Aufgabe 1: MapReduce

Ein optisches Meßverfahren erfasst die Oberfläche von Objekten als Menge von 3D-Ortsvektoren $(x, y, z)^T$ und speichert diese in der Datenbanktabelle $S(\underline{vectorId}, \underline{dimension}, value)$ ab. Durch Störeinflüsse werden beim Messen nicht alle Ortsvektoren vollständig erfasst.

Für eine grafische Simulation werden nun die Vektorlängen *length* aller **vollständig erfasster** Ortsvektoren benötigt, $length = \sqrt{x^2 + y^2 + z^2}$. Aufgrund der großen Datenmenge soll die Berechnung auf einem MapReduce-Cluster erfolgen.

a) Erzeuge als Ausgabe eine Tabelle $T(\underline{vectorId}, length)$, in der für jeden vollständig erfassten Ortsvektor die Vektorlänge steht. Die Ausgabe wird partitioniert auf den verschiedenen Nodes ins verteilten Dateisystem geschrieben. Löse die Aufgabe mit nur einem MapReduce-Job. Verwende auch die in der Vorlesung vorgestellte Funktion *combine*, um Netzwerklast zu reduzieren. Beschreibe dein Vorgehen kurz in wenigen Sätzen. Erstelle den Pseudocode für *map*, *combine* und *reduce*.

Lösung

Die Map-Funktion emittiert als Schlüssel die ID des Vektors und als Wert jeweils den quadrierten Dimensionswert. **2 P**

```
map (key, tuple) {
  emit (tuple.vectorid, tuple.value * tuple.value);
}
```

Die Combine-Funktion erhält als Schlüssel die ID eines Vektors und als Wert die Liste der zugehörigen quadrierten Dimensionswerte, die dann aufsummiert werden. Für vollständige Vektoren wird die Vektorlänge ins verteilte Dateisystem geschrieben. Die Kennzahlen der unvollständigen Vektoren werden über das Netzwerk zur Reduce-Funktion geschickt. **3 P**

```
combine (key, values) {
  s = 0;
  foreach (values as v)
    s += v;
  if (values.size() == 3)
    write (key, sqrt(s));
  else
    emit (key, {'count' => values.size(), 'sum' => s});
}
```

Die Reduce-Funktion schreibt für vollständige Vektoren die Vektorlänge ins verteilte Dateisystem. **3 P**

```
reduce (key, tuples) {
  c = 0;
  s = 0;
  foreach (tuples as t) {
    c += t.count;
    s += t.sum;
  }
  if (c == 3)
    write (key, sqrt(s));
}
```

Hinweis: Wir benutzen hier eine Funktion `write`, um das Schreiben ins verteilte Dateisystem darzustellen.

b) Die Abbildung zeigt beispielhaft eine Verteilung von S auf drei MapReduce-Nodes. Zeige für dieses Beispiel die Ausgabe der einzelnen Phasen deines MapReduce-Jobs auf jeder der drei Nodes. **3 P**

Eingabe (3 Datanodes)

id	*dim*	*val*
1	x	2
2	z	4
1	y	3
3	y	2
4	y	4

id	*dim*	*val*
3	z	4
4	z	2
2	x	2
2	y	4

id	*dim*	*val*
5	y	3
3	x	4
4	x	4

Phase 1 - Ausgabe (map)

key	*value*
1	4
2	16
1	9
3	4
4	16

key	*value*
3	16
4	4
2	4
2	16

key	*value*
5	9
3	16
4	16

Phase 2 - Ausgabe (combine)

key	*value*
1	{2,13}
2	{1,16}
3	{1, 4}
4	{1,16}

key	*value*
3	{1,16}
4	{1, 4}
2	{2,20}

key	*value*
5	{1, 9}
3	{1,16}
4	{1,16}

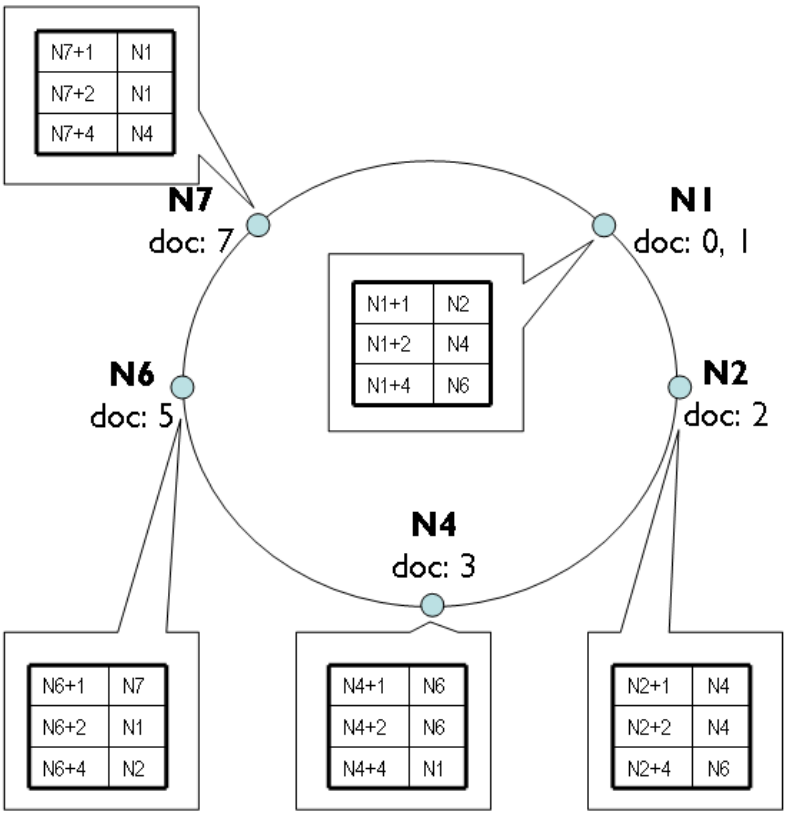
Phase 3 - Ausgabe (reduce)

key	*value*
2	6
3	6
4	6

Aufgabe 2: Distributed Hash Tables (DHT): Chord

Chord – Protokoll: Siehe auch, http://sarwiki.informatik.hu-berlin.de/Chord

a)



Lookup: Bei einer Suchanfrage an Peer *n* nach Schlüssel *k* wird zunächst der nächstgelegene Vorgänger von *k* über die *Finger Table* von *n* gesucht. Das wiederholt sich, bis der Peer gefunden ist, dessen direkter Nachfolger für *k* zuständig ist (erster Eintrag in der *Finger Table*).

b) 2 > 6 > 7 > 1

Peer 2: Peer 6 ist nächstgelegene Vorgänger für Schlüssel 0, den Peer 2 kennt. Somit wird die Suchanfrage an Peer 6 weitergereicht.
Peer 6: Peer 7 ist direkter Vorgänger für Schlüssel 0. Somit wird die Suchanfrage an Peer 7 weitergereicht.
Peer 7: Suchanfrage für Schlüssel 0 an den direkten Nachfolger, Peer 1, weiterleiten.

c) 7 > 1 > 2 > 6

Peer 7: Peer 1 ist nächstgelegene Vorgänger für Schlüssel 5, den Peer 7 kennt. Somit wird die Suchanfrage an Peer 1 weitergereicht.
Peer 1: Peer 2 ist nächstgelegene Vorgänger für Schlüssel 5, den Peer 1 kennt. Somit wird die Suchanfrage an Peer 2 weitergereicht.
Peer 2: Suchanfrage für Schlüssel 5 an den direkten Nachfolger, Peer 6, weiterleiten.

Aufgabe 3: Two Phase Commit (2PC)

Commit Request Phase	**INPUT**
	msg=READY or msg = EndTransaction;
	ACTION
	LocalState := CheckCommit();
	Log(LocalState); // READY / FAILED
	if hasNextWorker()
	send LocalState to nextWorker;
	else
	if LocalState = READY
	send COMMIT to self;
	elseif LocalState = FAILED
	send ABORT to self;
	fi;
	fi;
		INPUT
		msg = FAILED;
		ACTION
		Log(FAILED);
		if hasNextWorker()
		send FAILED to nextWorker;
		else
		send ABORT to self;
		fi;

Commit Phase	**INPUT**
	msg = COMMIT;
	ACTION
	Log(COMMIT);
	CommitLocalTransaction();
	FreeLocalResources();
	if(hasPreviousWorker())
	send COMMIT to previousWorker;
	fi;
		INPUT
		msg = ABORT;
		ACTION
		Log(ABORT);
		AbortLocalTransaction();
		FreeLocalResources();
		if(hasPreviousWorker())
		send ABORT to previousWorker;
		fi;