recursive_reducers

November 15, 2019

1 Reducers récursifs

J'utilise volontiers une terminologie découverte chez Microsoft pour illustrer une façon d'écrire le même calcul qui a un impact sur la facilité avec laquelle on peut le distribution : utiliser des comptes plutôt que des moyennes.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

Le notebook utilise des fonctions développées pour illustrer les notions, plus claires qu'efficaces.

1.1 Stream

Le map reduce s'applique à des jeux de données très grands. D'un point de vue mathématique, on écrit des algorithmes qui s'appliquent à des jeux de données infinis ou plutôt dont la taille n'est pas connu. Pour les distinguer des jeux de données, on les appelle des *flux* ou *stream* en anglais.
En aparté, écrits pour être parallélisés, ces traitements ont la particuliarité de ne pas conserver l'ordre dans lequel il traite les données. C'est particulièrement vrai lorsque le jeu de données est divisé sur plusieurs disques durs. Il est impossible de choisir un morceau en premier.

1.2 Mapper

Un *mapper* applique le même traitement à chaque observation du *stream* de façon indépendante.

[2]:
```
ens = [('a', 1), ('b', 4), ('a', 6), ('a', 3)]
```

[3]:
```
from sparkouille.fctmr import mapper
stream1 = mapper(lambda el: (el[0], el[1]+1), ens)
stream1
```

[3]:
```
<map at 0x1a222add470>
```

Le résultat n'existe pas tant qu'on ne demande explicitement que le calcul soit faut. Il faut parcourir le résultat.

[4]:
```
list(stream1)
```

[4]:
```
[('a', 2), ('b', 5), ('a', 7), ('a', 4)]
```

Et on ne peut le parcourir qu'une fois :

[5]:
```
list(stream1)
```

[5]:
```
[]
```

1.3 Coût du premier élément

Quand on a une infinité d'éléments à traiter, il est important de pouvoir regarder ce qu'un traitement donne sur les premiers éléments. Avec un mapper, cela correspond au coût d'un seul map.

[6]:
```
from sparkouille.fctmr import take
first = lambda it: take(it, count=1)
big_ens = ens * 100
```

[7]:
```
%timeit -n 1000 list(mapper(lambda el: (el[0], el[1]+1), big_ens))
```

```
124 µs ± 15.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[8]:
```
%timeit -n 1000 first(mapper(lambda el: (el[0], el[1]+1), big_ens))
```

```
2.46 µs ± 451 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

1.4 Reducer

Un vrai *reducer* réduit les éléments d'un ensemble, il ne répartit pas les données. En pratique, on réduit rarement un ensemble qu'on n'a pas distribué au préalable, comme avec un *groupby*. On ne réduit pas toujours non plus un ensemble à une seule ligne. On empile les opérations de streaming, on repousse également le moment d'évaluer. La distribution s'effectue selon une clé qui est hashée (voir Hash et distribution). La première lambda fonction décrit ce qu'est cette clé, le premier élément du couple dans ce cas.

[9]:
```
from sparkouille.fctmr import reducer
stream1 = mapper(lambda el: (el[0], el[1]+1), ens)
stream2 = reducer(lambda el: el[0], stream1, asiter=False)
stream2
```

[9]:
```
<generator object reducer at 0x000001A2229E3200>
```

[10]:
```
list(stream2)
```

[10]:
```
[('a', [('a', 2), ('a', 4), ('a', 7)]), ('b', [('b', 5)])]
```

Dans cet exemple, le *reducer* réduit chaque groupe à un seul résultat qui est l'ensemble des éléments. Quel est le coup du premier élément…

[11]:
```
def test2(ens, one=False):
    stream1 = mapper(lambda el: (el[0], el[1]+1), ens)
    stream2 = reducer(lambda el: el[0], stream1, asiter=False)
    return list(stream2) if one else first(stream2)

%timeit -n 1000 test2(big_ens)
```

```
1.75 µs ± 409 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[12]:
```
%timeit -n 1000 test2(big_ens, one=True)
```

```
720 µs ± 31.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

C'est plus court mais pas significativement plus court. Cela correspond au coût d'un tri de l'ensemble des observations et du coût de la construction du premier groupe.

1.5 Reducer et tri

Un stream est infini en théorie. En pratique il est fini mais on ne sait pas si un ou plusieurs groupes entiers tiendraient en mémoire. Une façon de faire est de limiter la présence des données en mémoire à un seul groupe et pour cela, il faut d'abord trier les données selon les clés. Ce n'est pas indispensable mais dans le pire des cas, c'est une bonne option. On pourrait avoir un stream comme suit :

[13]:
```
pas_cool = [(chr(int(c) + 96), i) for i, c in enumerate(str(11111111 ** 2))]
pas_cool
```

[13]:
```
[('a', 0),
 ('b', 1),
 ('c', 2),
 ('d', 3),
 ('e', 4),
 ('f', 5),
 ('g', 6),
 ('h', 7),
 ('g', 8),
 ('f', 9),
 ('e', 10),
 ('d', 11),
 ('c', 12),
 ('b', 13),
 ('a', 14)]
```

Le groupe a est au début et à la fin, si on regroupe en mémoire, le groupe associé à a doit rester en mémoire du début à la fin. On ne sait jamais si un groupe ne va pas réapparaître plus tard. En triant, on est sûr.

1.6 Un autre map

On ajoute un dernier map qui fait la somme des éléments de chaque groupe.

[14]:
```
def sum_gr(key_gr):
    key, gr = key_gr
    return key, sum(e[1] for e in gr)

stream1 = mapper(lambda el: (el[0], el[1]+1), ens)
stream2 = reducer(lambda el: el[0], stream1)
stream3 = map(sum_gr, stream2)
stream3
```

[14]:
```
<map at 0x1a222addf98>
```

[15]:
```
list(stream3)
```

[15]:
```
[('a', 13), ('b', 5)]
```

1.7 Combiner ou join

Un *combiner* ou *join* permet de fusionner deux bases de données qui ont en commun une clé.

[16]:
```
from sparkouille.fctmr import combiner
stream1 = mapper(lambda el: (el[0], el[1]+1), ens)
stream2 = reducer(lambda el: el[0], stream1)
stream3 = map(sum_gr, stream2)
```

```
stream4 = mapper(lambda el: (el[0], el[1]+10), pas_cool)
comb = combiner(lambda el: el[0], stream3, lambda el: el[0], stream4)
comb
```

[16]:
```
<generator object combiner at 0x000001A222ADFAF0>
```

[17]:
```
list(comb)
```

[17]:
```
[(('a', 13), ('a', 10)),
 (('a', 13), ('a', 24)),
 (('b', 5), ('b', 11)),
 (('b', 5), ('b', 23))]
```

Le coût du premier élément est un peu plus compliqué à inférer, cela dépend beaucoup des données.

[18]:
```
def job(ens, ens2, one=False, sens=True):
    stream1 = mapper(lambda el: (el[0], el[1]+1), ens)
    stream2 = reducer(lambda el: el[0], stream1)
    stream3 = map(sum_gr, stream2)
    stream4 = mapper(lambda el: (el[0], el[1]+10), ens2)
    if sens:
        comb = combiner(lambda el: el[0], stream3, lambda el: el[0], stream4)
    else:
        comb = combiner(lambda el: el[0], stream4, lambda el: el[0], stream3)
    return list(comb) if one else first(comb)

%timeit -n 1000 job(big_ens, pas_cool)
```

```
2.97 µs ± 793 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[19]:
```
%timeit -n 1000 job(big_ens, pas_cool, sens=False)
```

```
3.15 µs ± 1.16 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[20]:
```
%timeit -n 1000 job(big_ens, pas_cool, one=True)
```

```
401 µs ± 6.73 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[21]:
```
%timeit -n 1000 job(big_ens, pas_cool, one=True, sens=False)
```

```
389 µs ± 10.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Il y a différentes façons de coder un *combiner*, l'une d'elle consiste à réduire chacun des deux streams puis à faire le produit croisé de chaque groupe assemblé.

1.8 Reducers récursifs

C'est pas loin d'être un abus de langage, disons que cela réduit la dépendance au tri. Un exemple.

[22]:
```
def sum_gr(key_gr):
    key, gr = key_gr
    return key, sum(e[1] for e in gr)
```

```
def job_recursif(ens):
    stream2 = reducer(lambda el: el[0], ens)
    stream3 = map(sum_gr, stream2)
    return list(stream3)

job_recursif(ens)
```

[22]:
```
[('a', 10), ('b', 4)]
```

Et maintenant, on coupe en deux :

[23]:
```
n = len(ens) // 2
job_recursif(ens[:n])
```

[23]:
```
[('a', 1), ('b', 4)]
```

[24]:
```
job_recursif(ens[n:])
```

[24]:
```
[('a', 9)]
```

Et maintenant :

[25]:
```
job_recursif( job_recursif(ens[:n]) + job_recursif(ens[n:]))
```

[25]:
```
[('a', 10), ('b', 4)]
```

Le job ainsi écrit est associatif en quelque sorte. Cela laisse plus de liberté pour la distribution car on peut maintenant distribuer des clés identiques sur des machines différentes puis réappliquer le *reducer* sur les résultats de la première salve. C'est d'autant plus efficace que le *reducer* réduit beaucoup les données. Il reste à voir le cas d'un *reducer* **non récursif**.

[26]:
```
def mean(ens):
    s = 0.
    for i, e in enumerate(ens):
        s += e
    return s / (i + 1)

def mean_gr(key_gr):
    key, gr = key_gr
    return key, mean(e[1] for e in gr)

def job_non_recursif(ens):
    stream2 = reducer(lambda el: el[0], ens)
    stream3 = map(mean_gr, stream2)
    return list(stream3)

job_non_recursif(ens)
```

[26]:
```
[('a', 3.3333333333333335), ('b', 4.0)]
```

[27]:
```
n = len(ens) // 2
job_non_recursif(ens[:n])
```

[27]:
```
[('a', 1.0), ('b', 4.0)]
```

[28]:
```
job_non_recursif(ens[n:])
```

[28]:
```
[('a', 4.5)]
```

[29]:
```
job_non_recursif( job_non_recursif(ens[:n]) + job_non_recursif(ens[n:]))
```

[29]:
```
[('a', 2.75), ('b', 4.0)]
```

Ce *job* ne doit pas être distribué n'importe comment.

[30]:
```

```