Ατομική Διπλωματική Εργασία

ΜΕΛΕΤΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΤΕΧΝΙΚΩΝ ΔΥΝΑΜΙΚΗΣ ΑΝΑΔΙΑΝΟΜΗΣ ΔΕΔΟΜΕΝΩΝ ΣΕ ΣΥΣΤΗΜΑΤΑ ΡΟΩΝ ΜΕΓΑΛΩΝ ΔΕΔΟΜΕΝΩΝ

Μάριος Βλαδιμήρου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2016

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μελέτη και υλοποίηση τεχνικών δυναμικής αναδιανομής δεδομένων σε συστήματα ροών μεγάλων δεδομένων

Μάριος Βλαδιμήρου

Επιβλέπων Καθηγητής
Γιώργος Πάλλης

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2016

Περίληψη

Σκοπός της διπλωματικής ήταν να μελετηθεί πως η ανομοιομορφία στην κατανομή των δεδομένων επηρεάζει την επεξεργασία ροής δεδομένων, να προταθεί μια λύση και να γίνει αξιολόγηση της.

Αρχικά έγινε έρευνα για τις διάφορες μηχανές επεξεργασίας ροής δεδομένων που υπάρχουν και είναι διαδεδομένες, για την αρχιτεκτονική τους, τις διαφορές τους και τα πλεονεκτήματα της κάθε μίας.

Για να μελετήσουμε πως επηρεάζει η ανομοιομορφία έγινε ανάπτυξη εργαλείου το οποίο αυξάνει την ανομοιομορφία σε ένα σύνολο δεδομένων, έτσι θα μπορούμε να συγκρίνουμε για διάφορα σενάρια. Για τα πειράματα χρησιμοποιήθηκαν tweets από το Twitter.

Στην συνέχεια δείξαμε πως η ανομοιομορφία στην κατανομή των δεδομένων επηρεάζει την επίδοση για ένα κοινό σενάριο επεξεργασίας ροής δεδομένων χρησιμοποιώντας παράθυρο.

Ακολούθως ορίστηκαν κάποιες προδιαγραφές που πρέπει να έχει η λύση που θα προταθεί. Μετά έγινε η ανάπτυξη του αλγορίθμου σε ψευδοκώδικα και στην συνέχεια υλοποιήθηκε για μία μηχανή επεξεργασίας ροής δεδομένων.

Τέλος, έγιναν μετρήσεις για το ίδιο σενάριο που χρησιμοποιήσαμε αρχικά για να δείξουμε το πρόβλημα αλλά χρησιμοποιώντας την προτεινόμενη λύση και είδαμε την βελτίωση στην επίδοση. Εκτός από την επίδοση, εξετάσαμε και εάν η προτεινόμενη λύση τηρούσε τις προδιαγραφές που ορίσαμε αρχικά.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Εισαγωγή

Το κίνητρο πίσω από το θέμα είναι η τεράστια αύξηση των δεδομένων που παράγονται ή καταγράφονται ηλεκτρονικά τα τελευταία χρόνια αλλά και οι εκτιμήσεις για ακόμα μεγαλύτερη αύξηση τα επόμενα χρόνια.

Η επεξεργασία ροής δεδομένων, δηλαδή μόλις τα δεδομένα παράγονται, είναι αναγκαία όταν τα δεδομένα έχουν ημερομηνία λήξης και πρέπει να επεξεργαστούν σε κάποιο χρονικό διάστημα αλλιώς η πληροφορία μετά είναι άχρηστη. Για παράδειγμα εάν παρακολουθούμε δεδομένα για σεισμικές δονήσεις, θα θέλουμε άμεσα να τα επεξεργαστούμε και να προβλέψουμε τυχόν πιο δυνατούς μετασεισμούς, ενώ εάν καθυστερήσουμε πολύ πιθανών αυτά να μην έχουν και μεγάλη αξία.

Ο άλλος λόγος που η επεξεργασία ροής δεδομένων είναι σημαντική τελευταία είναι επειδή λόγω της ανάπτυξης του υλικού των υπολογιστών αλλά και του λογισμικού, η ανάλυση δεδομένων που παραδοσιακά γινόταν εκ των υστέρων, τώρα μπορεί να γίνεται άμεσα. Για παράδειγμα, πριν μπορεί να είχαμε ένα αρκετά μεγάλο όγκο δεδομένων και θα χρειαζόταν να μαζευτούν τα δεδομένα, ίσως να γίνει κάποια προεργασία σε αυτά και μετά θα γινόταν η επεξεργασία που θα έπαιρνε ώρες. Τώρα όμως σε αρκετά σενάρια είναι δυνατό αυτή η επεξεργασία να γίνεται άμεσα, με την παραγωγή των δεδομένων. Αυτή η εξοικονόμηση χρόνου από την παραγωγή των δεδομένων μέχρι την εξαγωγή των πληροφοριών είναι πολύ σημαντική για αρκετές εταιρείες και κάποιες φορές

μπορεί να αποκτούν σημαντικό πλεονέκτημα σε σχέση με τον ανταγωνισμό εάν είναι οι πιο γρήγοροι.

Το κύριο πρόβλημα με το οποίο ασχολείται η διπλωματική είναι η ανομοιόμορφη κατανομή στα δεδομένα. Αυτό είναι πρόβλημα επειδή δημιουργεί αναποτελεσματική χρησιμοποίηση των μηχανών και τελικά πιο αργή επεξεργασία. Ο λόγος είναι επειδή πολλές φορές θέλουμε να επεξεργαστούμε τα δεδομένα χωρίζοντας τα με κάποιο κλειδί, εάν για κάποια κλειδιά έχουμε πολλά περισσότερα δεδομένα σε σχέση με άλλα τότε πιθανόν να δημιουργηθεί προβληματική ανομοιομορφία. Δηλαδή στο τέλος κάποιες μηχανές που επεξεργάζονται δεδομένα από τα πιο συχνά κλειδιά να έχουν μεγαλύτερο φόρτο από τις άλλες.

Στην πραγματικότητα πολύ σπάνια έχουμε ιδανική για μας κατανομή των δεδομένων και τις περισσότερες φορές υπάρχει κάποια ανομοιομορφία. Εάν γνωρίζαμε ή μπορούσαμε να προβλέψουμε από πριν την ανομοιομορφία, τότε ίσως το πρόβλημα να ήταν μικρότερο, αντίθετα όμως τις περισσότερες φορές είναι αδύνατο να προβλεφθεί. Αυτή η απρόβλεπτη ανομοιομορφία στα δεδομένα οφείλεται στους παραγωγούς των δεδομένων, δηλαδή τους ανθρώπους και γενικότερα την φύση. Και αφού έχουμε να κάνουμε με ένα τόσο πολύπλοκο σύστημα τότε οι προβλέψεις είναι πολύ δύσκολες έως αδύνατες κάποιες φορές.

Για την εξέταση του προβλήματος αρχικά έπρεπε να γίνει επιλογή κάποιας μηχανής επεξεργασίας ροής δεδομένων. Υπήρχαν αρκετές πολύ καλές επιλογές, και τελικά μετά από ανάλυση και σύγκριση των υποψήφιων επιλέγηκε το Spark.

Τα δεδομένα που χρησιμοποιήθηκαν για τα πειράματα ήταν ένα σύνολο 10 εκατομμυρίων tweets από το Twitter. Ο λόγος που επιλέγηκε το Twitter είναι λόγω της μικρής «διάρκειας ζωής» ενός tweet[1], που αυτό σημαίνει ότι είναι χρήσιμο να επεξεργάζονται μόλις δημιουργηθούν, σε ροή δεδομένων δηλαδή. Στην συνέχεια, έπρεπε να δούμε πως η ανομοιομορφία των δεδομένων μας επηρεάζει και γι' αυτό τον λόγο αναπτύχθηκε ένα πρόγραμμα που παράγει σύνολα δεδομένων με διαφορετικό βαθμό ανομοιομορφίας. Πιο συγκεκριμένα, χρησιμοποιώντας σαν βάση το αρχικό

[1] http://www.wiselytics.com/blog/tweet-isbillion-time-shorter-than-carbon14/

σύνολο δεδομένων προσθέτει επιπλέον tweets με αποτέλεσμα να αυξάνει την ανομοιομορφία. Ακολούθως, αναπτύχθηκαν διεργασίες για επεξεργασία των δεδομένων στο Spark για να διερευνηθεί πως επηρεάζεται ο χρόνος εκτέλεσης τους αναλόγως του βαθμού ανομοιομορφίας των δεδομένων.

Μετά έγινε η ανάπτυξη της λύσης η οποία σκοπό είχε να μειώσει τον χρόνο εκτέλεσης εκεί όπου η ανομοιομορφία των δεδομένων είχε αρνητικό αντίκτυπο στην επίδοση. Στα δεδομένα που έχουμε το κάθε tweet είναι ένα μήνυμα και χρησιμοποιείται το ID του χρήστη που έγραψε το tweet σαν κλειδί. Η προκαθορισμένη τεχνική διαμοιρασμού των μηνυμάτων είναι το hashing, δηλαδή με βάση το κλειδί (το ID του χρήστη). Όμως έτσι δημιουργείται πρόβλημα σε περίπτωση που έχουμε σημαντική ανομοιομορφία στην κατανομή των tweets ανά χρήστη. Εάν για παράδειγμα κάποιοι χρήστες έχουν πολύ μεγαλύτερο αριθμό tweets από τους άλλους, πιθανόν να οδηγήσει στο σενάριο όπου κάποιες μηχανές έχουν πολύ περισσότερα δεδομένα να επεξεργαστούν με αποτέλεσμα να έχουμε καθυστέρηση. Η γενική ιδέα του αλγορίθμου είναι να ανιχνεύει πότε υπάρχει σημαντική ανομοιομορφία στα δεδομένα και να «θυσιάζει» λίγο την τοπικότητα δεδομένων με σκοπό να αποφορτίζει τις μηχανές που έχουν αυξημένο όγκο εργασίας. Πιο συγκεκριμένα, αντί να στέλνει τα μηνύματα εκεί που κανονικά θα πήγαιναν, τα στέλνει σε μηχανές που έχουν λιγότερο φόρτο.

Τέλος, έγινε η αξιολόγηση της λύσης και κατά πόσο η προτεινόμενη λύση όντως βοηθά σε καλύτερη επίδοση στα σενάρια όπου υπάρχει μεγάλη ανομοιομορφία στην κατανομή των δεδομένων.

1.2 Συνεισφορά

Αυτή η ΑΔΕ δείχνει τα προβλήματα όταν έχουμε ανομοιομορφία στα δεδομένα και πως αυτή επηρεάζει την επίδοση. Στην συνέχεια, προτείνει μια λύση η οποία βελτιώνει την επίδοση σε σενάρια με διαφορετική ανομοιομορφία.

Η προτεινόμενη λύση είναι σχετικά απλή, δεν απαιτεί κάποιο συγκεκριμένο cluster manager για να εφαρμοστεί, ούτε είναι συνδεδεμένη με μόνο ένα message broker (το

kafka στην περίπτωση μας), ούτε προϋποθέτει εγκατάσταση άλλου λογισμικού. Επίσης, αν και υλοποιήθηκε μόνο για το Spark, μπορεί να μεταφερθεί εύκολα και σε άλλες μηχανές επεξεργασίας δεδομένων.

Κεφάλαιο 2

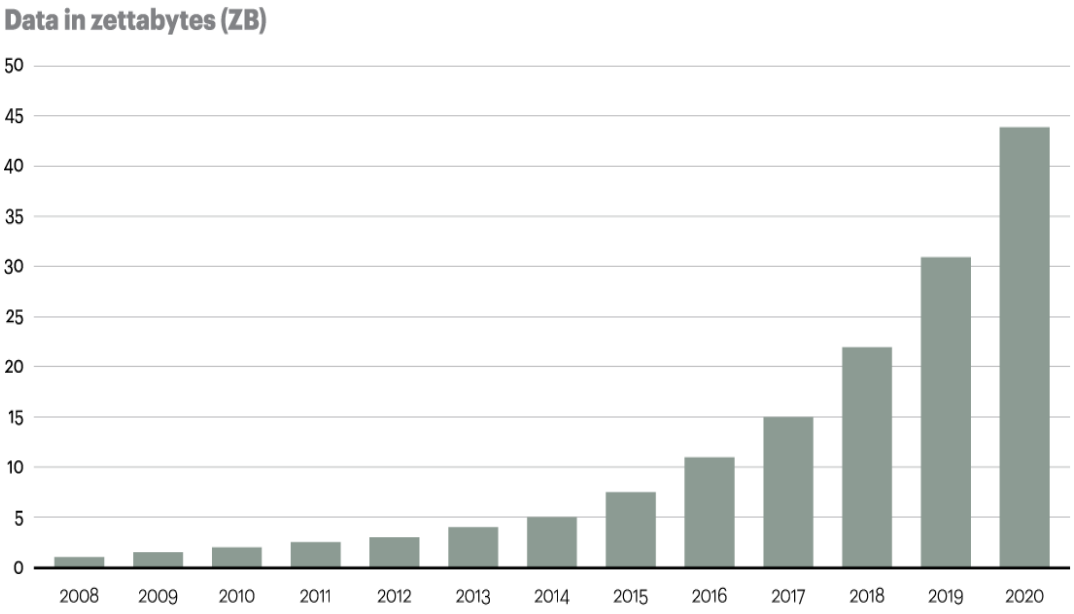
Background

2.1 Big Data

Ο όρος «Big Data» έχει σταματήσει να είναι απλά hype, ή ένα buzzword, είναι πλέον η καθημερινότητα για πολλές εταιρείες/οργανισμούς. Ο όγκος των δεδομένων που δημιουργείται μεγαλώνει εκθετικά.

Figure 1

Data is growing at a 40 percent compound annual rate, reaching nearly 45 ZB by 2020

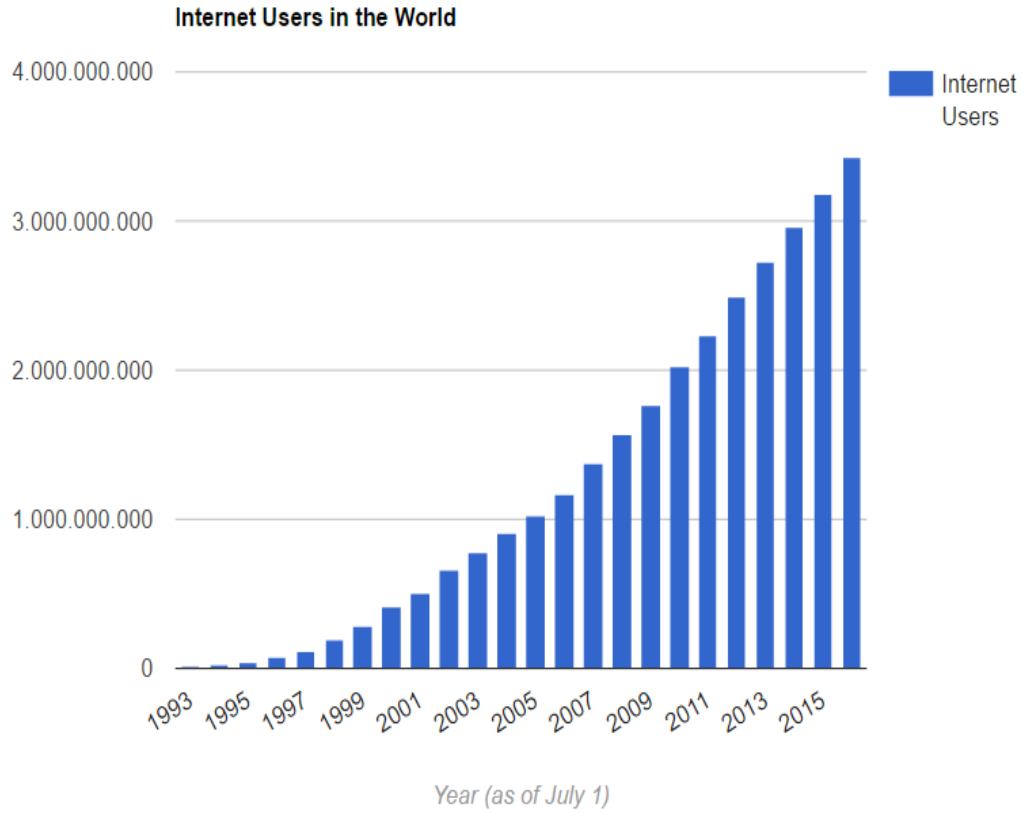


Source: Oracle, 2012

Σχήμα 2.1: Εκτίμηση όγκου δεδομένων παγκόσμια

Ο όγκος των δεδομένων είναι άμεσα συνδεδεμένος με την ποσότητα των χρηστών του διαδικτύου και τα μέσα που μπορούν να παράγουν δεδομένα. Για παράδειγμα, ένα smartphone μπορεί να παράγει δεδομένα από την κάμερα, το μικρόφωνο, το GPS, το accelerometer, το gyroscope, το heart rate monitor, το fingerprint sensor και άλλα.

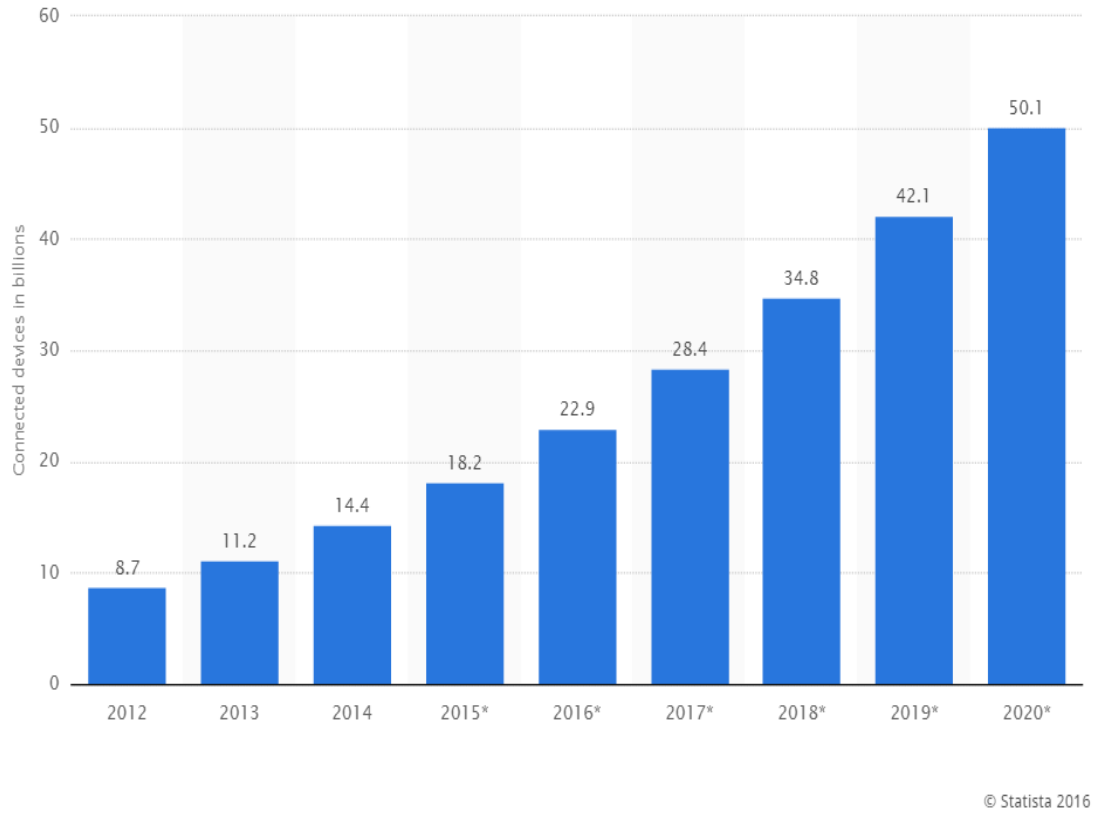
Έτσι, τεράστιο ρόλο στην αύξηση του όγκου δεδομένων έχει και η μείωση του κόστους των smartphones η οποία οδήγησε σε μεγάλη αύξηση των χρηστών που είναι ενωμένοι στο διαδίκτυο σε αναπτυσσόμενες χώρες με μεγάλο πληθυσμό όπως η Ινδία και η Κίνα.



Σχήμα 2.2: Αριθμός χρηστών παγκόσμια[2]

Ωστόσο, αυτό που αναμένετε να εκτοξεύσει τα δεδομένα που θα δημιουργούνται είναι αυτό που λέμε «Internet of Things», δηλαδή όταν τα πάντα (αυτοκίνητα, οικιακές συσκευές, κλπ.) θα είναι ενωμένα στο διαδίκτυο.

[2] http://www.internetlivestats.com/internet-users-by-country/



Σχήμα 2.3: Εκτίμηση για αριθμό ενωμένων συσκευών στο διαδίκτυο

Εκτιμάτε ότι μέχρι το 2020, θα δημιουργούνται 1.7 megabyte δεδομένα το δευτερόλεπτο για κάθε άνθρωπο στον πλανήτη[3], αυτός ο αριθμός είναι περίπου 8 φορές μεγαλύτερος από το average upload speed της Κύπρου.

Τα δεδομένα όμως από μόνα τους είναι άχρηστα, η κάθε εταιρεία/οργανισμός θα πρέπει να εξάγει χρήσιμες πληροφορίες από τον μεγάλο όγκο δεδομένων που έχει.

Όταν όμως έχουμε να κάνουμε με τεράστια μεγέθη δεδομένων, η επεξεργασία με τον παραδοσιακό τρόπο, δηλαδή «τα βάζω σε ένα δίσκο σε ένα υπολογιστή και τα επεξεργάζομαι» είναι είτε αδύνατο λόγο περιορισμένου χώρου είτε θα είναι πολύ αργή η επεξεργασία.

[3] http://www.forbes.com/sites/bernardmarr/2015/09/30/big-data-20-mind-boggling-facts-everyone-must-read/#568260616c1d

Δηλαδή, το vertical scaling (πιο γρήγορος επεξεργαστής, περισσότερη μνήμη) έχει φτάσει στα όρια του και αναγκαστικά πρέπει να πάμε στο horizontal scaling όπου χρησιμοποιούμε πολλές μηχανές.

Ωστόσο, η κατανεμημένη επεξεργασία σε πολλές μηχανές έχει πολλές προκλήσεις και είναι σχετικά ένα πρόσφατο πρόβλημα αφού ενόσω τα δεδομένα ήταν λίγα, οι περισσότερες εταιρείες δεν χρειάζονταν κατανεμημένη επεξεργασία.

Η πιο δημοφιλής λύση σε αυτό το πρόβλημα είναι το μοντέλο «Map Reduce» και μια υλοποίηση του το Hadoop. Αν και το Hadoop πλησιάζει τα 10 χρόνια από το πρώτη έκδοση, και έχουν αναπτυχθεί μηχανές επεξεργασίας δεδομένων με καλύτερη επίδοση, το σύστημα αποθήκευσης του Hadoop (HDFS) θεωρείται ακόμα μία αξιόπιστη λύση και χρησιμοποιείται ευρέως σε πολλά συστήματα. .

2.2 Streaming data processing

Η ιδιαιτερότητα της επεξεργασίας ροής δεδομένων έχει να κάνει με το ότι θέλουμε να εξάγουμε αποτέλεσμα από κάποια δεδομένα όσο πιο σύντομα γίνεται σε σχέση με τον χρόνο που δημιουργήθηκαν.

Αν και τα περισσότερα δεδομένα που δημιουργούνται είναι streaming αφού οι πλήστες συσκευές είναι ενωμένες στο διαδίκτυο συνεχώς, λόγω του μεγάλου όγκου και της πολύπλοκης επεξεργασίας τους, τις περισσότερες φορές η επεξεργασία των δεδομένων γινόταν σε μεταγενέστερο στάδιο. Όμως ο ανταγωνισμός ανάγκασε εταιρείες από όλους τους τομείς να θέλουν τα εκμεταλλευτούν την γνώση που παράγεται από τα δεδομένα όσο πιο γρήγορα γίνεται. Οι εφαρμογές ποικίλουν, από εταιρείες τεχνολογίας που θέλουν να έχουν στοχευμένες διαφημίσεις μέχρι μέσα μαζικής ενημέρωσης που πρέπει να βρουν τα trending topics για να έχουν μεγαλύτερη αναγνωσιμότητα.

Η επεξεργασία μεγάλης κλίμακας δεδομένων σε πραγματικό χρόνο μέσω της ανάπτυξης εφαρμογών στο πλαίσια του «Internet of Things» είναι αναγκαία αφού όλο και περισσότερες συσκευές ενωμένες στο διαδίκτυο στέλνουν δεδομένα. Παράλληλα, όλο και περισσότερες χρονικά κρίσιμες εφαρμογές αναπτύσσονται, όπως για παράδειγμα συστήματα ιατρικής (π.χ. heart rate monitors) ή με αυτόνομα αυτοκίνητα, οι οποίες σε αντίθεση με τις πλείστες εφαρμογές σήμερα (π.χ. διαφημίσεις, προτεινόμενα προϊόντα) δεν έχουν περιθώρια για μεγάλες καθυστερήσεις.

Η κυριότερη αιτία για αυξημένη καθυστέρηση είναι στις περιπτώσεις που έχουμε να επεξεργαστούμε δεδομένα περισσότερα από τις δυνατότητες των μηχανών. Επιπρόσθετα, το πρόβλημα αυξάνεται όταν έχουμε ανομοιομορφία στην κατανομή των δεδομένων με αποτέλεσμα να μην χρησιμοποιούνται ιδανικά οι μηχανές.

Μια άλλη προσέγγιση για την αντιμετώπιση των θεμάτων σχετικά με την επίδοση όταν έχουμε πολλά δεδομένα είναι χρησιμοποιώντας δειγματοληψία αφού έτσι θα επεξεργαζόμαστε μέρος των δεδομένων. Όπως για παράδειγμα η λύση που προτείνεται στο «AdaM: an Adaptive Monitoring Framework for Sampling and Filtering on IoT Devices» όπου γίνεται δυναμική δειγματοληψία αναλόγως των μεταβολών των τιμών, με στόχο να καταγράφονται μόνο τα πιο σημαντικά δεδομένα. Βέβαια, η συγκεκριμένη προσέγγιση επικεντρώνεται περισσότερο στην εξοικονόμηση ενέργειας και έχει εφαρμογή κυρίως σε συσκευές με αισθητήρες που καταγράφουν κάποιες μετρήσεις.

Η λύση που προτείνεται στην διπλωματική μπορεί να συνδυαστεί και με το AdaM framework αφού στο AdaM τα δεδομένα από τις μετρήσεις δεν στέλνονται ανά σταθερά χρονικά διαστήματα. Για παράδειγμα μπορεί να έχουμε κάποιες συσκευές που δεν έχουν μεγάλες αλλαγές στα δεδομένα και δεν στέλνουν συχνά μετρήσεις ενώ κάποιες άλλες έχουν σημαντικές αυξομειώσεις και στέλνουν πολύ πιο συχνά. Αυτό θα έχει σαν αποτέλεσμα να δημιουργηθεί ανομοιομορφία στην κατανομή των μετρήσεων που είναι το πρόβλημα που αντιμετωπίζουμε.

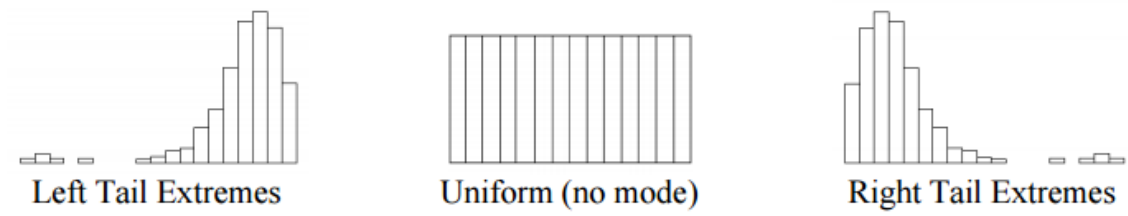
2.3 Skewness στα δεδομένα

Skewness λέμε ότι υπάρχει στα δεδομένα όταν, αναλόγως του τρόπου που τα διαχωρίζουμε για την επεξεργασία, δεν είναι ομοιόμορφα κατανεμημένα. Για παράδειγμα εάν έχουμε μια εφαρμογή με αυτόνομα οχήματα, ομοιόμορφα θα ήταν εάν σε κάθε δρόμο ή περιοχή είχαμε περίπου τον ίδιο αριθμό οχημάτων για να στέλνουν δεδομένα.

Όταν τα δεδομένα είναι ομοιόμορφα κατανεμημένα είναι το ιδανικό σενάριο για τους προγραμματιστές επειδή θα μοιραστεί η εργασία στους διαθέσιμους υπολογιστές. Αυτό είναι το ιδανικό σενάριο για 2 λόγους:

Ο πρώτος λόγος είναι ότι αφού θα έχουν όλοι οι υπολογιστές παρόμοιο φόρτο εργασίας, τότε θα τελειώσουν πολύ πιο γρήγορα παρά εάν κάποιος υπολογιστής είχε διπλάσιες εργασίες επειδή θα καθυστερούσε ενώ οι άλλοι θα ήταν αδρανείς.

Ο δεύτερος λόγος είναι επειδή θα είχαμε τοπικότητα δεδομένων, δηλαδή στο παράδειγμα με τα αυτοκίνητα, σε κάθε υπολογιστή θα είχαμε τα δεδομένα από τα αυτοκίνητα της ίδιας περιοχής. Οπότε, αν θα θέλαμε να κάνουμε επεξεργασία να βρούμε το ιδανικό δρομολόγιο για κάποιο αυτοκίνητο τότε τα απαραίτητα δεδομένα των άλλων αυτοκινήτων θα βρίσκονται σε αυτόν τον υπολογιστή. Οπότε, θα γλυτώναμε αρκετό χρόνο αφού σε αντίθετη περίπτωση, όπου τα δεδομένα βρίσκονταν σε άλλο υπολογιστή του δικτύου, θα ήταν πολύ μεγαλύτερος ο χρόνος ανάκτησης.



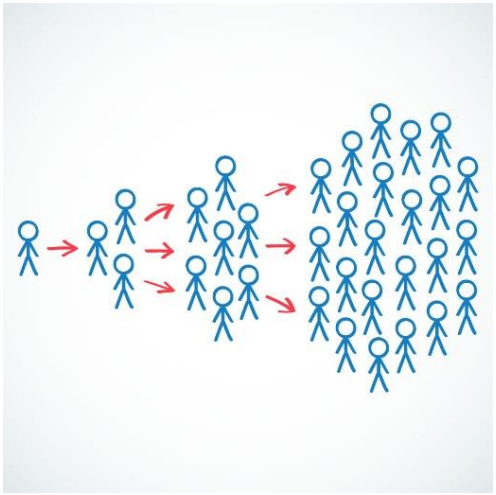
Σχήμα 2.4: Είδη κατανομής δεδομένων[4]

[4] http://www.amstat.org/publications/jse/v19n2/doane.pdf

Όμως αυτό το ιδανικό σενάριο σπάνια συμβαίνει. Ωστόσο, ακόμα και όταν υπάρχει μεγάλη ανομοιομορφία στα δεδομένα, κάποιες φορές οι προγραμματιστές μπορούν να βρουν λύσεις. Στο παράδειγμα με τα αυτοκίνητα μπορεί να ξέρουν από πριν τις πιο δημοφιλείς περιοχές και να κατανέμουν περισσότερους πόρους σε αυτές.

Αυτή η πρόβλεψη όμως στην πραγματικότητα είναι πολύ δύσκολη και κάποιες φορές αδύνατη. Τα δεδομένα δημιουργούνται από εκατομμύρια ανθρώπους που συναναστρέφονται με άλλους ανθρώπους και όλοι λαμβάνουν χιλιάδες αποφάσεις κάθε μέρα.

Ας πάρουμε ένα υποθετικό σενάριο για παράδειγμα: επεξεργαζόμαστε τα trending hashtags του twitter σχετικά με τις προεδρικές εκλογές στις ΗΠΑ γνωρίζοντας ότι οι 2 δημοφιλέστεροι υποψήφιοι είναι η Hillary Clinton και ο Donald Trump. Για αυτό το λόγο έχουμε δεσμεύσει πόρους αποκλειστικά για αυτά τα hashtags. Όμως για κάποιο λόγο, ο Gary Johnson από το Libertarian Party, που πήρε μόλις 1% στις προηγούμενες εκλογές, έχει γίνει trending hashtag και ξεπέρασε τους άλλους 2. Για αυτό το hashtag όμως δεν προβλέψαμε και αυτό είχε σαν αποτέλεσμα να δημιουργήσει μεγάλες καθυστερήσεις στο σύστημα επειδή οι δεσμευμένες μηχανές για τους 2 δημοφιλείς είχαν μικρότερο φόρτο ενώ το hashtag του Gary Johnson ήταν σε μια μηχανή μαζί με άλλα hashtags. Αυτό το ίσως υπεραπλουστευμένο σενάριο είναι πολύ συχνό φαινόμενο αφού στα social media κάτι μπορεί να γίνει «viral» από το «τίποτα».



Σχήμα 2.5: Δημιουργία ενός ξαφνικού trend

2.4 Stateful και Stateless

Είναι σημαντικό να διαχωρίσουμε σε 2 είδη την επεξεργασία σε ροή δεδομένων. Ο διαχωρισμός έχει να κάνει με το εάν χρειάζεται η προηγούμενη γνώση για παραγωγή της επόμενης κατάστασης.

Stateless είναι όταν δεν μας ενδιαφέρει το τι έγινε πριν για να υπολογίσουμε την επόμενη κατάσταση. Για παράδειγμα, εάν θέλουμε να υπολογίσουμε το άθροισμα κάποιων αριθμών, αυτή η πράξη μπορεί να γίνει με ένα απλό map-reduce όπου η κάθε μηχανή παίρνει ίσο αριθμό αριθμών και μετά προστίθενται τα ενδιάμεσα αποτελέσματα από την κάθε μια και βγάζουμε το τελικό. Σε αυτή την κατηγορία δεν μας ενδιαφέρει η τοπικότητα δεδομένων και για αυτό η καλύτερη πρακτική είναι απλά να κάνουμε shuffle τα δεδομένα στις διάφορες μηχανές για να έχουν όλες παρόμοιο όγκο εργασίας. Άρα εδώ δεν μας επηρεάζει αρνητικά η ανομοιομορφία στην κατανομή των δεδομένων.

Αντίθετα, η άλλη κατηγορία η οποία μας ενδιαφέρει είναι τα Stateful. Εδώ είναι όταν χρειαζόμαστε πληροφορίες της προηγούμενης κατάστασης για κάνουμε την επεξεργασία. Για παράδειγμα εάν κάνουμε επεξεργασία χρησιμοποιώντας κινούμενο παράθυρο και κρατούμε πληροφορίες για τον κάθε χρήστη, τότε το ιδανικό θα ήταν όλα τα νέα δεδομένα που αφορούν τον ίδιο χρήστη να πηγαίνουν στην ίδια μηχανή, σε διαφορετική περίπτωση θα πρέπει να γίνει μεταφορά της κατάστασης σε πολλές μηχανές. Οπότε, στις περιπτώσεις όπου δεν υπάρχει ομοιόμορφη κατανομή των δεδομένων δημιουργείται το πρόβλημα που θα προσπαθήσουμε να λύσουμε.

2.5 Άλλες προσεγγίσεις επεξεργασίας ροής δεδομένων

Μια σχετική έρευνα με το θέμα της διπλωματικής έχει τίτλο «When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing» η οποία προτείνει 2 αλγορίθμους και δείχνει πως υπερέχουν σε σχέση με προηγούμενες συνηθισμένες τακτικές όταν υπάρχει ανομοιομορφία στα δεδομένα.

Η λύση που προτείνεται στην συνέχεια είναι παρόμοια με την μία από τις 2 προτάσεις αυτής της έρευνας. Η υλοποίηση των αλγορίθμων στην εν λόγο έρευνα έγιναν στην μηχανή Apache Storm.

Κεφάλαιο 3

Σχεδίαση και Υλοποίηση

3.1 Προδιαγραφές λύσης

Οι προδιαγραφές της προτεινόμενης λύσης είναι οι εξής:

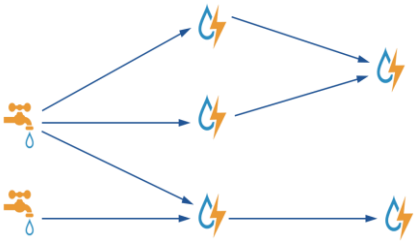
- Επίδοση: Η λύση πρέπει να αντιμετωπίζει το πρόβλημα του skewness στα δεδομένα και να προσφέρει καλύτερη επίδοση σε σχέση με την προκαθορισμένη λειτουργία (Hashing).
- Φορητότητα: Η προτεινόμενη λύση παρόλο που θα αναπτυχθεί σε μόνο μια μηχανή επεξεργασίας δεδομένων, θα πρέπει να μπορεί εύκολα να μεταφερθεί σε άλλη μηχανή.
- Συμβατότητα: Είναι σημαντικό η λύση να μην είναι στενά συνδεδεμένη με κάποιο συγκεκριμένο message broker (π.χ kafka) ούτε με κάποιο συγκεκριμένο cluster manager (π.χ YARN) έτσι ώστε να μπορεί να εφαρμοστεί σε διάφορα περιβάλλοντα.
- Παραμετροποιησιμότητα: Η λύση δεν θα μπορεί να εφαρμοστεί μόνο για κάποιο συγκεκριμένο πρόβλημα, φόρτο εργασίας και δυνατότητες υλικού. Θα πρέπει να μπορεί να παίρνει παραμέτρους για καλύτερη επίδοση αναλόγως του είδους του προβλήματος αλλά και του περιβάλλοντος.

3.2 Επιλογή μηχανής επεξεργασίας δεδομένων για υλοποίηση

Λόγω των τεράστιων αναγκών για γρήγορη επεξεργασία δεδομένων όπως αναφέραμε και πριν, τα τελευταία χρόνια έχουν αναπτυχθεί πολλές μηχανές. Έτσι, αρχικά καταλήξαμε σε μια λίστα από 4 υποψήφιες μηχανές: Spark, Storm, Flink και Samza.

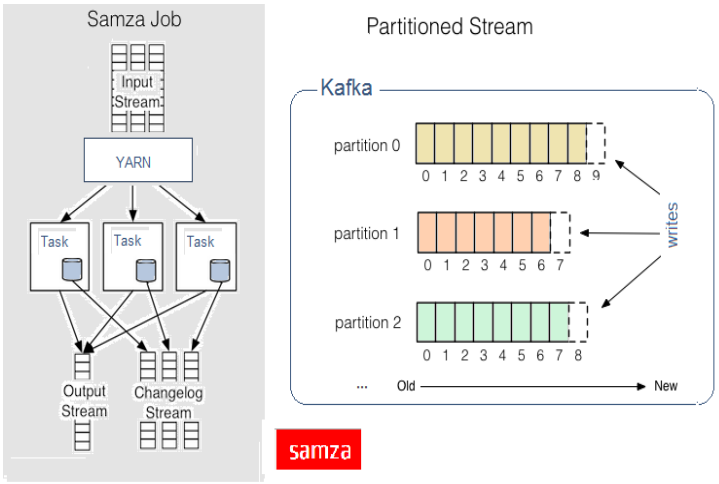
Και οι 4 είναι open source και υπάγονται στο Apache Foundation.

Storm[5]: Είναι η πρώτη που δημιουργήθηκε από τις υποψήφιες γι' αυτό και είναι και η πιο διαδεδομένη, έχει δοκιμαστεί σε πολύ μεγάλα cluster σε μεγάλες εταιρείες για μερικά χρόνια[6]. Ειδικεύεται αποκλειστικά στο stream processing και επικεντρώνεται στην χαμηλή καθυστέρηση.



Σχήμα 3.1: Επεξεργασία στο Storm

Samza[7]: Όπως και το Storm, ασχολείται αποκλειστικά με stream processing. Αναπτύχθηκε στο LinkedIn για τις ανάγκες της εταιρείας αλλά αργότερα έχει ανοικτού κώδικα. Σε σύγκριση με το Storm προσφέρει περισσότερες δυνατότητες όσων αφορά την διαχείριση των καταστάσεων σε stateful operations[8].

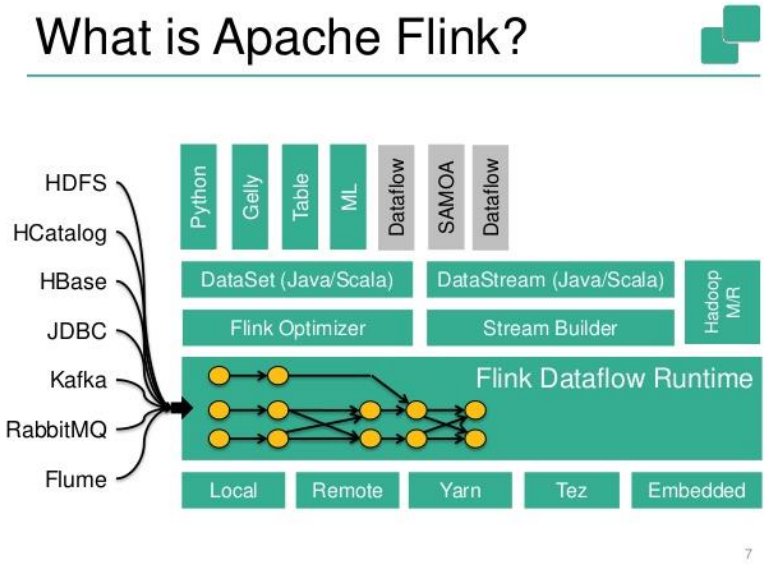


Σχήμα 3.2: Επεξεργασία στο Samza

[5] http://storm.apache.org/
[6] http://storm.apache.org/releases/current/Powered-By.html
[7] http://samza.apache.org/
[8] https://samza.apache.org/learn/documentation/0.10/comparisons/storm.html

Spark[9]: Σε αντίθεση με τα προηγούμενα, το stream processing είναι μία από τις πολλές δυνατότητες του Spark αφού έχει βιβλιοθήκες και για άλλες χρήσεις (machine learning, graphs, SQL). Είναι σχετικά καινούργια αλλά έχει γίνει πολύ δημοφιλής κυρίως λόγο των πολλών δυνατοτήτων της. Σημαντική διαφορά με τις υπόλοιπες είναι ότι μπορεί να επεξεργαστεί τα δεδομένα μόνο σε δέσμες (batches) και όχι ένα-ένα.

Flink[10]: Μοιάζει πολύ με το Spark ως προς τις δυνατότητες της αφού και αυτή έχει βιβλιοθήκες για πολλές πιο εξιδεικευμένες χρήσεις. Σε αντίθεση όμως με το Spark, δεν τα επεξεργάζεται με δέσμες αλλά όπως και τα Storm, Samza, δηλαδή μόλις φθάσουν τα δεδομένα.



του Flink

[9] http://spark.apache.org/
[10] https://flink.apache.org/

	Spark	**Storm**	**Flink**	**Samza**
Target	General data processing	Stream only	General data processing	Stream only
Compatible Cluster Managers	Standalone, YARN, MESOS	zookeper, MESOS, YARN (partially supported)	Standalone, YARN	YARN
Processing Model	Σε δέσμες	Ένα κάθε φορά	Ένα κάθε φορά	Ένα κάθε φορά
Delivery guarantees	All guarantee “at least once”			
Community (github)	894 contributors/ 8719 stars[11]	214 contributors/ 3202 stars[12]	185 contributors/ 1326 stars[13]	54 contributors / 202 stars[14]

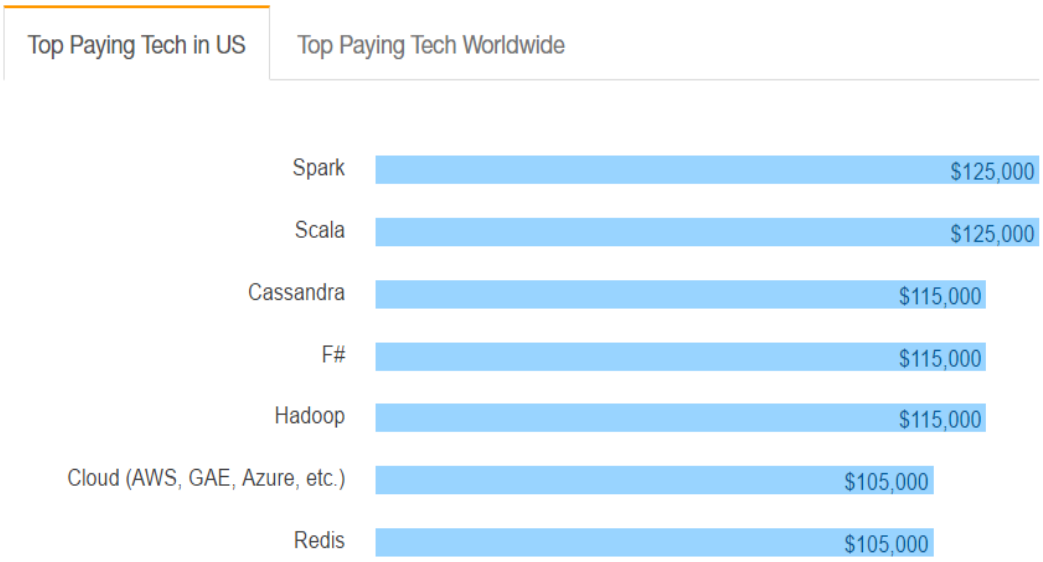
Πίνακας 3.1: Σύγκριση μηχανών επεξεργασίας

Όσον αφορά τον cluster manager, οι περισσότερες μηχανές υποστηρίζουν αρκετές επιλογές και όλες το YARN. Επίσης όλες οι μηχανές εγγυούνται ότι το κάθε μήνυμα θα επεξεργαστεί τουλάχιστο 1 φορά.

Όπως έγινε αναφορά και προηγουμένως, υπάρχει μια σημαντική διαφορά στον τρόπο που επεξεργάζονται τα δεδομένα αφού το Spark περιμένει να μαζευτούν κάποια μηνύματα πρώτα και μετά να ξεκινήσει την επεξεργασία, σε αντίθεση με τα υπόλοιπα συστήματα που ξεκινούν μόλις φτάσει κάποιο μήνυμα. Θεωρητικά, το μοντέλο του Spark έχει μεγαλύτερο latency αλλά μεγαλύτερο throughput.

Ακόμα ένα χαρακτηριστικό είναι ότι το Spark έχει πολλούς περισσότερους contributors στο git repository του ενώ γενικά δείχνει να είναι αυτό που εξελίσσεται πιο γρήγορα αφού έχει και πάρα πολλά Pull requests.

[11] https://github.com/apache/spark
[12] https://github.com/apache/storm
[13] https://github.com/apache/flink
[14] https://github.com/apache/samza

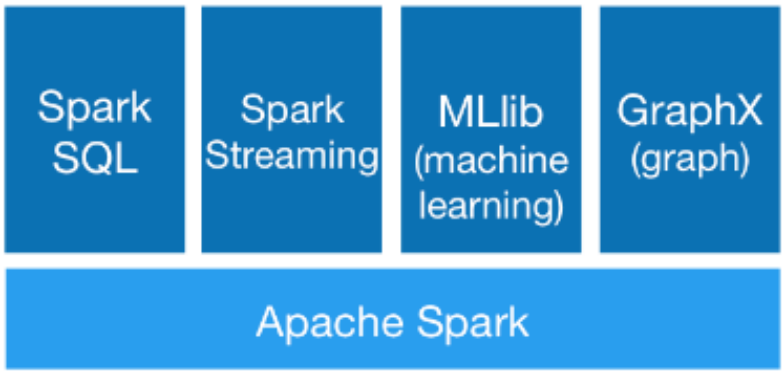


Σχήμα 3.4: Μέσος όρος ετήσιου μισθού ανά τεχνολογία

Στο φετινό ερωτηματολόγιο του stackoverflow[15], το Spark είναι πρώτο στις πιο ψηλά αμειβόμενες τεχνολογίες. Αυτό είναι σημαντικό στοιχείο επειδή δείχνει ότι οι εταιρείες ενδιαφέρονται για το Spark.

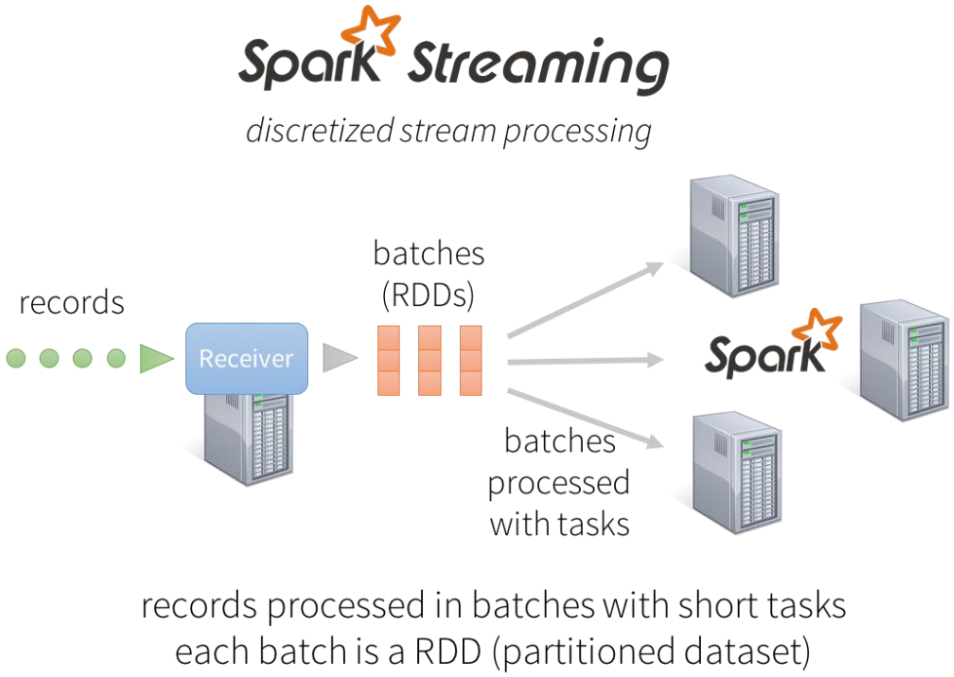
Όπως βλέπουμε, η μεγάλη ζήτηση για γρήγορη επεξεργασία δεδομένων οδήγησε στην παράλληλη ανάπτυξη πολλών μηχανών και ίσως κάποιος να ισχυριστεί ότι η κατηγορία είναι υπερπλήρης. Τελικά επιλέγηκε το Spark, κυρίως λόγω του momentum και της δημοσιότητας που έχει ανάμεσα στους προγραμματιστές αλλά και το γεγονός ότι αν και σχετικά καινούργιο οι εταιρείες ήδη το χρησιμοποιούν σε production environments. Επίσης είναι ενδιαφέρον η αρχιτεκτονική του επειδή μπορεί κανείς εύκολα να συνδυάσει διάφορες κατηγορίες επεξεργασίας χωρίς να χρειάζεται διαφορετικές μηχανές, για παράδειγμα να παίρνει δεδομένα ροής και μετά να εκτελεί αλγόριθμους μηχανικής μάθησης.

[15] http://stackoverflow.com/research/developer-survey-2016#technology-top-paying-tech



Σχήμα 3.5: Βιβλιοθήκες του Spark

Το Spark, όπως φαίνεται και στο πιο πάνω σχήμα, έχει ένα κοινό πυρήνα και πάνω στον πυρήνα στηρίζονται οι εξειδικευμένες βιβλιοθήκες (streaming, machine learning κτλπ). Για αυτό τον λόγο, για να είναι συμβατό το streaming library με τον πυρήνα, επεξεργάζεται τα δεδομένα που έρχονται σε μικρές ομάδες και όχι το καθένα μόνο του.



Σχήμα 3.6: Διαδικασία επεξεργασίας ροής δεδομένων στο Spark

Αυτή η ιδιαιτερότητα έχει εκτός από θετικά και αρνητικά θέματα. Το προφανές αρνητικό είναι ότι έχουμε περισσότερη καθυστέρηση καθώς το spark περιμένει κάποιο χρόνο (batch interval) για να συλλεχθούν τα δεδομένα. Αυτός ο χρόνος όμως μπορεί να ρυθμιστεί και έτσι εάν θέλουμε περισσότερο throughput να το μεγαλώσουμε ή αν θέλουμε μικρότερη καθυστέρηση να το μειώσουμε.

Σχήμα 3.7: Διαδικασία δημιουργίας batches δεδομένων από την ροή

3.3 Περιγραφή σχετικών τεχνολογιών

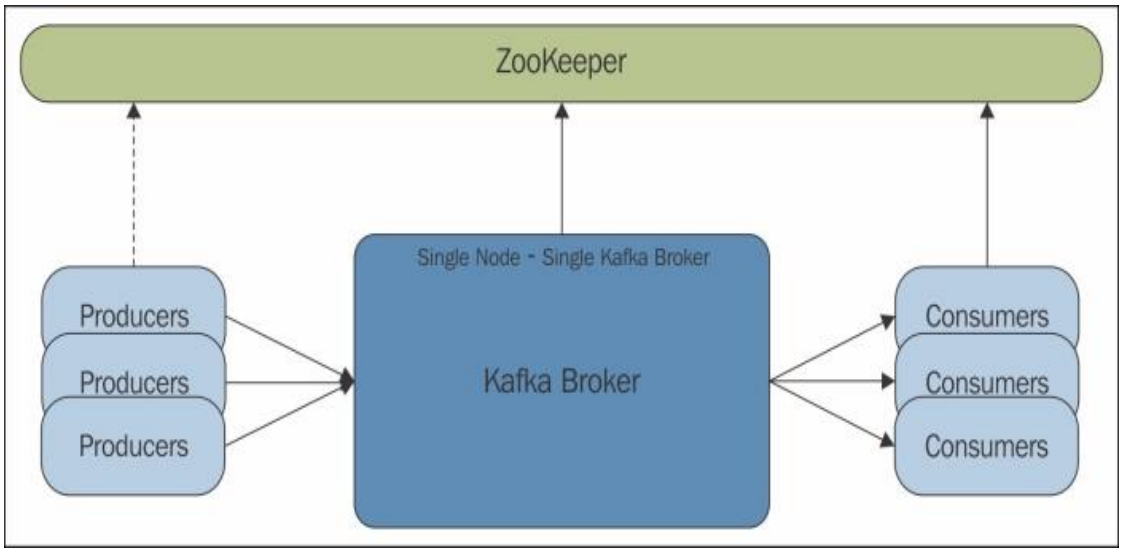
Για την υλοποίηση της λύσης αλλά και των πειραμάτων, εκτός από το Spark, χρησιμοποιούνται και άλλες τεχνολογίες.

Πρώτα έπρεπε να γίνει επιλογή της γλώσσας στην οποία θα γίνει η υλοποίηση του αλγορίθμου. Το Spark έχει API για Java, Scala και Python. Κυρίως η επιλογή ήταν μεταξύ Java και Scala επειδή το API της Python στο Spark δεν έχει όλες τις δυνατότητες με αυτό της Java και Scala.

Τελικά επιλέγηκε η Scala επειδή και το ίδιο το Spark είναι γραμμένο σε Scala. Αυτό έχει 2 πλεονεκτήματα, πρώτον ότι το API του Spark για Scala είναι το πιο ενημερωμένο, με τις περισσότερες δυνατότητες και δεύτερο το ότι είναι πιο εύκολο να αντιληφθεί κάποιος τι γίνεται στην μηχανή βλέποντας τον πηγαίο κώδικα, άρα πρέπει να γνωρίζει Scala.

Ακόμα ένα πλεονέκτημα της Scala, το οποίο δεν έχει σχέση με το Spark, είναι το ότι προσφέρει δυνατότητες συναρτησιακού προγραμματισμού, ο οποίος είναι πολύ σχετικός όταν έχουμε να κάνουμε με ανάλυση δεδομένων. Έτσι, είναι ακόμα πιο βοηθητικό εάν κάποιος ασχολείται με ανάλυση δεδομένων και «Big Data» να γνωρίζει και Scala.

Στην συνέχεια, έπρεπε να γίνει επιλογή για το πως θα δημιουργείται η ροή δεδομένων, όπου επιλέχθηκε το Kafka cluster. Το Kafka είναι ένα σύστημα για μεταφορά μηνυμάτων το οποίο είναι σχεδιασμένο για πολύ μεγάλο όγκο δεδομένων και πολλές μηχανές. Δημιουργήθηκε στο LinkedIn αλλά αργότερα έγινε ανοικτού κώδικά και αυτό κάτω από το Apache Foundation, σήμερα χρησιμοποιείτε σε μεγάλη κλίμακα από εταιρείες όπως Yahoo, Twitter, Netflix, Uber, PayPal, Cisco και πάρα πολλές άλλες.

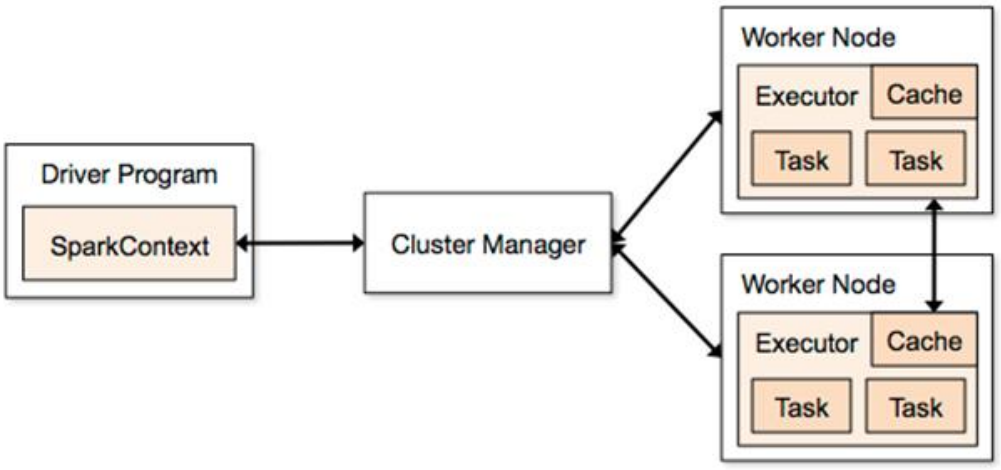


Σχήμα 3.8: Τοπολογία του Kafka cluster

Το Kafka διαχωρίζει τα μηνύματα σε διάφορα topics (κατηγορίες). Οι producers δημιουργούν τα μηνύματα κάποιου topic και τα στέλνουν σε κάποιο Broker. Οι consumers κάνουν «συνδρομή» σε κάποιο topic και ο broker τους στέλνει τα μηνύματα του topic που είναι «συνδρομητές». Ο zookeeper έχει το ρόλο του cluster manager όταν

υπάρχουν πολλοί brokers αλλά κι άλλες λειτουργείες για τον συντονισμό ανάμεσα στα διάφορα μέρη.

Για Cluster Manager έχουμε να επιλέξουμε ανάμεσα σε Mesos, YARN και τον Standalone του Spark. Αν και σε production environments χρησιμοποιούνται περισσότερο οι Mesos, YARN, δεν επηρεάζει και πολύ την συμπεριφορά στο πείραμα. Αυτοί οι πιο πολύπλοκοι και με περισσότερες δυνατότητες cluster managers κυρίως είναι χρήσιμοι όταν θέλουμε να τρέξουμε πολλά διαφορετικά applications στις ίδιες μηχανές και για πολύ μεγάλο αριθμό μηχανών. Άρα επιλέγουμε τον Standalone του Spark αφού είναι ιδανικός για το πείραμα, εξάλλου όπως είπαμε δεν θέλουμε να συνδέσουμε στενά την λύση με κάποιο cluster manager. Αυτό θα ήταν αρνητικό σε περίπτωση που χρησιμοποιούσαμε ως μέρος της λύσης κάποια δυνατότητα που έχει για παράδειγμα μόνο το Mesos και όχι το YARN, έτσι θα δούλευε μόνο στο Mesos.



Σχήμα 3.9: Πως διανέμονται οι δουλειές στους workers του Spark cluster

3.4 Dataset και προεργασία

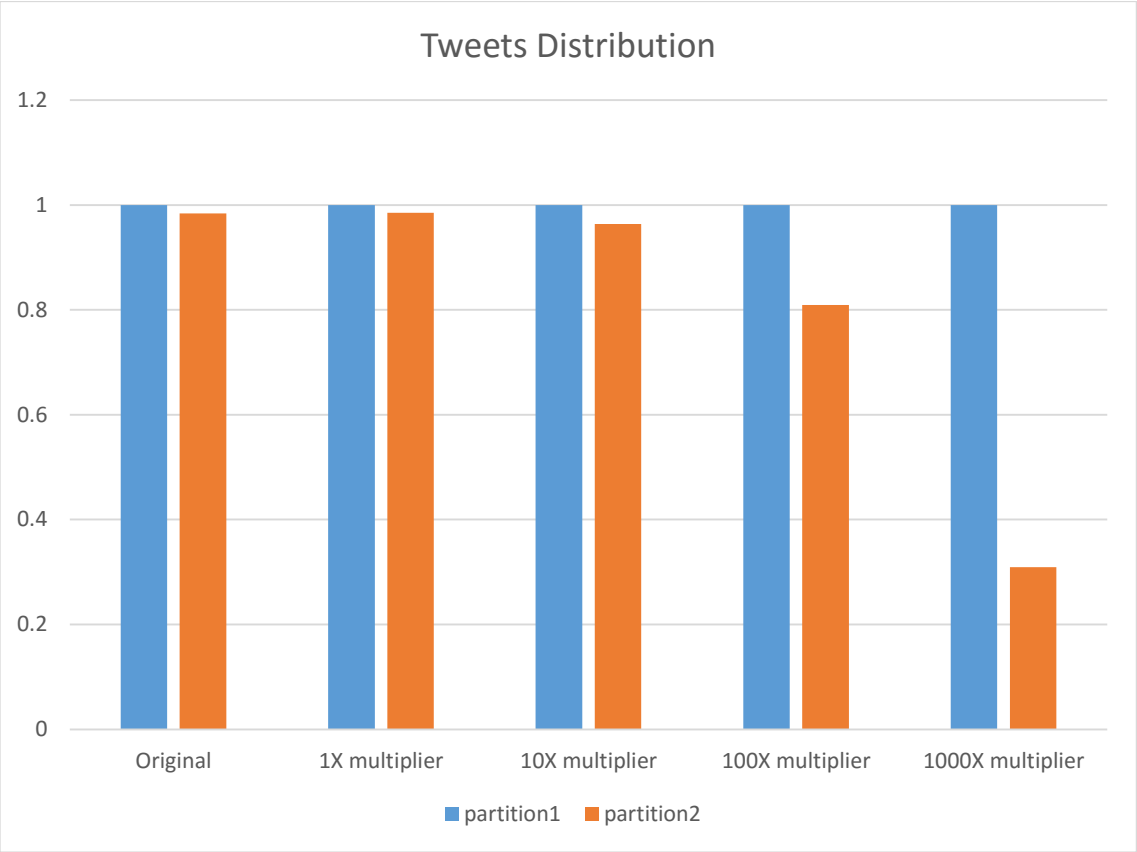
Τα δεδομένα με τα οποία ασχοληθήκαμε είναι πραγματικά tweets από το Twitter. Συγκεκριμένα είχαμε ένα σύνολο δεδομένων με περίπου 10 εκατομμύρια tweets.

Αρχικά έπρεπε να γίνει επεξεργασία στο αρχικό σύνολο έτσι ώστε να δημιουργηθούν διαφορετικά σύνολα δεδομένων με διαφορετικό skewness. Έτσι θα δούμε πως ανταποκρίνεται ο αλγόριθμος σε διαφορετικά σενάρια skewness.

Θα έπρεπε να γίνει ένα μικρό πρόγραμμα το οποίο να παίρνει σαν είσοδο το πόσο skewed θέλουμε να είναι το νέο σύνολο δεδομένο, χρησιμοποίησα Python για την υλοποίηση του αφού είναι πολύ εύκολη γλώσσα με πολλές έτοιμες συναρτήσεις.

Τα δεδομένα ήταν σε μορφή “ <user id>$%<tweet id>$%<tweet> “.

Ο τρόπος για την δημιουργία του skewed συνόλου ήταν να βρεθούν οι X χρήστες με τα περισσότερα tweets από όλο το σύνολο, και μετά όπου είχαν tweet οι χρήστες αυτοί, να προστεθούν Y νέα tweets με τυχαία συμβολοσειρά. Ο λόγος που είναι τυχαία η συμβολοσειρά και όχι η ίδια με το πραγματικό tweet είναι για να αποφευχθούν τυχόν βελτιστοποιήσεις με caching από το Spark και να είναι ήταν πιο γρήγορο απ’ ότι σε πραγματικό σενάριο που είναι διαφορετικά τα tweets. Επίσης, κατά την διάρκεια της δημιουργίας του νέου συνόλου, έσβηνε τυχαία tweets για να μειωθεί ο αριθμός των άλλων tweet σε σχέση με αυτά των δημοφιλών χρηστών για να υπάρχει μεγαλύτερο skewness.



Σχήμα 3.10: Κατανομή των δεδομένων για κάθε σύνολο

Για το πείραμα είχαμε 2 partitions για να διαχωρίζονται τα δεδομένα. Δημιουργήθηκαν 4 νέα σύνολα δεδομένων με διαφορετικό αριθμό νέων tweet στον χρήστη με τα περισσότερα tweets, συγκεκριμένα με 1,10,100 και 1000. Η κατανομή στο γράφημα υποθέτει ότι ο διαμοιρασμός των tweets στα 2 partitions γίνεται με Hashing που είναι και ο προκαθορισμένος και ο πιο συνηθισμένος τρόπος.

3.5 Περιγραφή και υλοποίηση αλγορίθμου

Η βασική ιδέα του προτεινομένου αλγορίθμου είναι να βρίσκει μια ισορροπία μεταξύ shuffle partitioning και hash partitioning. Σκοπός είναι να κρατά όλα τα πλεονεκτήματα της τοπικότητας χρησιμοποιώντας το hash partitioning μέχρι το σημείο όπου το skewness είναι τόσο μεγάλο τέτοιο ώστε να είναι προτιμότερο να γίνει και κάποιο shuffle σε άλλες μηχανές.

Δηλαδή αρχικά θα κάνει κανονικά hashing μέχρι να εντοπίσει skewness στα δεδομένα. Εάν το skewness ξεπεράσει ένα προκαθορισμένο threshold για κάποιο συγκεκριμένο partition, τότε θα τα δεδομένα που κανονικά γίνονταν hash στο συγκεκριμένο θα σταλούν σε άλλο partition το οποίο υπολογίζουμε ότι είναι το λιγότερο βαρυφορτωμένο.

Ο ψευδοκώδικας για τον Partitioner που θα αποφασίζει σε πιο partition θα πάει το κάθε μήνυμα είναι ως ακολούθως:

Αρχικοποίηση πίνακα κατανομών
Για κάθε μήνυμα:
    Υπολόγισε Η = partition με hash
    Αν η κατανομή του Η partition > threshold:
        Στείλε μήνυμα στο partition με την μικρότερη κατανομή
    Αλλιώς:
        Στείλε στο Η partition
    Ανανέωσε Counters

Κάθε Χ μηνύματα ανανέωσε πίνακα κατανομών

Για την ανίχνευση του skewness κρατούμε μια σύνοψη των πρόσφατων δεδομένων. Πιο συγκεκριμένα, το «Counters» είναι μια ουρά η οποία κρατάει counters για όλα τα partitions ανά κάποιο χρονικό διάστημα. Δηλαδή εάν κρατούμε counters ανά 5 seconds και κρατούμε συνολικά για μισό λεπτό τότε η ουρά θα έχει 6 πίνακες counters και κάθε φορά που περνούν 5 δευτερόλεπτα τότε οι παλιοί counters θα βγαίνουν από την ουρά.

Εδώ γίνεται η υπόθεση ότι μετά από κάποιο χρονικό διάστημα αποκλείεται να υπάρχουν δεδομένα στις ουρές των μηχανών. Σε διαφορετική περίπτωση για να μαθαίναμε θα έπρεπε να γίνει επικοινωνία είτε στο επίπεδο της μηχανής επεξεργασίας, είτε με τον cluster manager, είτε σε επίπεδο λειτουργικού. Ωστόσο αποφεύχθηκε αυτή η λύση επειδή θα χρειαζόταν η συνεργασία με κάποιο από τα προηγούμενα επίπεδα και αυτό θα αφαιρούσε από την απλότητα της λύσης αλλά και για λόγους συμβατότητας αφού ίσως αυτή η λειτουργία να μην είναι δυνατή σε όλες της μηχανές επεξεργασίας.

Ο πίνακας των κατανομών είναι η άλλη βοηθητική δομή για να υπολογίζουμε πως είναι ο φόρτος στις διάφορες μηχανές. Ο υπολογισμός θα γίνεται κάθε κάποιο αριθμό μηνυμάτων και όχι σε κάθε μήνυμα για να μην υπάρχει καθυστέρηση. Για τον υπολογισμό θα χρησιμοποιεί τα δεδομένα που έχουμε στους Counters ενώ ο αριθμός των μηνυμάτων που καταγράφηκαν πρόσφατα έχουν μεγαλύτερη βαρύτητα αφού είναι πιο πιθανό να παίρνουν σειρά για επεξεργασία. Για παράδειγμα εάν έχουμε καταγραμμένο ότι πριν 30 δευτερόλεπτα είχε μεγάλο φόρτο η μηχανή Α και στα προηγούμενα 5 η μηχανή Β, τότε ο αλγόριθμός θα λάβει περισσότερο υπόψη αυτά της Β αφού αυτά της Α το πιο πιθανό είναι ότι μέχρι τώρα ήδη επεξεργάστηκαν.

Πιο συγκεκριμένα, κάθε Χ μηνύματα, θα υπολογίζεται ο μέσος όρος ανάμεσα σε όλα τα partitions και με βάση αυτό η μετρική των κατανομών. Οι counters για τα λιγότερο πρόσφατα θα μειώνονται και θα αποθηκεύεται το partition με την λιγότερη κατανομή έτσι ώστε μετά να ξέρουμε πιο έχει τον λιγότερο φόρτο για να στέλνουμε σε αυτό μετά.

Στον αλγόριθμο υπάρχουν οι παρακάτω παραμετροποιήσημες σταθερές:

- Threshold: Πόσο ευαίσθητος στο skewness θα είναι ο partitioner, με μεγάλο threshold τότε μόνο σε πολύ ακραία σενάρια με μεγάλο skewness θα ενεργοποιείται ο αλγόριθμος για να αποφορτώνει της μηχανές με μεγάλο φόρτο, ενώ με μικρό threshold θα προσπαθεί συνεχώς να κρατά ισορροπημένο φόρτο.
- Χρόνος μνημόνευσης counters: Πόσο «παλιά» θα θυμούνται οι counters για να λαμβάνουν υπόψη. Με μεγάλη τιμή τότε ο αλγόριθμος σε κάποιες περιπτώσεις θα αργεί περισσότερο να αντιδράσει ωστόσο θα ανταποκρίνεται καλύτερα σε

σενάρια όπου ο όγκος δεδομένων είναι πολύ μεγαλύτερος από την δυνατότητα των μηχανών και έχουμε πολύ μεγάλες ουρές.

Αυτή είναι και μια προδιαγραφή που ορίσαμε αρχικά, δηλαδή αναλόγως της εφαρμογής αλλά και του περιβάλλοντος να μπορούμε να εκμεταλλευτούμε καλύτερα τον αλγόριθμο. Εάν για παράδειγμα έχουμε να επεξεργαστούμε δεδομένα που έχουν μεγάλη ανομοιομορφία στην κατανομή των δεδομένων και το γνωρίζουμε από πριν αλλά ορίσουμε μικρό Threshold τότε ίσως έχουμε χειρότερα αποτελέσματα. Πιο συγκεκριμένα, λόγο της υπερβολική προσπάθειας του αλγορίθμου να ισορροπήσει τις κατανομές ίσως να δημιουργήσει περισσότερη κίνηση στο δίκτυο ή να χάσουμε επίδοση λόγο της μειωμένης τοπικότητας των δεδομένων.

Ακόμα ένα σημαντικό σημείο είναι ότι ο αλγόριθμος θα πρέπει να μην είναι πολύ πολύπλοκος έτσι ώστε να επιβαρύνει το πρόγραμμα. Από την στιγμή που προσπαθούμε να βελτιώσουμε την επίδοση δεν θα είχε νόημα να είχαμε ένα αλγόριθμο που καθυστερούσε το πρόγραμμα περισσότερο από την βελτίωση που θα έκανε. Βέβαια θα φανεί και μετά στα πειράματα εάν επιβαρύνει το πρόγραμμα αλλά ακόμα και πριν την υλοποίηση φαίνεται ότι δεν θα έχει ιδιαίτερο κόστος επεξεργασίας. Εκτός από το hashing που θα γινόταν ούτως ή άλλως, η άλλη διαδικασία που πρέπει να γίνεται σε κάθε μήνυμα είναι η ανανέωση ενός μετρητή που είναι απλή πράξη. Οι υπόλοιπες πράξεις που είναι πιο ακριβές υπολογιστικά δεν εκτελούνται σε κάθε μήνυμα, αλλά μετά από κάποιο αριθμό μηνυμάτων, έτσι ακόμα και εάν φανεί ότι επιβαρύνει το πρόγραμμα μπορεί να ρυθμιστεί.

Κεφάλαιο 4

Μετρήσεις και αποτελέσματα

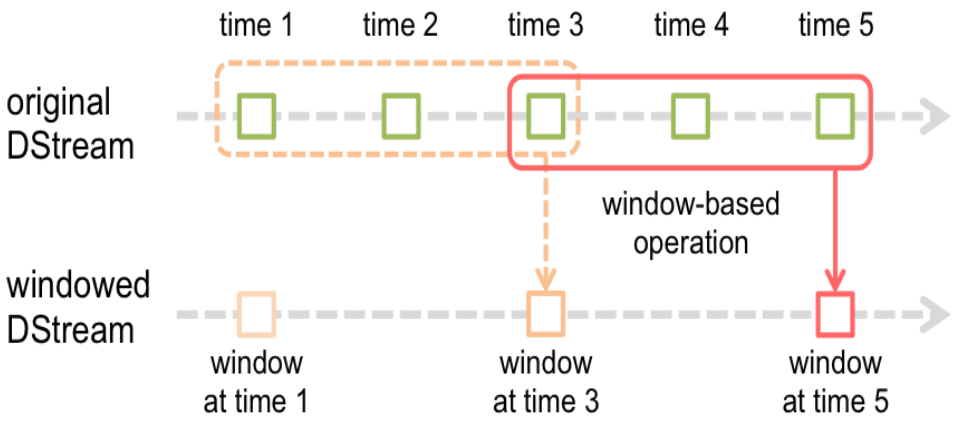
4.1 Περιγραφή πειράματος

Η αρχιτεκτονική των πειραμάτων αποτελείται από το σύστημα του Kafka παράγει τα δεδομένα και το Spark cluster που γίνεται η επεξεργασία.

Το kafka αποτελείται από ένα zookeeper (cluster manager) και ένα Broker. Ένα πρόγραμμα σε python είναι ο producer ο οποίος διαβάζει το αρχείο με τα tweets και τα στέλνει στον broker ο οποίος τα στέλνει στον Spark receiver και έτσι δημιουργείται ένα περιβάλλον με δεδομένα ροής παρόλο που τα δεδομένα είναι σε αρχείο.

Ο Spark cluster αποτελείται από ένα Master και 2 workers. Ο master είναι ο συντονιστής και δεν εκτελεί κάποια διεργασία για την επεξεργασία των δεδομένων, ενώ οι 2 workers είναι σε αυτούς που θέλουμε να διαμοιράζουμε τις εργασίες.

Το πρόγραμμα που εκτελείται για τα πειράματα όπως αναφέρθηκε και πριν είναι γραμμένο σε Scala, ενώ η κύρια επεξεργασία θα γίνεται χρησιμοποιώντας το reduceByKeyAndWindow του Spark.



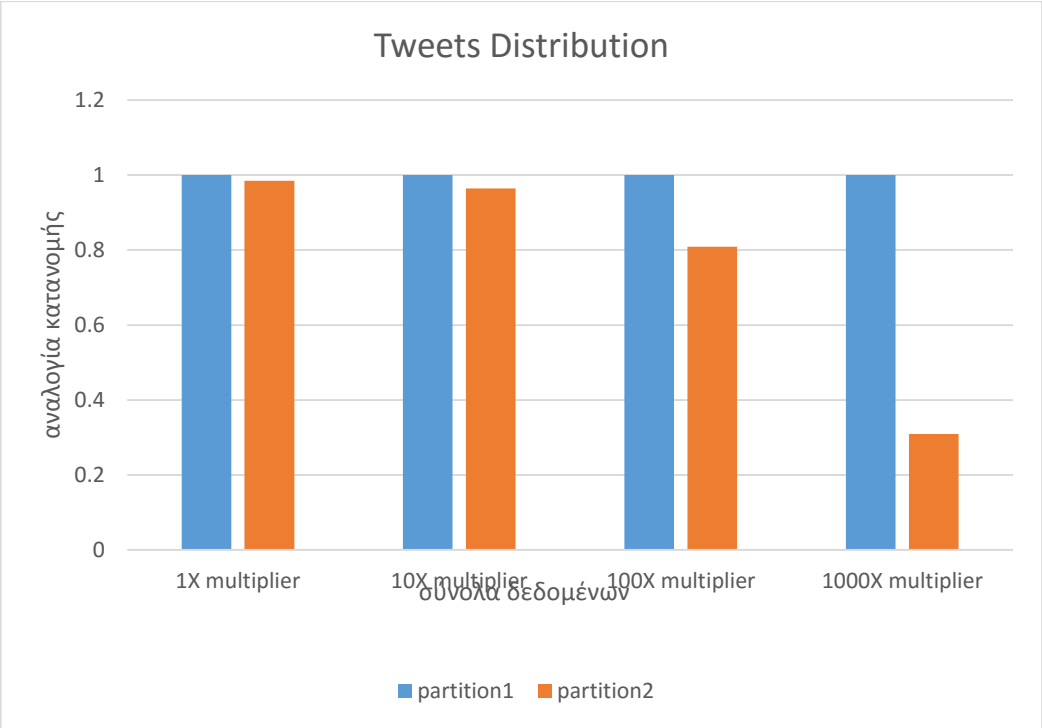
Σχήμα 4.1: Επεξεργασίες χρησιμοποιώντας παράθυρο

Αυτή η λειτουργεία επεξεργάζεται τα δεδομένα με βάση το κλειδί σε κάποιο παράθυρο, στην περίπτωση μας το κλειδί είναι το id του user από το Twitter.

4.2 Μετρήσεις με τις προεπιλεγμένες ρυθμίσεις

Η προεπιλεγμένη επιλογή για τον partitioner του Spark είναι χρησιμοποιώντας τον Hash Partitioner, ο οποίος κάνει hash με βάση το κλειδί και έτσι στέλνει πάντα τα tweets του ίδιου user στο ίδιο partition/μηχανή.

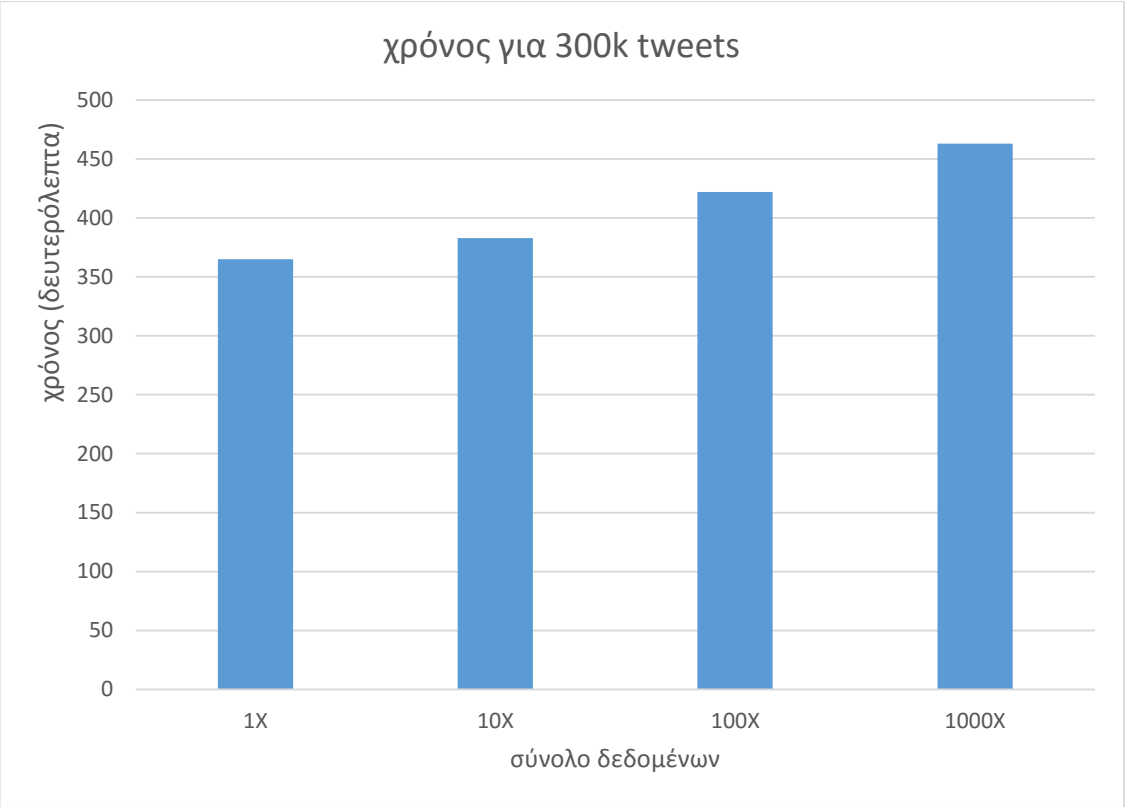
Έχουμε αρχικά τα 4 σύνολα δεδομένων, το κάθε ένα με διαφορετική κατανομή όπως δημιουργήθηκαν από το πρόγραμμα. Επίσης έγιναν πειράματα και για διαφορετικό όγκο δεδομένων.

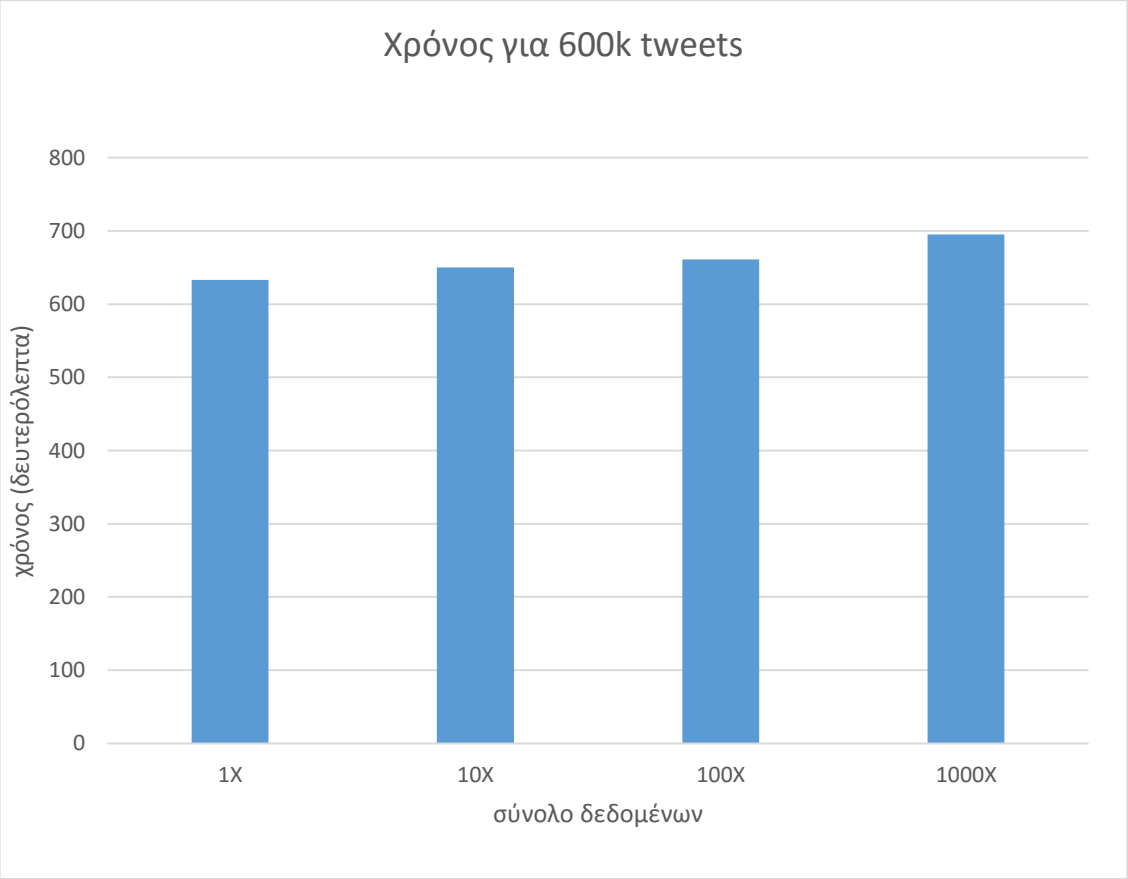


Σχήμα 4.2: Κατανομές των δεδομένων στα 4 σύνολα

Τα 4 σύνολα (1X, 10X, 100X, 1000X) είναι τα αποτελέσματα από το python script που περιεγράφηκε προηγουμένως, δηλαδή στο 100X στον χρήστη με τα πιο πολλά tweets πολλαπλασιάστηκαν επί 100 για να τονίσουμε το skewness.

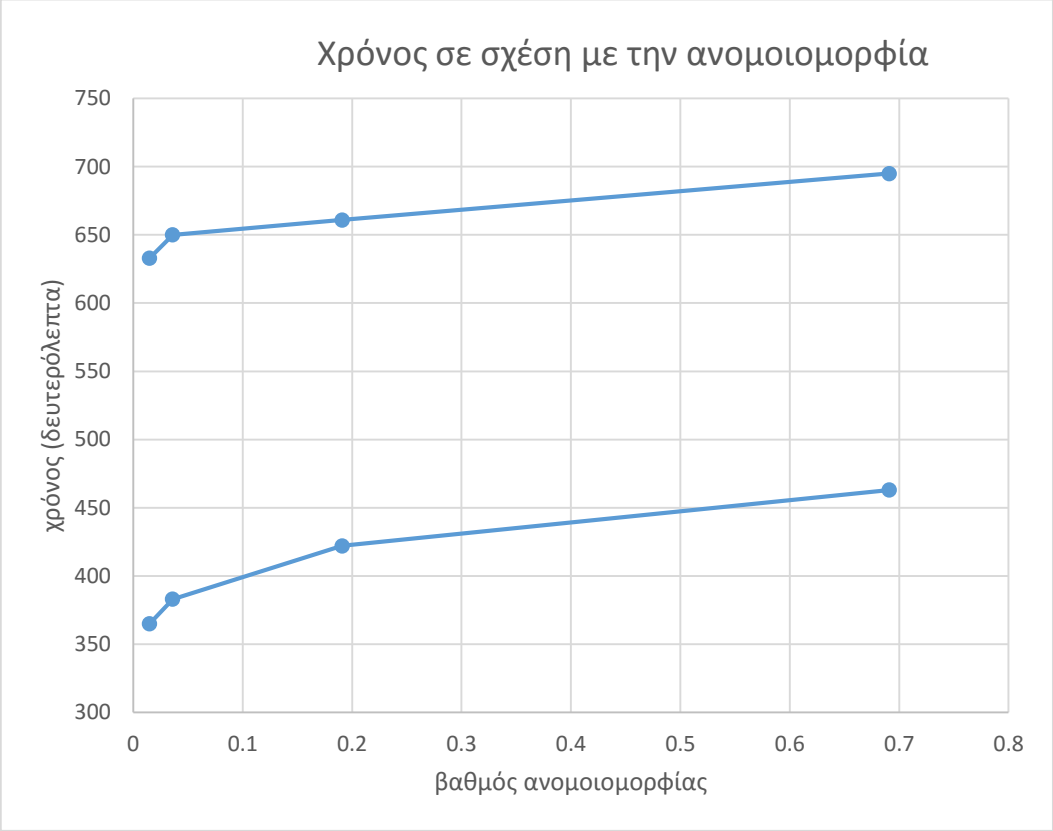
Η αλλαγή στην κατανομή είναι περισσότερο εμφανής στα 2 τελευταία σύνολα, το 100X και το 1000X.


χρόνος για 300k tweets
χρόνος (δευτερόλεπτα)
500
450
400
350
300
250
200
150
100
50
0
1X
10X
100X
1000X
σύνολο δεδομένων


Χρόνος για 600k tweets
χρόνος (δευτερόλεπτα)
800
700
600
500
400
300
200
100
0
1X
10X
100X
1000X
σύνολο δεδομένων

Στις 2 πιο πάνω γραφικές βλέπουμε τον χρόνο που χρειάστηκαν (σε δευτερόλεπτα) να τελειώσουν την επεξεργασία, την πρώτη φορά με 300 χιλιάδες tweets και τη δεύτερη με 600 χιλιάδες tweets.

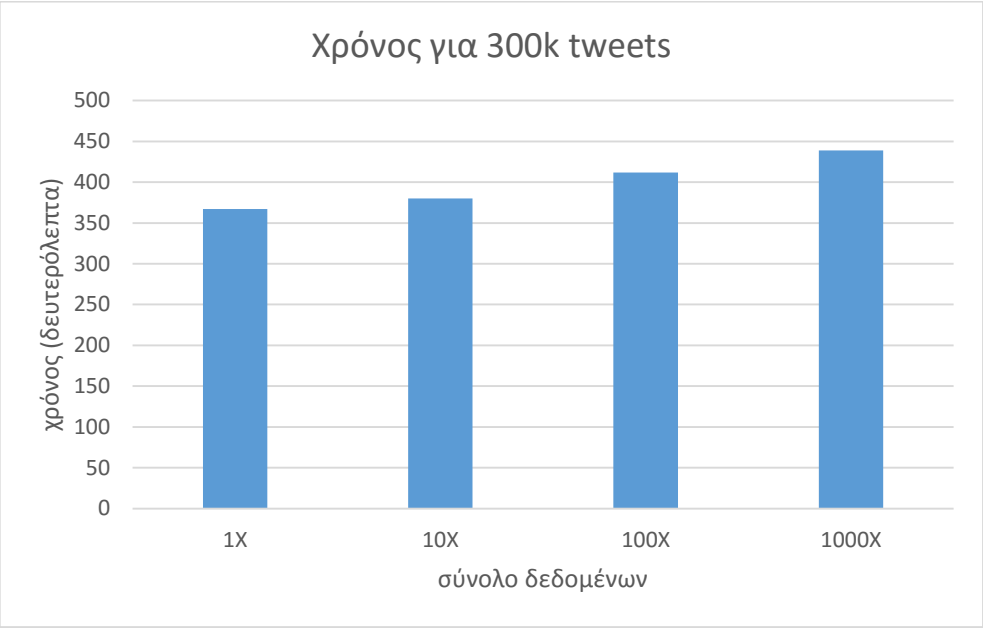
Όπως παρατηρούμε υπάρχει αρνητική επίδραση του skewness όσο αφορά το χρόνο εκτέλεσης, με τις πιο σημαντικές διαφορές να παρατηρούνται στα 100X και 1000X αφού εκεί είναι και πολύ μεγαλύτερο το skewness.

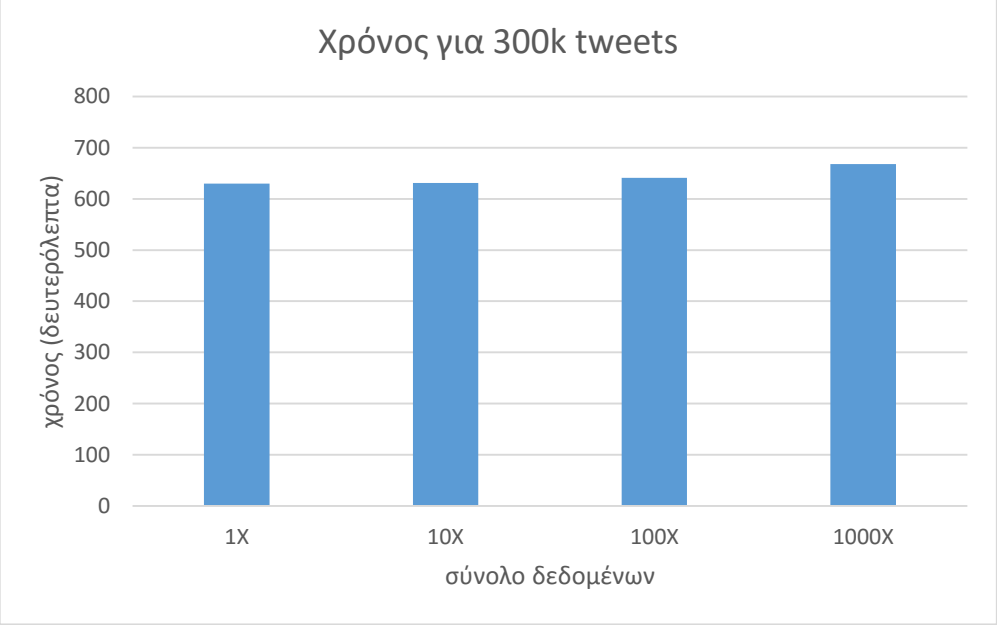


Σε αυτό το γράφημα βλέπουμε τον χρόνο εκτέλεσης σε σχέση με το skewness του συνόλου, όπου φαίνεται καλύτερα πως το skewness επηρεάζει την επίδοση.

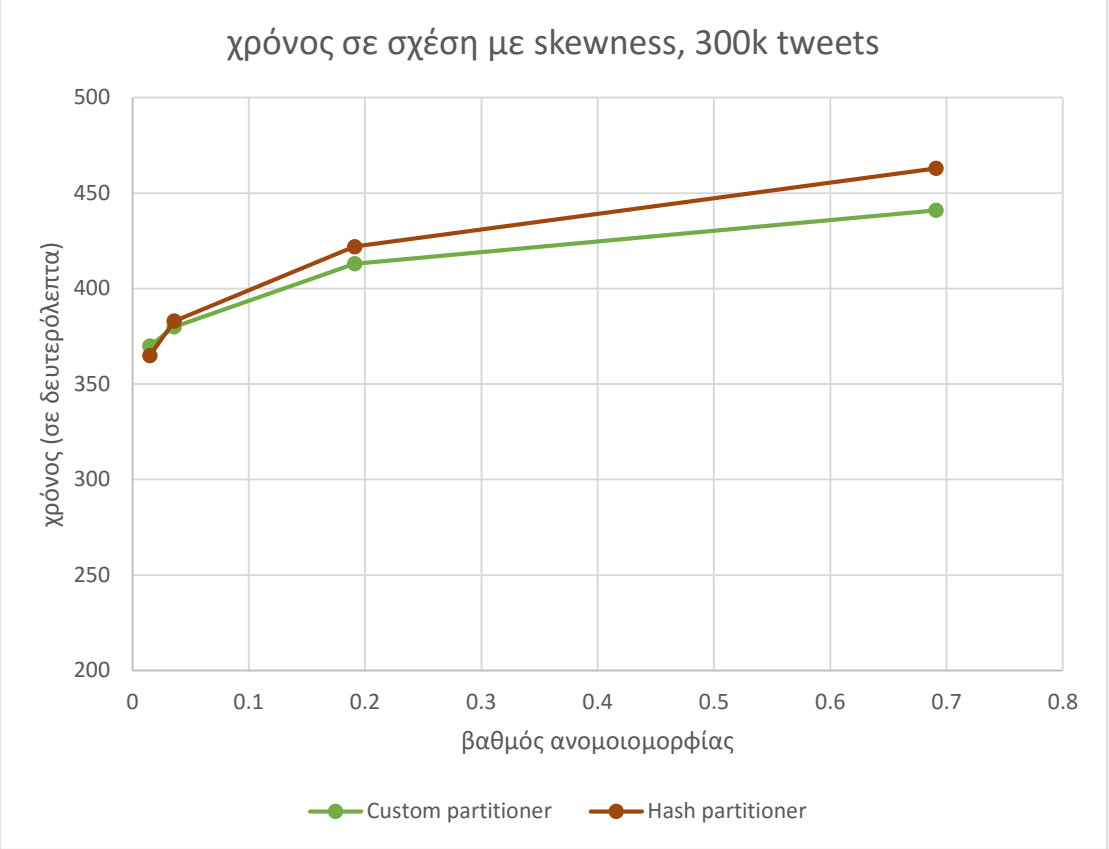
4.3 Μετρήσεις χρησιμοποιώντας την προτεινόμενη λύση

Στην συνέχεια έγιναν τα ίδια πειράματα, αλλά χρησιμοποιώντας τον νέο partitioner και όχι τον Hash. Τα δεδομένα που χρησιμοποιήθηκαν είναι τα ίδια με το προηγούμενο πείραμα.





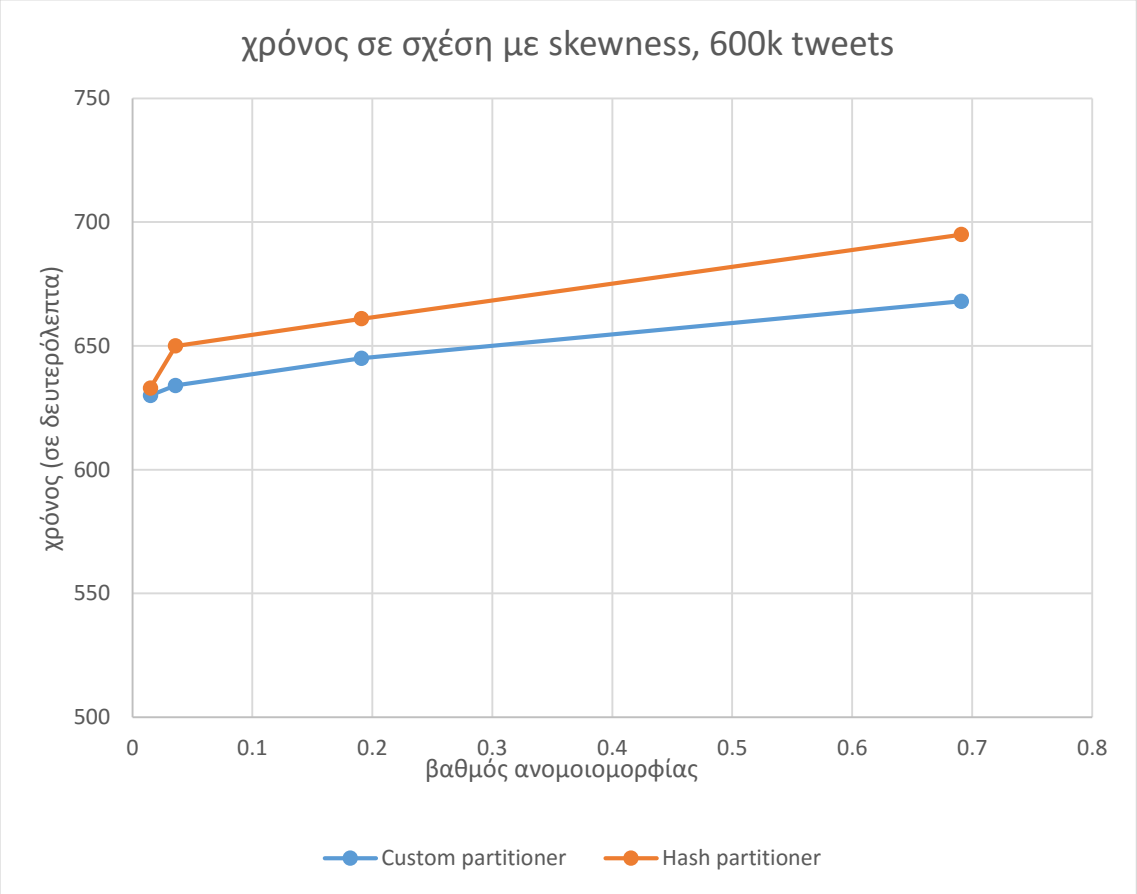
Στα 2 πρώτα γραφήματα βλέπουμε τα αποτελέσματα χρησιμοποιώντας τον νέο partitioner, βλέπουμε ότι ακόμα και τώρα παρατηρείται σημαντική διαφορά ανάμεσα στα σύνολα δεδομένων με διαφορετικό skewness. Ωστόσο το σημαντικό είναι να γίνει η σύγκριση με τον Hash partitioner.



Σε αυτή την γραφική παράσταση συγκρίνουμε τον χρόνο σε σχέση με το πόσο μεγάλη ανομοιομορφία υπάρχει στα δεδομένα δείχνοντας τα αποτελέσματα του προηγούμενου πειράματος με τον hash partitioner αλλά και τα καινούργια με τον νέο partitioner που υλοποιήθηκε.

Για μικρό βαθμό ανομοιομορφίας βλέπουμε ότι δεν υπάρχουν ουσιαστικές διαφορές ανάμεσα στα 2, μάλιστα ο νέος partitioner είναι ελαφρώς πιο αργός από τον hash στο σύνολο που τα δεδομένα είναι σχετικά ομοιόμορφα κατανεμημένα. Στα άλλα σύνολα

όμως βλέπουμε ότι υπάρχει ουσιαστική βελτίωση στην επίδοση, ειδικά στο τελευταίο σύνολο με το μεγάλο skewness.



Η αντίστοιχη γραφική της προηγούμενης αλλά με μεγαλύτερο όγκο δεδομένων. Και εδώ βλέπουμε ότι η διαφορά είναι ασήμαντη αρχικά, ωστόσο στην συνέχεια φαίνεται η επίδραση του skewness και η διαφορά ανάμεσα στους 2 partitioners μεγαλώνει.

Τελικά είδαμε ότι χρησιμοποιώντας τον αλγόριθμο που περιγράψαμε στον partitioner, είδαμε κάποια βελτίωση στην επίδοση όταν έχουμε σύνολα με μεγάλο skewness. Στα σύνολα με μικρό skewness δεν παρατηρήσαμε σχεδόν καθόλου διαφορές επειδή ο

αλγόριθμος που υλοποιήθηκε δεν δραστηριοποιείται εάν το skewness είναι μικρό. Όταν ανιχνεύει skewness μικρότερο του threshold που του ορίσαμε, ουσιαστικά συμπεριφέρεται ακριβώς όπως και ο Hash partitioner.

Κεφάλαιο 5

Συμπεράσματα

Η επεξεργασία ροής δεδομένων τα τελευταία χρόνια έγινε τάση όχι μόνο σε μεγάλες εταιρείες τεχνολογίας αλλά σε πολλούς τομείς και σε μικρές εταιρείες/οργανισμούς. Επίσης αν και το Spark είναι μόλις 2 χρονών, έχει μαζέψει τεράστιο ενδιαφέρον. Αυτά τα στοιχεία δείχνουν ότι δεν υπάρχει αρκετή μελέτη και πολλές βελτιστοποιήσεις για όλα τα προβλήματα, σε σύγκριση με πιο παραδοσιακές τεχνολογίες όπως για παράδειγμα οι σχεσιακές βάσεις.

Έτσι, αυτή η ΑΔΕ δείχνει ένα πρόβλημα το οποίο όπως είδαμε οι μηχανές επεξεργασίας ροών δεδομένων δεν προσφέρουν λύση αλλά το αφήνουν στον προγραμματιστή να το αντιμετωπίσει.

Ας δούμε πως η προτεινόμενη λύση πληροί τις προδιαγραφές που έχουμε ορίσει:

- Επίδοση: Δείξαμε πως βελτιώθηκε η επίδοση για τα 4 διαφορετικά σύνολα δεδομένων με διαφορετικό βαθμό ανομοιομορφίας.
- Φορητότητα: Η προτεινόμενη λύση δεν απαιτεί εξειδικευμένες δυνατότητες που είναι διαθέσιμες μόνο στο Spark. Χρησιμοποιεί μόνο τον Partitioner που είναι βασική λειτουργία για κάθε μηχανή επεξεργασίας ροής δεδομένων. Επίσης δεν απαιτεί επέμβαση στον πηγαίο κώδικα της μηχανής για να μπορεί να λειτουργήσει.
- Συμβατότητα: Η λύση που προτάθηκε μπορεί να δουλέψει χρησιμοποιώντας διαφορετικό message broker, διαφορετικό cluster manager και δεν απαιτεί εγκατάσταση άλλων προγραμμάτων. Αυτό είναι αρκετά σημαντικό επειδή οι διάφοροι συνδυασμοί για πιθανά περιβάλλοντα είναι πάρα πολλοί και έτσι προσθέτοντας άλλες απαιτήσεις αποκλείονται όσοι χρησιμοποιούν άλλες τεχνολογίες.

- Παραμετροποιησιμότητα: Όπως γνωρίζουμε, γενικά, δεν υπάρχουν λύσεις που είναι πάντα ιδανικές, έτσι και η προτεινόμενη λύση. Έχοντας όμως την δυνατότητα αναπροσαρμογής κάποιων παραμέτρων δίνει περισσότερες δυνατότητες στον προγραμματιστή να την προσαρμόσει αναλόγως του προβλήματος. Τα διάφορα είδη επεξεργασίας, το είδος και ο όγκος των δεδομένων, το πόσες μηχανές υπάρχουν και η επεξεργαστική ισχύ τους, όλα αυτά είναι σημαντικά στοιχεία που επηρεάζουν την επίδοση.

Βιβλιογραφία

[1] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, Marco Serafini.
When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing.

[2] [Trihinas, 2015] D. Trihinas, G. Pallis, and M. D. Dikaiakos. AdaM: an Adaptive Monitoring Framework for Sampling and Filtering on IoT Devices. In 2015 IEEE International Conference on Big Data, of IEEE BigData 2015, pages 717--726, 2015.

[3] [Trihinas, 2014] D. Trihinas, G. Pallis, and M. D. Dikaiakos. JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud. In 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014.

Παράρτημα Α

Σε αυτό το παράρτημα είναι ο κώδικα σε Python που χρησιμοποιήθηκε για την δημιουργία των συνόλων δεδομένων με διαφορετικό βαθμό skewness, όπως επίσης και για αποστολή των μηνυμάτων στο kafka.

Δημιουργία συνόλων με skewness:

```
import time
import csv
from sys import argv
import random
import string

originalFile = "Tweets_Demetris.csv"

def randomString(N):
   return ''.join(random.choice(string.ascii_lowercase + ' ') for i in range(N))

count=0;
counters = dict()
with open(originalFile, "rU") as f:
   reader = csv.reader(f, delimiter="%")
   for i, line in enumerate(reader):
      if (len(line)==3):
         count+=1
         userid = line[0].translate(None, '$')
         counters[userid] = counters.get(userid, 0) + 1

print "total tweets: " + str(count)
print len(counters)
sorterUsers = sorted(counters, key=counters.get, reverse=True)
```

```
i = 0
topUsersCount = 1
tweetMultiplier = int(argv[1])
topUsers = [None]*(topUsersCount)
for w in sorterUsers:
    if i>=topUsersCount:
        break
    topUsers[i] = w
    i+=1

removed = 0
multiplied = 0
with open("Tweets_"+str(tweetMultiplier)+"X.csv", "a") as edited:
    with open(originalFile, "rU") as f:
        reader = csv.reader(f, delimiter="%")
        for i, line in enumerate(reader):
            if (len(line)==3):
                userid = line[0].translate(None, '$')
                if userid in topUsers:
                    multiplied += 1
                    for x in range(0, tweetMultiplier):
                        tweetmsg = randomString(random.randrange(30, 100, 1))

                        edited.write(topUsers[count%topUsersCount]+"$%"+str(random.randint
                        (1000000000,100000000000))+"$%"+tweetmsg+"\n")
                elif (random.random()>0.7):
                    edited.write(line[0]+'%'+line[1]+'%'+line[2]+'\n')

print "multiplied tweets:" + str(multiplied)
```

Αποστολή μηνυμάτων στο Kafka:

```
from kafka import SimpleProducer, KafkaClient
import time
import sys
import csv
import time
from sys import argv

kafka = KafkaClient('localhost:9092')
producer = SimpleProducer(kafka)
count = 0
with open(argv[1], "rU") as f:
    reader = csv.reader(f, delimiter="%")
    for i, line in enumerate(reader):
        if (len(line)==3):
            count += 1
            producer.send_messages(b'tweets', str(line[0]).translate(None, '$')+" "+line[2])
        if (count%10000==0):
            print time.strftime('%X') + " " + str(count)
            if (count>=600000):
                print time.strftime('%X')
                sys.exit(1)
```

Παράρτημα Β

Σε αυτό το παράρτημα είναι ο κώδικας σε Scala που χρησιμοποιήθηκε για τα πειράματα στο Spark.

Χρησιμοποιώντας τον Hash partitioner:

```
import kafka.serializer.StringDecoder
import org.apache.log4j.{Level, LogManager}
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.{HashPartitioner, Partitioner, SparkConf}

import scala.collection.mutable.ArrayBuffer

object charcalcHash {

 val rnd = new scala.util.Random

 def removeWrongTweets(line: String): Boolean=
 {
  line.length>0 && line.split(" ").nonEmpty
 }

 def charChalculation = (key:String, tweet:String) => {
  var output = 0.1
  tweet.foreach(c => {
   tweet.foreach(d => {
    val v = c.toInt
    val v2 = d.toInt
    val l1 = rnd.nextInt(10000000)
    val l2 = rnd.nextInt(10000000)
    val f1 = rnd.nextFloat()
```

```
      val f2 = rnd.nextFloat()
      output = output + (f1*l1)/v + (f2*l2)/(v+f1)*v2
      output = output + (f2*(l2+l1)*f1)/v
    })
  })
  output.toString
}

def main(args: Array[String]) {
  if (args.length < 2) {
    System.err.println(s"""
                  |Usage: DirectKafkaWordCount <brokers> <topics>
      """.stripMargin)
    System.exit(1)
  }

  val Array(brokers, topics) = args

  val sparkConf = new SparkConf().setAppName("CharCalculationHash")

  //sparkConf.setMaster("local[3]")

  val ssc = new StreamingContext(sparkConf, Seconds(1))


  LogManager.getRootLogger.setLevel(Level.ERROR)
  val topicsSet = topics.split(",").toSet
  val kafkaParams = Map[String, String]("metadata.broker.list" -> brokers)
  val messages = KafkaUtils.createDirectStream[String, String, StringDecoder,
StringDecoder](
    ssc, kafkaParams, topicsSet)

  val lines = messages.map(_._2)
```

```
    val tweets = lines
      .filter(removeWrongTweets)
      .map(x => (x.split(" ")(0), x))
      .reduceByKeyAndWindow(charChalculation, Seconds(60), Seconds(1), new
HashPartitioner(2))
      .persist()

    tweets.print()

    ssc.start()
    ssc.awaitTermination()
  }
}
```

Χρησιμοποιώντας τον νέο partitioner:

```
import kafka.serializer.StringDecoder
import org.apache.log4j.{Level, LogManager}
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.{Partitioner, SparkConf}

import scala.collection.mutable.ArrayBuffer

object CharacterCalculation {

  val counts = Array(0, 0)
  val distributions = Array(1.0, 1.0)
  val threshold = 1.3
  var min = 0
  var mindist = 2.0
  var counter = 0

  val rnd = new scala.util.Random
  class myPartitioner extends Partitioner {

    def numPartitions = 2

    def getPartition(key: Any): Int = try {
      var partition = math.round(key.asInstanceOf[String].toInt/2)%numPartitions
      counts(partition) = counts(partition) + 1
      counter = counter + 1

      if (counter%1000==0) {

        // calculate average counts
        val average = (counts(0)+counts(1))/counts.length
```

```
      // calculate distributions
      for (i <- 0 until partition-1) {
        distributions(i) = counts(0)/average
        counts(i) = counts(i)/2
      }

      // save distribution with lowest distribution
      min = 0
      mindist = 2.0
      for (i <- 0 until partition-1) {
        if (distributions(i)<mindist) {
          min = i
          mindist = distributions(i)
        }
      }
    }

    if (distributions(partition)>threshold) {
      partition = min
    }
    partition
  } catch {
    case e: NumberFormatException => rnd.nextInt(numPartitions)
  }

  override def equals(other: Any): Boolean = other match {
    case part: myPartitioner =>
      part.numPartitions == numPartitions
    case _ =>
      false
  }
}
```

```
def removeWrongTweets(line: String): Boolean=
{
 line.length>0 && line.split(" ").nonEmpty
}

def charChalculation = (key:String, tweet:String) => {
 var output = 0.1
 tweet.foreach(c => {
  tweet.foreach(d => {
   // some random calculations to increase processing time
   val v = c.toInt
   val v2 = d.toInt
   val l1 = rnd.nextInt(10000000)
   val l2 = rnd.nextInt(10000000)
   val f1 = rnd.nextFloat()
   val f2 = rnd.nextFloat()
   output = output + (f1*l1)/v + (f2*l2)/(v+f1)*v2
   output = output + (f2*(l2+l1)*f1)/v
  })
 })
 output.toString
}

def main(args: Array[String]) {
 if (args.length < 2) {
  System.err.println(s"""
                |Usage: DirectKafkaWordCount <brokers> <topics>
   """.stripMargin)
  System.exit(1)
 }
```

```
    val Array(brokers, topics) = args

    val sparkConf = new SparkConf().setAppName("CharacterCalculation")

    //sparkConf.setMaster("local[3]")

    val ssc = new StreamingContext(sparkConf, Seconds(1))



    LogManager.getRootLogger.setLevel(Level.ERROR)
    val topicsSet = topics.split(",").toSet
    val kafkaParams = Map[String, String]("metadata.broker.list" -> brokers)
    val messages = KafkaUtils.createDirectStream[String, String, StringDecoder,
StringDecoder](
      ssc, kafkaParams, topicsSet)

    val lines = messages.map(_._2)

    val tweets = lines
      .filter(removeWrongTweets)
      .map(x => (x.split(" ")(0), x))
      .reduceByKeyAndWindow(charChalculation, Seconds(60), Seconds(1), new
myPartitioner())
      .persist()

    tweets.print()

    ssc.start()
    ssc.awaitTermination()
  }
}
```