` und `reduce()` umsetzen zu können; alle anderen Aufgaben, wie zum Beispiel das Ausführen von Worker-Threads oder die Verteilung der Daten, übernimmt das Framework.



Die Bearbeitung eines MapReduce-Jobs erfolgt in mehreren Phasen: Zunächst werden die Eingabedaten aus einer Datei gelesen und gleichmäßig auf die Mapper-Worker-Threads verteilt. Anschließend erfolgt in der Map-Phase die Vorverarbeitung der Daten. Die hierbei erzeugten Zwischenergebnisse werden von jedem Mapper in sortierter Form in einer separaten Datei abgelegt. Als nächstes erfolgt die Zusammenführung der Zwischenergebnisse in einer einzelnen Datei, die Sortierung bleibt dabei erhalten. Analog zur Map-Phase, erfolgt in der Reduce-Phase eine Aufteilung der Zwischenergebnisse auf die Reducer. Abschließend sichert jeder Reducer die von ihm berechneten Endergebnisse in einer eigenen Datei, eine Zusammenführung findet in diesem Fall nicht mehr statt.

3.1.1 Kontexte für Mapper und Reducer

Die Bereitstellung der zu verarbeitenden Daten erfolgt für Mapper und Reducer unter Zuhilfenahme eines Kontext-Objekts der Klasse `MWContext`. Da MapReduce vorsieht, dass mehrere Werte mit dem selben Schlüssel assoziiert werden können, stellt `MWContext` die Daten über eine Schnittstelle `MWKeyValuesIterator` zur Verfügung, die es ermöglicht, über Schlüssel-Wertemengen-Paare zu iterieren:

```
public interface MWKeyValuesIterator<KEY, VALUE> {
    public boolean nextKeyValues();
    public KEY getCurrentKey();
    public Iterable<VALUE> getCurrentValues();
}
```

Die Methode `nextKeyValues()` setzt den Zeiger des Iterators auf den nächsten Datensatz und gibt solange `true` zurück, bis das Ende der einzulesenden Daten erreicht ist. Ein Aufruf von `getCurrentKey()` liefert den aktuellen Schlüssel, `getCurrentValues()` die dazugehörige Menge von Werten zurück.

Beim Einsatz von mehreren Worker-Threads pro Phase, muss jeder Mapper bzw. Reducer nur den für ihn bestimmten Teil der zu verarbeitenden Daten einlesen. Im Konstruktor soll sich daher angeben lassen, bei welchem Byte-Index in der Eingabedatei die relevanten Daten beginnen und wie viele Bytes sie umfassen.

```
public abstract class MWContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
implements MWKeyValuesIterator<KEYIN, VALUEIN>
{
    public MWContext(String inFile, long startIndex, long length);
    [...] // Methoden der MWKeyValuesIterator-Schnittstelle
    public abstract void write(KEYOUT key, VALUEOUT value) throws IOException;
    public abstract void outputComplete() throws IOException;
}
```

Neben der Bereitstellung der zu verarbeitenden Daten ist der Kontext auch für die Aufnahme der Resultate der jeweiligen Phase verantwortlich. Hierzu werden ihm die vom Worker berechneten Ergebnisse in Form von Schlüssel-Wert-Paaren per `write()`-Methode übergeben. Durch einen Aufruf von `outputComplete()` wird der Kontext darüber informiert, dass die Phase abgeschlossen ist und daher keine weiteren Ergebnisse mehr folgen.

Da für die Ausgabe der (Zwischen-)Ergebnisse für Mapper und Reducer unterschiedliche Anforderungen gelten, ist dieser Teil jeweils in einer Unterklasse von `MWContext` zu realisieren. Der Mapper-Kontext (`MWMapContext`) muss die beiden Methoden `write()` und `outputComplete()` dabei so implementieren, dass die Zwischenergebnisse am Ende der Map-Phase nach Schlüssel sortiert ausgegeben werden; das Sortierkriterium ist dem Mapper-Kontext im Konstruktor durch einen `java.util.Comparator`-Parameter bekannt zu geben. Da die Ergebnisse vom Reduce-Kontext (`MWReduceContext`) nicht nochmals sortiert werden, kann ihre Ausgabe dort direkt erfolgen.

Aufgaben:

→ Implementierung der abstrakten Basisklasse `MWContext` in einem Subpackage `mw.mapreduce.core`
→ Implementierung der Klassen `MWMapContext` und `MWReduceContext`

Hinweise:

- Zum wahlfreien Einlesen von Dateien steht im Pub-Verzeichnis die Klasse `MWTextFileReader` zur Verfügung. Diese stellt sicher, dass das Einlesen eines Datenblocks auch dann mit einer ganzen Zeile beginnt bzw. endet, wenn der übergebene Start-Index bzw. das Ende des Datenblocks in der Mitte einer Zeile liegen.
- Zur Vereinfachung darf in `MWContext` angenommen werden, dass Schlüssel und Werte immer `String`s sind.
- Enthalten die eingelesenen Datensätze keine Schlüssel, soll `getCurrentKey()` die Zeilennummer zurückgeben.

3.1.2 Mapper und Reducer

Unter Zuhilfenahme der in Teilaufgabe 3.1.1 entwickelten Kontexte soll nun in `mw.mapreduce.core` jeweils eine Basisklasse für Mapper (`MWMapper`) und Reducer (`MWReducer`) implementiert werden. Die beiden Klassen sind dabei so aufzubauen, dass Mapper und Reducer für konkrete Anwendungen nur die `map()`- bzw. die `reduce()`-Methode überschreiben müssen. Die Basisklassen sollen für diese beiden Methoden jeweils die Default-Implementierung (siehe Tafelübung) bereitstellen, welche die Schlüssel-Wert-Paare unverändert durchreicht.

```
public class MWMapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> implements Runnable {
    public void setContext(MWMapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> context);
    public void run();
    protected void map(KEYIN key, VALUEIN value,
                       MWContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> context);
}

public class MWReducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> implements Runnable {
    public void setContext(MWReduceContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> context);
    public void run();
    protected void reduce(KEYIN key, Iterable<VALUEIN> values,
                          MWContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> context);
}
```

Mit der Methode `setContext()` wird dem Mapper bzw. Reducer der ihm zugeordnete Kontext übergeben. Aufgabe der `run`-Methode ist es jeweils, die dem Worker per Kontext zugewiesenen Daten unter Einbeziehung der `map()`- bzw. der `reduce()`-Methode zu verarbeiten.

Aufgaben:

→ Implementierung einer Mapper-Basisklasse `MWMapper`
→ Implementierung einer Reducer-Basisklasse `MWReducer`

3.1.3 Fertigstellung des Frameworks

Abschließend kann nun das eigentliche Framework zur Ausführung von MapReduce-Jobs implementiert werden, das für ein schrittweises Durchlaufen der Verarbeitungsphasen sorgt. Um dabei sicherzustellen, dass die verwendeten Mapper und Reducer zur selben Anwendung gehören, soll eine Factory `MWJob` für die Erzeugung der Worker zum Einsatz kommen; Anzahl und Typen der Parameter der beiden `create()`-Methoden sind freigestellt. Des Weiteren legt `MWJob` über eine Methode `getComparator()` das Sortierkriterium für die Anwendung fest.

```
public interface MWJob {
    public MWMapper createMapper([...]);
    public MWReducer createReducer([...]);
    public java.util.Comparator getComparator();
}
```

In Fällen, in denen mehr als ein Mapper zum Einsatz kommt, ist in MapReduce vor Beginn der Reduce-Phase eine Zusammenführung der Zwischenergebnisse in einer einzelnen Datei vorgesehen, die pro Schlüssel nur noch einen Datensatz enthält. Da jeder Mapper die von ihm berechneten Zwischenergebnisse nach Schlüssel sortiert in einer eigenen Datei ablegt, ist es aus Effizienzgründen nicht sinnvoll, bei der Vereinigung der Zwischenergebnisse eine globale Sortierfunktion zu verwenden, die nochmals alle Datensätze sortiert. Stattdessen soll ein effizienterer Weg gewählt werden, der den Vorteil ausnutzt, dass die zusammenzuführenden Teile bereits vorsortiert sind.

Das MapReduce-Framework soll sich von der Kommandozeile wie folgt aufrufen lassen:

```
java -cp <classpath> mw.mapreduce.MWMapReduce <app> <infile> <tmpprefix> <outprefix>
```

Der Parameter `app` bestimmt dabei die auszuführende Applikation (siehe Teilaufgabe 3.2), `infile` enthält den Pfad auf die Datei mit den Eingabedaten. Da bei Verwendung mehrerer Mapper bzw. Reducer mehrere Dateien mit Zwischen- bzw. Endergebnissen erzeugt werden, ist für diese jeweils nur ein Pfad-Präfix (`tmpprefix` bzw. `outprefix`) anzugeben; die einzelnen Dateinamen sollen zur Unterscheidung eine Indexnummer enthalten.

Aufgabe:

→ Implementierung der Klasse `MWMapReduce` im Subpackage `mw.mapreduce`

Hinweise:

- Die Anzahl der zu verwendenden Mapper und Reducer soll unabhängig voneinander konfigurierbar sein.
- Bei der Aufteilung der zu verarbeitenden Daten auf Mapper bzw. Reducer ist jeweils darauf zu achten, dass jeder Worker in etwa die gleiche Menge an Daten erhält.
- Das Starten der Mapper- bzw. Reducer-Threads soll ein *Executor-Service* übernehmen.

3.2 Implementierung von MapReduce-Anwendungen

In dieser Teilaufgabe soll nun eine Reihe von typischen Anwendungen für das MapReduce-Framework implementiert werden. Wie in MapReduce üblich, erfolgt dies durch Überschreiben der `map()`- bzw. `reduce()`-Methode in einer anwendungsspezifischen Mapper- bzw. Reducer-Unterklasse; sollte die Default-Implementierung eines Workers bereits ausreichen, ist diese zu verwenden. Hinzu kommt für jede Anwendung die Bereitstellung einer zugehörigen `MWJob`-Factory. Alle für eine Anwendung implementierten Klassen sind jeweils in einem Subpackage `mw.mapreduce.jobs.<Anwendung>` zusammenzufassen. Bei der Implementierung der Anwendungen ist darauf zu achten, dass die im Pub-Verzeichnis zur Verfügung gestellten Eingabedaten nicht verändert werden dürfen.

3.2.1 Sortieren von Textdateien (für alle)

Eine der naheliegendsten mit Hilfe von MapReduce zu lösenden Aufgaben ist die Sortierung von Textdateien, da sich bei einer Implementierung dieser Anwendung ausnutzen lässt, dass das MapReduce-Framework im Anschluss an die Map-Phase automatisch eine Sortierung der Zwischenergebnisse durchführt. Als Eingabe soll hierbei die Datei `/proj/i4mw/pub/aufgabe3/num_friends.list` dienen, in der für jeden beim Facebook-Service aus Aufgabe 1 registrierten Nutzer die Größe seines Freundeskreises aufgelistet ist (→ „`friendsort`“). Es gilt dabei zu beachten, dass die Eingabedaten im Format „`<ID>\t<#Freunde>`“ vorliegen, die Ausgabe jedoch im Format „`<#Freunde>\t<ID>`“ erfolgen soll. Neben den anwendungsspezifischen Mapper- und Reducer-Klassen ist außerdem ein `Comparator` zu implementieren, der dafür sorgt, dass die Ergebnisse **absteigend** nach Anzahl der Freunde sortiert werden.

Aufgabe:

→ Implementierung der `friendsort`-Anwendung (inklusive der Factory-Klasse `MWFriendSortJob`)

3.2.2 Extrahieren von Daten aus Dokumenten (optional für 5,0 ECTS)

Ein häufiger Anwendungsfall für MapReduce in der Praxis ist das Extrahieren von Daten aus Dokumenten, typischerweise Web-Seiten. Hierbei werden die Rohdaten (HTML-Code) zeilenweise eingelesen und anschließend die relevanten Informationen mit Hilfe bekannter Muster identifiziert und weiterverarbeitet. Ziel dieser Teilaufgabe ist es, eine MapReduce-Anwendung `friendextract` zu entwickeln, welche die Freundschaftsbeziehungen zwischen Nutzern des Übungs-Facebook-Service aus deren Profil-Webseiten ermittelt. Die Ausgabe soll dabei jeweils in Form zweier Schlüssel-Wert-Paare erfolgen („`<ID_A>\t<ID_B>`“ und „`<ID_B>\t<ID_A>`“). Ein Abzug der zu verarbeitenden Nutzerprofile ist unter `/proj/i4mw/pub/aufgabe3/data.dump` erhältlich.

Bei der Implementierung muss berücksichtigt werden, dass das Aufteilen der Eingabedaten auf mehrere Mapper zu Situationen führen kann, in denen die Grenze eines Datenblocks innerhalb eines Profils verläuft. Um in einem solchen Fall den Verlust von Daten zu verhindern, muss sichergestellt werden, dass ein Mapper ein angefangenes Profil auch dann vollständig verarbeitet, wenn die ihm zugewiesene Blockgrenze zwischenzeitlich erreicht wurde.

Aufgabe:

→ Implementierung der `friendextract`-Anwendung (inklusive der Factory-Klasse `MWFriendExtractJob`)

Hinweise:

- Es bietet sich an, das Erzwingen der vollständigen Verarbeitung der Nutzerprofile in einer anwendungsspezifischen Unterklasse von `MWMapContext` zu implementieren.
- Die Hilfsklasse `MWTextFileReader` (siehe Teilaufgabe 3.1.1) stellt für das Einlesen von Zeilen, die nach dem Blockende liegen, die Methode `forceReadLine()` zur Verfügung.
- Im Pub-Verzeichnis liegt eine Datei `data.readme` mit näheren Informationen darüber bereit, in welcher Form Nutzer-IDs und Freundschaftsbeziehungen im HTML-Code repräsentiert sind. Außerdem befindet sich dort zum Testen eine Eingabedatei `data-small.dump` mit einer geringeren Anzahl an Profilen.

3.2.3 Zusammenführen von Informationen (für alle)

MapReduce dient nicht nur der Aufbereitung von Rohdaten, sondern kann auch dazu eingesetzt werden, mit MapReduce-Anwendungen erzeugte Daten weiter zu verarbeiten. Die Eingabedatei dieser Teilaufgabe (`/proj/i4mw/pub/aufgabe3/friends.list`) besteht aus den Ergebnissen der Teilaufgabe 3.2.2, an die eine Liste der (Klar-)Namen von Nutzern angehängt wurde. Die Namensdatensätze weisen dabei folgendes Format auf: „`<ID>\t|<Name>`“. Das Trennzeichen „|“ („Pipe“) dient in diesem Zusammenhang als Kennzeichnung dafür, dass es sich bei diesem Schlüssel-Wert-Paar um einen Namensdatensatz handelt.

Ziel dieser Teilaufgabe ist es, eine MapReduce-Anwendung `friendcount` zu implementieren, die für jeden Nutzer die Anzahl seiner Freunde ermittelt, und das Resultat im Format „`<Name>\t<#Freunde>`“ ausgibt; eine Sortierung der Ergebnisse nach Namen ist hierbei nicht erforderlich.

Aufgaben:

→ Implementierung der `friendcount`-Anwendung (inklusive der Factory-Klasse `MWFriendCountJob`)

Hinweise:

- In der Ausgabe soll das Trennzeichen „|“ nicht mehr vorkommen.
- Da die in `data.dump` bereitgestellten Nutzerprofile nur einen geringen Teil des sozialen Netzwerks abdecken, weichen die hier berechneten Ergebnisse stark von den Zahlen aus Teilaufgabe 3.2.1 ab.

Abgabe: am 16.12.2010