**ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ**

Ακ. έτος 2013-2014, 9ο Εξάμηνο ΗΜ&ΜΥ

Εξαμηνιαία Εργασία

*Εισαγωγή στο MapReduce και στις βάσεις NoSQL*

# 1. Εισαγωγή

Σκοπός της συγκεκριμένης εργασίας είναι η εισαγωγή και η εξοικείωση με το προγραμματιστικό μοντέλο MapReduce και τις βάσεις NoSQL. Συγκεκριμένα, το MapReduce θα μελετηθεί με τη βοήθεια του framework Hadoop, ενώ οι NoSQL βάσεις μέσω τις HBase. Και τα 2 αυτά εργαλεία τρέχουν σε ένα κατανεμημένο υπολογιστικό σύστημα το οποίο μπορεί να αυξομειώνεται ανάλογα με τις ανάγκες τις εφαρμογής. Για τις ανάγκες τις άσκησης χρησιμοποιήθηκα 2 virtual machines από τον ~okeanos τα οποία έτρεχαν Debian Wheezy.

Σαν αντικείμενο μελέτης είχαμε στη διάθεσή μας 2 διαφορετικά datasets, ένα από την America On Line με 20 εκατομμύρια ερωτήματα 650.000 χρηστών και ένα από την Wikipedia με τίτλους άρθρων.

Τα datasets αυτά ανέβηκαν αρχικά στο HDFS (κατανεμημένο σύστημα αρχείων που χρησιμοποιεί το Hadoop) ώστε να μπορεί να γίνει η κατανεμημένη επεξεργασία.

# 2. Εισαγωγή στο MapReduce

**2.1 Yπολογισμός αριθμού αναζητήσεων ανά ημέρα**

Στο ερώτημα αυτό πρέπει να εξάγουμε ένα διάγραμμα από το dataset της AOL το οποίο θα δείχνει το πλήθος των αναζητήσεων που έγιναν ανά μέρα.

Από το αρχείο των αναζητήσεων οι mappers για κάθε γραμμή θα κάνουν emit ένα keyvalue της μορφής <data, 1>. Στη συνέχεια οι reducers θα μαζεύουν τα key-values ανά μέρα. Συνοπτικά:

MAP(key, value) // value is the file line

date = getDate(line)

emit(date,1)

REDUCE(key, value\_list) //

sum = 0

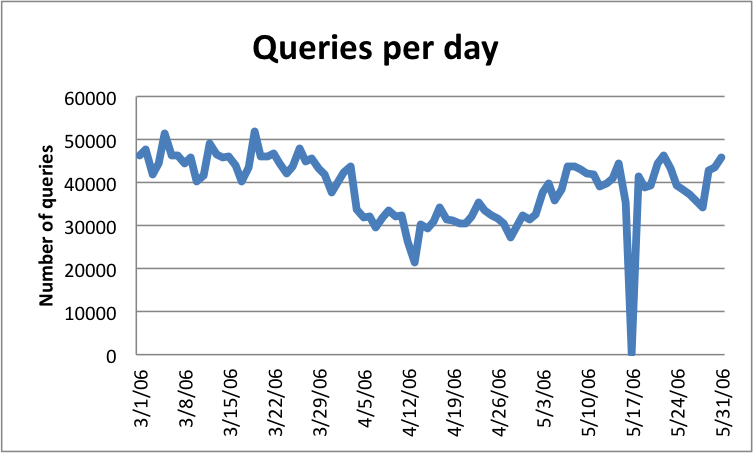
for v in value\_list

sum += v

emit(key, sum)

Μπορούμε να χρησιμοποιήσουμε μόνο έναν reduce ή και περισσότερους. Στην περίπτωση των περισσότερων θα πρέπει να μοιράσουμε τα key-values ανάλογα διατηρώντας τη σειρά. Επιλέξαμε 2 reducers και γνωρίζοντας το εύρος των ημερομηνιών κατασκευάσαμε έναν custom partitioner που μοιράζει τα key-values στα 2.

Παρακάτω βλέπουμε το διάγραμμα του αποτελέσματος.



*Διαγραμμα 1: Ερωτήματα ΑΟL ανά μέρα*

## 2.2 Υπολογισμός ποσοστού «επιτυχών» και «ανεπιτυχών» αναζητήσεων

Στο ερώτημα αυτό ζητείται να υπολογιστούν τα ποσοστά επιτυχών και ανεπιτυχών αναζητήσεων. Επιτυχής αναζήτηση θεωρείται αυτή στην οποία ο χρήστης επέλεξε να μεταβεί σε κάποιο URL, δηλαδή εκείνα τα entries του dataset τα οποία έχουν 4ο και 5ο πεδίο.

Οι mappers διαβάζουν γραμμή-γραμμή το dataset και αν η γραμμή έχει επιτυχή αναζήτηση κάνουν emit keyvalue της μορφής <1,success>, διαφορετικά <1, failure>. Όπως φαίνεται θα χρησιμοποιήσουμε έναν reducer στον οποίο θα καταλήξουν όλα τα αποτελέσματα αφού θέλουμε ποσοστά και χρειαζόμαστε το συνολικό πλήθος αναζητήσεων. O reducer θα μετρήσει τις συνολικές αναζητήσεις και θα υπολογίσει τα ποσοστά επιτυχών και ανεπιτυχών αναζητήσεων. Συνοπτικά:

MAP(key, value):

if check\_success(value):

emit(1,success)

else:

emit(1, failure)

REDUCE(key, value\_list):

success\_count = 0

failure\_count = 0

count = 0

for v in value\_list:

if v==success:

success\_count++

else:

failure\_count++

count++

emit(“Success”, success\_count/count)

emit(“Failure”, failure\_count/count)

Τα αποτελέσματα είναι

Successful searches (%): 53.13

Unsuccessful searches (%): 46.87

## 2.3 Ιστοσελίδων που επισκέφτηκαν πάνω από 10 διαφορετικοί χρήστες

Στο ερώτημα αυτό με βάση το AOL dataset θα πρέπει να να βρούμε τις ιστοσελίδες με πάνω από 10 unique visits και για κάθε μια από αυτές να τυπώσουμε τον αριθμό των επισκέψεων.

Oι mappers κάνουν emit keyvalues της μορφής <url, userId>. Μπορούμε να χρησιμοποιήσουμε πολλούς reducers χωρίς κάποια αλλαγή στο job. Ωστόσο χρησιμοποιήθηκε ένας reducer. Ο reducer παίρνει τα key-values ανά url και τοποθετεί τα userIds μέσα σε ένα HashSet για να υπολογίσουμε τα unique visits. Αν στο τέλος το μέγεθος του set είναι παραπάνω από 10 τότε γίνεται emit το <url, set\_size>. Συνοπτικά:

ΜAP(key, value): // value is the dataset line

url = getUrl(value)

userId = getUser(value)

if (url):

emit(url, userId)

REDUCE(key, value\_list):

Set set;

for id in value\_list:

set.add(id)

if (set.size>10):

emit(key, set.size)

## 2.4 Εύρεση δημοφιλών λέξεων κλειδιών των ερωτημάτων της AOL

Με βάση τα ερωτήματα των χρηστών καλούμαστε να υπολογίσουμε τις πιο δημοφιλείς λέξεις για τις οποίες γίνεται αναζήτηση. Επειδή πρέπει να ταξινομήσουμε το αποτέλεσμα σε φθίνουσα σειρά φορών αναζήτησης δεν μπορούμε να πραγματοποιήσουμε το συγκεκριμένο task με ένα MapReduce job. Στο πρώτο job γίνεται υπολογισμός του πλήθους των αναζητήσεων για κάθε keyword και στο δεύτερο γίνεται η ταξινόμηση.

Ο πρώτος mapper διαβάζει τις γραμμές του dataset και για κάθε λέξη κάνει emit <keyword, 1>. Αντίστοιχα, οι reducers, για κάθε keyword υπολογίζουν το πόσες φορές έγινε αναζήτηση για αυτό και τα ενδιάμεσα αποτελέσματα αποθηκεύονται στο hdfs. Να σημειώσουμε ότι θέλοντας να απαλλάξουμε τα αποτελέσματά μας από «θόρυβο», λέξεις δηλαδή των αγγλικών που χρησιμοποιούνται συχνά όπως άρθα, χρησιμοποιήθηκε μια λίστα από τέτοιες λέξεις, η οποία δινόταν, και οι λέξεις γίνοταν emit στο map σταδιο μόνο στην περίπτωση που δεν υπήρχαν στη συγκεκριμένη λίστα. Η λίστα αυτή έφτασε σε κάθε mapper με τη βοήθεια της DistributedCache class. Τα περιεχόμενα αυτής μπήκαν σε ένα set στο οποίο γινόταν έλεγχος αν υπάρχει ή όχι η εκάστοτε λέξη. Συνοπτικά:

MAP(key, value):

keywords = getKeyWords(value)

word\_list = keywords.split(“ ”)

for w in word\_list:

if (not inStopList(w))

emit(w,1)

REDUCE(key, value\_list):

count=0

for v in value\_list:

count += v

emit(count, key)

Επομένως, τώρα υπάρχουν στο hdfs αποθηκευμά αρχεία που περιέχουν την πληροφορία σχετικά με το πόσες φορές έγινε αναζήτηση για κάποιο keyword. Τα αρχεία αυτά είναι τόσα όσοι και οι reducers του πρώτου jobs, στο οποίο μπορούμε να χρησιμοποιήσουμε όσους θέλουμε αφού γίνεται χρήση του HashPartitioner.

Στο δεύτερο job έχοντας το πλήθος αναζητήσεων για την κάθε λέξη πρέπει να κάνουμε μια ταξινόμηση με βάση το πλήθος. Χρειαζόμαστε identity mappers οι οποίοι θα κάνουν emit ό,τι key-value διαβάσουν. Για το λόγο αυτό στο προηγούμενο job χρησιμοποιήσαμε SequenceFileOutputFormat. Οι reducers θα πρέπει να κάνουν emit αρχεία τα οποία απλά εμείς θα πρέπει να κάνουμε append ώστε να έχουμε την τελική ταξινόμηση. Δεν μας κάνει ο HashPartitioner στην περίπτωση περισσότερων από εναν reducers γιατί θέλουμε ταξινόμηση. Χρειαζόμαστε έναν total-order partitioner. Δεδομένου ότι έχουμε 2 vms στη διάθεσή μας δώσαμε στο 2ο job 2 reducers αλλά μετά από κάποιες δοκιμές διαπιστώθηκε ότι τα δεδομένα δεν ήταν ισοκατανεμημένα και υπήρχε τεράστιος αριθμός keywords με πολύ λίγες αναζητήσεις, δηλαδή κάτω από 10. Επομένως, φτιάξαμε και έναν custom total order partitioner ο οποίος έστελνε keys με τιμή κάτω από 10 στον ένα reducer και τα υπόλοιπα στον άλλον. Ωστόσο, θα μπορούσε να χρησιμοποιηθεί απλά ένας reducer και να μην γίνει όλη η διαδικασία. Τέλος, δεδομένου ότι θέλουμε φθίνουσα ταξινόμηση χρειάστηκε να κάνουμε overwrite τον Comparator. Συνοπτικά για το reduce έχουμε:

REDUCE(key, value\_list): //key=count, value\_list = keywords

for v in value\_list:

emit(v, count)

Η λίστα με τα 50 πιο δημοφιλή keywords είναι και τη συχνότητά τους είναι:

- 104052

free 45149

google 34970

http 24394

yahoo 23491

county 22435

pictures 21416

lyrics 18476

school 18091

myspace 16549

florida 15779

ebay 15603

sale 14612

american 14479

city 14331

home 13794

state 13622

www 12171

music 12161

pics 11704

.com 11691

games 11639

york 11556

girls 10980

beach 10887

bank 10707

texas 10666

online 10664

black 10567

high 10506

nude 10457

aol 10083

yahoo.com 10007

map 9791

news 9614

myspace.com 9577

college 9510

car 9111

mapquest 8895

jobs 8881

2006 8854

homes 8798

ohio 8706

google.com 8682

real 8483

university 8449

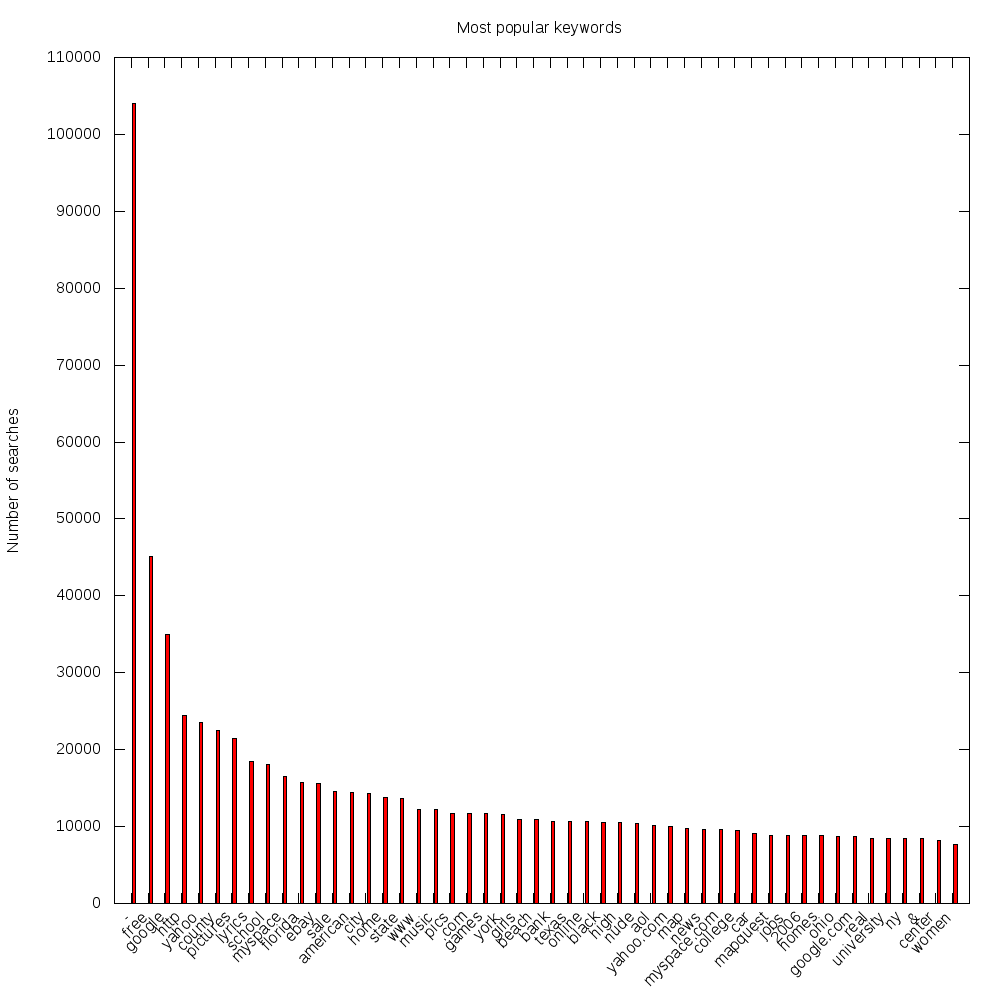
ny 8421

& 8388

center 8169

women 7713

Και σχηματικά,



*Διαγραμμα 2: 50 πιο δημοφιλείς λέξεις και η συχνότητα αναζήτησής τους*

Αν μετρήσουμε τις γραμμές των αρχείων αποτελεσμάτων βρίσκουμε το πλήθος των μοναδικών λέξεων αναζήτησης οι οποίες είναι συνολικά 120703+459234 = 579937

## 2.5 Υπολογισμός ιστογράμματος της λεξικογραφικής κατανομής των λέξεων

## κλειδιών των άρθρων των τίτλων της wikipedia

### 2.5.1 Κατανομή λέξεων

Με βάση το dataset των τίτλων της Wikipedia, στο ερώτημα αυτό καλούμαστε να κατασκευάσουμε έναν πίνακα ο οποίος για κάθε κατηγορία (έχουμε συνολικά 28 κατηγορίες, μια για κάθε γράμμα του αγγλικού αλφαβήτου, μια για σύμβολα και μια νούμερα) θα περιέχει το ποσοστό εμφάνισης των λέξεων που ανήκουν στην κατηγορία αυτή συνολικά στο αρχείο των τίτλων τηνς wikipedia. Για να το πετύχουμε αυτό θα χρειαστούμε 2 MapReduce jοbs. Το πρώτο θα υπολογίσει το πλήθος των λέξεων ανά κατηγορία και το δεύτερο θα υπολογίσει τα ποσοστά και θα παρουσιάσει ταξινομημένα τα αποτελέσματα. Και πάλι θα γίνει χρήση του αρχείου με τις stop words.

Για το πρώτο job, οι mappers διαβάζουν τους τίτλους και για κάθε λέξη του τίτλου που δεν ανήκει στις stop words κάνουν emit ένα key-value της μορφής <category, 1>. Στη συνέχεια οι reducers αθροίζουν τους άσσους και βρίσκουν το πλήθος λέξεων για τη συγκεκριμένη κατηγορία. Κάνουν emit ένα pair της μορφής <null, category\_count > αφού θέλουμε όλα τα pairs να καταλήξουν στον δεύτερο reducer, χρησιμοποιώντας ένα sequence file. Συνοπτικά,

MAP(key, value): // value ο τίτλος

words = value.split(“\_”)

for w in words:

if (not inStopWords(w)):

emit(category(w), 1)

To category είναι μια συνάρτηση που υπολογίζει σε ποια κατηγορία βρίσκεται η κάθε λέξη και επιστρέφει έναν ακέραιο αριθμό, το id της κατηγορίας.

REDUCE(key, value\_list):

count = 0

for v in value\_list:

count++

emit(null, key+”\_”+count)

Στο δεύτερο job χρησιμοποιούμε έναν identity mapper και όλα τα key-values που παράχθηκαν από το πρώτο job καταλήγουν στον ίδιο reducer. Χρησιμοποιήσαμε το null σαν κλειδί ώστε να έχουμε και τα 28 pairs διαθέσιμα μαζί. Έτσι μπορούμε να υπολογίσουμε αρχικά το συνολικό άθροισμα των λέξεων που μετρήσαμε και μετά το ποσοστό για κάθε κατηγορία. Συνοπτικά,

REDUCE(key, value\_list):

count = 0;

for v in value\_list:

c = v.split(“\_”)[1]

count += c

for v in value\_list:

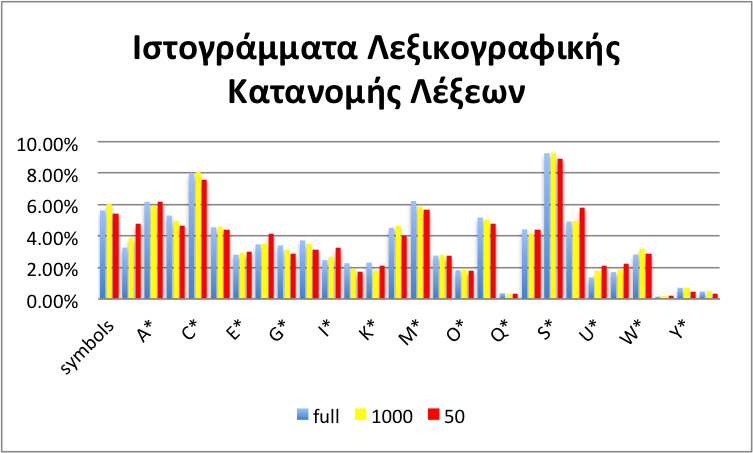
id = v.split(“\_”)[0]

c = v.split(“\_”)[1]

emit(getCategory(id), c/count )

όπου το getCategory μας επιστρέφει τα ονόματα των κατηγοριών, Α\*, Β\* κλπ.

Η παραπάνω διαδικασία πραγματοποιήθηκε 3 φορές. Προσπαθώντας να μειώσουμε το βάρος του υπολογισμού κάναμε δειγματοληψία στον αριθμό των key-values που κάνουν process οι reducers. Για να το καταφέρουμε αυτό κάναμε overwrite τη συνάρτηση run του mapper και ρυθμίσαμε πόσες φορές θα κληθεί η συνάρτηση map στον κάθε mapper. Έτσι δειγματοληπτικά πάνω στο αρχείο εισόδου κάναμε process με αποτέλεσμα να παραχθούν λιγότερα key-vaues για τους reducers. Αρχικά εκτελέσαμε το job χωρίς δειγματοληψία. Στη συνέχεια αφήνοντας τους πρώτους mapper να κάνουν process μόνο 50 key-values και τέλος αφήνοντας τους mappers να κάνουν process 1000 key-values. Τα αποτελέσματα παρουσιάζονται στο ακόλουθο διάγραμμα.



*Διαγραμμα 3: Ιστογράμματα λεξικογραφικής κατανομής λέξεων των τιτλων της wikipedia*

### 2.5.2 Ταξινόμηση

Στο συγκεκριμένο ερώτημα καλούμαστε να ταξινομήσουμε λεξικογραφικά τις λέξεις των άρθρων της Wikipedia. Η ταξινόμηση θα πρέπει να γίνει κατανεμημένα και το αποτέλεσμα να αποτελείται από διαφορετικά αρχεία τα οποία μπορούν να γίνουν append το ένα στο άλλο και η λεξικογραφική διάταξη να διατηρείται. Για να γίνει κατανεμημένα η ταξινόμηση και όχι σε έναν reducer θα χρειαστούμε εναν partitioner ο οποίος διατηρεί την ταξινόμηση όπως ο TotalOrderPartitioner. Απαραίτητο για τη χρήση του partitioner αυτού είναι ένα αρχείο το οποίο περιέχει τα splits, key-values δηλαδή τα οποία καθορίζουν τα όρια των κλειδιών που δέχεται ο κάθε reducer.

Επομένως, θα χρειαστούμε 2 jobs. Το πρώτο θα δημιουργήσει το αρχείο των splits κάνοντας κάποιο sampling στο αρχείο εισόδου, ώστε να ξέρουμε σε ποια κομμάτια του αλφαβήτου να δώσουμε περισσότερους reducers και ποια κομμάτια δεν έχουν τόσο μεγάλη συχνότητα. Το δεύτερο job είναι αυτό το οποίο θα κάνει και την ταξινόμηση. Οι mappers του θα διαβάζουν τους τιτλους των άρθρων και θα κάνουν emit τις λέξεις. Οι reducers απλά θα γράφουν τις λέξεις αυτές ταξινομημένα στο hdfs. Να σημειωθεί ότι και εδώ έγινε χρήση των stop words. Συνοπτικά,

* για το πρώτο ΜR

MAP(key, value): // value = η γραμμή του dataset

words = value.split(“\_”)

for w in words:

if (not inStopWords(w))

emit(w, 0)

global\_count++

Η συνάρτηση αυτή καλείται 50 ή 1000 φορές. Αυτό καθορίζεται στη συνάρτηση run του mapper. Το key-value είναι της μορφής <string,int>. Αυτό συμβαίνει διότι στο reducer χρειαζόμαστε να ξέρουμε πόσα key-values παρήχθησαν στη φάση του map. Έτσι ο κάθε mapper στο τέλος κάνει emit ένα key-value <””, global\_count>. Τα key-values με κενό string είναι τα πρώτα τα οποία θα διαβάσει ο μοναδικός reducer και θα υπολογίσει τα διαθέσιμα κλειδιά εισόδου. Δουλειά του reducer είναι να κάνει emit τόσα κλειδιά όσα και ο αριθμός των reducer για το job της ταξινόμησης -1. Γνωρίζοντας πόσα διαθέσιμα κλειδιά έχουμε και πόσα θέλουμε να κάνουμε emit μπορούμε να υπολογίσουμε το βήμα με το οποίο θα κάνουμε emit ως step = sample / (reducersNo -1). Επομένως, κάθε step keys που διαβάζουμε κάνουμε emit το τρέχον και αυτό γράφεται στο αρχείο το οποίο θα διαβάσει ο TotalOrder partitioner για να κάνει την ταξινόμηση.

* για το δεύτερο ΜR

Στο job αυτό γίνεται η ταξινόμηση. Οι mappers κάνουν emit τις λέξεις των τίτλων και οι reducers τυπώνουν ταξινομημένα το αποτέλεσμα. Ο partitioner φροντίζει ώστε να γίνει σωστά και όσο το δυνατόν δίκαια με βάση το partition file ο διαμοιρασμός των key-values.

MAP(key, value): // value = η γραμμή του dataset

words = value.split(“\_”)

for w in words:

if (not inStopWords(w))

emit(w, null)

Χρησιμοποείται ο identity reducer αφού δεν χρειάζεται να γίνει κάποια συγκεκριμένη επεξεργασία, παρά μόνο να μαζευτούν τα μοναδικά keys και να γίνουν emit.

Και πάλι ακολουθήθηκαν 2 προσεγγίσεις σχετικά με το sampling. Στην πρώτη οι mappers έκαναν 50 επαναλήψεις, ενώ στη δεύτερη 1000. Στόχος ήταν φυσικά τα αρχεία με τις λέξεις να έχουν όσο το δυνατόν λιγότερες αποκλείσεις στο μέγεθός τους αλλά και με το μεγαλυτερο δυνατό sampling. Ενδεικτικά αναφέρουμε:

* Για samping 50

1421979 /user/root/2\_5\_2\_50\_results/part-r-00000

1457687 /user/root/2\_5\_2\_50\_results/part-r-00001

2256524 /user/root/2\_5\_2\_50\_results/part-r-00002

1870106 /user/root/2\_5\_2\_50\_results/part-r-00003

2144550 /user/root/2\_5\_2\_50\_results/part-r-00004

2154282 /user/root/2\_5\_2\_50\_results/part-r-00005

1648754 /user/root/2\_5\_2\_50\_results/part-r-00006

2903366 /user/root/2\_5\_2\_50\_results/part-r-00007

2399396 /user/root/2\_5\_2\_50\_results/part-r-00008

13427031 /user/root/2\_5\_2\_50\_results/part-r-00009

* Για sampling 1000

2219194 /user/root/2\_5\_2\_1000\_results/part-r-00000

3013629 /user/root/2\_5\_2\_1000\_results/part-r-00001

2865982 /user/root/2\_5\_2\_1000\_results/part-r-00002

2988760 /user/root/2\_5\_2\_1000\_results/part-r-00003

3642139 /user/root/2\_5\_2\_1000\_results/part-r-00004

3103469 /user/root/2\_5\_2\_1000\_results/part-r-00005

3291799 /user/root/2\_5\_2\_1000\_results/part-r-00006

3496447 /user/root/2\_5\_2\_1000\_results/part-r-00007

2950653 /user/root/2\_5\_2\_1000\_results/part-r-00008

4111603 /user/root/2\_5\_2\_1000\_results/part-r-00009

Παρατηρούμε λοιπόν πόσο πιο ισοκατανεμημένα είναι τα δεδομένα στην περίπτωση που η συχνότητα της δειγματοληψίας είναι μεγαλύτερη.

### 2.5.3 Ερωτήσεις

* *Πόσο χρόνο έκαναν να εκτελεστούν τα προγράμματα histogram\_full, histogram\_50 και histogram\_1000?*

Από τα logs των αντίστοιχων jobs βρίσκουμε τις εξής πληροφορίες :

|  |  |  |  |
| --- | --- | --- | --- |
|  | full | 1000 | 50 |
| job1 | 3min 23 sec | 2 min 19 sec | 2 min 15 sec1 |
| job2 | 1 min 5 sec | 1 min 8 sec | 1 min 5 sec |

Υπενθυμίζουμε ότι το πρώτο job είναι αυτό που κάνει το processing των λέξεων και το δεύτερο απλά υπολογίζει τα ποσοστά έχοντας τις απόλυτες τιμές από το πρώτο. Άρα αυτό που μας ενδιαφέρει είναι ο χρόνος του πρώτου, γιατί το δεύτερο είναι σταθερό.

Παρατηρούμε λοιπόν ότι όσο μειώνουμε το πλήθος των key-values που παράγονται από τους mappers τόσο μειώνεται και ο χρόνος.

* *Πιο ιστόγραμμα προσεγγίζει καλύτερα το histogram\_full?*

Όπως παρατηρούμε στο διάγραμμα 3, το histogram\_full (μπλε), προσεγγίζεται καλύτερα από το histogram\_1000 (κίτρινο) όπως είναι και το αναμενόμενο. Όσο πιο συχνά κάνουμε δειγματοληψία τόσο καλύτερα το αποτέλεσμα προσεγγίζει τα πλήρη δεδομένα. Ωστόσο, να σημειωθεί ότι και το histogram\_50 δεν έχει τραγικά μεγάλες αποκλίσεις. Παρόλα αυτά κρίνοντας και από τους αντίστοιχους χρόνους δεν υπάρχει λόγος να κάνουμε sampling με 50 αφού histogram\_1000 και histogram\_50 έκαναν περίπου τους ίδιους χρόνους.

* *Με τι κόστος έγινε αυτή η προσέγγιση?*

Για να κάνουμε την προσέγγιση διατρέξαμε όλα τα δεδομένα εισόδου και κάναμε sampling πάνω σε αυτά. Είχαμε δηλαδή το κόστος της ανάγνωσης όλων των δεδομένων παρόλο που χρειαστηκε να επεξεργαστούμε μόνο ένα μικρό μέρος από αυτά.

* *Ποιο γράμμα έχει τα περισσότερα αποτελέσματα? Για ποιο λόγο συμβαίνει αυτό?*

Από το διάγραμμα 3 αλλά και το output των jobs βλέπουμε ότι τα περισσότερα αποτελέσματα τα έχει το γράμμα S. (Γιατί?)

* *Σε ποια εκτέλεση από τις ordered\_sample\_50 και ordered\_sample\_1000 τα αρχεία File1..File10 που προέκυψαν περιέχουν πιο ισοκατανεμημένο αριθμό κλειδιών?*

Όπως βλέπουμε από το μέγεθος των αρχείων παραπάνω στην περίπτωση του ordered\_sample\_1000 πετυχαίνουμε πολύ καλύτερη ισοκατανομή με το μέγεθος των αρχείων να μεταβάλλεται από 2.12 MB – 3.92 MB. Η αντίστοιχη μεταβολή για το orderd\_sample\_50 είναι 1.36 ΜΒ – 12.81ΜΒ.

* *Ποια εκτέλεση από τις ordered\_sample\_50 και ordered\_sample\_1000 εκτελέστηκε πιο γρήγορα?*

Από τα logs παίρνουμε τις εξής τιμές για τους χρόνους εκτέλεσης:

|  |  |  |
| --- | --- | --- |
|  | ordered\_sample\_1000 | ordered\_sample\_50 |
| sampling | 40 sec | 35 sec |
| sorting | 2 min 36 sec | 2 min 34sec |

Όπως είναι λογικό το sampling με λιγότερα βήματα κράτησε λιγότερο χρόνο. Αντίθετα, για την ταξινόμηση βλέπουμε ότι και στις 2 περιπτώσεις κράτησε περίπου τον ίδιο χρόνο ενώ αναμενόμενο θα ήταν να κρατούσε περισσότερο στην περίπτωση της μικρής δειγματοληψίας, αφού σε κάποιον reducer θα έπεφτε περισσότερος φορτος εργασίας με αποτέλεσμα να καθυστερήσει. Κάτι τέτοιο, ωστόσο, δεν έγινε.

## 2.6 Υπολογισμός ποσοστού ερωτημάτων που μπορούν να απαντηθούν από την wikipedia

Στο ερώτημα αυτό καλούμαστε να συνδιάσουμε τα 2 datasets, αυτό της Wikipedia και αυτό της AOL, με σκοπό να υπολογίσουμε ποιο ποσοστό των ερωτημάτων της AOL μπορεί να απαντηθεί από τη Wikipedia. Ουσιαστικά αυτό σημαίνει να βρούμε το ποσοστό των λέξεων που είναι κοινές στους τίτλους της Wikipedia και στα ερωτήματα των χρηστών της AOL.

Για να το κάνουμε αυτό θα χρειαστούμε 2 MapReduce jobs. Σκοπός μας είναι για κάθε λέξη να ελέγξουμε αν υπάρχει και στα 2 datasets. Επομένως, το πρώτο job θα βρει όλες τις λέξεις και πού υπάρχουν ενώ το δεύτερο θα κάνει τους υπολογισμούς. Συγκεκριμένα, οι mappers του πρώτου job, για κάθε λέξη από τα ερώτηματα των χρηστών της AOL θα κάνουν emit keyword της μορφής <word, 1>, ενώ για κάθε λέξη των τίτλων της Wikipedia θα κάνουν emit keyword της μορφής <word, 2>. Στη συνέχεια, οι reducers θα μαζέψουν τα key-values ανά λέξη. Στα κλειδιά στα οποία υπάρχει και 2 τα ερωτήματα θεωρούνται επιτυχημένα και επομένως οι reducers κάνουν emit keyvalue της μορφής <1, count> όπου count είναι το πλήθος των 1 στη λίστα. Διαφορετικά, τα ερωτήματα είναι αποτυχημένα και κάνουν emit <2, count>. Συνοπτικά:

MAP(key, value): // value η γραμμή του dataset

if AOL(value):

keywords = getKeyWords(value)

for word in keywords:

emit(word, 1)

else:

words = value.split(“\_”)

for word in words:

emit(word,2)

REDUCE(key, value\_list):

ones = countOnes(value\_list)

twos = countTwo(value\_list)

if (twos > 0):

emit(1,ones)

else:

emit(2,ones)

Δουλειά του δεύτερου MapReduce job είναι να κάνει τα αθροίσματα και να υπολογίσει το πλήθος των επιτυχημένων και αποτυχημένων ερωτημάτων. Χρησιμοποιούμε SequenceFormat για τα ενδιάμεσα αποτελέσματα και έτσι χρειαζόμαστε έναν identity mapper. Δουλειά επομένως, του reducer είναι να αθροίσει τα values που υπάρχουν στο κάθε κλειδί. Χρησιμοποιούμε 2 reducers, ώστε ο ένας να πάρει τα κλειδιά με 1 και ο άλλος να πάρει τα κλειδιά με 2. Έτσι, στο τέλος του job θα έχουμε 2 διαφορετικά αρχεία, το ένα θα περιέχει το πλήθος των επιτυχών queries και το δεύτερο το πλήθος των ανεπιτυχών. Συνοπτικά,

REDUCE(key, value\_list):

count = 0

for v in value\_list:

count++

emit(key, count)

Μετά την εκτέλεση και των 2 jobs καταλήγουμε τα εξής αποτελέσματα:

Successful queries 6973321 (79.38 %)

Unccessful queries 1810865 (20.62 %)

# 3. Εισαγωγή στις βάσεις NoSQL (HBase)

Στο κομμάτι αυτό της εργασίας γίνεται μια εισαγωγή και εξοικείωση με τις μη σχεσιακές βάσεις δεδομένων και συγκεκριμένα την κατανεμημένη βάση HBase. Παρόλο που με το Hadoop μπορούμε να κάνουμε κατανεμημένη επεξεργασία και ανάγνωση δεδομένων, έχουμε περιορισμένη δυνατότητα τροποποίησης των αρχικών δεδομένων. Αντίθετα, με την HBase μπορούμε να διαβάζουμε και να γράφουμε πολύ πιο εύκολα μεγάλα σύνολα δεδομένων μέσα από το API της σε μια ημιδομημενη μορφή. Συγκεκριμένα, θα εισάγουμε τα δεδομένα από τα 2 datasets που μας δίνονται (AOL και Wikipedia) και θα κάνουμε αναζητήσεις πάνω σε αυτά.

## 3.1 Εισαγωγή των τίτλων της wikipedia στην Hbase.

Αρχικά, θα πρέπει να βάλουμε σε έναν πίνακα της HBase τα ονόματα των τίτλων της Wikipedia. Για να το κάνουμε αυτό θα φτιάξουμε ένα MapReduce job το οποίο θα εκτελεστεί από το Hadoop και θα γεμίσει τον αντίστοιχο πίνακα. Το job αυτό θα διαβάσει το αντίστοιχο dataset από το hdfs και θα παράξει key-value pairs τα οποία είναι αναγνωρίσιμα από την HBase. Παρόλο που θα μπορούσαμε να ακολουθήσουμε το HBase API και να κάνουμε διαδοχικά Put στον εν λόγω πίνακα, επιλέγουμε αυτή την τεχνική του bulk loading του πίνακα, της μαζικής δηλαδή εισαγωγής δεδομένων σε έναν πίνακα της HBase. Σαν κλειδί για τον πίνακα content (πίνακας των τίτλων) θα χρησιμοποιηθεί το md5 του τίτλου. Θα χρειαστούμε ένα column family (wikipedia) και έναν qualifier (titles).

Για το συγκεκριμένο job δεν χρειαζόμαστε κάποιο reducer αφού οι mappers παράγουν τα απαραίτητα key-values για το bulk loading. Χρησιμοποιούνται οι τύποι ImmutableBytesWritable για τα κλειδιά και KeyVaule για τις τιμές. Συγκεκριμένα,

MAP(key, value): //value μια γραμμή του dataset της Wikipedia

hbase\_key = md5(value)

emit(hbase\_key, value)

**Ερώτηση:** *Γιατί ο μηχανισμός bulk import είναι πιο γρήγορος από διαδοχικά Put για κάθε αντικείμενο? Πως εξηγείται αυτό λαμβάνοντας υπόψη την αρχιτεκτονική της HBase κατά την εισαγωγή νέων αντικειμένων?*

Οι λόγοι για τους οποίους επιλέγουμε το bulk loading και για τους οποίους έχουμε καλύτερο performance με αυτή την τεχνική είναι δύο. Ο πρώτος αφορά το WAL (write ahead log) το οποίο δεν γράφεται στην περίπτωση του bulk loading, ενώ γράφεται στην περίπτωση των απλών Put. Επομένως, αφού παραλείπουμε την εγγραφή εκεί, κερδίζουμε cpu χρόνο αφού εγγραφές στο δίσκο γίνονται μόνο όσες χρειάζονται για την εισαγωγή των δεδομένων. Το WAL δεν είναι απαραίτητο στη συγκεκριμένη περίπτωση και γι’αυτό παραλείπεται. Επιπλέον, το bulk loading σημαίνει ότι όλες οι εγγραφές θα γίνουν στο ίδιο region μέχρι αυτό να γίνει αρκετά μεγάλο και να χωριστεί. Επομένως, έτσι δεν χρειάζεται extra επικοινωνία για τον συντονισμό και τον συγχρονισμό των διαφορετικών regions.