Big Data Analysis
Group 6

Cedric Mingneau (1232611)
William Thenaers (1746077)
2019-11-13

Similar Items: Plagiarism Detection

Initial analysis (ter referentie, implementatieverslag vanaf pagina 3)

1. *How can we establish plagiarism in python source files? Keep in mind that students often tend to change solely the names of variables and functions within the code.*

   - In de eerste plaats *tokenizen* we de source files. Met behulp van een Python language parser identificeren we woorden uit een source file die de naam van een variabele moeten voorstellen. Op die manier kunnen alle variabelen, klassen- en functienamen vervangen worden door "ID"-tokens. We zouden deze tokens dan als één karakter kunnen voorstellen (bijv. "$"). Zo voorkomen we dat de gelijkenis van de documenten negatief beïnvloed wordt door variabelen anders te noemen.

   - Eens de broncode uniforme ID's bevat, kunnen we deze opsplitsen in woorden van een bepaalde lengte die we vervolgens verder in k-shingles samenvoegen.

     Vb "`my_list = { x * x for x in range(10) }`" => "`$={$*$for$inrange(10)}`"
     => 5-shingles: {`$={$*`, `={$*$`, `{$*$f`, `$*$fo`, `*$for`, …, `(10)}`}

2. *Explain how Minhash signatures can be computed in a MapReduce environment and argue about the correctness of your approach. Describe in detail how hashes are computed.*

   - Indien alle bestanden gelijkmatig als chunks worden verdeeld over enkele compute nodes, kan het parallel berekenen van MinHashes voor problemen zorgen. De nodes kunnen vlot de tokenizing- en shingling-stappen uitvoeren. Bij de chunk-aanpak verliezen we echter de bundeling van een document. Je kan de MinHashing-stap enkel uitvoeren op bit-vectoren die op het volledige document zijn gebaseerd. Om de shingles naar behoren te hashen, zouden we daarom eerst alle chunks moeten defragmenteren in documenten. Deze stap lijkt ons omslachtig.

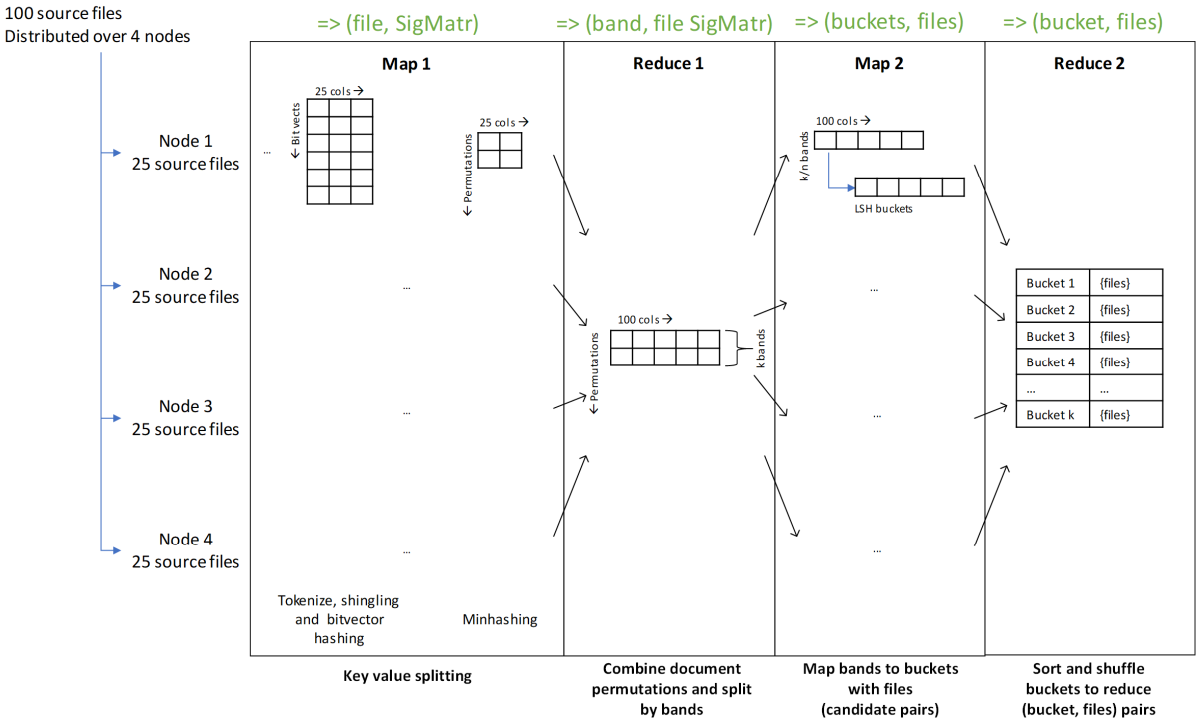
   - Vermits het ontdekken van plagiaat wordt gebruikt binnen de context van schooltaken, lijkt het ons veroorloofd om te veronderstellen dat de documentgroottes niet erg variabel zijn. Dit betekent dat we documenten over nodes kunnen verdelen zonder een echt ongelijkmatige werkverdeling te veroorzaken. Alle voorgenoemde stappen kunnen dan op de nodes gebeuren zonder de omslachtige defragmentatiestap. Omdat alle documenten 'heel' blijven zullen de MinHash signaturen correct worden berekend.

   - Om een goede Signature Matrix te bekomen, moeten we een aantal pseudo-random permutaties uitvoeren op de rij-(/bit-)indices van de gehashte bitvectoren van shingles. Deze permutaties worden gedaan via een aantal hashfuncties H die de rijindices afbeelden op andere indices. Zulke hashfuncties kunnen bijvoorbeeld de vorm hebben van $(\mathrm{ax} + \mathrm{b})\ \%\ \mathrm{length}$, met a en b distinct verschillende getallen voor elke functie en length het aantal shingle-bits in de document input matrix.

Cedric Mingneau (1232611)
William Thenaers (1746077)

2019-11-13

3. *Present the end-to-end architecture of the pipeline, preferably by means of a visualization. Draw specific attention to the input and output of each of the MapReduce stages involved. Clearly indicate the key used during reduce phases and describe the logic of non-trivial map operations.*



1. De bronbestanden worden gelijkmatig verdeeld over de beschikbare compute nodes.
2. Elk bestand wordt gestreamd door een tokenizer die identifiers vervangt door bvb. `$` en whitespace negeert. N-shingles worden vervolgens opgesteld en gehasht naar een bitvector waarin elke bit aanduidt of de shingle uit de shingle-set voorkomt in het document.
3. Minhashing wordt toegepast en de Signature Matrix wordt opgesteld afhankelijk van een aantal pseudo-random gekozen hashfunctie, en dit voor elk document op de node.
4. In de volgende stap worden de Signature Matrices “samengevoegd” om voor elk bestand enkele banden uit de bitvector te kiezen voor verdere behandeling op elke node.
5. In deze banden worden mogelijke kandidaatparen beschouwd en de gelijkenis berekend. Paren waarvoor de kans op gelijk zijn boven een bepaalde grens ligt, worden naar eenzelfde bucket gehasht (Locality Sensitive Hashing).
6. In de voorlaatste stap worden de buckets van elke node opnieuw samengevoegd om een unieke set te creëren van elke file in elke bucket.
7. Ten slotte kan men de lijsten uit elke bucket nu samenvoegen om tot het finaal resultaat te komen.

In stap 1-3 worden *(file, Signature Matrix)*-paren aangemaakt in de mapping. Stap 4 maakt hiervan nieuwe *(band, file Signature Matrix)*-paren, waarbij een aantal banden per node verdeeld worden, samen met elke kolom (elke file). Stap 5 zorgt vervolgens dat deze files naar *(bucket, files)*-paren worden omgezet, die in stap 6 over elke node gelijkmatig verzameld worden, waarbij node 1 bijvoorbeeld $buckets/4$ aantal buckets krijgt die hij moet samenvoegen. In de laatste stap produceren we dan opnieuw *(bucket, files)*-paren, maar deze keer werden de lijst met bestanden reeds samengevoegd in de vorige stap. Elke bucket bevat een indicatie van de waarschijnlijkheid dat de bestanden in die bucket gelijkwaardig waren. Deze files kunnen vervolgens in meer detail vergeleken worden om na te gaan wat er precies in voorkomt dat gelijkaardig was aan de andere bestanden in diens bucket.

Cedric Mingneau (1232611)
William Thenaers (1746077)

Implementation report

1. *Explain how your final implementation differs from the one presented in the initial analysis. Elaborate on the computation of the Minhash signatures and argue the correctness of your approach.*

   - In onze analyse gingen we er impliciet vanuit dat de karakteristieke matrix (of bit vector) in het geheugen van een node past. Vermits dit uiteraard niet het geval is, moesten we een nieuwe minhashing-aanpak zoeken.
   - Bij onze eerste poging maakten we gebruik van het feit dat een hashfunctie iedere shingle omzette naar een hash. Dit betekende dus dat als we het volledige beeld van de hashfunctie controleerde, we alle mogelijke shingles hadden gecontroleerd, alle posities uit de bit vector dus, zonder hierbij alle shingles in memory te moeten laden. Uiteraard zouden ook heel veel niet relevante hashwaardes onderzocht worden. Veronderstellende dat we MD5 als hash-functie gebruikten, zouden we dan 2^{128} verschillende hashes moeten controleren, hetgeen uiteraard eveneens niet haalbaar leek.
   - Na het minhashing-algoritme uit de cursus opnieuw te bestuderen, konden we een verband vinden tussen de gehashte shingle en de hash van die gehashte shingle. In plaats van het hele beeld van MD5 te onderzoeken, gaan we nu enkel de shingles van één file beschouwen. Deze subset komt namelijk exact overeen met de bits die '1' zouden zijn in de overeenkomstige bit vector. Iedere permutatie hashfunctie gaat dan elk van deze shingles hashen en vindt voor elke shingle een nieuwe hash. Uit die verzameling wordt dan de kleinste waarde geselecteerd als entry in de signatuurmatrix.
   - Correctheid:
     - In het boek en in de slides wordt het minhashing-algoritme voorgesteld uitgaande van de karakteristieke matrix. Je kan dan die matrix rij per rij afgaan en indien er 1'en voorkomen in een bepaalde kolom van een rij controleer, je de corresponderende hashwaarde voor die rij op die kolom en vervang je de waarde in de signatuurmatrix als de nieuwe hashwaarde kleiner is dan die van de huidige entry.
     - Voor een enkele file is het echter niet nodig om de volledige karakteristieke matrix ter beschikking te hebben indien je zijn kolom binnen de signatuurmatrix wilt berekenen. De aanpak uit het boek zorgt ervoor dat je in een enkele iteratie door de karakteristieke matrix de volledige signatuurmatrix kan berekenen.
     - In feite bestaat de kolom van een file binnen de signatuurmatrix uit niets meer dan de kleinst mogelijke waardes die de hashfuncties konden outputten voor de verzameling van shingles uit een bestand.
     - Het komt daarom op hetzelfde neer om enkel hashes te berekenen van de 'shingle-hashes' en vervolgens voor iedere hashfunctie de kleinste outputwaarde te selecteren.

Cedric Mingneau (1232611)
William Thenaers (1746077)

2. *Present the final architecture of the pipeline, preferably by means of a visualization. Draw specific attention to the input and output of each of the MapReduce stages involved. Clearly indicate the key used during reduce phases and describe the logic of non-trivial map operations.*

Onderstaand schema, Figure 1, geeft visueel de flow weer zoals de pipeline eruit ziet in Apache Beam. Daaronder een beschrijving van de meest prominente stappen en hun input, intermediary en ouput key-value pairs. Het schema geeft deze pairs ook weer met een iets concreter voorbeeld.

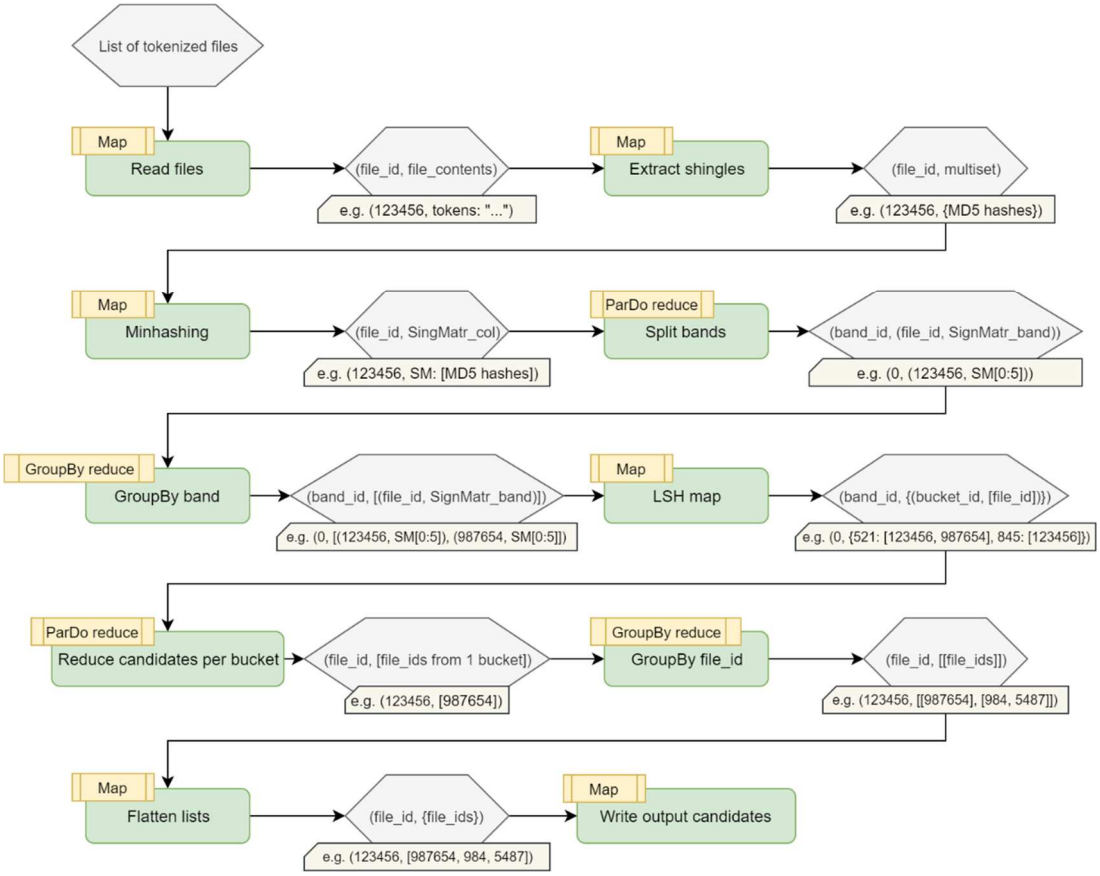


Figure 1 MapReduce dataflow volgens Apache Beam

Beschrijving

1. Input files lezen
   Input: (list_of_file_paths)
   a. Lijst van files in queue;
2. **Voor elke file:** K-shingles opstellen van tokens
   Deze stap kan apart op elke node worden uitgevoerd en komt overeen met een Map-fase.
   Input: (list_of_file_paths)
   Intermediary: (filename, file_contents)
   Intermediary: (filename, shinglelist: [shingles])
   Output: (filename, multiset: {shingle_hashes})
   a. K factor bepaald door input parameter;
   b. Lijn per lijn inlezen van een file
      - Omzetten naar tokens (reeds gebeurd);
      - Negeren van irrelevante whitespace tokens;
      - *NAME* tokens omzetten naar vb "$" in plaats van de naam zelf;

Cedric Mingneau (1232611)
William Thenaers (1746077)

- Andere tokens opnieuw aggregeren met hun oorspronkelijke waarde;

c. Elke *k* tokens samenvoegen in shingle en hashen naar de hashset (mulitset) van de ingelezen file;
   - De implementatie gaat uit van een waarde van 7 voor k, waarbij er dan 7 individuele characters worden samengevoegd om een shingle te vormen.

d. Goede hashfunctie nodig die groot genoeg is om collisions op shingles te vermijden.
   - MD5 werd in de implementatie voor alle hashes gebruikt.

3. **Voor elke file:** Minhashing

   Voor de volgende stap, kan een node telkens een afgewerkte file uit de vorige stap nemen en die remappen naar zijn signature matrix representatie. Aangezien files geen onderlinge afhankelijkheden hebben, kunnen alle files parallel behandeld worden.

   Input: (filename, multiset: {shingle_hashes})

   Output: (filename, [signature_matrix_col])

   a. Permutatiehashes opstellen (zelfde hashes worden gebruikt op elke node)
      - Vb `pHash = (value) => MD5(id + value)`
        Met *id* het id van de permutatie en *value* de waarde die gehasht moet worden.
      - De implementatie gebruikt 100 verschillende permutatiehashes.

   b. Signature Matrix samenstellen
      - Normaal gebeurt dit door een permutatie te maken van de indices in de bit vector representatie van de shingle multiset. Het aantal mogelijke shingles is wellicht zo groot dat dit niet mogelijk blijkt. In plaats daarvan lopen we bij elke file enkel de hashes in de multiset af en kijken we welk van deze hashes een minimum waarde oplevert door er *pHash* op uit te voeren. Deze minima voor elke *pHash* vormen dan de Signature Matrix.
      - Voorbeeld implementatie:

```
# For each col/file in SignMatrix
for col in range(len(SignMatrix[0])):
    # For each hashed shingle in the file's multiset
    # This is the domain we are interested in
    for x in hashed_shingle_multiset[col]:
        # Iterate over permutation hashes (rows from SM)
        for row, h in enumerate(pHashes):
            hashval = h(x)
            # Check if the new value is lower, and replace
            if hashval < SignMatrix[col][row]:
                SignMatrix[col][row] = hashval
```

4. **Herverdelen van Signature Matrices over nodes**: Locality Sensitive Hashing

   Het LSH-proces in onafhankelijk van welke files of banden *b* beschouwd worden. We kunnen deze dan ook opsplitsen en op elke node een set van buckets opstellen die de gehashte banden bevat. Daarna kunnen de buckets van elke node gereduceerd worden naar één set van buckets die aangeven hoe groot de kans is dat files in dezelfde buckets zitten ook effectief gelijk zijn met een bepaalde kans *t* (threshold).

   Input: (filename, [signature_matrix_col])

   Intermediary: (band, (filename, signature_matrix_band))

   Output: (band, buckets: [file_names])

   a. Kiezen van aantal banden en rijen per band (*b* en *n*)

Cedric Mingneau (1232611)
William Thenaers (1746077)

- Bepaling van deze factoren zorgen voor een verhouding tussen false-positives en -negatives. We verdeelden de signature matrix in 20 banden van elk 5 rijen.

b. Elke node stelt buckets op voor de LSH-hash functie
c. Splitsen van de Signature Matrices van elke file in banden als tussenresultaat, daarna worden de nieuwe items gegroepeerd volgens band id. Elk van deze stukjes van de Signature matrix kan namelijk apart beschouwd worden. Een node neemt alle stukjes voor 1 band en hasht deze naar buckets met de LSH-hashfunctie (ook MD5);
d. Elke band krijgt zijn eigen buckets toegekend.

5. **Reduceren** van buckets over elke node

   Als laatste stap voegen we alle buckets (per band) van elke node samen en krijgen we een set van verschillende lijsten. Elk item in dezelfde lijst had een band gemeenschappelijk met elk ander item en de kans dat deze items effectief gelijk zijn is minstens *t*, de kans die we in de vorige stap beschouwden.

   Input: (band, buckets: [file_names])

   Output: (file_name, candidates: [files])

   a. Alle overeenkomstige buckets van dezelfde band over elke node kunnen worden samengevoegd;
   b. Indien items voorkomen in dezelfde bucket, is de kans dat ze *similar* zijn, bepaald door de *b* en *n* factors bij het kiezen van banden met een threshold *t*. Deze items kunnen dan ook nader vergeleken worden om de echte gelijkwaardigheid te verifiëren.
   c. Voor elke file gaan we na met welke andere file deze in een bucket zat en stellen we een lijst op van die files, met name als files A, B en C in dezelfde bucket zitten, voegen we een nieuwe output toe voor A met een lijst bestaande uit B en C, B met A en C, en C met A en B.
   d. Deze items kunnen dan opnieuw gereduceerd worden door te groeperen op keys (defile identifier). Elke file vormt dan een kandidaatpaar met elke andere file dat er in de lijst achter staat.

Cedric Mingneau (1232611)
William Thenaers (1746077)

3. *Discuss the results produced by your implementation, describe potential improvements and issues encountered along the way*

   - De output resulteert in een bestand dat op elke lijn een filenaam heeft staan, gevolgd door een lijst met andere filenamen.

```
2375523: [2375494, 2375553]
2201211: [2201219]
2409328: [2409348, 2409259]
3587177: [3587126, 3587166, 3587164, 3587162, 3587130, 3587161, 3587174,
2297256: [2297280, 2297253, 2297198]
2575763: [2575830, 2575772]
2299087: [2299123, 2298352, 2298854, 2298468, 2299321, 2299164, 2298828]
2201831: [2201824, 2202454, 2202633, 2202169, 2201978, 2201930, 2201018,
2292224: [2292210, 2292243, 2292220, 2292188]
2557990: [2418324, 2557984, 2418350, 2557980]
```

Figure 2 De finale output: telkens een bestand gevolgd met een lijst van kandidaten die mogelijk gelijkaardig zijn.

   - Figure 3 vergelijkt de kandidaten uit regel 1 hierboven. We zien dat deze inderdaad grotendeels overeenkomen en we kunnen dus besluiten dat de studenten in kwestie de code van elkaar hebben overgenomen.

```
# write your code here
word_1=input("give a word")
word_2=input("give a word")
total=0
for i in range(len(word_1)):
    for j in range(len(word_2)):
        if word_1[i]==word_2[j]:
            total+=1
            if total==len(word_1):
                print(word_1 + " and " + word_2 + " are anagrams")
```

```
# write your code here
word_1=input("give a word")
word_2=input("give a word")
total=0
for i in range(len(word_1)):
    for j in range(len(word_2)):
        if word_1[i]==word_2[j]:
            total+=1
            if total==len(word_1):
                print(word_1 + " and " + word_2 + " are anagrams")
                else:
                print(word_1 + " and " + word_2 + " are not anagrams")


```

```
# write your code here
word_1=input("give a word")
word_2=input("give a word")
total=0
for i in range(len(word_1)):
    for j in range(len(word_2)):
        if word_1[i]==word_2[j]:
            total+=1
            if total==len(word_1):
                print(word_1 + " and " + word_2 + " are anagrams")
            else:
                print(word_1 + " and " + word_2 + " are not anagrams")
```

Figure 3 De eerste kandidaten uit Figure 2 vergeleken.

Cedric Mingneau (1232611)
William Thenaers (1746077)

- De kans dat een kandidaat key-value paar effectief voldoende gelijkenissen vertoont hangt af van het aantal banden en het aantal rijen per band dat we kozen namelijk 100 permutaties die verdeeld werden over 20 banden met 5 rijen elk.
  - We hebben de pipeline initieel uitgevoerd met een shinglegrootte van 7, 100 permutatiehashes en 20 banden van elk 5 rijen.
  - Alles opnieuw uitvoeren met een shinglegrootte van 4, 200 permutatiehashes en 25 banden van elk 8 rijen, resulteert in ongeveer dezelfde output. We zagen dat sommige bestanden enkele kandidaten bij kregen waar de code ook zeer gelijkaardig leek, maar sommige regels van plaats wisselden. Deze kunnen we nog steeds als plagiaat beschouwen. Een goede verhouding tussen de paramaters lijkt dus nodig om optimale kandidaatparen te vinden.
- Na het berekenen van de gelijkaardige files, kunnen ze nader worden vergeleken om precieze verschillen en overstemmingen vast te stellen.
- Eventueel zouden we ook de signatuurmatrices van elke file als tussenresultaat kunnen opslaan, zodat bij het vergelijken van nieuwe files de signatuurmatrices van de reeds beschouwde files terug ingelezen kan worden. Dit voorkomt dat alle 8328 bestanden opnieuw geparsed worden. Dit neemt namelijk het grootste deel van het proces in beslag, zoals te zien in Figure 5 op een volgende pagina.
- De huidige uitvoering op GCS duurt tussen de acht en de negen minuten, afhankelijk van het aantal workers dat automatisch wordt toegekend. Dezelfde pipeline lokaal uitvoeren met de DirectRunner (op een snelle 8 core machine), duurt ongeveer dubbel zo lang, maar hierbij wordt de input ook van het GSC gehaald en niet lokaal. Dit is nog steeds veel sneller dan elk bestand met elk ander bestand te vergelijken, maar de voordelen van GCS en Apache Beam komen zo nog meer tot uiting.

Job summary

Job name	bda-plagiarism
Job ID	2019-11-12_07_27_21-12269026941372379991
Region	europe-west1
Job status	Succeeded
SDK version	Apache Beam SDK for Java 2.14.0
Job type	Batch
Start time	12 November 2019 at 16:27:22 UTC+1
Elapsed time	9 min 34 sec
Encryption type	Google-managed key

Figure 4 Google Cloud Service dataflow job summary

- Om verder parallellisme mogelijk te maken, zouden we de verwerking van het minhashing algoritme kunnen opsplitsen tot op shingle niveau. Onze opsplitsing loopt momenteel tot op document niveau. Parallelliseren over de shingles van ieder document zou inhouden dat elke multiset opnieuw opgesplitst zou worden. Zo kan met een 'Parallel Do' elke permutatiehash berekenen op elke aparte shingle hash en die toevoegen aan een tussentijdse output. Deze kunnen vervolgens gecombineerd worden met file_id en permutatie_id om elke overeenkomstige hash te reduceren en het minimum te vinden voor dat permutatie_id. Het reduceren zou dan een custom Combiner functie kunnen zijn. In de stap erna kunnen we nog steeds hetzelfde doen door deze minima per band te groeperen voor elke file en te hashen naar buckets.

Cedric Mingneau (1232611)
William Thenaers (1746077)

Figure 5 Google Cloud Service Dataflow job diagram