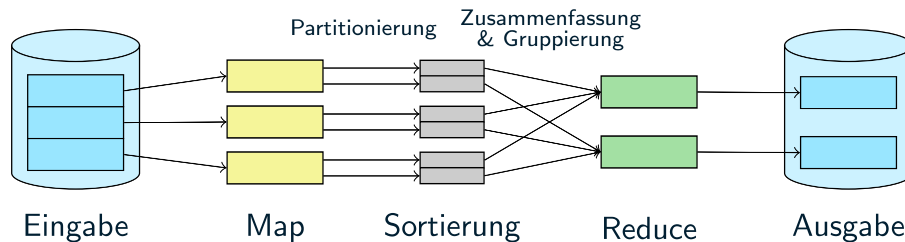
4 Übungsaufgabe #4: MapReduce-Framework

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 *Apache Hadoop*, die an einigen Stellen jedoch vereinfacht oder abweichend umgesetzt wird. Abschließend ist unter Verwendung des Frameworks eine Reihe von einfachen Anwendungen zu realisieren.

4.1 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, wobei durch die Verteilung auf mehrere Map- und Reduce-Tasks möglichst viel Parallelität erreicht werden soll. Dazu wird die Eingabedatei zu gleich großen Abschnitten auf die Map-Tasks verteilt und die Map-Phase beginnt. Jeder Map-Task liest unabhängig seinen zugewiesenen Datenbereich ein und verarbeitet ihn mittels der von der Anwendung vorgegebenen `map()`-Funktion. Die dabei erzeugten Schlüssel-Wert-Paare werden von jedem Mapper anhand des Schlüssels zunächst einem bestimmten Reduce-Task zugeordnet (Partitionierung) und die einzelnen Teile danach im selben Arbeitsschritt sortiert. Die sortierten Teile sollen jeweils in einer eigenen Datei gespeichert werden, so dass am Ende der Map-Phase für jede Kombination aus Map-Task und Reduce-Task eine Datei mit dem Zwischenergebnis vorliegt. In der darauffolgenden Reduce-Phase führt jeder Reduce-Task seine Partition aus den Teilen der Mapper-Ausgaben zusammen und führt sie der `reduce()`-Funktion zu. Die Vorsortierung der Map-Tasks wird dabei genutzt, um alle zu einem Schlüssel zugehörigen Werte zu einem `reduce()`-Aufruf zusammenzufassen. Jeder Reduce-Task speichert die Ausgabe, welche das Endergebnis darstellt, in jeweils einer eigenen Datei.

4.1.1 Eingabedaten

Um Daten mit dem Framework verarbeiten zu können, müssen diese von verschiedenen Quellen gelesen und als Wertpaare aufbereitet werden. Die dazu erforderlichen Klassen sollen im Package `mw.mapreduce.reader` zusammengefasst werden und auf der Schnittstelle `MWReader<MWPair<KEY, VALUE>>` aufbauen. Diese spezifiziert eine `read()`-Methode, welche entweder einen gelesenen Datensatz als `MWPair` liefern soll oder `null` zurückgibt, wenn das Ende des Eingabestroms erreicht ist.

Im Package `mw.mapreduce.util` steht als Grundlage die Klasse `MWSplitTextFileInput` zum zeilenweisen Einlesen von Textdaten zur Verfügung. Für deren Konstruktor ist zusätzlich zum Dateinamen die Angabe einer Startposition und Länge in Bytes möglich. Beim Einlesen der Datei wird dann nur der übergebene Bereich verarbeitet. Dabei ist sichergestellt, dass das Lesen 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.

Eine eingelesene Textzeile muss zur Verarbeitung durch MapReduce noch in ein Schlüssel-Wert-Paar aufgetrennt werden. Dies soll in der Klasse `MWKeyValueReader` erfolgen. Zur Repräsentation dieser Paare soll durchgängig die bereitgestellte Klasse `MWPair<String, String>` zum Einsatz kommen.

Des Weiteren müssen die Zwischenergebnisse der Mapper für die Reducer vorbereitet werden. Dieses Zusammenführen der vorsortierten Schlüssel-Wert-Paar-Datenströme in einen vollständig sortierten Datenstrom für den jeweiligen Reducer soll in der Klasse `MWMergingReader` im Reader-Package erfolgen.
Für die Reduce-Phase müssen außerdem alle Werte eines Schlüssels in gruppierter Form bereitgestellt werden. In der Übungsaufgabe soll diese Gruppierung auf Seiten des Reducer erfolgen und einen `Iterator` verwenden. Dazu wird die Adapter-Klasse `MWReduceReader` bereitgestellt, die eine Reihe normaler, sortierter Schlüssel-Wert-Paare eines Readers (z.B. `MWMergingReader`) in die Form `MWPair<String, Iterable<String>>` wandelt. Der Iterator läuft dabei so lange, bis sich zwischen zwei aufeinanderfolgenden Eingaben der Schlüssel ändert. Dann muss ein erneuter `read()`-Vorgang angestoßen werden, um die zugehörigen Werte abzurufen.

Aufgaben:

→ Implementierung einer Klasse `MWKeyValueReader` zum Einlesen von Textdateien als Schlüssel-Wert-Paare über die Schittstelle `MWReader`. Trennzeichen ist Tabulator (`"\t"`).

→ Erstellen der Klasse `MWMergingReader` zum parallelen Einlesen aus mehreren vorsortierten Textdateien. Die Vorsortierung soll dabei genutzt werden, um effizient eine vollständig sortierte Ausgabe über die `MWReader`-Schnittstelle bereitzustellen. Das Sortierkriterium muss konfigurierbar sein.

Hinweise:

- Die vorgegebenen Klassen befinden sich im `pub`-Verzeichnis zur Aufgabe.
- Enthält ein eingelesener Datensatz keinen Schlüssel, soll als Schlüssel stattdessen die (relative) Zeilennummer verwendet werden.

4.1.2 Kontexte für Mapper und Reducer

Die Ausführung eines Map- und Reduce-Tasks erfolgt im Rahmen eines Kontexts, welcher für die Verwaltung der Ein- und Ausgaben zuständig ist. Der Kontext soll dabei folgende gemeinsame Schnittstelle bereitstellen:

```
public interface MWContext<VALUETYPE> {
  // Input
  MWReader<MWPair<String, VALUETYPE>> getReader();

  // Output
  void write(String key, String value) throws IOException;
  void outputComplete() throws IOException;
}
```

Zur Übergabe der Eingabedaten an den eigentlichen Mapper- bzw. Reducer dient die Methode `getReader()`, die Zugriff auf den darunterliegenden Reader zum Einlesen der Eingabedaten ermöglicht.
Neben der Bereitstellung der zu verarbeitenden Daten ist der Kontext auch für die Aufnahme der Resultate des jeweiligen Tasks verantwortlich. Hierzu werden ihm die vom Map-/Reduce-Worker berechneten Ergebnisse in Form von Schlüssel-Wert-Paaren per `write()`-Methode übergeben. Durch einen Aufruf von `outputComplete()` wird der Kontext darüber informiert, dass der Task abgeschlossen ist und 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 eigenen Unterklasse von `MWContext` zu realisieren. Der Mapper-Kontext (`MWMapContext`) muss die beiden Methoden `write()` und `outputComplete()` dabei so implementieren, dass die Zwischenergebnisse am Ende des Map-Tasks nach zuständigem Reduce-Task partitioniert in einzelne Dateien ausgegeben werden. Der Inhalt der Dateien muss dabei nach Schlüssel sortiert sein, wobei das Sortierkriterium dem Mapper-Kontext im Konstruktor durch einen `java.util.Comparator<String>`-Parameter bekannt zu geben ist. Ein Reduce-Kontext (`MWReduceContext`) kann die übergebenen Ergebnisse direkt ausgeben, da diese bereits das finale Ergebnis darstellen.

Aufgaben:

→ Implementierung der Schnittstelle `MWContext` in einem Subpackage `mw.mapreduce.core`

→ Implementierung der Klassen `MWMapContext` und `MWReduceContext`

4.1.3 Mapper und Reducer

Unter Zuhilfenahme der in Teilaufgabe 4.1.2 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 implements Runnable {
  public void run();
  public void setContext(MWMapContext context);
  protected void map(String key, String value, MWContext<?> context) throws Exception;
}

public class MWReducer implements Runnable {
    public void run();
    public void setContext(MWReduceContext context);
    protected void reduce(String key, Iterable<String> values,
                          MWContext<?> context) throws Exception;
}
```

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`

Hinweis:

- In der `reduce()`-Methode werden alle einem Schlüssel zugehörigen Werte zusammengefasst per `Iterator` übergeben. Dies wird im zuständigen Reader für die Eingabe umgesetzt (siehe Teilaufgabe 4.1.1).

4.1.4 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 die verwendeten Mapper und Reducer einer Anwendung konfigurieren zu können, soll eine Factory `MWJob` für die Erzeugung der Worker zum Einsatz kommen. Zudem soll ein Konfigurieren des Readers für die Eingabedaten mittels `createInputReader()` sowie des verwendeten Sortierkriteriums für Schlüssel mittels `getComparator()` möglich sein. Anzahl und Typen der Parameter der `create...()`-Methoden sind freigestellt. Werden die Methoden aus der Anwendung nicht überschrieben, soll eine Standardimplementierung bereitgestellt werden.

```
public class MWJob {
    public MWMapper createMapper([...]);
    public MWReducer createReducer([...]);
    public MWReader<MWPair<String, String>> createInputReader([...]) throws Exception;
    public java.util.Comparator<String> getComparator();
}
```

Das MapReduce-Framework selbst 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 4.2), `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 `MWJob` im Subpackage `mw.mapreduce.core`

→ 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. Reduce-Threads soll ein *Executor-Service* übernehmen.

4.2 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.

Hinweise:

- Zur Kontrolle, dass in den Anwendungen keine Daten verloren gegangen sind, kann die Anzahl der erzeugten Zeilen der Zwischen- oder Ausgabedateien mittels `wc -l <datei1> <datei2> ...` ermittelt werden.

4.2.1 Sortieren von Textdateien (für alle)

Eine der naheliegendsten mithilfe 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/aufgabe4/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)

4.2.2 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 mithilfe 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/aufgabe4/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 einem anwendungsspezifischen MWReader zu implementieren.
- Die Hilfsklasse MWSplitTextFileInput (siehe Teilaufgabe 4.1.2) 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.

4.2.3 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/aufgabe4/friends.list) besteht aus den Ergebnissen der Teilaufgabe 4.2.2, 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 4.2.1 ab.

Abgabe: am 27.01.2021 in der Übungssprechstunde