Uitwerkingen oefeningen MapReduce

Versie 20230502

Denk altijd op de volgende regels:

- Elke Map en Reduce worker kan alleen bij zijn eigen lokale datastructuren. Hij kan niet bij anderen kijken of schrijven.
- Het Map model houdt in dat je een generieke functie voor elk <key, value>-paar schrijft. Je hebt geen mogelijkheid tot het expliciet manipuleren van je eigen input chunk (bijvoorbeeld herhaald scannen).
- De enige output-functie is *emit*.
- Reduce workers krijgen hun input uitsluitend via de <key,list>-paren. Ook hier heb je geen mogelijkheid tot het expliciet manipuleren van je eigen input chunk (bijvoorbeeld herhaald scannen). Wel kun je natuurlijk de list die je onder handen hebt herhaald scannen, want die zit in een lokale datastructuur.

[1] INIT_MAP: Maak een binaire relatie (C#: dictionary) waarin je woorden en hun frequenties opslaat.
MAP <docid, text>:
Houdt voor elk woord in text de frequentie bij via de dictionary. Je aggregeert dus al in de MAP.
FINALIZE_MAP: emit je alle <word, freq> paren in de dictionary

Ga ervan uit dat elke doc door 1 MAP worker behandeld wordt. <key, value> pairs van de input blijven heel in de segmentering van de input van Map.

```
REDUCE < w, flist >:
  int fsum = 0;
  FOR EACH f IN flist fsum += f;
  emit(< w, fsum >);
```

[2]
MAP <g,v>: IF v >= 100 emit <g,v>

```
REDUCE <g, vlist>:
  int  sumv = 0;
  FOR EACH v IN VLIST sumv += v;
  IF sumv >= 10000 emit <g, sumv>;
```

[2++] INIT_MAP: Maak een binaire relatie d (C#: dictionary) waarin je g en sum(v) bijhoudt;
MAP <g,v>: IF v >= 100 { add <g,v> to d; } // als g al in de dictionary zit, verhoog je de sumv met v
FINALIZE_MAP: emit alle <g,v> paren in d;
De REDUCE blijft onveranderd.

[3]
MAP <T,<x,y>>:
IF T= R THEN emit <y, <x, R>>; ELSE emit <x, <y, S>>;

```
REDUCE <b, tlist>:
Rlist := emptylist; Slist := emptylist;
FOR EACH t IN tlist
   IF t.component2 = R THEN add <t.component1 > to Rlist;
   ELSE add <t.component1 > to Slist;
FOR EACH t1=<a > IN Rlist
    FOR EACH t2=<c > IN Slist
      emit <a, b, c>;
```

[4]

```
MAP <docid, text>:
FOR EACH word IN text  emit  <word, docid> ;

REDUCE <w, wlist>:
 sort (wlist);  emit <w, wlist>
// de output staat dan klaar voor verwerking in een Dictionary+Posting List-structuur
```

[5] eerste poging:

```
MAP <docid, text>:
 int pos = 0;
 list wlist = emptylist;
 FOR EACH word w IN text: { pos++; emit<w, <docid, pos>>;  }
 // waarbij de FOR EACH consecutief door de text heen gaat

REDUCE <w, dplist>:
// bij een woord w hebben we een lijst van entries van de vorm <docid, pos>
 dplist = sort(dplist);      // sorteer eerst op docid en bij gelijke docid op pos
 wlist = emptylist;  // total list for word w
 while (dplist is not empty) {
      curlist = emptylist;  // list for current docid
      curdocid = dplist.docid;  add(curlist, dplist.pos);  // first positional entry
      dplist = next(dplist);
      WHILE (dplist.docid == curdocid)  { add(curlist, dplist.pos); dplist = next(dplist); }
      add(wlist, <curdocid, curlist>);
 }
 emit <w, wlist>;
```

[5] tweede poging:

```
(we hebben als aanname dat een document door één map worker verwerkt wordt;
daarom kunnen we de positielist voor een combinatie <w, docid> al in de Map aanmaken;
dat maakt de Reduce eenvoudiger en vermindert het aantal emits)
MAP <docid, text>:
 creëer een dictionary dict met entries van het type [w, <list of int>];
 FOR EACH word w IN text: { pos++; dict.add(w,pos);  }
 // in de dictionary wordt pos achteraan in de list achter w toegevoegd, zodat dat deze gesorteerd blijft
 // lees de dictionary nu uit in volgorde op w
```

```
FOR EACH entry <w, poslist> IN dict { emit <w, <docid, poslist>>; }
  // merk op dat het aanmaken en het dumpen van de dictionary hier niet via INIT_MAP en FINALIZE_MAP geschiedt!
  // één instantie van Map verwerkt namelijk een document in zijn geheel

 REDUCE <w, docposlist> :
// bij een woord w hebben we een lijst van entries van de vorm <docid, poslist>
docposlist.sort();  // sorteer op docid
emit<w, docposlist>;
```