4 Übungsaufgabe #4: Verteiltes MapReduce

In dieser Aufgabe sollen aus Freundschaftsbeziehungen Cliquen mit Hilfe mehrerer verketteter MapReduce-Schritte auf Basis des *Hadoop*-Frameworks identifiziert werden. Um auch größere Datenmengen in kurzer Zeit verarbeiten zu können, soll schließlich im zweiten Teil das Programm auf der I4 Private Cloud ausgeführt werden.

4.1 Finden von Cliquen

Im ersten Aufgabenteil sollen Cliquen einer bestimmten Mindestgröße in einem ungerichteten Graphen gefunden werden (siehe Tafelübung). Die grundlegende Idee hinter dem Algorithmus zur Implementierung mit MapReduce stammt dabei aus dem Artikel *Graph Twiddling in a MapReduce World* [Cohen09]. Die Knoten des Graphen stellen die einzelnen Personen dar, während die Kanten den Freundschaften zwischen den einzelnen Personen entsprechen. Die zu findenden Cliquen zeichnen sich dadurch aus, dass es sich dabei um eine Menge von Personen handelt, die alle in gegenseitiger Freundschaftsbeziehung stehen. Der zu implementierende Algorithmus filtert dazu in mehreren Iterationen Personen aus dem Graphen, die aufgrund ihrer Freundeskreise nicht als Teil einer Clique in Frage kommen.

4.1.1 Implementierung der grundlegenden Datentypen (für alle)

Bevor mit der Implementierung des eigentlichen Graph-Algorithmus in Teilaufgabe 4.1.2 begonnen werden kann, müssen zunächst einige grundlegenden Datentypen erstellt werden. Diese sind notwendig, um die Eingabedaten einzulesen, Zwischenergebnisse zu speichern, und schließlich die gewünschten Ausgaben zu erzeugen. Die Klassen für diese Datentypen sind im Subpackage `mw.cliquefind.datatypes` abzulegen und müssen die Schnittstelle `org.apache.hadoop.io.WritableComparable` von Hadoop zur (De-)Serialisierung implementieren.
Einzelne Personen (= Knoten) werden durch positive Zahlen eindeutig identifiziert und durch den Java-Typ `int` dargestellt. Die vorgegebenen Eingabedaten und das zu erzeugende Endergebnis besteht aus einer Liste von gegenseitigen Freundschaftsbeziehungen. Die Speicherung erfolgt im Binärformat durch jeweils zwei direkt aufeinander folgende `int`-Werte und gibt jeweils ein Personen-Paar an, zwischen denen eine Beziehung besteht. Diese sollen in Java durch die Klasse `Edge` repräsentiert werden. Da die letzten Schritte des Algorithmus zum Finden von Cliquen auf Dreiecken basiert, ist weiterhin die Klasse `Triangle` notwendig, die drei jeweils direkt miteinander befreundete Personen zusammenfasst.
Weiterhin sind im Verlauf des Graph-Algorithmus aus Teilaufgabe 4.1.2 verschiedene Elemente des Graphen abzuzählen und zu gewichten. Dazu soll der Java-Typ `long` zum Einsatz kommen, um ein Überlaufen bei großen Datenmengen und eine Verwechslung mit Personen-IDs (Typ `int`) zu vermeiden. Zur Darstellung gewichteter Graphen sind zudem noch die Klassen `WeightedVertex` und `WeightedEdge` notwendig, die einen Knoten beziehungsweise eine Kante zusätzlich mit einem Zählerwert (Typ `long`) gewichten können.

Aufgabe:

→ Implementierung der Klassen `Edge`, `Triangle`, `WeightedVertex`, `WeightedEdge` zur Repräsentation und Serialisierung entsprechender Datentypen

Hinweise:

- Zur Vereinfachung der Ein-/Ausgabe von Dateien ist im Pub-Verzeichnis das Package `mw.cliquefind.io` vorgegeben. Darin enthalten sind unter anderem die Klassen `RecordInputFormat` und `RecordOutputFormat` zum Einlesen und Ausgeben von Binärdateien mit Datensätzen fester Länge.
- Ein- und Ausgaben erfolgen über diese Klassen nur über das Wert-Feld der Schlüssel/Wert-Paare. Der Schlüssel ist immer vom Typ `NullWritable` und damit leer.
- Die Klassennamen sollten möglichst den Vorgaben der Aufgabenstellung entsprechen. Die vorgegebenen Hilfsklassen zum Einlesen der verschiedenen Datentypen (`...InputFormat`) im Package `mw.cliquefind.io` benötigen den jeweiligen Namen der Klasse um Objekte vom entsprechenden Typ beim Einlesen instanziieren zu können.
- Die zum Entwickeln zur Referenzierung notwendigen Klassenbibliotheken `commons-logging-api-1.0.4.jar` und `hadoop-core.jar` sind im `/proj/i4mw/pub/aufgabe4/lib`-Verzeichnis hinterlegt.

4.1.2 Graph-Algorithmus zum Finden von Trägern (für alle)

Der Algorithmus zum Finden von tragenden Kanten lässt sich in sechs verschiedenen Teilschritten implementieren. Die ersten drei Teilschritte dienen dabei dazu, den zunächst ungerichteten Graphen in einen gerichteten zu verwandeln. Dabei wird versucht, die Anzahl der möglichen Kandidaten für Dreiecks-Überlappungen für die folgenden Schritte vorab zu minimieren. Die restlichen drei Teilschritte identifizieren dann alle möglichen Dreiecke und zählen, wie oft eine Kante Teil eines Dreiecks ist. Nur wenn eine Kante mindestens $n-2$ mal verwendet wird, kann sie Teil einer Clique mit mindestens n Knoten sein und bildet damit einen Träger.

Im **1. Schritt** soll zunächst bestimmt werden, wie oft eine Person in den Eingabedaten auftaucht, und damit auch wie viele Freundschaftsbeziehungen sie besitzt. Dies bestimmt den *Grad* des entsprechenden Knotens im Graph. Die Ausgabe besteht aus dem Knoten und dem ermittelten Grad. Dieser Teilschritt soll in der Klasse `S1CalculateVertexDegreesJob` implementiert werden.

Im **2. Schritt** des Algorithmus sollen die Kanten aus den Eingabedaten und die im ersten Schritt bestimmten Grade der Knoten zusammengeführt werden, so dass ein gerichteter Graph entsteht, der die Kanten entsprechend des Grades des Anfangsknotens gewichtet. Dabei verdoppelt sich die Anzahl der Kanten gegenüber der Eingabe, da jede ungerichtete Kante in zwei gerichtete Kanten mit Gewichtung transformiert wird. Für diesen Schritt soll der Klassenname `S2JoinVertexDegreesJob` verwendet werden.

Im **3. Schritt** geht es nun darum, aus den gerichteten Kanten diejenige zwischen zwei Knoten herauszufinden, welche die niedrigste Gewichtung aufweist. Weisen die beiden möglichen Kanten die gleiche Gewichtung auf, ist diejenige mit der niedrigsten Personen-Identifikationsnummer im Anfangsknoten zu bevorzugen. Die Ausgabe erfolgt jeweils als einfache Kante ohne Angabe einer Gewichtung, da diese im weiteren Verlauf nicht mehr benötigt wird. Die Implementierung dieses Schritts soll in der Klasse `S3GraphDirectionJob` abgelegt werden.

Nachdem der ursprünglich ungerichtete Graph nun in Richtung der Knoten mit dem jeweils höheren Grad ausgerichtet wurde, sollen im **4. Schritt** alle möglichen Dreiecke ausgegeben werden. Diese lassen sich bilden, indem man ausgehend von jedem Knoten des gerichteten Graphen die jeweiligen Endknoten in jeder möglichen Kombination miteinander verbindet. Die Dreiecke bilden die Ausgabe dieses Teilschritts, der in der Klasse `S4ListTriangleCandidatesJob` implementiert werden soll.

Im **5. Schritt** muss noch ermittelt werden, ob die im vorherigen Schritt erzeugten Dreiecke auch wirklich in den ursprünglichen Eingangsdaten existieren. Nicht alle Kanten, die zum Schließen eines möglichen Dreiecks notwendig sind, müssen auch im ursprünglichen, ungerichteten Graphen vorkommen. Dazu wird der ungerichtete Graph aus den Eingabedaten des Algorithmus mit den Dreiecks-Kandidaten zusammengeführt. Eine Ausgabe des Dreiecks erfolgt nur, wenn die in Schritt 4 zusätzlich erzeugte Kante auch in den ursprünglichen Eingabedaten des Algorithmus vorhanden ist. Dieser Schritt ist in der Klasse `S5FindTrianglesJob` zu abzulegen.

Im abschließenden **6. Schritt** ist nun noch herauszufinden, welche der Kanten Träger darstellen. Dazu werden die vom vorherigen Schritt ausgegebenen Dreiecke wieder in einzelne, ungerichtete Kanten zerlegt und gezählt, wie oft jede Kante vorkommt. Kanten, die weniger als $n-2$ mal vorkommen, sollen *nicht* ausgegeben werden. Alle anderen Kanten bilden die Ausgabe des Algorithmus zum Finden von Trägern. Die Klasse zur Implementierung dieses Schritts soll unter dem Namen `S6FindTrussJob` abgelegt werden.

Aufgabe:

→ Implementierung der einzelnen Schritte des Graph-Algorithmus in Form von MapReduce-Jobs (Basisklasse `org.apache.hadoop.mapreduce.Job`) für Hadoop

Hinweise:

- Da die Personen-IDs nur positiv sind, lassen sich negative Werte in den Zwischenschritten zur Kennzeichnung von speziellen Einträgen verwenden.
- Zum Zusammenführen verschiedener Datentypen mit unterschiedlichen Mapper-Klassen in einer Map-Phase kann die Klasse `org.apache.hadoop.mapreduce.lib.input.MultipleInputs` verwendet werden. Diese erlaubt das Setzen von Eingabedatei und zugehörigem Mapper für einen Job mit Hilfe der statischen Methode `addInputPath()`. Bei Verwendung von `MultipleInputs` darf kein Mapper für das Job-Objekt mittels `setMapperClass()` gesetzt werden.
- Zur Entwicklung und Fehlersuche empfiehlt es sich jeweils die Ausgaben der Map- und Reduce- Schritte zusätzlich in Textform auf der Konsole ausgeben zu lassen.
- Zum Testen kann der einfache Graph aus `/proj/i4mw/pub/aufgabe4/data/small.bin` verwendet werden. Graphen (nur ungewichtete Kanten, Binärformat) lassen sich mit Hilfe des Programms `visualize.sh` aus dem Verzeichnis `/proj/i4mw/pub/aufgabe4/tools`-Verzeichnis grafisch darstellen.

4.1.3 Combiner zur effizienteren Verarbeitung (optional für 5,0 ECTS)

Um die Datenmenge zu reduzieren, die vom Map-Schritt an die Reduce-Phase weitergereicht werden muss, sollen sogenannte *Combiner* eingesetzt werden. Diese führen bereits einen Reduce-Vorgang auf die vorsortierten Daten der Mapper aus, womit sich die Menge der zu sortierenden Daten und damit die benötigte Ausführungszeit verringern lässt. Bei dem Algorithmus zum Finden von Trägern aus Teilaufgabe 4.1.2 lässt sich dies insbesondere in Schritt **1** und **6** ausnutzen. Daher sollen diese Schritte in dieser Aufgabe noch um entsprechende Combiner ergänzt werden.

Aufgabe:

→ Ergänzung der Schritte 1 und 6 um Combiner

4.1.4 Zusammenfassung der Einzelschritte und mehrfache Iteration (für alle)

Der Algorithmus aus Teilaufgabe 4.1.2 besteht aus mehreren einzelnen MapReduce-Schritten und muss zudem mehrfach hintereinander auf den Graphen angewendet werden, um die gesuchten Cliquen zu finden. Da das manuelle Ausführen der einzelnen Jobs zu mühsam wäre, soll zunächst eine Klasse implementiert werden, welche die einzelnen Teilschritte miteinander verbindet und automatisch hintereinander ausführt. Dabei ist zu beachten, dass für Zwischenergebnisse temporäre Dateien nach einem Schema erzeugt werden sollen, die ein gegenseitiges Überschreiben durch verschiedene Schritte beziehungsweise Iterationen verhindert. Um Speicherplatz zu sparen, sollen diese temporären Dateien zum frühest möglichen Zeitpunkt wieder gelöscht werden, sobald sie von keinem weiteren MapReduce-Schritt mehr benötigt werden.
Hinweis:

- Zur Fehlersuche ist es unter Umständen hilfreich, das Löschen der Dateien aus den Zwischenschritten bei Bedarf möglichst einfach abschalten zu können.

Um nicht nur Träger zu identifizieren, sondern die gesuchten Cliquen, muss der Algorithmus zudem so lange auf den Graphen angewendet werden, bis keine Kanten mehr entfernt werden können. Daher ist schließlich das Programm noch dementsprechend zu ergänzen, dass die notwendige Zahl an Iteration durchgeführt wird. Die Ein- und Ausgabedatei, sowie die minimale Größe der gesuchten Clique sollen sich über Parameter auf der Kommandozeile einstellen lassen. Die angegebene Eingabedatei darf nicht überschrieben werden.

Aufgaben:

→ Implementierung der Klasse `FindTrussIteration` zur Verkettung der einzelnen MapReduce-Schritte
→ Ableiten der `Tool`-Klasse von Hadoop zur Implementierung der Klasse `FindCliqueTool` zum Starten des Algorithmus mit den angegebenen Parametern
→ Finden von Cliquen mit mindestens 5 Personen in `/proj/i4mw/pub/aufgabe4/data/medium.bin`

4.1.5 Anfertigung von Statistiken (optional für 5,0 ECTS)

Ergänzend zum Suchen der Cliquen sollen noch verschiedene statistische Informationen über die Daten beim Durchlaufen der MapReduce-Schritte ermittelt werden. Eine interessante statistische Größe ist zum einen die durchschnittliche Anzahl an Freundschaften, die jede Person in dem angegebenen Graphen besitzt. Weiterhin kann durch Betrachtung der Zahl der Dreiecks-Kandidaten und der tatsächlichen Zahl der gefundenen Dreiecke herausgefunden werden, wie effizient dieses Verfahren bei den bearbeiteten Graphen ist. Um einen einfachen Überblick über die verarbeiteten Datenmengen zu bekommen, soll außerdem bei jedem Map- und Reduce-Schritt gezählt werden, wie viele Kanten zur Verarbeitung eingelesen wurden. Diese Zahlen sollen ohne Hinzufügen zusätzlicher MapReduce-Durchläufe bestimmt werden.

Aufgaben:

→ Bestimmung der durchschnittlichen Anzahl an Freunden pro Person
→ Vergleich der Zahl der in Schritt 4 erzeugten Dreiecke mit der Zahl der in Schritt 5 ausgegebenen Dreiecke
→ Ermittlung der Anzahl von Kanten, die in jedem Schritt verarbeitet wurden

4.2 Verteilte Ausführung in der Cloud (für alle)

Da durch das Aufzählen aller möglichen Dreiecke aus Graph-Bestandteilen sehr schnell große Datenmengen entstehen können, soll schließlich zur Verarbeitung des Graphen in der Datei `s3n://aufgabe4/large.bin` Hadoop mit dem Algorithmus auf der Private Cloud des Lehrstuhls 4 ausgeführt werden. Dazu sollen drei Instanzen des VM-Abbilds `emi-74700EB2` vom Typ **c1.medium** verwendet werden. Zum Verteilen der Konfiguration und zum Starten von Hadoop ist das vorgefertigte Skript in `pub/aufgabe4/tools/start-hadoop.sh` zu verwenden.

Aufgabe:

→ Finden der Cliquen mit mindestens 35 Personen in den vorgegebenen Daten in der I4-Cloud

Hinweis:

- Die Datei `large.bin` muss zunächst ins HDFS kopiert werden, da der Speicher der Java-VM nicht ausreicht, um deren kompletten Inhalt zwischenzuspeichern.

Abgabe: am 18.1.2013

Cohen09: Jonathan Cohen. Graph Twiddling in a MapReduce World. In *Computing in Science & Engineering Volume 11 Issue 4*, S. 29-41, ISSN 1521-9615, IEEE Computer Society, 2009
`http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5076317`
(Kopie: `/proj/i4mw/pub/aufgabe4/Cohen-Graph_Twiddling.pdf`)