Similitud entre documents (Part II)

Guillem Vidal

6 de desembre de 2024

1 Introducció

M'ha semblat adient modificar el `MapReduce` per tal d'optimitzar més les operacions, i fer-lo més genèric de manera que pogués acceptar més diversitat de tipus.

La funció que genera tot el funcionament amb actors es diu `groupMapReduce`, i té la següent signatura:

```
def groupMapReduce[K, A, B, C](
  input: Iterable[A],
  compute: A => Iterable[B],
  key: B => K,
  mapper: B => C,
  reducer: (C, C) => C,
  nmappers: Int = 16,
  nreducers: Int = 16
): Map[K, C]
```

El paràmetre `compute` és redundant: permet que durant el mapeig es generi una sèrie de valors a agrupar i reduir, que no sigui una relació 1 a 1 amb l'input, sinó 1 a N (opcionalment).

Això és útil quan hem de calcular les aparicions de les paraules en els documents, per exemple, ja que podem passar com a input els documents i generar totes les combinacions a dins del `groupMapReduce` aprofitant les seves aptituds de paral·lelisme.

2 Demostracions

2.1 La funció `timeMeasurement`

L'he definida de la següent manera:

```
def timeMeasurement[A](
  compute: Unit => A,
  name: String = "Function"
): A = {
```

```
  val beg = System.currentTimeMillis()

  val ret = compute()

  val end = System.currentTimeMillis()

  val elapsed = (end - beg).toFloat / 1000;
  println(s"$name took $elapsed seconds!")

  return ret
}
```

No he volgut utilitzar la sintaxi especial per l'avaluació *lazy* de les expressions, perquè m'agrada la claredat que el que s'està enviant és una expressió, no el valor d'aquesta.

2.2 Nombre promig de referències

`112 references on average.`

2.3 *Query* de recomanació mitjançant PR

Si busquem per 'Guerra', aquests són els primers resultats:

```
Natal (Rio Grande do Norte),
Heinkel He 280,
Fokker F.VII,
Raymond Collishaw,
Yokosuka MXY-7,
Diàspora basca,
Països àrabs,
Edat Contemporània als Països Catalans,
Història de Bielorússia,
...
```

2.4 Pàgines que s'assemblen sense referenciar-se

Donada una *query* de 'Guerra' i agafant els 100 primers elements retornats, obtinc els següents resultats:

```
(Economia de la Unió Soviètica, Èxode rural),
(Història d'Estònia, Èxode rural),
(Persona desplaçada, Èxode rural),
(Feixisme italià, Èxode rural),
(Història d'Amèrica, Èxode rural),
(Llista de diàspores, Èxode rural),
...
```

2.5 Eficiència

La màquina usada té una CPU Intel Core i7 amb 8 nuclis.

Els resultats de l'eficiència de cada operació, segons els *mappers* i *reducers* usats, són els següents, les unitats en segons:

Mappers	**Reducers**	**Load**	**Query**	**Similar**
1	1	15.88	0.729	1.788
1	8	15.198	0.615	2.05
1	16	15.582	1.863	1.757
1	32	15.298	0.36	1.673
1	64	14.579	0.387	1.664
8	1	5.103	0.407	1.259
8	8	5.087	0.21	0.779
8	16	5.244	0.212	0.801
8	32	5.608	0.22	1.108
8	64	4.857	0.286	1.384
16	1	5.072	0.426	1.251
16	8	6.079	0.251	1.076
16	16	4.92	0.263	0.837
16	32	5.229	0.208	1.028
16	64	4.926	0.243	2.197
32	1	4.809	0.42	1.242
32	8	5.739	0.234	0.682
32	16	4.834	0.21	0.835
32	32	5.877	0.224	1.047
32	64	4.937	0.219	1.31
64	1	4.947	0.41	2.14
64	8	4.845	0.244	0.698
64	16	4.925	0.21	0.829
64	32	5.157	0.219	1.043
64	64	4.831	0.265	1.349

Podem veure un increment d'eficiència substancial quan comencem a utilitzar múltiples actors, sobretot pel que fa a *mappers*. Però a mesura que n'afegim els beneficis respecte a la quantitat anterior no incrementen gaire. Fins al punt que veiem que usar-ne 32 o 64 és completament indiferent.

És normal que l'increment de l'eficiència comença a decaure quan ja hem sobrepassat els nuclis de la màquina (8). Això pel que fa a operacions intenses de CPU, si es tracta d'entrada i sortida barrejat amb operacions de CPU, possiblement quedi més optimitzat amb encara més actors.

3 Estructura

3.1 QueryDocument

Un fitxer que conté les operacions per a realitzar *queries*:

```
case class WikiContents(freqs: Map[String, Int], refs: Set[String])
case class RankInformation(rank: Double, refs: Set[String])

case class QueryDocument(wiki: Map[String, WikiContents], wordcount: Int) {

  final val brakefactor = 0.85
  final val tolerance = 1e-3

  @tailrec
  final def pagerank(
    ranking: Map[String, RankInformation],
    references: String => Set[String]
  ): Map[String, RankInformation] = {

    val newRanking = MapReduce.groupMapReduce(
      ranking.values,
      (info: RankInformation) => info.refs.toIterable
        .map(ref => (ref, info.rank / info.refs.size)),
      (pair: (String, Double)) => {
        val (title, _) = pair
        title
      },
      (pair: (String, Double)) => {
        val (_, rank) = pair
        rank
      },
      (lhs: Double, rhs: Double) => lhs + rhs
    )

    val head = (1 - brakefactor).toDouble / newRanking.size
    var diff = 0.0

    val dampedRanking = newRanking.map(keyval => {
      val (title, rank) = keyval
      val dampedRank = head + brakefactor * rank

      diff += Math.abs(ranking(title).rank - dampedRank)
      (title, RankInformation(dampedRank, references(title)))
    })

    if (diff < tolerance) {
      return dampedRanking
    }
```

```
    return pagerank(dampedRanking, references)
  }

  def query(query: String): List[String] = {

    val keywords = ngrames(Iterable(query), wordcount).toList

    // Filtrem pels documents que contenen la query.
    val references = wiki.view.filter(keyval => {
      val (_, content) = keyval
      keywords.exists(keyword => content.freqs.contains(keyword))
    }).map(keyval => {
      val (title, content) = keyval
      (title, content.refs)
    }).to(mutable.Map)

    // Limitem les referencies, perque es quedin dins del subset.
    for ((page, refs) <- references)
    references.put(page, refs.filter(references.contains))

    val shared = 1.0 / references.size
    val initialRanking = references.view.map(keyval => {
      val (title, refs) = keyval
      (title, RankInformation(shared, refs))
    }).toMap

    val finalRanking = pagerank(initialRanking, references)

    finalRanking.view.map(keyval => {
      val (title, info) = keyval
      (title, info.rank)
    }).toList.sortBy(keyval => {
      val (_, rank) = keyval
      rank
    }).map((keyval) => {
      val (title, _) = keyval
      title
    })
  }

}
```

Com a plus, el *pagerank* és *tail recursive*.

3.2 SimilarPages

És un fitxer que conté les operacions per comparar les pàgines més similars, que no es referencien directament, mitjançant *inverse document frequency*.

```
case class SimilarPages(wiki: Map[String, WikiContents], wordcount: Int) {
```

```
def computeAppearances(
  subset: Map[String, WikiContents]
): Map[String, Int] = {

  val keys = subset.keys.toList
  val appearances = MapReduce.groupMapReduce(
    keys.indices,
    (i: Int) => {
      val words = subset(keys(i)).freqs.keys

      (i until keys.length).flatMap(j => {
        val freqs = subset(keys(j)).freqs
        words.filter(word => freqs.contains(word))
        .map(word => (word, j))
      })
    },
    (keyval: (String, Int)) => {
      val (word, _) = keyval
      word
    },
    (keyval: (String, Int)) => {
      val (_, index) = keyval
      BitSet() + index
    },
    (lhs: BitSet, rhs: BitSet) => lhs.concat(rhs)
  )

  return appearances.view.mapValues(bits => bits.count(_ => true)).toMap
}

def topNonReferenced(
  query: String,
  limit: Int = 1000
): List[(String, String)] = {

  val documents = QueryDocument(wiki, wordcount).query(query)
  .slice(0, limit)

  val subset = documents.view
  .map(document => (document, wiki(document)))
  .to(mutable.Map)

  val appearances = computeAppearances(subset.toMap)

  val docs_idf = subset.view.map(keyval => {
    val (title, contents) = keyval
    val freqs = contents.freqs.map(keyval => {
      val (word, freq) = keyval
      val count = appearances(word)
```

```
    val idf = subset.size.toDouble / count
    (word, freq * idf)
  })
  (title, freqs)
}).toMap

val similarPages = MapReduce.groupMapReduce(
  docs_idf,
  (fst: (String, Map[String, Double])) => {
    val (fst_title, _) = fst
    val fst_refs = subset(fst_title).refs
    docs_idf.dropWhile(sec => {
      val (sec_title, _) = sec
      sec_title != fst_title
    }).drop(1)
    .filter(sec => {
      val (sec_title, _) = sec

      if (fst_refs.contains(sec_title)) {
        false
      } else {
        val sec_refs = subset(sec_title).refs
        !sec_refs.contains(fst_title)
      }
    })
    .map(sec => (fst, sec))
  },
  pair => {
    val (fst, sec) = pair
    val (fst_title, _) = fst
    val (sec_title, _) = sec
    (fst_title, sec_title)
  },
  pair => {
    val (fst, sec) = pair
    val (_, fst_freqs) = fst
    val (_, sec_freqs) = sec

    Cosinesim.simil(fst_freqs, sec_freqs)
  },
  (_: Double, _: Double) => throw new Exception()
)

val sorted = similarPages.toList
.sortBy(keyval => {
  val (_, simil) = keyval
  -simil
})

return sorted.map(keyval => {
```

```
      val (docs, _) = keyval
      docs
    })
  }

}
```

3.3 Càrrega dels documents en memòria

```
val wordcount = 1
val dir = FileSystems.getDefault.getPath ("viqui_files")
val docs = asScala(Files.list(dir).iterator())
  .to(Iterable)
  .map(_.toString)

val wiki = TimeMeasurement.timeMeasurement(_ => MapReduce.groupMapReduce (
  docs,
  (filename: String) => Iterable (parseViquipediaFile (filename) ),
  (file: ResultViquipediaParsing) => file.titol,
  (file: ResultViquipediaParsing) => WikiContents (
    freq (ngrames (Iterable (file.contingut), wordcount) ),
    file.refs.map (ref => ref.substring (2, ref.length - 2) ).toSet
  ),
  (_: WikiContents, _: WikiContents) =>
    throw new Exception ("Non-unique values!"),
), "File fetching")
```