A tanulmány Klaus Berberich és Srikanta Bedathur Computing n-Gram Statistics in MapReduce cikkét dolgozza fel.

A cikk elérhető itt: http://arxiv.org/pdf/1207.4371.pdf

Győző Csóka (M7BVLS) (jelen tanulmány)
Polgár Ákos (prezentáció)
Zvara Zoltán (bemutató program)

n-Gram statisztikák kiszámítása MapReduce-ban

Bevezetés

A szövegelemzések egyik alapvető statisztikája a szövegben előforduló n-hosszú szavak vagy kifejezések, tetszőleges karaktersorozatok (vagyis n-gramok) előfordulásának gyakorisága. A bemutatott cikk[1] a MapReduce modellben [2] rendelkezésre álló elosztott adatfeldolgozási módszerrel végzett n-gram statisztikák készítésével foglalkozik.

Az n-gram statisztikák [3] fontosságát a szövegelemzésben jelzi, hogy a Google és a Microsoft is közzétett a webes tartalmak n-gram statisztikáit tartalmazó adathalmazokat. Azonban ezek a statisztikák a legfeljebb 5 szó hosszú n-gramokról szólnak, így idézetek, idiómák, keresésére, véleményelemzésre, vagy plágiumellenőrzésre nem alkalmasak.

A Hadoop MapReduce implementációjával végzett n-gram statisztikák készítéséhez több algoritmust is adaptáltak, azonban látni fogjuk, hogy ezek teljesítménye lényegesen javítható. A cikkben bemutatott suffix-σ algoritmus kihasználja a MapReduce modell csoportosítás és rendezés műveleteit, és a jelenleg használt algoritmusoknál (bizonyos karakterisztikánál 12-szer) hatékonyabb működésre képes.

Az alapprobléma

Alapfogalmak, jelölések

A V véges ábécé feletti szövegsorozatok univerzumát jelöljük S-sel.
Egy adott s sorozatnál $s =< s0, \ldots, sn-1 >$, ahol $si \in V, |s| = n$ jelölje s hosszát, s[i..j] jelölje s következő részsorozatát: $< si, \ldots, sj >$, és hivatkozzon si-re s[i].
r||s jelöje r és s sorozatok konkatenálását.

r az s prefixe($r \lhd s$) ha

$$\forall\, 0 \leq i < |r| : r[i] = s[i]$$

r az s szuffixe($r \rhd s$) ha

$$\forall\, 0 \leq i < |r| : r[i] = s[|s| - |r| + i]$$

r az s részsorozata($r \diamond s$) ha

$$\exists\, 0 \leq j < |s| : \forall\, 0 \leq i < |r| : r[i] = s[i + j]$$

r s-beli gyakorisága

$$f(r,s) = |\{\, 0 \leq j < |s| \mid \forall\, 0 \leq i < |r| : r[i] = s[i + j] \,\}| \,.$$

Az algoritmusok bemenetként szövegek egy halmazát kapják, ezt jelöljük D-vel.

MapReduce modell

A MapReduce egy programozási modell nagy adathalmazok feldolgozására párhuzamosan és egy szerverfürtön elosztottan.

A MapReduce tartalmaz egy map funkciót, amely szűrést és rendezést végez, valamint egy reduce funkciót, amely összegzi az eredményt. A MapReduce rendszer osztja el a feladatokat a szervereken párhuzamosan futtatva azokat, irányítva minden adatátvitelt, egyúttal hibatűrést is biztosít redundancián keresztül.

A MapReduce elve szerint a nagy adathalmazokon végzett műveleteket két fázisban célszerű elvégezni, először a map fázisban, hasonlóan a mi Haskell példánkhoz, a "szétszórt" adatokon egy műveletet végez el a rendszer, majd ezt aggregálja a Reduce fázisban a rendszer. Így nem kell a programozónak foglalkoznia a köztes eredményekkel, az elosztott adatokon sokkal egyszerűbb és gazdaságosabb előbb helyben elvégezni a Map fázist, majd valahol összegyűjteni a Reduce fázis eredményeit. [4]

A modellt a funkcionális programozásból ismert map és reduce funkciók inspirálták, bár a használatuk nem egészen ugyanaz a MapReduce rendszerben, mint az eredeti formában.

A MapReduce algoritmust számos különböző területen használják, például elosztott minta-alapú keresés, elosztott rendezés, web log statisztika feldolgozás, gépi tanulás. A Google-nél a MapReduce algoritmust használták a világháló indexének felépítésére. [5]

A megoldandó feladat

Adott D szövegek halmazához keressük meg az összes olyan n-gramot és hozzátartozó előfordulási gyakoriságot, ahol a gyakoriság legalább τ, az n-gramok hossza legfeljebb σ.

Egy példán bemutatva: tartalmazzon D három szöveget:

$$d1 = < a\ x\ b\ x\ x >$$
$$d2 = < b\ a\ x\ b\ x >$$
$$d3 = < x\ b\ a\ x\ b >$$

Legyen τ = 3 és σ = 3, ekkor a következő megoldást kell kapnunk:

$$< a >: 3$$
$$< b >: 5$$
$$< x >: 7$$
$$< a\ x >: 3$$
$$< x\ b >: 4$$
$$< a\ x\ b >: 3$$

Korábbi megoldások

Naiv algoritmus

Ez a MapReduce modell bemutatásához gyakran használt példaprogram egy szövegben szereplő összes szó gyakoriságának kiszámításához. A pszeudokódban látható, hogy a map függvény kiválasztja az összes legfeljebb σ hosszúságú n-gramot, és elhagyja azokat, amik τ-nál kevesebbszer szerepelnek.

Algorithm 1: NAÏVE

```
   // Mapper
1  map(long did, seq d) begin
2      for b = 0 to |d| − 1 do
3          for e = b to min(b + σ − 1, |d| − 1) do
4              emit(seq d[b..e], long did)

   // Reducer
1  reduce(seq s, list<long> l) begin
2      if |l| ≥ τ then
3          emit(seq s, int |l|)
```

Legrosszabb esetben, azaz σ >|d| esetén a Naive algoritmus $O(|d|^2)$ kulcs-érték párt választ ki d szövegből, minden pár O(|d|) hosszú, így ez a módszer $O(|d|^3)$ byte transzfert végez a map és reduce fázisok között.

Apriori alapú módszerek - bevezetés

Az ötlet, amit felhasználunk a naiv algoritmus javításához az ún. apriori elv, ami formálisan:

$$r \diamond s \Rightarrow cf(r) \geq cf(s)$$

Vagyis ha r részsorozata s-nek, akkor s legfeljebb annyiszor fordul elő a szövegben, mint r.

Apriori-keresés

Az apriori-keresés többször olvassa végig a szöveget. A k. olvasásnál a módszer meghatározza azon k-gramokat, amik legalább τ-szor szerepelnek a szövegben. Ehhez felhasználja a korábban meghatározott (k-1)-gramokat, hiszen az apriori elv miatt csak azon k-gramoknak lehet megfelelő előfordulási gyakoriságuk, amik az előző körben meghatározott lista elemeinek egy karakterrel való kibővítései. A metódus legfeljebb σ olvasás után befejezi működését (korábban akkor, ha valamelyik körben egyetlen megfelelő gyakoriságú k-gramot sem talál).

Algorithm 2: APRIORI-SCAN

```
int k = 1
repeat
    hashset<int[]> dict = load(output-(k − 1))
    // Mapper
1   map(long did, seq d) begin
2       for b = 0 to |d| − k do
3           if k = 1 ∨
4           (contains(dict, d[b..(b + k − 2)]) ∧
5            contains(dict, d[(b + 1)..(b + k − 1)])) then
6               emit(seq d[b..(b + k − 1)], long did)
    // Reducer
1   reduce(seq s, list<long> l) begin
2       if |l| ≥ τ then
3           emit(seq s, int |l|)
    k += 1
until isEmpty(output-(k − 1)) ∨ k = σ + 1;
```

A pszeudokód map függvényében látható, hogy azon k-gramokat, amihez nincs hozzátartozó (k-1)-gram kiszűrjük. A reduce függvény a naiv algoritmusban használtnak megfelelően összegzi a k. olvasás eredményeit.

A már ismertetett példán bemutatva az input harmadik olvasásánál az algoritmus map fázisa csak az (< a x b >, d_i) kulcs-érték párt választja ki, míg a több trigramot (pl.: < b x x >), amely tartalmaz egy túl ritka bigramot (pl.: < x x >) elhagyja.

Legrosszabb esetben azaz σ>|d| esetén az apriori keresés $O(|d|^2)$ kulcs-érték párt választ ki d szövegből, minden pár O(|d|) hosszú, így ez a módszer szintén $O(|d|^3)$ byte transzfert végez a map és reduce fázisok között.

Apriori-index

Algorithm 3: APRIORI-Index

```
int k = 1
repeat
    if k ≤ K then
        // Mapper #1
1       map(long did, seq d) begin
2           hashmap<seq, int[]> pos = ∅
3           for b = 0 to |d| − 1 do
4               add(get(pos, d[b..(b + k − 1)]), b)
5           for seq s : keys(pos) do
6               emit(seq s, posting(did, get(pos,s)))

        // Reducer #1
1       reduce(seq s, list<posting> l) begin
2           if cf(l) ≥ τ then
3               emit(seq s, list<posting> l)
    else
        // Mapper #2
1       map(seq s, list<posting> l) begin
2           emit(seq s[0..|s| − 2],
3                (r-seq, list<posting>) (s, l))
4           emit(seq s[1..|s| − 1],
5                (l-seq, list<posting>) (s, l))

        // Reducer #2
1       reduce(seq s, list<(seq, list<posting>)> l)
        begin
2           for (l-seq, list<posting>) (m, l_m) : l do
3               for (r-seq, list<posting>) (n, l_n) : l do
4                   list<posting> l_j = join(l_m, l_n)
5                   if cf(l_j) ≥ τ then
6                       seq j = m || ⟨n[|n| − 1]⟩
7                       emit(seq j, list<posting> l_j)

    k += 1
until isEmpty(output-(k − 1)) ∨ k = min(σ, K);
```

Második apriori elvű algoritmusunk nem olvassa többször az inputot, hanem egy invertált indexet[6] épít a gyakori n-gramokhoz. Az ábrán (Algorithm 3) megadott algoritmus első szakaszában (Mapper #1 és Reducer#1) létrehoz egy indexet az összes gyakori, legfeljebb K hosszú n-gram pozíciójához. Utána az ennél hosszabb n-gramok meghatározásához az első fázis kimenetét felhasználja. Egy gyakori k-gram meghatározásához (pl. < b a x >) a módszer összefűzi az egymással összefűzhető (k-1)-gramokat (pl.: a < b a > és < a x > párt). Ezt MapReduce-ban így érhetjük el (Mapper #2 és Reducer #2): a map függvény minden gyakori (k-1)-gramhoz kiválaszt két kulcs-érték párt. Az értékek maga a gyakori (k−1)-gram a címkelistájával (posting list). A kulcsok a (k-2) hosszú prefix és szuffix. A pszeudokódban a módszer az r-seq és l-seq altípusok segítségével tárolja, hogy a kulcs prefix vagy szuffix. A reduce függvény kiválasztja egy adott kulcshoz tartozó összes illeszkedő értéket, összefűzi a címkelistáikat és kiválasztja az eredmény k-gramot a hozzátartozó címkelistájával együtt, ha gyakorisága legalább τ. Két sorozat akkor összefűzhető, ha az egyik kulcsa prefix, a másiké szuffix. A beágyazott for ciklusokban a módszer ellenőrzi minden lehetséges kombinációját a sorozatoknak.

A korábbi példánkhoz visszatérve, K = 2 értékkel a módszer a harmadik iterációjánál csak egy megfelelő sorozatpárt lát (címkelistájukkal együtt), az < x > kulcshoz:

$$< a\ x >: < d1 : [0], d2 : [1], d3 : [2] >$$
$$< x\ b >: < d1 : [1], d2 : [2], d3 : [0, 3] >.$$

Ezek összefűzésével az apriori-index megtalálja az egyetlen gyakori trigramot címkelistájával:

$$< a\ x\ b >: < d1 : [0], d2 : [1], d3 : [2] >.$$

Az algoritmus implementálásakor felmerülő nehézség, hogy a címkelisták száma és mérete a gyakorlatban igen nagy lehet. Továbbá az megfelelő sorozatok összefűzéséhez ezeket a listákat bufferelni kell, és kezelni kell a problémát, ha túl sok memóriát foglalnak le.
Az elvégzendő iterációk számát a σ paraméter vagy a leghosszabb gyakori n-gram hossza határozza meg. Legrosszabb esetben az apriori-index $O(|d|^2)$ kulcs-érték párt választ ki d szövegből, minden pár O(|d|) hosszú, így ez a módszer szintén $O(|d|^3)$ byte transzfert végez a map és reduce fázisok között.

A Suffix-σ algoritmus

Az eddig megismert algoritmusoknál nagy mennyiségű adatot kell továbbítani és rendezni, illetve sok MapReduce feladatot vagy sok memóriát használnak fel. A bemutatásra kerülő suffix-σ módszer kikerüli ezeket a problémákat: egyetlen MapReduce feladatot használ, kis mennyiségű adatot továbbít és kis memóriafelhasználású.

Vizsgáljuk meg újra a naiv algoritmus map függvényét. Az alábbi 3 n-gram kiválasztása (< b a x >, < b a >, < b >) láthatóan pazarló, hiszen az elsőből meghatározható a másik kettő, mint prefixek.

A szuffix-tömböt[7] használó és más szövegelemző módszerek használják ezt az ötletet. Ebből a megfigyelésből következik, hogy elég kiválasztanunk az n-gramok egy részhalmazát. Pontosabban elég kiválasztani a szöveg minden pozíciójához egyetlen kulcs-érték párt a pozíciótól kezdődő szuffixet használva kulcsként. Ezek a szuffixek σ hosszúra vághatók - ez adja a módszer nevét.

Algorithm 4: SUFFIX-σ

```
   // Mapper
1  map(long did, seq d) begin
2      for b = 0 to |d| − 1 do
3          emit(seq d[b..min(b + σ − 1, |d| − 1)], long did)

   // Reducer
   stack<int> terms = ∅
   stack<int> counts = ∅
1  reduce(seq s, list<long> l) begin
2      while lcp(s,seq(terms)) < len(terms) do
3          if peek(counts) ≥ τ then
4              emit(seq seq(terms), int peek(counts))
5          pop(terms)
6          push(counts, pop(counts) + pop(counts))
7      if len(terms) = |s| then
8          push(counts, pop(counts) + |l|)
       else
10         for i = lcp(s, seq(terms)) to |s| − 1 do
11             push(terms, s[i])
12             push(counts, (i == |s| − 1 ? |l| : 0))

1  cleanup() begin
2      reduce(seq ∅, list<long> ∅)

   // Partitioner
1  partition(seq s) begin
2      return hashcode(s[0]) mod R

   // Comparator
1  compare(seq r, seq s) begin
2      for b = 0 to min(|r|, |s|) − 1 do
3          if r[b] < s[b] then
4              return +1
5          else if r[b] > s[b] then
6              return −1
7      return |s| − |r|
```

Egy adott r n-gram előfordulási gyakoriságának meghatározásához össze kell számolnunk, hogy a map fázis alatt kiválasztott szuffixek közül mennyinek prefixe r. Ahhoz, hogy ezt egyetlen MapReduce job-bal megtegyük, biztosítanunk kell, hogy minden szóba jöhető szuffixet lát ugyanaz a Reducer. Ezt úgy érhetjük el, ha a szuffixek partícionálását az első karakterük alapján végezzük. Ez garantálja, hogy egyazon reducer megkapja az összes azonos karakterrel kezdődő szuffixet. Ez a reducer lesz felelős azért, hogy meghatározza az összes ilyen terminálissal kezdődő n-gram előfordulási gyakoriságát. Egy módszer ennek eléréséhez, hogy felsoroljuk egy megkapott szuffix összes prefixét és összeszedjük ezek gyakoriságát a memóriában. Mivel nem tudhatjuk, hogy egy n-gramot egy más, még nem látott beérkező szuffix is reprezentálhat, nem tudjuk elég korán kiválasztani az előfordulási gyakoriságával. Ezért nyilván kell tartsunk sok n-gramot, és ez sok memóriát igényel.
A memóriaigény csökkentéséhez a következő ötletet használjuk fel: a kulcs-érték párok rendezési sorrendje, amiben a reducerek megkapják őket, befolyásolható. A suffix-σ fordított lexikografikus sorrendben rendezi a kulcs-érték párokat, formálisan:

$$r < s \Leftrightarrow (|r| > |s| \wedge s \lhd r) \vee$$
$$\exists\, 0 \leq i < min(|r|,|s|) : r[i] > s[i] \wedge \forall\, 0 \leq j < i : r[j] = s[j]\,.$$

Ez azért hasznos, mert minden egyes bejövő szuffix reprezentálja az összes olyan n-gramot, amit ezek prefixeként kaphatunk. Jelölje s az aktuális input szuffixet. A fordított lexikografikus rendezés biztosítja, hogy biztosan kiválaszthatunk bármely r n-gramot ahol r < s, hiszen egyik még látatlan input szuffix sem reprezentálhatja r-et. Ekkor csak azokat az n-gramokat kell nyilvántartanunk, amelyek s prefixei. Példánkon bemutatva: a b-vel kezdődő szuffixekért felelős reducer a következő listát kapja:

$$< b\ x\ x\ >:< d1 >$$
$$< b\ x\ >:< d2 >$$
$$< b\ a\ x\ >:< d2, d3 >$$
$$< b\ >:< d3 >$$

Amikor a harmadik szuffixet, < b a x > -et látjuk, azonnal véglegesíthetjük < b x > előfordulási gyakoriságát, hiszen már további még látatlan szuffix nem reprezentálhatja.

Erre a megfigyelésre építve hatékony nyilvántartást építhetünk az aktuális s szuffix prefixeihez, előfordulási gyakoriságukat két verem segítségével számolva. Az első veremben (nevezzük terms-nek) az s-et alkotó terminálisokat tartjuk. A második verem (counts) s minden prefixéhez tart egy számlálót. A reduce függvény hívásai között két invariánst biztosítunk. Először is a két verem azonos m méretű. Másodjára, sum(counts[j]) jelentése, hogy eddig hányszor láttuk az inputban a <terms[0], .. ,terms[i]> n-gramot. Ezen szabályok betartásához, amikor egy s szuffixet feldolgozunk az inputból először is egyszerre kiveszünk egy-egy elemet mindkét veremből, amíg a terms tartalma s prefixe nem lesz. Minden egyes kivételnél kiválasztjuk a terms tartalmát és a counts legfelső elemét, ha utóbbi eléri τ értékét. Amikor a counts-ból kiveszünk egy elemet, annak értékét hozzáadjuk az új legfelső értékhez. Ezt folytatva frissítjük terms-et, hogy tartalma megegyezzen s szuffixszel. Az utolsón kívül minden terminus hozzáadásánál egy 0-t írunk counts-ba. Az utolsó terminálisnál s gyakoriságát írjuk. Az alábbi ábra bemutatja hogyan változik a két verem tartalma az input feldolgozása során:

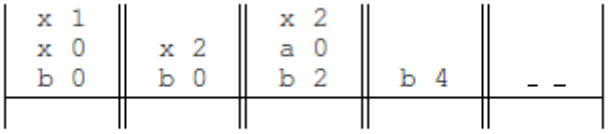

Fig. 1. SUFFIX-σ's bookkeeping illustrated

Az algoritmus pszeudokódjában a map függvény kiválasztja minden szövegből az összes szuffixet σ hosszúságúra csonkolva. A reduce függvény fordított lexikografikus sorrendben olvassa a szuffixeket és végrehajtja a nyilvántartást két veremmel, tárolja az n-gramokat (terms) és a előfordulási gyakoriságukat (counts). A seq függvény visszaad egy n-gramot a terms veremnek megfelelően. Az lcp függvény visszadja a leghosszabb két n-gram leghosszabb közös prefixét. A partition függvény biztosítja, hogy az első terminális szerint kerüljenek a szuffixek a konkrét reducerhez.

A suffix-σ $O(|d|)$ kulcs-érték párt választ ki d szövegből, minden pár $O(|d|)$ hosszú, így ez a módszer $O(|d|^2)$ byte transzfert végez a map és reduce fázisok között.

Eredmények

A méréseket 10 Dell R410 szerveren végezték, gépenként 64GB memóriával, két Intel Xeon X5650 6 magos CPU-val, és 4 2 TB-os SAS 7200 rpm merevlemezzel. A gépeken Debian GNU/Linux 5.0.9 (Lenny) futott. A klaszteren belül 1 GBit-es Ethernet kapcsolatot használtak. A Hadoop 0.20.2 Cloudera CDH3u0 disztribúcióját használták Oracle Java 1.6.0_26-on futtatva. Egy gép masterként működött a Hadoop namenode-ját és jobtrackerét futtatja, a többi kilenc gép legfeljebb 10 map és 10 reduce taskot futtatott párhuzamosan.

Összehasonlítottuk a naiv, apriori-keresés, apriori-index és a suffix- σ algoritmusokat. Az apriori-indexnél K értékének 4-et választottuk, méréseink alapján ezzel teljesített legjobban.

A mérési eredmények a következő értékeket fejezik ki:

- Idő: a futás teljes időigénye
- Transzfer: a map és reduce fázisok közötti összes adattranszfer mérete
- Rekordok: a map és reduce fázisok közötti összes rekord száma, ami transzferre és rendezésre került

DATASET CHARACTERISTICS

	NYT	C09
# documents	1,830,592	50,221,915
# term occurrence	1,049,440,645	21,404,321,682
# distinct terms	345,827	979,935
# sentences	55,362,552	1,257,357,167
sentence length (mean)	18.96	17.02
sentence length (stddev)	14.05	17.56

A teszteket két adathalmazon végeztük:

- A The New York Times Annotated Corpus több mint 1,8 millió 1987 és 2007 közötti újságcikkből áll.
- ClueWeb09-B, egy web dokumentum gyűjtemény, több mint 50 millió 2009-es angol nyelvű web dokumentummal.

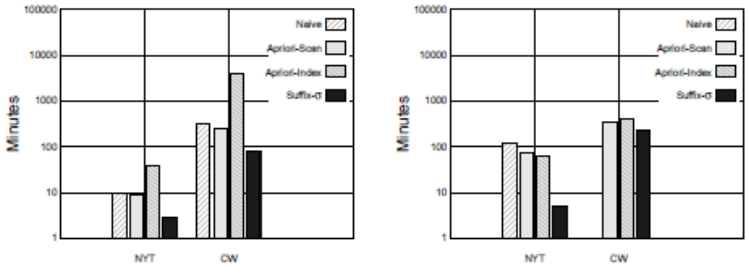

Fig. 3. Wallclock times in minutes for (a) *training a language model* ($\sigma = 5$, NYT: $\tau = 10$ / CW: $\tau = 100$) and (b) *text analytics* ($\sigma = 100$, NYT: $\tau = 100$ / CW: $\tau = 1,000$) as two typical use cases

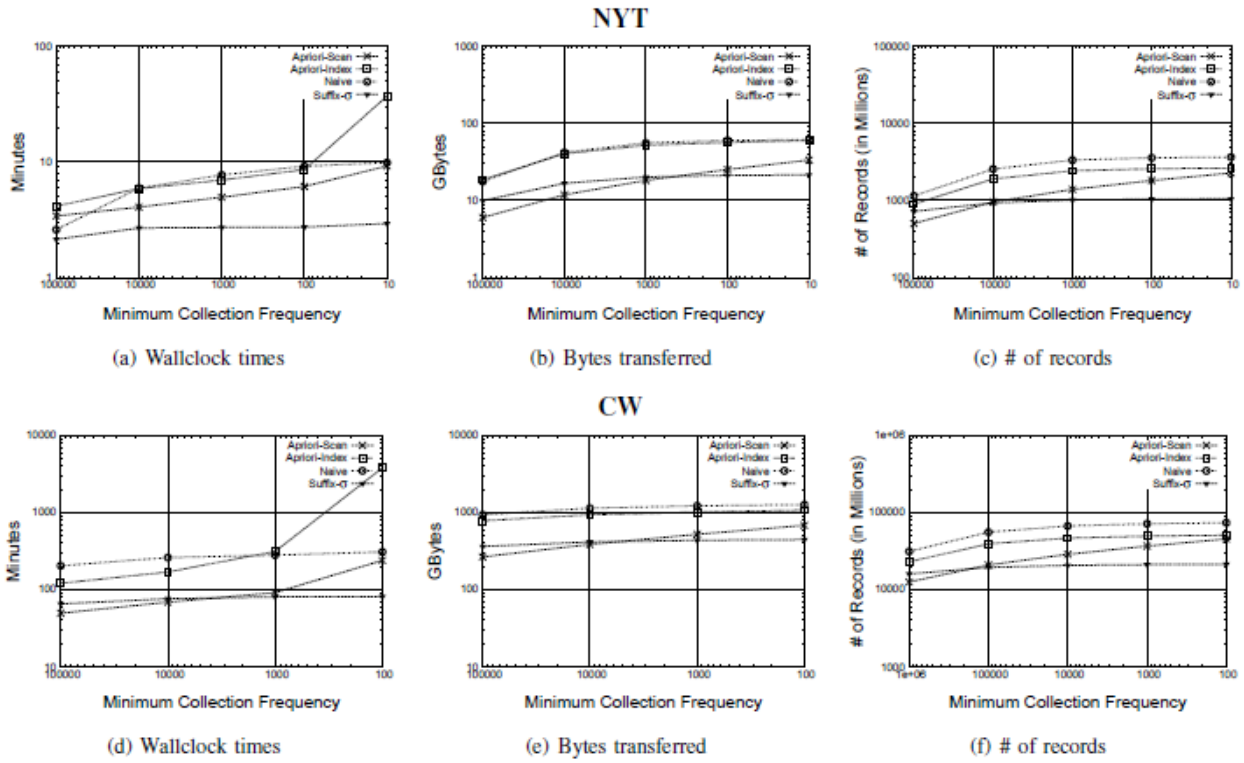

Fig. 4. Varying the minimum collection frequency τ

Mint az ábrákon látható a suffix-σ lényegesen jobb teljesítményt mutat versenytársainál, különösen a hosszú, vagy ritkán előforduló n-gramok gyűjtésénél.

Irodalomjegyzék

[1] Computing n-Gram Statistics in MapReduce
Klaus Berberich #, Srikanta Bedathur
(http://arxiv.org/pdf/1207.4371.pdf)

[2] MapReduce: Simplied Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
(http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/hu//archive/mapreduce-osdi04.pdf)

[3] http://en.wikipedia.org/wiki/N-gram

[4] http://szamitogepesnyelveszet.blogspot.hu/2010/10/mapreduce-paradigma-nagy-mennyisegu.html

[5] http://hu.wikipedia.org/wiki/MapReduce

[6] http://en.wikipedia.org/wiki/Inverted_index

[7] http://people.inf.elte.hu/csgqaai/bm.html