Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Δυναμική δρομολόγηση κατανεμημένων ροών εργασιών με χρήση τεχνικών μηχανικής μάθησης

Διπλωματικη Εργασια

του

Νικόλαου Καραβασίλη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Δυναμική δρομολόγηση κατανεμημένων ροών εργασιών με χρήση τεχνικών μηχανικής μάθησης

Διπλωματικη Εργασια

του

Νικόλαου Καραβασίλη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την κάτωθι τριμελή επιτροπή την 23η Ιουλίου 2021.

....	....	....
Γεώργιος Γκούμας Επίκουρος Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.

Νικόλαου Καραβασίλη
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Νικόλαος Καραβασίλης, 2021
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η ορολογία Big Data αναφέρεται σε δεδομένα που περιέχουν μεγαλύτερη ποικιλία, φτάνοντας σε αυξανόμενους όγκους και με μεγαλύτερη ταχύτητα. Με απλά λόγια, τα Big Data είναι μεγαλύτερα, πιο πολύπλοκα σύνολα δεδομένων, ειδικά από νέες πηγές δεδομένων. Αυτά τα σύνολα δεδομένων είναι τόσο ογκώδη που το παραδοσιακό λογισμικό επεξεργασίας δεδομένων δεν μπορεί να τα διαχειριστεί. Αλλά αυτοί οι τεράστιοι όγκοι δεδομένων μπορούν να χρησιμοποιηθούν για την αντιμετώπιση επιχειρηματικών προβλημάτων που δεν θα μπορούσαν να αντιμετωπιστούν πριν.

Στην σημερινή εποχή η έντονη παρουσία των Big Data στις περισσότετες επιχειρήσεις, έχει οδηγήσει στην δημιουργία πολυάριθμων συστημάτων επεξεργασίας δεδομένων. Ωστόσο, η μεταφορά από ένα τέτοιο σύστημα σε ένα άλλο, π.χ. για λόγους απόδοσης, απαιτεί την τροποποίηση ή ακόμα και την δημιουργία νέων εφαρμογών, προκειμένου να καλύπτουν τις προϋποθέσεις της νέας τεχνολογίας. Η τεχνολογία **Apache Beam** επιλύει αυτό το πρόβλημα, επιτρέποντας στο χρήστη να δημιουργήσει ένα πρόγραμμα και να το εκτελέσει στα συστήματα επεξεργασίας δεδομένων που υποστηρίζει. Ωστόσο, τις περισσότερες φορές ο χρήστης δεν γνωρίζει ποιο σύστημα είναι το πιο αποδοτικό για τα δεδομένα του.

Σκοπός της παρούσας διπλωματικής εργασίας, είναι η δημιουργία ενός αλγορίθμου μηχανικής μάθησης, ο οποίος θα μπορεί να προβλέπει το βέλτιστο σύστημα ανάλογα με τα δεδομένα του χρήστη. Πριν την δημιουργία του αλγορίθμου, γίνεται ένα **benchmark** για μελέτη της απόδοσης των συστημάτων για μια συγκεκριμένη μορφολογία δεδομένων και για ένα συγκεκριμένο είδος επεξεργασίας.

Για το λόγο αυτό, δημιουργήσαμε αρχεία με δεδομένα από διακριτές κατανομές αριθμών και ένα πρόγραμμα σύμφωνα με το Apache Beam που υπολογίζει την συχνότητα εμφάνισης των δεδομένων. Ύστερα, επιλέξαμε ένα υπολογιστικό σύστημα που ανήκει στο εργαστήριο CSLab του Ε.Μ.Π., στο οποίο εγκαταστήσαμε τις τεχνολογίες **Apache Spark** και **Apache Flink**. Τροφοδοτήσαμε κάθε αρχείο σε κάθε σύστημα με την βοήθεια του Apache Kafka προσομοιώνοντας την περίπτωση streaming. Με αυτόν τον τρόπο αναλύσαμε την απόδοση των συστημάτων επεξεργασίας ανάλογα με την μορφολογία των δεδομένων και με τα αποτελέσματα που λάβαμε εκπαιδεύσαμε τον αλγόριθμο **Random Forest**.

Τέλος παραθέτουμε τα συμπεράσματα που μπορούν να προκύψουν μέσα απ΄ την παραπάνω διαδικασία, καθώς και ιδέες ή προτάσεις βελτίωσης της απόδοσης των εφαρμογών μελλοντικά.

Λέξεις-κλειδιά: Apache Spark, Apache Beam, Apache Flink, Benchmark, Random Forest

Abstract

The definition of big data is data that contains greater variety, arriving in increasing volumes and with more velocity. Put simply, big data is larger, more complex data sets, especially from new data sources. These data sets are so voluminous that traditional data processing software just can't manage them. But these massive volumes of data can be used to address business problems you wouldn't have been able to tackle before.

Nowadays, the strong presence of Big Data in most companies, has led to the creation of numerous data processing systems. However, the transition from one such system to another, e.g. for performance reasons, requires modification or even the creation of new applications in order to meet the requirements of the new technology. **Apache Beam** solves this problem by allowing the user to create a program and run it on the data processing systems it supports. However, most of the time the user does not know which system is the most efficient for his data.

The purpose of this dissertation is to create a machine learning algorithm, which will be able to predict the optimal system based on user data. Before creating the algorithm, a **benchmark** study is performed of the performance of the systems for a specific data morphology and for a specific type of processing operation.

To accomplish that, we created files with data from distinct number distributions and a program according to Apache Beam that calculates the frequency of occurrence of data. Next, we selected a computer system belonging to CSlab located in NTUA, n which we installed the **Apache Spark** and **Apache Flink** technologies. We fed each file to each system with the help of Apache Kafka simulating the streaming phenomenon. In this way we analyzed the performance of the processing systems according to the morphology of the data and with the results we obtained we trained the **Random Forest** algorithm.

Finally, we present all of the conclusions that can be extracted from the above procedure, as well as ideas or propositions for future performance optimization.

Keywords: Apache Spark, Apache Beam, Apache Flink, Benchmark, Random Forest

Ευχαριστίες

Η εκπόνηση της Διπλωματικής μου εργασίας σηματοδοτεί το τέλος των προπτυχιακών σπουδών μου στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Πραγματοποιήθηκε στα πλαίσια του Εργαστηρίου Υπολογιστικών Συστημάτων του τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών, με επιβλέποντα καθηγητή τον κ. Νεκτάριο Κοζύρη, τον οποίο και θα ήθελα να ευχαριστήσω πρωτίστως για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο επιστημονικό πεδίο, αλλά και για τα άτομα με τα οποία ο ίδιος με έφερε σε επικοινωνία και υπήρξαν αρωγοί στην προσπάθειά μου.

Ιδιαιτέρως θα ήθελα να ευχαριστήσω την Δρ. Κατερίνα Δόκα, η οποία στάθηκε δίπλα μου από την αρχή, δίνοντας μου τις κατάλληλες συμβουλές και κατευθύνσεις, προκειμένου να έχουμε το επιθυμητό αποτέλεσμα, χάρη στην πολύτιμη βοήθεια της οποίας η συγκεκριμένη διπλωματική εργασία έγινε πραγματικότητα.

Θα ήθελα επίσης να ευχαριστήσω τον επίκουρο καθηγητή Γεώργιο Γκούμα και τον καθηγητή Διονύσιο Πνευματικάτο που συμπλήρωσαν την τριμελή επιτροπή.

Θα αποτελούσε παράλειψη να μην ευχαριστήσω όλους τους φίλους μου που με την στήριξή τους με βοηθούν όλα αυτά τα χρόνια να πετύχω τους στόχους μου.

Τέλος, ευχαριστώ βαθύτατα τους γονείς μου Χρήστο και Καλλιόπη καθώς και την αδελφή μου Αθηνά, για την αγάπη, την υπομονή και την στήριξη που μου έχουν προσφέρει όλα αυτά τα χρόνια.

Περιεχόμενα

Περίληψη **5**

Abstract **7**

Ευχαριστίες **9**

1 Εισαγωγή **15**
1.1 Κατανεμημένη ροή εργασίας 15
1.2 Επεξεργασία δεδομένων σε πραγματικό χρόνο 16
1.3 Σκοπός εργασίας 16
1.4 Διάρθρωση διπλωματικής εργασίας 17

2 Τεχνολογίες **19**
2.1 Apache Kafka 19
2.2 Streaming Data Processing Systems (SDPSs) 20
2.2.1 Apache Beam 20
2.2.2 Apache Spark Streaming 22
2.2.3 Apache Flink 23
2.3 Machine Learning 24
2.3.1 Τύποι προβλημάτων 24
2.3.2 Random Forest 25

3 Ανάλυση Benchmark **27**
3.1 Δεδομένα εισόδου 27
3.1.1 Ομοιόμορφη κατανομή 27
3.1.2 Zipf κατανομή 28
3.2 Αρχιτεκτονική Benchmark 29
3.2.1 Τροφοδοσία δεδομένων 30
3.2.2 Εκτέλεση προγράμματος 30
3.2.3 Query 31
3.2.4 Metrics 31
3.3 Χαρακτηριστικά χρησιμοποιηθέντος υλικού 33

4 Αποτελέσματα Benchmarking **35**

4.1 Παραμετροποίηση συστήματος 35
4.2 Αποτελέσματα ομοιόμορφης κατανομής 35
4.2.1 Window 5 seconds 36
4.2.2 Window 8 seconds 36
4.2.3 Window 10 seconds 37
4.3 Αποτελέσματα zipf κατανομής 38
4.3.1 Window 5 seconds 39
4.3.2 Window 8 seconds 39
4.3.3 Window 10 seconds 40
4.4 Συμπεράσματα 41

5 Αλγόριθμος πρόβλεψης SDPS 43
5.1 Διατύπωση προβλήματος 43
5.2 Predictor 43

6 Συμπεράσματα και μελλοντική δουλειά 47

Α′ Αρκτικόλεξο 49

Β′ Hadoop-Yarn Configuration Files 51

Βιβλιογραφία 55

Κατάλογος Σχημάτων

1.1 Κατανεμημένη επεξεργασία δεδομένων 15
1.2 Profile - High Level μορφολογία του συστήματος 17

2.1 Apache Beam Architecture 20
2.2 Αλληλουχία μετασχηματισμών PTransform 22
2.3 Apache Spark in Cluster Mode 23
2.4 Apache Flink Runtime 24

3.1 Συνάρτηση μάζας πιθανότητας της ομοιόμορφης διακριτής κατανομής 28
3.2 Συνάρτηση μάζας πιθανότητας της zipf κατανομής 29
3.3 Γενική μορφή του Benchmark 30
3.4 Αρχιτεκτονική μηχανημάτων clone 34

4.1 Avg Event-Time Latency per Uniform file, Window 5 sec 36
4.2 Avg Event-Time Latency per Uniform file, Window 8 sec 37
4.3 Avg Event-Time Latency per Uniform file, Window 10 sec 38
4.4 Avg Event-Time Latency per Zipf file, Window 5 sec 39
4.5 Avg Event-Time Latency per Zipf file, Window 8 sec 40
4.6 Avg Event-Time Latency per Zipf file, Window 10 sec 41

Κατάλογος Πινάκων

3.1 Χαρακτηριστικά αρχείων διακριτής ομοιόμορφης κατανομής 28
3.2 Χαρακτηριστικά αρχείων διακριτής ζιπφ κατανομής 30
3.3 Βασικά στοιχεία των μηχανημάτων clones 33
3.4 Εκδόσεις των τεχνολογιών 33

5.1 Μορφολογία συνδυαστικού αρχείου αποτελεσμάτων 44

Κεφάλαιο 1

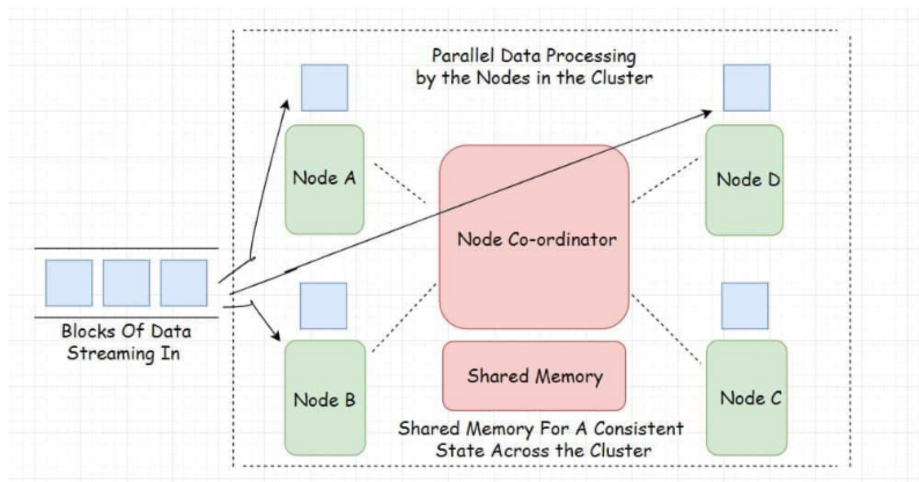
Εισαγωγή

1.1 Κατανεμημένη ροή εργασίας

Μία ροή εργασίας (workflow) μπορεί να οριστεί ως συλλογή βημάτων επεξεργασίας που οργανώνονται για να ολοκληρώσουν κάποια διαδικασία. Μια εργασία μπορεί να αντιπροσωπεύει μια χειροκίνητη λειτουργία από τον άνθρωπο ή έναν υπολογισμό που προϋποθέτει υπολογιστική ισχύ. Ένα workflow ορίζει τη σειρά επίκλησης εργασιών ή συνθηκών υπό τις οποίες οι επιμέρους εργασίες θα κληθούν (control-flow) και τα δεδομένα θα κατανεμηθούν μεταξύ αυτών των εργασιών (data-flow).

Στην κατανεμημένη εκτέλεση ενός workflow υπάρχει ένας κατανεμημένος οργανωτής της ροής (scheduler), ο οποίος εμπεριέχει επιμέρους οργανωτές σε διαφορετικούς κόμβους ενός δικτύου. Η κάθε υπολογιστική οντότητα εκτελεί κομμάτια της εργασίας. Μία τέτοια αρχιτεκτονική προσφέρει ανθεκτικότητα σε αστοχία και αυξημένη απόδοση στην εκτέλεση ενός workflow.

Σε αυτήν την διπλωματική θα αναφερθούμε στις κατανεμημένες εργασίες που αφορούν την επεξεργασία μεγάλου όγκου δεδομένων (Big Data).



Σχήμα 1.1: Κατανεμημένη επεξεργασία δεδομένων

1.2 Επεξεργασία δεδομένων σε πραγματικό χρόνο

Η επεξεργασία μεγάλων όγκων δεδομένων παρελθοντικού χρόνου (batch processing) συχνά δεν είναι επαρκής σε περιπτώσεις όπου τα νέα δεδομένα πρέπει να υποβληθούν σε επεξεργασία γρήγορα ώστε να προκύψουν συμπεράσματα που αλλάζουν τη μορφή των δεδομένων που έρχονται. Για αυτόν τον λόγο, η επεξεργασία δεδομένων σε πραγματικό χρόνο (stream processing) έχει αποκτήσει σημαντική προσοχή. Οι πιο δημοφιλείς τεχνολογίες, με τεράστια απήχηση στη βιομηχανία και στην ερευνητική κοινότητα, είναι το Apache Storm[1], το Apache Spark[2] και Apache Flink[3].

Η ροή δεδομένων (data stream) αναφέρεται σε ένα συγκεκριμένο είδος δεδομένων που λαμβάνεται ασταμάτητα για επεξεργασία χωρίς να αποθηκεύεται στο δίσκο ή στη μνήμη για μελλοντική τυχαία πρόσβαση. Η επεξεργασία ενός data stream εξαρτάται από την εφαρμογή και διαφέρει ανάλογα με την εταιρία. Για παράδειγμα οι εφαρμογές μπορεί να συμπεριλαμβάνουν οικονομικές δραστηριότητες (financial applications), παρακολούθηση δικτύου (web monitoring) και καταγραφή διαδικτυακού ιστού (web log)[4], [5]. Στην σύγχρονη εποχή παρατηρείται μια ταχέως αναπτυσσόμενη αύξηση δεδομένων κυρίως λόγω της επικράτησης του mobile Internet και της ραγδαίας χρήσης των social media [6]. Σύμφωνα με αναφορές το Facebook διαχειρίζεται εκατομμύρια δεδομένα το δευτερόλεπτο με χαμηλό latency, προκειμένου να παρέχει ακριβή στατιστικά στοιχεία για διαφημίσεις ανάλογα με τις σελίδες που έχει αλληλεπιδράσει ο χρήστης. Το Twitter επίσης χρειάζεται να αναγνωρίζει σε πραγματικό χρόνο sparm urls με ρυθμό εκατομμυρίων ανά ημέρα [7].

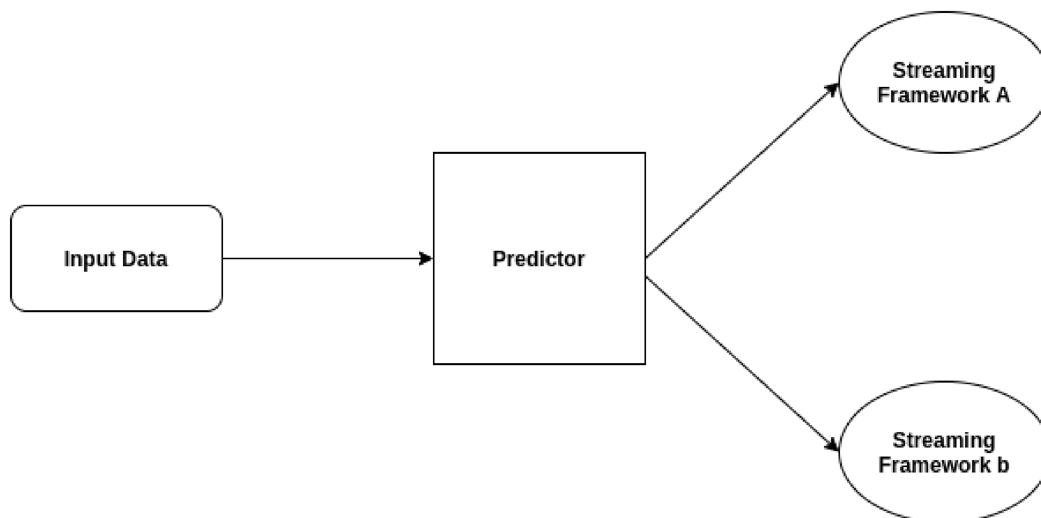
Όλες οι εταιρίες που χρειάζονται να επεξεργαστούν δεδομένα σε πραγματικό χρόνο βρίσκονται αντιμέτωπες με την επιλογή της καλύτερης τεχνολογίας που θα εξυπηρετήσει το σκοπό τους. Τα πιο βασικά κριτήρια είναι:

- **Μέγεθος Συστήματος (Cluster Size):** Η απόδοση ενός streaming framework εξαρτάται από το πλήθος των κόμβων που θα χρησιμοποιηθούν, την υπολογιστή ισχύ κάθε κόμβου και την μνήμη RAM.
- **Μορφολογία δεδομένων:** Τα δεδομένα, που εισέρχονται στο σύστημα σε πραγματικό χρόνο, τα χαρακτηρίζει μία συγκεκριμένη μορφολογία, η οποία μπορεί να επηρεάσει την απόδοση του συστήματος επεξεργασίας.
- **Ανάλυση των δεδομένων:** Η ανάλυση των δεδομένων που θα επιλεχθεί (operator) δεσμεύει υπολογιστικούς πόρους ανάλογα με το είδος της και κυρίως με τη δυσκολία της επεξεργασίας.

1.3 Σκοπός εργασίας

Αν και τα τελευταία χρόνια έχουν πραγματοποιηθεί πολλές έρευνες με στόχο την καταγραφή των διαφορών στην απόδοση των distributed stream data processing systems (SDPSs) δεν έχει αποτυπωθεί ένα σύστημα που να προβλέπει ποιο framework θα χρησιμοποιηθεί ανάλογα

με τις απαιτήσεις του χρήστη. Σκοπός της εργασίας είναι να παρουσιάσει μία προσέγγιση ενός μοντέλου που να μπορεί να αποφασίζει ποιο framework είναι καλύτερο ανάλογα με τα δεδομένα εισόδου και το είδος της κατανεμημένης εργασίας.



Σχήμα 1.2: Profile - High Level μορφολογία του συστήματος

1.4 Διάρθρωση διπλωματικής εργασίας

Στο δεύτερο κεφάλαιο παρατέθηκαν οι τεχνολογίες που χρησιμοποιήσαμε στην εργασία προκειμένου να πραγματοποιήσουμε τις πειραματικές μετρήσεις και για την δημιουργία του machine learning αλγόριθμου. Στο τρίτο κεφάλαιο γίνεται ανάλυση της αρχιτεκτονικής benchmark και γίνεται μια αναφορά στους πόρους και τα μηχανήματα που χρησιμοποιήσαμε στην εργασία. Στο τέταρτο κεφάλαιο παρουσιάζουμε τα αποτελέσματα των μετρήσεων και σχολιάζουμε τα αποτελέσματα. Στο πέμπτο κεφάλαιο παρουσιάζεται η προσέγγιση του αλγόριθμου που προβλέπει το βέλτιστο σύστημα επεξεργασίας. Τέλος, στο έκτο κεφάλαιο βρίσκονται τα συμπεράσματα που μπορούν να προκύψουν, καθώς και τα περιθώρια που υπάρχουν για μελλοντική έρευνα πάνω στο θέμα.

Κεφάλαιο 2

Τεχνολογίες

Στο συγκεκτιμένο κεφάλαιο παρουσιάζονται οι τεχνολογίες που χρησιμοποιήθηκαν για την μέτρηση της απόδοσης των SDPSs καθώς και ο αλγόριθμος machine learning που εφαρμόστηκε για το τελικό στάδιο της πρόβλεψης του βέλτιστου συστήματος.

2.1 Apache Kafka

Το Kafka [8],[9] είναι μια ουρά δημοσίευσης και κατανάλωσης μηνυμάτων (publish/subscribe message system), με τεράστια επιδεκτικότητα διεύρυνσης, σχεδιασμένη σαν ένα κατανεμημένο αρχείο καταγραφής συναλλαγών που αναπτύχθηκε από το LinkedIn. Το Kafka διατηρεί τις ροές μηνυμάτων σε κατηγορίες θεμάτων (topic categories), κάθε κατηγορία έχει πολλά διαμερίσματα (partitions), και κάθε διαμέρισμα περιέχει μια διατεταγμένη, αμετάβλητη ακολουθία μηνυμάτων που προσαρτώνται συνεχώς. Σε κάθε μήνυμα προσδίδεται ένα μοναδικό αναγνωριστικό για την αναγνώρισή του σε ένα partition. Σε ένα σύμπλεγμα από Kafka στιγμιότυπα (Kafka cluster), ένα διαμέρισμα διανέμεται σε πολλούς κόμβους για ανοχή σε σφάλματα. Ένα Kafka cluster διατηρεί τα δημοσιευμένα μηνύματα για μια χρονική περίοδο. Όταν λήξει η ώρα, τα μηνύματα απορρίπτονται ανεξάρτητα από το εάν έχουν καταναλωθεί ή όχι. Κάθε καταναλωτής φέρει την ετικέτα με το όνομα της ευρύτερης ομάδας καταναλωτών που ανήκει. Ένα topic partition μπορεί να παραδοθεί μόνο σε έναν καταναλωτή σε μια ομάδα καταναλωτών, γεγονός που εξασφαλίζει την σωστή παράδοση των μηνυμάτων.

Η αρχική χρήση του Kafka είναι η αναδημιουργία της καταγραφής της δραστηριότητας ενός χρήστη ως σύνολο ροών publish/subscribe σε πραγματικό χρόνο. Οι δραστηριότητες μιας ιστοσελίδας, όπως περιήγηση σε σελίδες, αναζήτηση και άλλες ενέργειες των χρηστών, δημοσιεύονται σε ένα Kafka cluster ως topics, και συγκεκριμένα topic ανά είδος δραστηριότητας. Τα topics είναι διαθέσιμα για συνδρομή για μια σειρά από διαφορετικές περιπτώσεις χρήσης, όπως επεξεργασία σε πραγματικό χρόνο, παρακολούθηση σε πραγματικό χρόνο και φόρτωση δεδομένων σε Hadoop ή αποθήκευση δεδομένων. Οι αγωγοί δεδομένων (data pipelines) στο Kafka παρακολουθούνται, π.χ. παρακολούθηση των στατιστικών πληροφοριών των aggregations από κατανεμημένες εφαρμογές. Επομένως, το Kafka είναι

κατάλληλο για καταστάσεις όπου οι χρήστες πρέπει να επεξεργάζονται δεδομένα σε πραγματικό χρόνο και να τα αναλύουν. Επί του παρόντος, το Kafka στο LinkedIn υποστηρίζει δεκάδες subscribing systems και παραδίδει περισσότερα από 55 δισεκατομμύρια μηνύματα στους καταναλωτές ανά ημέρα.

2.2 Streaming Data Processing Systems (SDPSs)

Στο συγκεκριμένο section θα αναφερθούν οι τεχνολογίες επεξεργασίας δεδομένων που θα χρησιμοποιηθούν για να πάρουμε μετρήσεις.

2.2.1 Apache Beam

Το Apache Beam [10] περιγράφεται ως ένα ενοποιημένο προγραμματιστικό μοντέλο, το οποίο επιτρέπει τον καθορισμό των εφαρμογών είτε είναι batch processing είτε είναι stream processing. Για αυτό το λόγο, παρέχονται διαφορεττικές εκδόσεις Apache Beam SDK. Επί του παρόντος, τρία SDK αποτελούν μέρος του αποθετηρίου Apache Beam: Java SDK, Python SDK και Go SDK [11].

Σχήμα 2.1: Apache Beam Architecture

Αντί να γραφεί μια εφαρμογή για ένα συγκεκριμένο SDPS, το Apache Beam επιτρέπει τη συγγραφή συμβατών προγραμμάτων με οποιαδήποτε υποστηριζόμενη μηχανή εκτέλεσης. Το Beam επιτρέπει τη δημιουργία ενός γενικού προγράμματος το οποίο μπορεί να τρέξει στο streaming framework της επιλογής μας, χωρίς να προαπαιτούνται αλλαγές στον κώδικα ή στην δομή του data pipeline. Μερικά από τα πιο βασικά στοιχεία του Apache Beam SDK είναι:

- **Pipeline:** Απεικονίζει τη συνολική εικόνα της εφαρμογής, συμπεριλαμβάνοντας την είσοδο δεδομένων, τους μετασχηματισμούς στα δεδομένα και το τελικό αποτέλεσμα που εξέρχεται.

- **PCollection:** Ενσωματώνει ένα κατανεμημένο σύνολο δεδομένων που μπορεί να είναι είτε οριοθετημένο (bounded) είτε χωρίς όρια (unbounded). Το unbounded PCollection χρησιμοποιείται για εφαρμογές streaming.

- **PTransform:** Αποτελεί μια λειτουργία που μετασχηματίζει δεδομένα. Λαμβάνει ένα ή περισσότερα PCollection και εφαρμόζει κάποια μετατροπή πάνω σε αυτά τα δεδομένα. Σαν έξοδο του μετασχηματισμού προκύπτουν μηδέν ή περισσότερα PCollection.

Το Apache Beam προσφέρει έναν μεγάλο αριθμό από μετασχηματισμούς PTrasform. Μερικές από αυτές είναι:

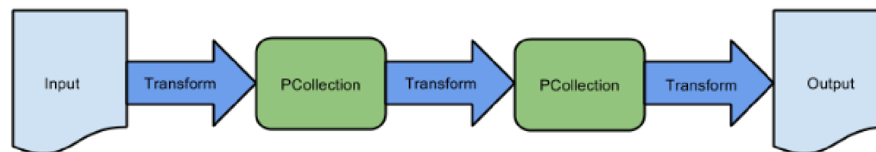
- **ParDo:** Αποτελεί έναν μετασχηματισμό του Beam για γενική παράλληλη επεξεργασία. Το ParDo σαν πρότυπο επεξεργασίας είναι παρόμοιο με το στάδιο map ενός τυπικού αλγορίθμου Map-Reduce αφού διαβάζει κάθε στοιχείο από το εισερχόμενο PCollection και στη συνέχεια εκτελεί κάποια λειτουργία επεξεργασίας σε κάθε στοιχείο και τελικά εκπέμπει μηδέν ή περισσότερα στοιχεία σε ένα PCollection εξόδου. Το ParDo είναι χρήσιμο για ένα μεγάλο εύρος operators, όπως φιλτράρισμα ενός συνόλου δεδομένων, καθώς μπορεί κανείς να χρησιμοποιήσει το ParDo για να ελέγξει κάθε ένα στοιχείο του PCollection και είτε να εξάγει αυτό το στοιχείο είτε να το απορρίψει. Το ParDo μπορεί επίσης να χρησιμοποιηθεί για μορφοποίηση ενός dataset ή αλλαγή του τύπου των δεδομένων, αφού εάν το εισερχόμενο PCollection περιέχει στοιχεία διαφορετικού τύπου, τότε το ParDo μπορεί να εφαρμόσει μετατροπή τύπου σε κάθε στοιχείο και να εξάγει το αποτέλεσμα σε ένα νέο PCollection. Εξαγωγή τμημάτων κάθε στοιχείου σε ένα σύνολο δεδομένων είναι επίσης δυνατή με το ParDo.

- **GroupByKey:** Είναι ένας μετασχηματισμός που αφορά την επεξεργασία συλλογών key-value pairs (KVPs). Ως μια παράλληλη λειτουργία μείωσης του μεγέθους της αρχικής συλλογής εισόδου, ανάλογη με τη φάση Shuffle ενός τυπικού αλγορίθμου Map-Reduce, η είσοδος στο GroupByKey είναι μια συλλογή μια συλλογή που περιέχει πολλά ζεύγη με το ίδιο κλειδί. Με δεδομένη μια τέτοια συλλογή, μπορεί κανείς να χρησιμοποιήσει το GroupByKey προκειμένου να συλλέξει όλες τις τιμές που σχετίζονται με κάθε μοναδικό κλειδί για να συγκεντρώσει δεδομένα που έχουν κάτι κοινό. Κατά τη χρήση unbounded PCollections, πρέπει να γίνει χρήση ενός window είτε κάποιου aggregation trigger προκειμένου να εφαρμοστεί ένα operation. Ο λόγος που απαιτείται η χρήση των παραπάνω τεχνικών είναι διότι στην περίπτωση που τα δεδομένα είναι bounded πρέπει να ληφθούν όλα τα δεδομένα πριν εφαρμοστεί κάποιο operation ενώ όταν τα δεδομένα είναι απεριόριστα και λαμβάνονται σε πραγματικό χρόνο η λειουργία window επιτρέπει να ομαδοποιήσουμε δεδομένα σε πεπερασμένα κομμάτια (bundles) και να τα επεξεργαστούμε. Τα unbounded δεδομένα που ομαδοποιούνται σύμφωνα με την GroupByKey χρησιμοποιούν την ίδια στρατηγική και το ίδιο μέγεθος window, προκειμένου να αποφευχθούν συγκρούσεις.

- **Combine:** Συνδυάζει συλλογές στοιχείων ή τιμών. Το Combine έχει παραλλαγές που λειτουργούν σε PCollections και ορισμένες που συνδυάζουν τα values για κάθε key από τα PCollections των key-value pairs. Κατά την εφαρμογή του μετασχηματισμού Combine, πρέπει να οριστεί μια συνάρτηση (combining function) που περιέχει τη λογική για τον

τρόπο με τον οποίο θα συνδυαστούν τα στοιχεία ή τα values. Επειδή τα δεδομένα εισόδου μπορεί να διανεμηθούν σε πολλά worker-nodes, combining function μπορεί να καλείται πολλές φορές για να εκτελέσει μερικό συνδυασμό σε υποσύνολα του αρχικού dataset.

- **Flatten:** Ενώνει πολλαπλά PCollection με τον ίδιο τύπο δεδομένων σε ένα PCollection. Τα unbounded δεδομένα που ενώνονται σύμφωνα με την Flatten χρησιμοποιούν την ίδια στρατηγική και το ίδιο μέγεθος window, προκειμένου να αποφευχθούν συγκρούσεις [12], [13].



Σχήμα 2.2: Αλληλουχία μετασχηματισμών PTransform

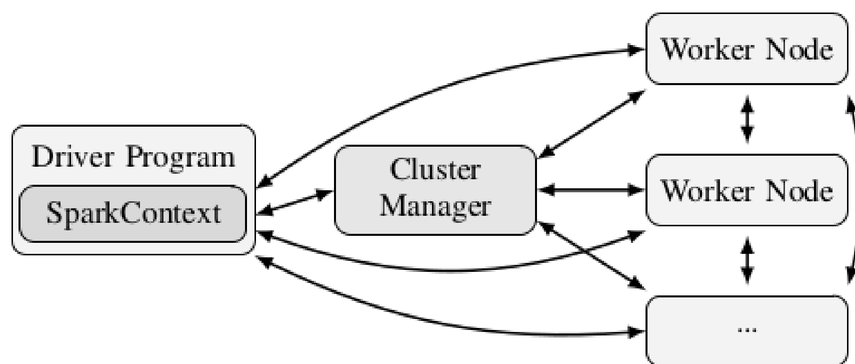
Εκτός από το Apache Flink και το Apache Spark υπάρχουν και άλλα frameworks που υποστηρίζουν το Apache Beam[14]. Μερικά από αυτά είναι το Apache Samza[15], IBM Streams[16], Apache Hadoop MapReduce[17] και το Google Cloud Dataflow[18].

2.2.2 Apache Spark Streaming

Το Apache Spark είναι ένα άλλο σύστημα ανοιχτού κώδικα για κατανεμημένη επεξεργασία δεδομένων. Προσφέρει, εκτός από λειτουργίες για batch processing, λειτουργίες για stream processing ως μέρος της βιβλιοθήκης Apache Spark Streaming. Ωστόσο, το stream processing εφαρμόζεται χρησιμοποιώντας micro-batches, δηλαδή δεν είναι επεξεργασία tuple-by-tuple όπως στο Apache Flink. Οι εφαρμογές Apache Spark Streaming μπορούν να γραφούν σε Java, Scala ή Python. Εκτός από το Apache Spark Streaming υπάρχουν και άλλες βιβλιοθήκες, όπως μια βιβλιοθήκη για machine learning καθώς και για επεξεργασία γραφημάτων (graph processing)[19], [20].

Η αρχιτεκτονική εγκατάστασης μιας εφαρμογής Apache Spark φαίνεται στην παρακάτω εικόνα. Μια εφαρμογή εκτελείται με τη μορφή πολλαπλών ανεξάρτητων διεργασιών που κατανέμονται σε ένα cluster. Το SparkContext συντονίζει αυτές τις διεργασίες. Αυτός ο συντονιστής είναι ένα object στην main() συνάρτηση της εφαρμογής, που ονομάζεται Driver Program. Επιπλέον, το SparkContext συνδέεται σε ένα Cluster Manager που φροντίζει για την κατανομή πόρων.

Προς το παρόν, υπάρχουν τέσσερις Cluster Managers που υποστηρίζονται από το Apache Spark - Spark Standalone, Apache Mesos [21], Apache Hadoop YARN (Yet Another Resource Negotiator) [22] και Kubernetes [23]. Μόλις η σύνδεση έχει εξασφαλιστεί, το SparkContext αποκτά τους λεγόμενους εκτελεστές (executors) στους Worker Nodes. Κάθε executor είναι μια διαδικασία που ανήκει σε μία μόνο εφαρμογή, η οποία αποθηκεύει δεδομένα



Σχήμα 2.3: Apache Spark in Cluster Mode

και εκτελεί υπολογισμούς. Ως εκ τούτου, διαφορετικές εφαρμογές που τρέχουν στον ίδιο Apache Spark Cluster εκτελούνται σε διαφορετικά JVMs. Έτσι, τα δεδομένα δεν μπορούν να ανταλλαχθούν μεταξύ διαφορετικών εφαρμογών Apache Spark χωρίς χρήση εξωτερικού συστήματος αποθήκευσης.

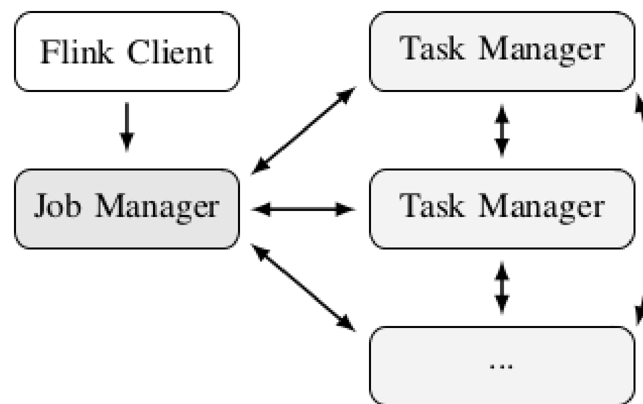
Μόλις ενεργοποιηθούν οι executors, δηλαδή βρεθούν οι πόροι που χρειάζονται για να τρέξουν, το SparkContext μεταδίδει το πρόγραμμα με τη μορφή αρχείων JAR ή Python σε αυτά. Στη συνέχεια, στέλνει tasks στις διεργασίες των executors. Μια διεργασία μπορεί να τρέξει πολλαπλά tasks σε διαφορετικά treads [24].

Μια κεντρική δομή δεδομένων που χρησιμοποιείται στο Apache Spark είναι η Resilient Distributed Dataset (RDD), που είναι μια κατανεμημένη αφαίρεση μνήμης δεδομένων. Το Σπαρκ εκτελεί υπολογισμούς στη μνήμη σε μεγάλα cluster με ανοχή σφαλμάτων μέσω RDDs. Ένα RDD βρίσκεται στην κύρια μνήμη, αλλά μπορεί να παραμείνει στο δίσκο όπως ζητήθηκε. Εάν χαθεί ένα διαμέρισμα του RDD, μπορεί να ξαναχτιστεί. Το Spark υποστηρίζει επίσης κοινόχρηστη μεταβλητή, μεταβλητή εκπομπής και μεταβλητή συσσωρευτή. Το Apache Spark Streaming αξιοποιεί ένα μοντέλο επεξεργασίας που ονομάζεται Discretized streams (D-Streams). Ένα τέτοιο D-Stream είναι μια ακολουθία των RDD. Μια εισερχόμενη ροή δεδομένων χωρίζεται σε batches που αποθηκεύονται σε RDD. Στη συνέχεια πραγματοποιούνται μετασχηματισμοί δεδομένων σε αυτά τα RDD, τα οποία εξάγουν ξανά ένα D-Stream [25].

2.2.3 Apache Flink

Το Apache Flink είναι ένα σύστημα ανοιχτού κώδικα με ικανότητες για batch και stream processing. Προσφέρει Java και Scala API για την ανάπτυξη εφαρμογών. Επιπλέον, υπάρχουν πολλές βιβλιοθήκες πάνω από το Apache Flink που παρέχουν, π.χ., λειτουργίες machine learning ή graph processing [3]. Η αρχιτεκτονική ενός Apache Flink Cluster φαίνεται στο παρακάτω σχήμα.

Το σχήμα δείχνει έναν Apache Flink Client, ένα Job Manager και τους Task Managers. Όταν ένα πρόγραμμα γίνεται deploy, ο client το μετατρέπει σε γράφημα ροής δεδομένων,



Σχήμα 2.4: Apache Flink Runtime

δηλαδή, κατευθυνόμενο ακυκλικό γράφημα (DAG) και το στέλνει στον Job Manager. Ο client δεν είναι μέρος της εκτέλεσης του προγράμματος και μπορεί, αφού μεταδώσει το γράφημα ροής δεδομένων, είτε να αποσυνδεθεί από τον Job Manager είτε να παραμείνει συνδεδεμένος για να λάβει πληροφορίες σχετικά με την πρόοδο εκτέλεσης.

Ο Job Manager είναι υπεύθυνος για τον προγραμματισμό των εργασιών μεταξύ των Task Manager και για τη διατήρηση του ελέγχου της εκτέλεσης. Μπορεί να υπάρχουν πολλαπλοί Job Manager, αλλά μόνο ένας μπορεί να είναι ο ηγέτης. Οι υπόλοιποι παραμένουν σε αναμονή και μπορούν να αναλάβουν σε περίπτωση αποτυχίας.

Οι Task Manager εκτελούν τα καθορισμένα μέρη του προγράμματος που τους έχουν ανατεθεί. Τεχνικά, ένας Task Manager είναι μια διαδικασία JVM. Πρέπει να υπάρχει τουλάχιστον ένας Task Manager σε μια Apache Flink εγκατάσταση. Με αυτόν τον τρόπο, ανταλλάσσουν δεδομένα μεταξύ τους όποτε χρειάζεται. Κάθε Task Manager παρέχει τουλάχιστον ένα task slot στο οποίο τα subtasks εκτελούνται σε πολλαπλά threads. Πολλά subtasks μπορούν να μοιράζονται ένα task slot εφόσον ανήκουν στην ίδια εφαρμογή, ακόμα και αν αποτελούν μέρος διαφορετικών tasks. Ενώ μία εργασία εκτελείται από ένα thread, το Apache Flink συνδυάζει πολλαπλά operator subtasks σε ένα κοινό task, όπως δύο συνεχόμενες λειτουργίες map. Ένα όφελος αυτής της βελτιστοποίησης είναι, π.χ., μειωμένη επιβάρυνση για επικοινωνία μεταξύ νημάτων.

Κάθε task slot έχει ένα υποσύνολο των πόρων που ανήκουν στον αντίστοιχο Task Manager. Ειδικότερα, η διαθέσιμη μνήμη κατανέμεται μεταξύ των task slots [26].

2.3 Machine Learning

2.3.1 Τύποι προβλημάτων

Σε ένα σύστημα ανάλογα με τα δεδομένα εξόδου, χαρακτηρίζεται το σύνολο εισόδου ως συνεχές ή διακριτό. Συνεχές είναι στην περίπτωση που το αποτέλεσμα για κάθε στοιχείου εισόδου μπορεί να πάρει οποιαδήποτε τιμή π.χ. δεδομένα απόστασης. Αντίθετα, στα διακριτά

δεδομένα τα αποτελέσματα μπορούν να πάρουν συγκεκριμένες τιμές από κάποιο πεπερασμένο σύνολο τιμών. Οι τιμές αυτές είναι πάντα αριθμήσιμες.

Με βάση την παραπάνω διαφοροποίηση, προκύπτει και η χαρακτηρισμός των προβλημάτων σε προβλήματα πρόβλεψης και προβλήματα κατηγοριοποίησης.

- **Προβλήματα Πρόβλεψης (Regression):** Στα προβλήματα πρόβλεψης τα δεδομένα είναι συνεχή. Τα αποτελέσματα από τα δεδομένα εισόδου προκύπτουν από μια συνάρτηση που εφαρμόζεται στα δεδομένα, χωρίς ωστόσο να είναι γνωστή η μορφή της. Κάνοντας χρήση όλων των γνωστών δεδομένων, προκύπτει η εκπαίδευση ενός αλγορίθμου, έτσι ώστε η συμπεριφορά του να πλησιάζει την άγνωστη συνάρτηση. Όταν στον αλγόριθμο αυτόν εισαχθούνε γνωστές τιμές δίνει σωστές τιμές εξόδου, ενώ όταν τροφοδοτηθεί με τιμές, για τις οποίες δεν είναι γνωστή η έξοδος, προβλέπει την τιμή που θα έδινε η άγνωστη συνάρτηση.
- **Προβλήματα Κατηγοριοποίησης (Classification):** Στα προβλήματα κατηγοριοποίησης τα δεδομένα είναι διακριτά. Τα αποτελέσματα εξόδου αντιπροσωπεύουν τις κατηγορίες στις οποίες ανήκουν τα διανύσματα εισόδου. Επομένως, όταν παρουσιάζονται νέα δεδομένα, ο αλγόριθμος δεν επιχειρεί να προσεγγίσει την τιμή μίας συνάρτησης, αλλά να κατατάξει την είσοδο σε μία γνωστή κατηγορία.

2.3.2 Random Forest

Ο Random Forest [27] είναι ένας αλγόριθμος μάθησης συνόλου (ensemble learning algorithm) που αναπτύχθηκε από τον Breiman [28]. Μία μέθοδος ensemble μάθησης δημιουργεί μεμονωμένους μαθητές (learners) και συγκεντρώνει τα αποτελέσματα. Ο Random Forest χρησιμοποιεί μια επέκταση στην προσέγγιση bagging. Bagging είναι η διαδικασία τυχαίας επιλογής ενός υποσυνόλου από το σύνολο των δεδομένων εκπαίδευσης, για την εκπαίδευση ενός συνδυαστικού αλγόριθμου ταξινόμησης. Το τυχαία επιλεγμένο υποσύνολο, επανατοποθετείται στο σύνολο δεδομένων, έτσι οι άλλοι ταξινομητές που συνθέτουν το συνδυαστικό μοντέλο μπορούν να επιλέξουν τυχαία ξανά μέρος ή ολόκληρο το υποσύνολο των δεδομένων για σκοπούς εκπαίδευσης. Στο Bagging, κάθε ταξινομητής χτίζεται ξεχωριστά δουλεύοντας με ένα bootstrap δείγμα των δεδομένων εισαγωγής. Σε έναν κανονικό ταξινομητή αποφάσεων (decision tree classifier) μια απόφαση που πραγματοποιείται στον διαχωρισμό κόμβων λαμβάνεται με βάση όλα τα χαρακτηριστικά γνωρίσματα.

Αλλά στον Random Forest, η καλύτερη παράμετρος στον κάθε κόμβο σε ένα δέντρο αποφάσεων γίνεται από τυχαία επιλεγμένο αριθμό χαρακτηριστικών. Αυτή η τυχαία επιλογή χαρακτηριστικών βοηθά τον Random Forest όχι μόνο να αποδώσει καλύτερα όταν υπάρχουν πολλά features, αλλά επίσης το βοηθά στη μείωση της αλληλεξάρτησης (συσχέτιση) μεταξύ των feature attributes [29].

Όπως αναφέρει και ο συγγραφέας [28], ο αριθμός των features που επιλέγονται ανα κόμβο απόφασης ενός δέντρου αποφασίζει το ποσοστό σφάλματος error rate του δάσους

κατηγοριοποίησης. Το ποσοστό σφάλματος του Random Forest classifier εξαρτάται από τη συσχέτιση μεταξύ δύο οποιονδήποτε δέντρων και από τη δύναμη ταξινόμησης του κάθε ατομικού δέντρου. Η μείωση των τυχαίων χαρακτηριστικών που επιλέγονται προκαλεί μείωση τόσο στη συσχέτιση μεταξύ των δέντρων ταξινόμησης όσο και στη δύναμη της ταξινόμησης κάθε μεμονωμένου δέντρου. Η αύξηση των τυχαίων χαρακτηριστικών αυξάνει τόσο τη συσχέτιση μεταξύ των δέντρων όσο και τη δύναμη κάθε δέντρου. Ο Breiman εξηγεί ότι το Out-of-Bag (OOB) error rate είναι μια ένδειξη του πόσο καλά αποδίδει ένα δάσος κατηγοριοποίησης.

Μια μέθοδος που χρησιμοποιείται όταν αναπτύσσεται ένας αλγόριθμος machine learning με χρήση του random forest είναι η μέθοδος cross-validation. Η μέθοδος περιγράφει το σπάσιμο του συνόλου των γνωστών δεδομένων σε δύο τμήματα. Αφού χρησιμοποιήσουμε το πρώτο τμήμα για τηνεκπαίδευση του αλγορίθμου, κατόπιν εισάγουμε σε αυτόν τα στοιχεία του δεύτερου τμήματος. Οι απαντήσεις που μας δίνει η μέθοδός μας ελέγχονται με τις γνωστές τιμές εξόδου των δεδομένων μας και προκύπτει ένα ποσοστό επιτυχίας της εκπαίδευσης. Η μέθοδος cross-validation βοηθάει να ελέγξουμε την ικανότητα του αλγορίθμου να απαντά σωστά, αξιοποιώντας την παρεχόμενη εκπαίδευση.

Κεφάλαιο 3

Ανάλυση Benchmark

Στο κάτωθι κεφάλαιο περιγράφουμε την ανάλυση της απόδοσης των Apache Spark και Apache Flink χρησιμοποιώντας την τεχνολογία Apache Beam. Αρχικά, αναφέρουμε τα δεδομένα που χρησιμοποιήθηκαν. Στη συνέχεια, περιγράφουμε την διαδικασία τροφοδότησης των δεδομένων ώστε να προκύπτει ένα σύστημα επεξεργασίας δεδομένων πραγματικού χρόνου και ορίζουμε τις μετρήσεις που λαμβάνουμε. Έπειτα, παρουσιάζουμε την αρχιτεκτονική του benchmark καθώς και τους υπολογιστικούς πόρους που χρησιμοποιήθηκαν. Τέλος, αναλύουμε τα αποτελέσματα.

3.1 Δεδομένα εισόδου

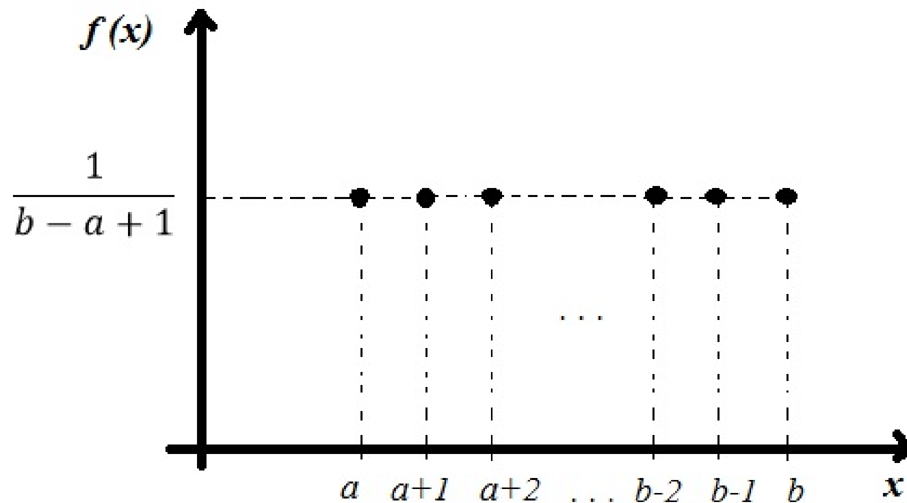
Προκειμένου να μπορέσουμε να μετρήσουμε την απόδοση των Streaming Data Processing Systems ανάλογα με την μορφολογία των δεδομένων, κάναμε χρήση δύο διαφορετικών κατανομών για διακριτές τιμές θετικών ακεραίων. Συγκεκριμένα, επιλέξαμε να μελετήσουμε την απόδοση για την ομοιόμορφη [30] και την zipf κατανομή [31].

3.1.1 Ομοιόμορφη κατανομή

Ο συμβολισμός $X \sim$ discrete uniform *(a,b)* δείχνει ότι η τυχαία μεταβλητή X ακολουθεί την διακριτή ομοιόμορφη κατανομή με ακέραιες παραμέτρους a και b, όπου a<b. Μια διακριτή ομοιόμορφη τυχαία μεταβλητή X με παραμέτρους a και b έχει συνάρτηση μάζας πιθανότητας:

$$f(x) = \frac{1}{b - a + 1} \qquad x = a, a + 1, ..., b. \tag{3.1}$$

Δημιουργήσαμε 10 αρχεία μεγέθους 1.000.000 αριθμών το καθένα ανάλογα με το πόσους διαφορετικούς αριθμούς επιθυμούμε σε κάθε αρχείο. Έτσι, μελετήσαμε την απόδοση των συστημάτων ανάλογα με το πλήθος των διαφορετικών αριθμών. Συμβολίζουμε το πλήθος των διαφορετικών αριθμών ως cardinality. Ειδικότερα, δημιουργήθηκαν 10 αρχεία με διαφορετικές τιμές cardinality όπως φαίνεται και από τον πίνακα 3.1.



Σχήμα 3.1: Συνάρτηση μάζας πιθανότητας της ομοιόμορφης διακριτής κατανομής

File	Cardinality	Size
1	10	1m
2	100	1m
3	1000	1m
4	10,000	1m
5	99,992	1m
6	289,198	1m
7	431,959	1m
8	532,426	1m
9	603,957	1m
10	632,079	1m

Πίνακας 3.1: Χαρακτηριστικά αρχείων διακριτής ομοιόμορφης κατανομής

3.1.2 Zipf κατανομή

Ο νόμος του Zipf είναι ένας εμπειρικός νόμος που αναφέρεται στο γεγονός ότι πολλοί τύποι των δεδομένων που μελετήθηκαν και στις φυσικές και στις κοινωνικές επιστήμες, ακολουθούν μια κατανομή γνωστή ως νόμος των δυνάμεων (power law).

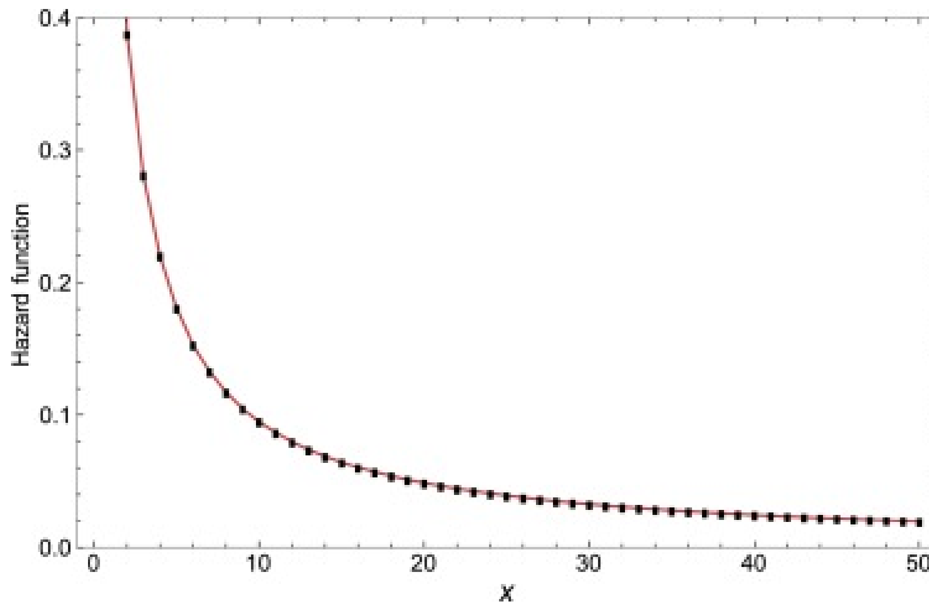
Ο νόμος αναδύεται σε φαινόμενα όπου έχουμε πολλά και μικρά γεγονότα και λίγα αλλά μεγάλα γεγονότα, πχ. έχουμε πολύ λίγους και μεγάλους σεισμούς και πολλούς μικρούς σεισμούς, έχουμε λίγους και πλούσιους και πολλούς αλλά φτωχούς, έχουμε λίγες αλλά μεγάλες πόλεις και πολλές αλλά μικρές, έχουμε λίγους αλλά πολύ δημοφιλείς ιστότοπους αλλά πολλούς που δεν τους έχει επισκεφτεί κανείς κλπ.

Ο ίδιος ο Ζιπφ ασχολήθηκε με το να προβλέψει τη συχνότητα της εμφάνισης των λέξεων μέσα σε ένα κείμενο. Δηλώνει δε ότι «αν οι λέξεις ταξινομηθούν κατά φθίνουσα σειρά του αριθμού εμφάνισής τους σε ένα σχετικά μεγάλο κείμενο, τότε η θέση/σειρά μιας λέξης σε αυτόν τον κατάλογο όταν πολλαπλασιάζεται με τη συχνότητα εμφάνισής της είναι ίση με μια σταθερά».

Η εξίσωση για αυτή τη σχέση είναι:

$$rxf = k \tag{3.2}$$

όπου ρ είναι η θέση/σειρά της λέξης, φ είναι η συχνότητα και κ είναι η σταθερά.



Σχήμα 3.2: Συνάρτηση μάζας πιθανότητας της zipf κατανομής

Για διακριτούς ακέραιους αριθμούς η συνάρτηση μάζας πιθανότητας της κατανομής zipf δίνεται από την εξίσωση:

$$f(k, a) = \frac{1}{\zeta(a)k^a} \qquad k \geq 1, a > 1 \tag{3.3}$$

όπου κ = μέγεθος του dataset, α = power law, ζ = Riemann zeta function

Δημιουργήσαμε 10 αρχεία μεγέθους 1.000.000 αριθμών το καθένα ανάλογα με το επιθυμητό power law. Έτσι, μελετήσαμε την απόδοση των συστημάτων ανάλογα με διαφορετικές τιμές του power law. Όσο αυξάνεται το power law τόσο μειώνεται το πλήθος των διαφορετικών διακριτών αριθμών. Ειδικότερα, δημιουργήθηκαν 10 αρχεία με διαφορετικά power law όπως φαίνεται και από τον πίνακα 3.2.

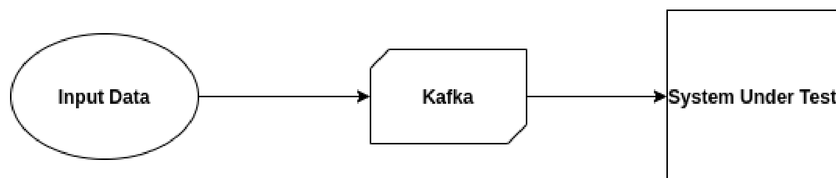
3.2 Αρχιτεκτονική Benchmark

Η γενικότερη μορφή της αρχιτεκτονικής φαίνεται στο σχήμα 3.3. Το benchmark χωρίζεται σε 3 στάδια.

Αρχικά, κάθε αρχείο (dataset) διαβάζεται και στέλνεται στους message brokers του Apache Kafka. Το πρόγραμμα που διαβάζει και αναλαμβάνει να στείλει τα δεδομένα εισόδου είναι γραμμένο σε python και λαμβάνει ως είσοδο τον ρυθμό με τον οποίο θέλουμε να στείλουμε δεδομένα (data ingestion rate). Το πρόγραμμα κάνει χρήση της βιβλιοθήκης python της

File	Power Law	Size
1	1.00001	1m
2	1.0001	1m
3	1.001	1m
4	1.01	1m
5	1.1	1m
6	1.3	1m
7	1.5	1m
8	1.7	1m
9	2	1m
10	3	1m

Πίνακας 3.2: Χαρακτηριστικά αρχείων διακριτής zipf κατανομής



Σχήμα 3.3: Γενική μορφή του Benchmark

εταιρίας Confluent [32], η οποία παρέχει ένα πιο αποδοτικό API που παράγει τα δεδομένα στους Kafka brokers πιο γρήγορα και αποτελεσματικά. Στη συνέχεια, το σύστημα που εξετάζεται κάθε φορά, γνωστό και ως system under test (SUT), δηλαδή είτε το Apache spark είτε το Apache Flink, εκτελεί το πρόγραμμα με το επιθυμητό query γραμμένο με την Java SDK της τεχνολογίας Apache Beam. Τέλος, το SUT γράφει σε αρχεία το αποτέλεσμα του query καθώς και τις μετρήσεις που λαμβάνουμε (το είδος των μετρήσεων αναλύεται παρακάτω).

3.2.1 Τροφοδοσία δεδομένων

Τα δεδομένα διαβάζονται από τα αρχεία που έχουμε δημιουργήσει και προκειμένου να προσομοιώσουμε την λειτουργία stream processing με δεδομένα πραγματικού χρόνου, για κάθε αριθμό δημιουργούμε ένα object το οποίο περιέχει τον αριθμό και την τωρινή χρονική στιγμή (timestamp). Στη συνέχεια, αυτό το object/record αποστέλλεται στους kafka brokers με τον python script που αναφέραμε προηγουμένως. Για το σκοπό αυτό, δημιουργήσαμε ένα kafka topic με συντελεστή αναπαραγωγής 2 (replication factor) και 2 διαμερίσεις (partitions).

3.2.2 Εκτέλεση προγράμματος

Στη φάση της εκτέλεσης, το query εκτελείται και στο Apache spark και στο Apache flink. Όταν εκτελείται το πρόγραμμα, στο σύστημα δεν υπάρχει κάποιο άλλο πρόγραμμα που να

εκτελείται. Επίσης, μετά από κάθε ολοκλήρωση εκτέλεσης του προγράμματος γίνεται επανεκκίνηση του συστήματος. Το query εκτελείται με παραλληλισμό επιπέδου 2, αφού έχουμε δημιουργήσει 2 topic partitions και το SUT δημιουργεί δύο στιγμιότυπα του κώδικα (instances), ώστε να καταναλώσει δεδομένα και από τα 2 partition. Το επίπεδο παραλληλισμού ορίζεται διαφορετικά για κάθε σύστημα. Στην περίπτωση του Apache Spark Streaming εξασφαλίζεται δίνοντας στην είσοδο μια τιμή στην παράμετρο *spark.default.parallelism*, ενώ στην περίπτωση του Apache Flink η παράμετρος δίνεται κατα την υποβολή του προγράμματος με την παράμετρο *-parallelism*. Επιπλέον, ο κώδικας του pipeline έχει γραφεί, έτσι ώστε τα streaming frameworks να επεξεργάζονται τους αριθμούς με βάση την χρονική στιγμή που χαρακτηρίζει το καθένα και όχι με βάση την χρονική στιγμή που εισέρχονται στο σύστημα (processing time). Έτσι, προσομοιώνουμε την περίπτωση που τα δεδομένα λαμβάνονται σε πραγματικό χρόνο και πρέπει να αναλυθούν. Ως εκ τούτου, αποφεύγεται και η περίπτωση που το πρόγραμμα τρέχει σε τουλάχιστον 2 διαφορετικούς κόμβους για τους οποίους το ρολόι επεξεργασίας είναι διαφορερικό (cpu clock time).

3.2.3 Query

Το query που χρησιμοποιήσαμε είναι αθροιστικό (aggregation) με χρήση window. Ειδικότερα, δεδομένου ενός παραθύρου διάρκειας t βρίσκουμε τους αριθμούς που ανήκουν σε αυτό το παράθυρο και στη συνέχεια υπολογίζουμε την συχνότητα εμφάνισης κάθε αριθμού σε αυτό το παράθυρο. Αν ο πρώτος αριθμός που εισήχθη στο παράθυρο έχει timestamp t_1 και ο τελευταίος έχει timestamp t_2, τότε η απόλυτη τιμή της διαφοράς των 2 αυτών χρονικών στιγμών είναι το πολύ t.

$$|t_1 - t_2| \leq t \tag{3.4}$$

Η διαδικασία αυτή επαναλαμβάνεται μέχρι να τροφοδοτηθούν όλα τα δεδομένα και για κάθε παράθυρο προκύπτει η συχνότητα εμφάνισης των αριθμών που ανήκουν σε αυτό το παράθυρο.

3.2.4 Metrics

Τα σύγχρονα συστήματα επεξεργασίας δεδομένων real-time [3], [33] χρησιμοποιούν την μέτρηση event-time για να μετρήσουν την απόδοση ενός συστήματος. Παρόμοια θα κάνουμε χρήση του ορισμού για το event-time latency για να αξιολογήσουμε την απόδοση του κάθε streaming framework.

Ορισμός 1 (Event Time για Windowed γεγονότα): Το event-time κάθε window είναι η μέγιστη χρονική στιγμή από όλους τους αριθμούς που συγκαταλέγονται σε αυτό το window.

Όπως αναφέρθηκε και στην περιγραφή του query κάθε window συγκεντρώνει ένα πλήθος αριθμών ανάλογα με τη χρονική στιγμή που τα χαρακτηρίζει. Το event-time για κάθε window είναι η μέγιστη χρονική στιγμή των αριθμών που το απαρτίζουν.

Ορισμός 2 (Event Time Latency ενός Window): Έστω ότι σε ένα συγκεκριμένο SDPS, όπου η επεξεργασία ενός αρχείου δεδομένων με τη μέθοδο που προσομοιάζει το φαινόμενο real-time streaming, προέκυψαν n παράθυρα διάρκειας t το καθένα. Κάθε παράθυρο έχει event-time t_i και εφαρμόζει aggegation query στα δεδομένα που έχει συμπεριλάβει. Το aggegation query ολοκληρώνεται την χρονική στιγμή p_i (χρονική στιγμή που έχει το μηχάνημα) ανάλογα με το παράθυρο. Το event-time latency ενός window t_k είναι η απόλυτη τιμή της διαφοράς του p_k από το t_k:

$$|p_k - t_k| \tag{3.5}$$

Ορισμός 3 (Event Time Latency): Με βάση τον παραπάνω ορισμό το Event Time Latency που χαρακτηρίζει την απόδοση ενός συστήματος για ένα συγκεκριμένο αρχείο εισόδου είναι ο μέσος όρος των event-time latency όλων window:

$$Latency = \frac{\sum_{k=1}^{n}(|p_k - t_k|)}{n} \tag{3.6}$$

Προσδιορίζουμε το throughput ενός συστήματος επεξεργασίας δεδομένων, ως ο αριθμός των δεδομένων που ένα σύστημα μπορεί να επεξεργαστεί μια δεδομένη χρονική στιγμή. Το throughput και το event time latency συνήθως δεν συσχετίζονται. Για παράδειγμα, ένα σύστημα streaming το οποίο ομαδοποιεί γεγονότα (events) σε batches μπορεί να εξασφαλίσει καλύτερο throughput. Ωστόσο, ο χρόνος που καταναλώνεται στην ομαδοποίηση επηρεάζει το event time latency.

Γενικότερα, μια εφαρμογή πρέπει να λάβει υπόψη τον ρυθμό με τον οποίο καταφθάνουν τα δεδομένα. Όταν ο ρυθμός αυξάνεται το σύστημα πρέπει να αντεπεξέλθει, ώστε να μπορέσει να επεξεργαστεί τα νέα δεδομένα απευθείας χωρίς να χρειαστεί να τα τοποθετήσει σε ουρά (backpressure).

Για να μπορέσει το σύστημά μας να επιφέρει το μέγιστο δυνατό throughput χωρίς να παρουσιάσει το φαινόμενο backpressure τροφοδοτούμε τους kafka brokers με δεδομένα σύμφωνα με ένα συγκεκριμένο ρυθμό (records/second). Αν αυξάνοντας το ρυθμό τροφοδοσίας παρατηρήσουμε αύξηση του event time latency, τότε το σύστημα δεν μπορεί να αντιμετωπίσει τον ρυθμό που έρχονται τα δεδομένα και μερικά από αυτά τα αποθηκεύει σε ουρά, προκειμένου να τα επεξργαστεί σε μεταγενέστερο χρόνο. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να βρεθεί ο μέγιστος δυνατός ρυθμός τροφοδοσίας, ο οποίος είναι και το throughput.

Η μεθοδολογία αυτή εκτελείται για κάθε αρχείο εισόδου και για κάθε διαφορετικό window duration.

Ορισμός 4 (Throughput): Είναι ο μέγιστος ρυθμός που μπορούμε να τροφοδοτήσουμε δεδομένα το σύστημα χωρίς να παρατηρηθεί συνεχόμενη αύξηση του event time latency.

3.3 Χαρακτηριστικά χρησιμοποιηθέντος υλικού

Προκειμένου να πραγματοποιήσουμε μετρήσεις και να αναλύσουμε τη συμπεριφορά του aggregation query στο Apache Spark Streaming και στο Apache Flink για κάθε αρχείο διακριτών αριθμών, ήταν απαραίτητη η πρόσβαση σε μερικά φυσικά μηχανήματα, server, ο οποίος είχε αρκετούς πόρους για τις ζητούμενες μετρήσεις. Τέτοιας μορφής μηχανήματα δανειστήκαμε από το εργαστήριο CSlab - Computer Systems Laboratory του Εθνικού Μετσόβιου Πολυτεχνείου.

Τα μηχανήματα που επιλέξαμε είναι της σειράς clone που βρίσκονται στο εργαστήριο. Συγκεκριμένα, δανειστήκαμε 6 μηχανήματα, ενώ σε κάθε μηχάνημα η CPU έχει 8 cores και τοπική μνήμη 8GB εκτός από ένα που έχει τοπική μνήμη 6GB. Το ΛΣ του μηχανήματος είναι Debian Linux 8 (jessie). Ο αποθηκευτικός χώρος των μηχανημάτων είναι κοινός για όλους, δηλαδή πρόκειται για Network File System (NFS). Για την εγκατάσταση των τεχνολογιών δύο clones λειτουργούν σαν kafka brokers, ενώ για τον Zookeeper που είναι υπεύθυνος για τους brokers χρησιμοποιήσαμε ένα ξεχωριστό μηχάνημα. Επιπλέον, το Hadoop και το YARN εγκαταστάθηκαν στα υπόλοιπα τρία μηχανήματα. Στον πίνακα 3.3 φαίνονται τα μηχανήματα μαζί με κάποιες βασικές πληροφορίες.

Clone Name	**Usage**	**CPU**	**RAM**
clone11	Zookeeper	Intel Xeon CPU E5335 2.00GHz	6GB
clone19	Kafka Broker	Intel Xeon CPU E5405 2.00GHz	8GB
clone20	Kafka Broker	Intel Xeon CPU E5405 2.00GHz	8GB
clone13	Namenode, ResourceManager	Intel Xeon CPU E5335 2.00GHz	8GB
clone14	Datanode, NodeManager	Intel Xeon CPU E5335 2.00GHz	8GB
clone15	Datanode, NodeManager	Intel Xeon CPU E5335 2.00GHz	8GB

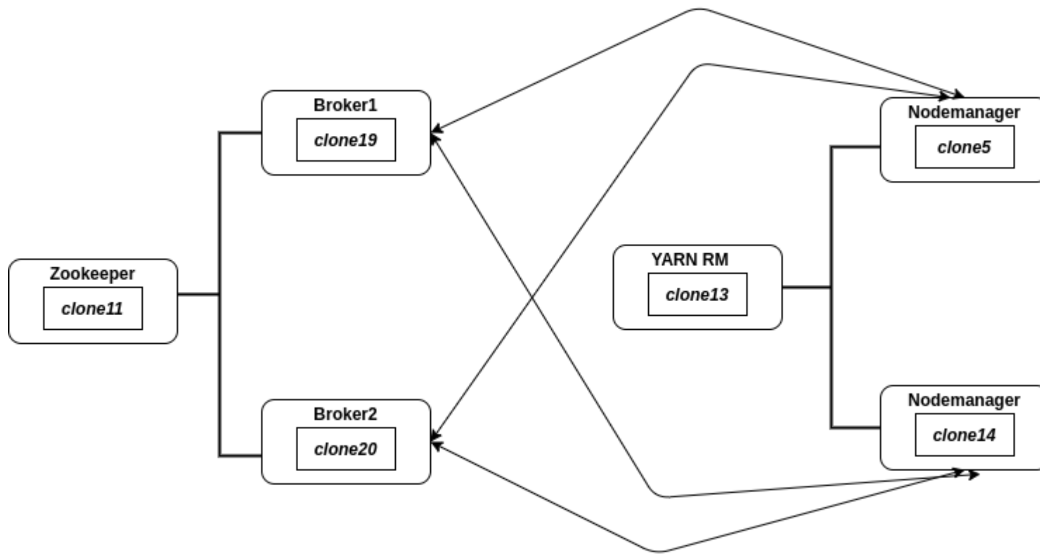
Πίνακας 3.3: Βασικά στοιχεία των μηχανημάτων clones

Όπως φαίνεται και στον πίνακα 3.3, διαθέτουμε συνολικά 7 nodes. Οι αποστάσεις μεταξύ των nodes είναι ίδιες, οπότε και το latency πρόσβασης δεδομένων ενός node προς οποιοδήποτε άλλο, είναι ακριβώς το ίδιο. Τα Apache Spark και Apache Flink εγκαθίστανται τόσο στο βασικό node που λειτουργεί σαν ResourceManager όσο και στα 2 nodes που είναι οι NodeManager. Στους NodeManager γίνεται δέσμευση πόρων για την εκτέλεση ενός προγράμματος. Ο ResourceManager αναλαμβάνει να δεσμεύσει τους πόρους που χρειάζεται το πρόγραμμα. Στην εικόνα 3.4 φαίνονται οι εκδόσεις των τεχνολογιών.

Technology	**version**
Apache Hadoop	2.7.7
Apache Kafka	2.4.1
Apache Beam	2.30.0
Apache Spark	2.4.7
Apache Flink	1.11.2

Πίνακας 3.4: Εκδόσεις των τεχνολογιών

Η τοπολογία των μηχανημάτων φαίνεται στο σχήμα 3.4.



Σχήμα 3.4: Αρχιτεκτονική μηχανημάτων clone

Κεφάλαιο 4

Αποτελέσματα Benchmarking

Στο κεφάλαιο που ακολουθεί, Θα αξιολογήσουμε την απόδοση των Apache Spark και Apache Flink. Πριν γίνει η παρουσίαση των αποτελεσμάτων θα αναφερθούν καποιες παραμετροποιήσεις που έγιναν κατά την εκτέλεση των streaming frameworks.

4.1 Παραμετροποίηση συστήματος

Επιλέγουμε ένα μόνο κεντρικό node, 1 driver node, 3 workers στην περίπτωση του Apache Spark και 3 task slot στην περίπτωση του Apache Flink. Ενεργοποιούμε την λειτουργία backpressure σε κάθε σύστημα, δηλαδή δεν επιτρέπουμε το σύστημα να απορροφήσει περισσότερα δεδομένα από όσα αντέχει και στη συνέχεια να τερματίσει απρόσμενα.

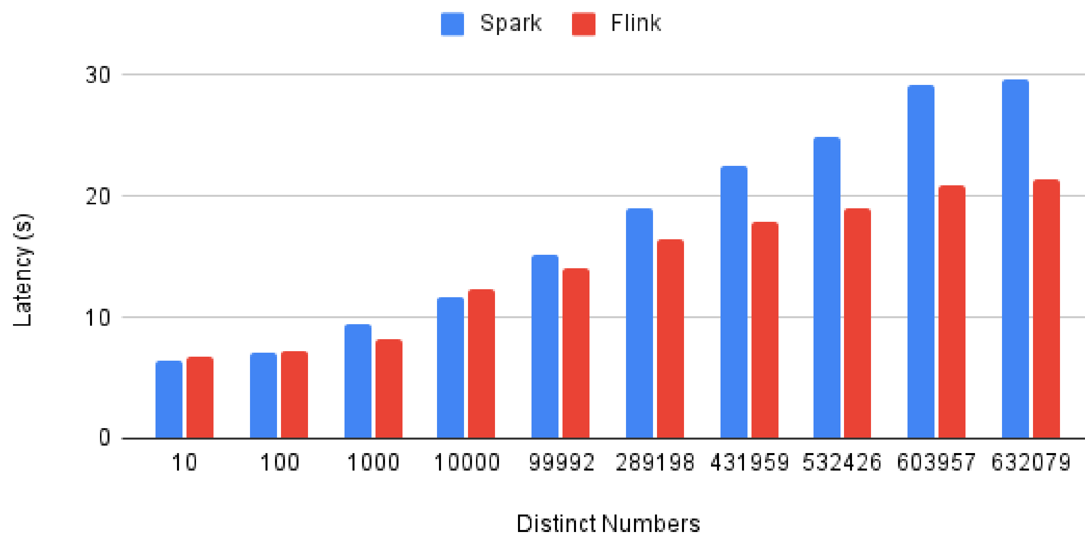
Για να αποδώσει το σύστημα βέλτιστα είναι απαραίτητο να παραμετροποιήσουμε το σύστημα. Ειδικότερα, προσαρμόσαμε το block interval που είναι υπεύθυνο για την διαμέριση των RDDs στο Spark. Ο αριθμός των RDD διαμερίσεων ενός batch είναι το πολύ $\frac{batch\ interval}{block\ interval}$. Όσο αυξάνεται το μέγεθος ενός cluster, μειώνοντας το block interval είναι πιθανό να αυξηθεί ο παραλληλισμός. Ένας από τους κύριους λόγους που το Spark αποδίδει καλά είναι η διαμέριση των RDDs. Ωστόσο, ανάλογα με την περίπτωση, ο ιδανικός αριθμός από διαμερίσεις αλλάζει. Για όλα τα συστήματα, η επιλογή του σωστού επιπέδου παραλληλισμού είναι απαραίτητη για την ισορροπία μεταξύ καλής χρήσης των πόρων και εξάντλησης του δικτύου.

4.2 Αποτελέσματα ομοιόμορφης κατανομής

Χρησιμοποιήσαμε το aggregation query που περιγράψαμε για διάρκεια window 5, 8 και 10 sec, ενώ το window slide ήταν 0. Παρουσιάζονται τα αποτελέσματα για κάθε window και συγκεκριμένα η τιμή του event-time latency ανάλογα με το πλήθος των διαφορετικών αριθμών ή αλλιώς ανάλογα με το ποιο αρχείο τροφοδοτήσαμε κάθε φορά.

4.2.1 Window 5 seconds

Για το Spark το throughput είχε τη μέγιστη τιμή των 1500 για cardinality 1 και όσο ανέβαινε το cardinality, μειωνόταν μέχρι που έφτασε τα 1000 στο τέλος. Για το Flink όμοια ξεκίνησε από 2500 και στο τελευταίο αρχείο έπεσε στα 2000.



Σχήμα 4.1: Avg Event-Time Latency per Uniform file, Window 5 sec

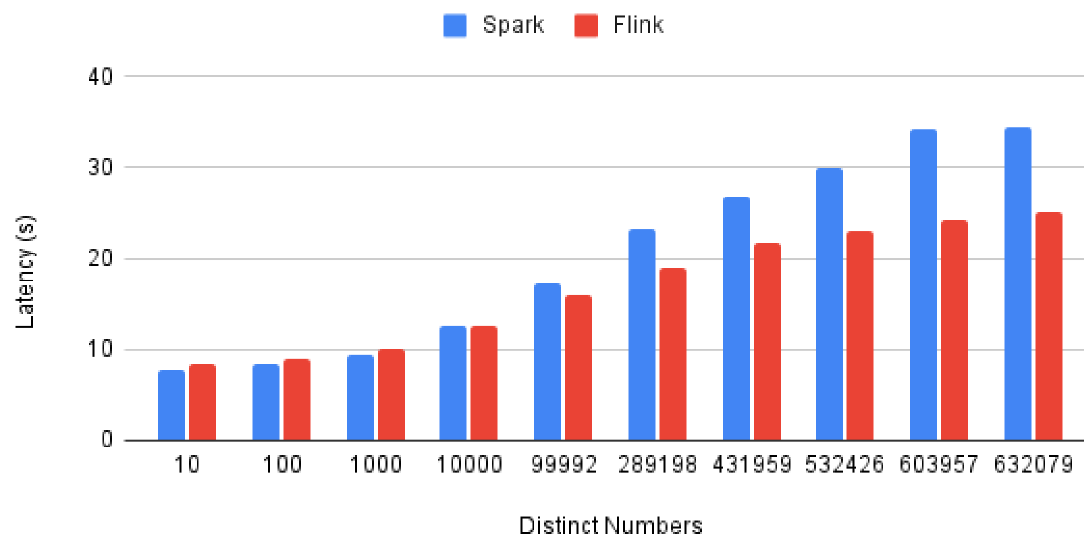
Στο σχήμα 4.1 παρουσιάζεται το Event-Time Latency για κάθε cardinality/αρχείο. Όπως φαίνεται και από το σχήμα, το Apache Spark αποδίδει καλύτερα όταν το cardinality είναι μικρό και συγκεκριμένα για τιμές 10, 1000, 10000. Καθώς ο αριθμός των διαφορετικών αριθμών αυξάνεται το Apache Flink επιφέρει καλύτερο latency σε σχέση με το Apache Spark. Για τις τελευταίες τιμές του cardinality η διαφορά στο latency είναι της τάξεως των 5-8 sec.

Επιπρόσθετα, όσο αυξανόταν το cardinality χρειάστηκε να αλλάξουμε και το batch Interval στην περίπτωση του Apache Spark, λόγω της αδυναμίας του να επεξεργαστεί τα δεδομένα δημιουργώντας πολλά RDD partition. Μειώνοντας το batch Interval μειώνεται το throughput αλλά ταυτόχρονα μειώνεται και το event-time latency.

4.2.2 Window 8 seconds

Για το Spark το throughput είχε τη μέγιστη τιμή των 1300 για cardinality 1 και όσο ανέβαινε το cardinality, μειωνόταν μέχρι που έφτασε τα 750 στο τέλος. Για το Flink όμοια ξεκίνησε από 2300 και στο τελευταίο αρχείο έπεσε στα 1900.

Στο σχήμα 4.2 παρουσιάζεται το Event-Time Latency για κάθε cardinality/αρχείο. Όπως φαίνεται και από το σχήμα, το Apache Spark αποδίδει καλύτερα όταν το cardinality είναι μικρό



Σχήμα 4.2: Avg Event-Time Latency per Uniform file, Window 8 sec

και συγκεκριμένα για τιμές 10, 100, 1000, 10000. Καθώς ο αριθμός των διαφορετικών αριθμών αυξάνεται το Apache Flink επιφέρει καλύτερο latency σε σχέση με το Apache Spark. Για τις τελευταίες τιμές του cardinality η διαφορά στο latency είναι της τάξεως των 4-8 sec.

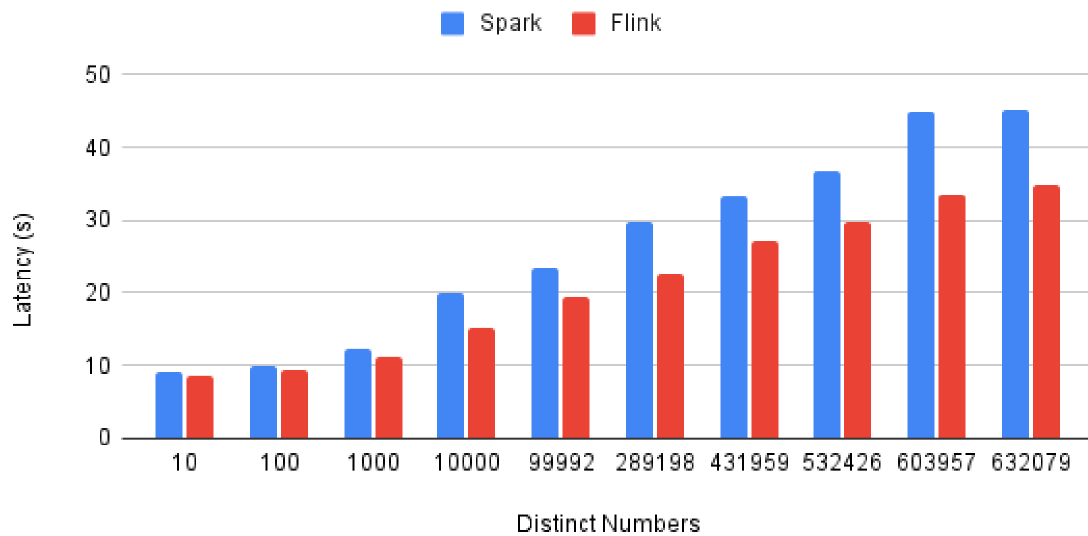
Επιπρόσθετα, όσο αυξανόταν το cardinality χρειάστηκε να αλλάξουμε και το batch Interval στην περίπτωση του Apache Spark, λόγω της αδυναμίας του να επεξεργαστεί τα δεδομένα δημιουργώντας πολλά RDD partition. Μειώνοντας το batch Interval μειώνεται το throughput αλλά ταυτόχρονα μειώνεται και το event-time latency.

Επίσης, παρατηρείται αύξηση του latency στο συγκεκριμένο window σε σχέση με το προηγούμενο που ήταν μικρότερης διάρκειας. Ό λόγος που συμβαίνει αυτό στο Spark είναι λόγω της ιδιότητας να αποθηκεύει στην μνήμη cache. Ειδικότερα, το Spark αποθηκεύει στην μνήμη τα αποτελέσματα ενός window operation, με σκοπό να τα εκμεταλλευτεί αργότερα. Η συμπεριφορά αυτή οδηγεί στη γρήγορη κατανάλωση της μνήμης. Σε αντίθεση, το Flink δεν μοιράζεται τα αποτελέσματα ενός aggregation query μεταξύ διαφορετικών window. Το Flink εκτελεί το aggregation αμέσως, χωρίς να περιμένει να κλείσει το παράθυρο.

4.2.3 Window 10 seconds

Για το Spark το throughput είχε τη μέγιστη τιμή των 1050 για cardinality 1 και όσο ανέβαινε το cardinality, μειωνόταν μέχρι που έφτασε τα 650 στο τέλος. Για το Flink όμοια ξεκίνησε από 2000 και στο τελευταίο αρχείο έπεσε στα 1600.

Στο σχήμα 4.3 παρουσιάζεται το Event-Time Latency για κάθε cardinality/αρχείο. Όπως φαίνεται και από το σχήμα, το Apache Flink σε σχέση με το Apache Spark. Αν και για μικρά



Σχήμα 4.3: Avg Event-Time Latency per Uniform file, Window 10 sec

cardinality η διαφορά είναι μικρή, όταν το πλήθος των διαφορετικών αριθμών είναι μεγάλο η διαφορά αυξάνεται αισθητά. Για τις τελευταίες τιμές του cardinality η διαφορά στο latency είναι της τάξεως των 4-9 sec.

Επιπρόσθετα, όσο αυξανόταν το cardinality χρειάστηκε να αλλάξουμε και το batch Interval στην περίπτωση του Apache Spark, λόγω της αδυναμίας του να επεξεργαστεί τα δεδομένα δημιουργώντας πολλά RDD partition. Μειώνοντας το batch Interval μειώνεται το throughput αλλά ταυτόχρονα μειώνεται και το event-time latency.

Επίσης, παρατηρείται αύξηση του latency στο συγκεκριμένο window σε σχέση με το προηγούμενο που ήταν μικρότερης διάρκειας. Ό λόγος που συμβαίνει αυτό στο Spark είναι λόγω της ιδιότητας να αποθηκεύει στην μνήμη cache. Ειδικότερα, το Spark αποθηκεύει στην μνήμη τα αποτελέσματα ενός window operation, με σκοπό να τα εκμεταλλευτεί αργότερα. Η συμπεριφορά αυτή οδηγεί στην γρήγορη κατανάλωση της μνήμης. Σε αντίθεση, το Flink δεν μοιράζεται τα αποτελέσματα ενός aggregation query μεταξύ διαφορετικών window. To Flink εκτελεί το aggregation αμέσως, χωρίς να περιμένει να κλείσει το παράθυρο.

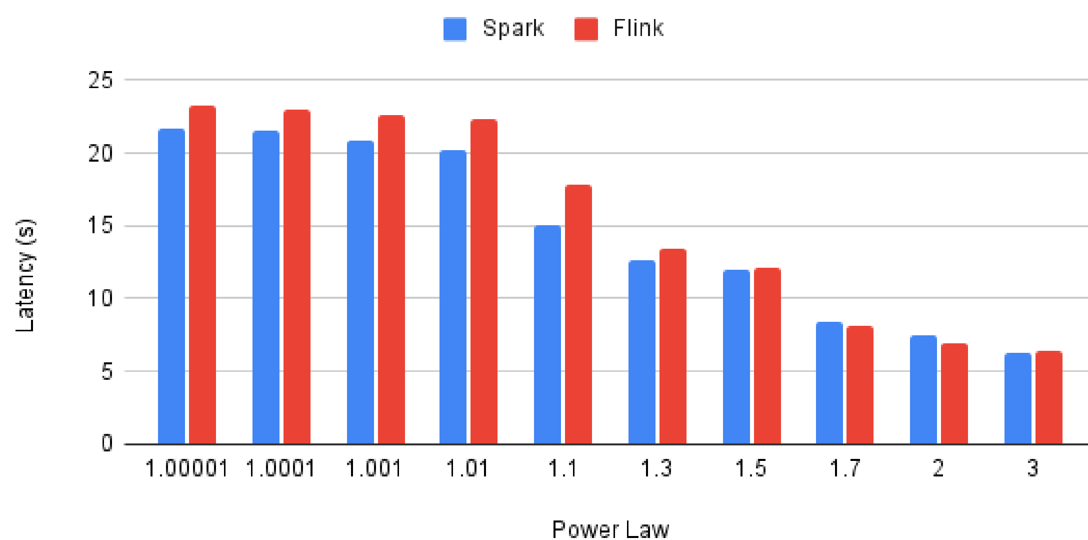
4.3 Αποτελέσματα zipf κατανομής

Χρησιμοποιήσαμε το aggregation query που περιγράψαμε για διάρκεια window 5, 8 και 10 sec, ενώ το window slide ήταν 0. Παρουσιάζονται τα αποτελέσματα για κάθε window και συγκεκριμένα η τιμή του event-time latency ανάλογα με το power law ή αλλιώς ανάλογα με το ποιο αρχείο τροφοδοτήσαμε κάθε φορά.

Όσο μικρότερο είναι το power law τόσο περισσότερους διαφορετικούς αριθμούς έχουμε. Ενδεικτικά, στην περίπτωση που το power law είναι 1.0001, το cardinality είναι 808010, ενώ όταν το power law είναι 3, το cardinality είναι 114.

4.3.1 Window 5 seconds

Για το Spark το throughput είχε τη μέγιστη τιμή των 1700 για power law 3 και όσο έπεφτε το power law μειωνόταν μέχρι που έφτασε τα 1200 στο τέλος. Για το Flink όμοια ξεκίνησε από 2700 και στο τελευταίο αρχείο έπεσε στα 2200.



Σχήμα 4.4: Avg Event-Time Latency per Zipf file, Window 5 sec

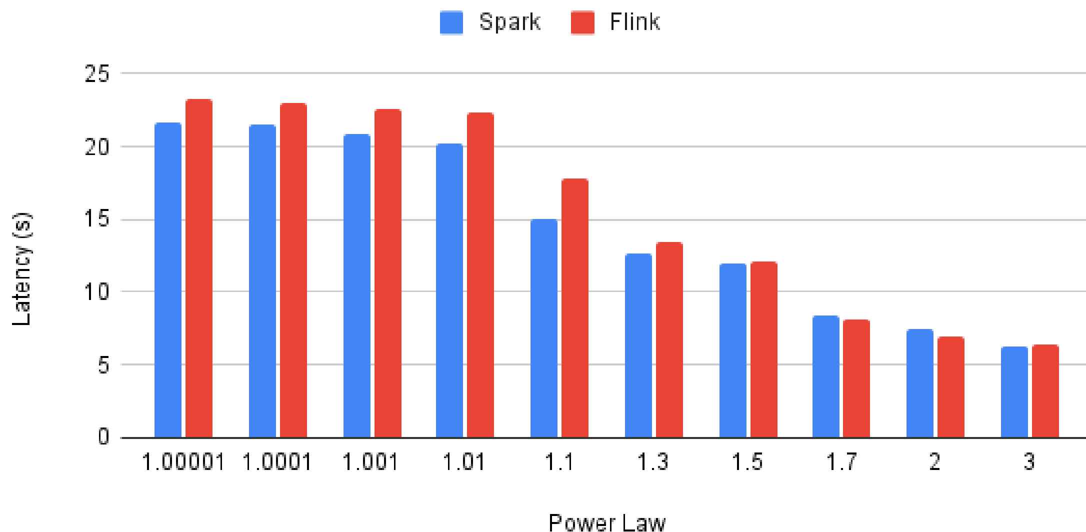
Στο σχήμα 4.4 παρουσιάζεται το Event-Time Latency για κάθε power law/αρχείο. Όπως φαίνεται και από το σχήμα, το Apache Flink αποδίδει καλύτερα όταν το power law είναι μεγάλο και συγκεκριμένα για τιμές 2 και 3. Καθώς το power law μειώνεται το Apache Spark επιφέρει καλύτερο latency σε σχέση με το Apache Flink. Για τις χαμηλότερες τιμές του power law η διαφορά στο latency είναι της τάξεως των 2-3 sec.

Αν και για μικρά power law, το cardinality είναι μεγάλο, το Apache Spark επιφέρει καλύτερα αποτελέσματα. Ο κύριος λόγος που συμβαίνει αυτό είναι η μορφολογία των δεδομένων και συγκεκριμένα το γεγονός ότι οι αριθμοί είναι συγκεντρωμένοι στο αριστερό άκρο (skew), σε αντίθετη με την ομοιόμορφη κατανομή.

4.3.2 Window 8 seconds

Για το Spark το throughput είχε τη μέγιστη τιμή των 1500 για power law 3 και όσο έπεφτε το power law μειωνόταν μέχρι που έφτασε τα 1100 στο τέλος. Για το Flink όμοια ξεκίνησε

από 2500 και στο τελευταίο αρχείο έπεσε στα 1950.



Σχήμα 4.5: Avg Event-Time Latency per Zipf file, Window 8 sec

Στο σχήμα 4.5 παρουσιάζεται το Event-Time Latency για κάθε power law/αρχείο. Όπως φαίνεται και από το σχήμα, το Apache Flink αποδίδει καλύτερα όταν το power law είναι μεγάλο και συγκεκριμένα για τιμές 1.7, 2 και 3. Καθώς το power law μειώνεται το Apache Spark επιφέρει καλύτερο latency σε σχέση με το Apache Flink. Για τις χαμηλότερες τιμές του power law η διαφορά στο latency είναι της τάξεως των 2-3 sec.

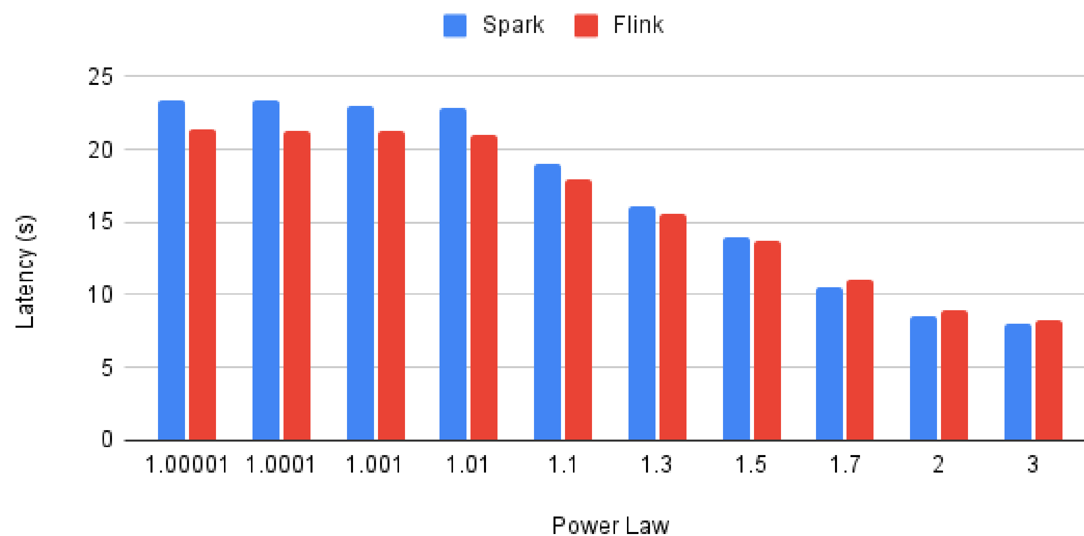
Αν και για μικρά power law, το cardinality είναι μεγάλο, το Apache Spark επιφέρει καλύτερα αποτελέσματα. Ο κύριος λόγος που συμβαίνει αυτό είναι η μορφολογία των δεδομένων και συγκεκριμένα το γεγονός ότι οι αριθμοί είναι συγκεντρωμένοι στο αριστερό άκρο (skew), σε αντίθετη με την ομοιόμορφη κατανομή.

Δεν παρατηρούμε μεγάλη μεταβολή στο latency για την περίπτωση του Apache Spark καθώς αυξήθηκε το window. Αντιθέτως, στο Apache Flink σημειώθηκε ανύψωση του latency.

4.3.3 Window 10 seconds

Για το Spark το throughput είχε τη μέγιστη τιμή των 1240 για power law 3 και όσο έπεφτε το power law μειωνόταν μέχρι που έφτασε τα 780 στο τέλος. Για το Flink όμοια ξεκίνησε από 2100 και στο τελευταίο αρχείο έπεσε στα 1670.

Στο σχήμα 4.6 παρουσιάζεται το Event-Time Latency για κάθε power law/αρχείο. Όπως φαίνεται και από το σχήμα, το Apache Spark αποδίδει καλύτερα όταν το power law είναι μεγάλο και συγκεκριμένα για τιμές 1.7, 2 και 3. Καθώς το power law μειώνεται το Apache Flink επιφέρει καλύτερο latency σε σχέση με το Apache Spark. Για τις χαμηλότερες τιμές του



Σχήμα 4.6: Avg Event-Time Latency per Zipf file, Window 10 sec

power law η διαφορά στο latency είναι της τάξεως των 2-3 sec.

Δεν παρατηρούμε μεγάλη μεταβολή στο latency για την περίπτωση του Apache Spark καθώς αυξήθηκε το window. Αντιθέτως, στο Apache Flink σημειώθηκε βελτίωση του latency.

Σε αντίθεση με τα προηγούμενα μεγέθη window, για 10 seconds το Spark δεν καταγράφει παρόμοια αποτελέσματα. Όπως παρατηρήθηκε και προηγουμένως, η μορφολογία των δεδομένων της zipf κατανομής, ευνοεί το Spark, αλλά καθώς αυξάνεται το window έχουμε την ίδια αντίδραση που είχαμε και για την ομοιόμορφη κατανομή. Ό λόγος που συμβαίνει αυτό στο Spark είναι λόγω της ιδιότητας να αποθηκεύει στην μνήμη cache. Ειδικότερα, το Spark αποθηκεύει στην μνήμη τα αποτελέσματα ενός window operation, με σκοπό να τα εκμεταλλευτεί αργότερα.

4.4 Συμπεράσματα

Παρακάτω παρουσιάζονται συγκεντρωτικά τα συμπεράσματα από το Benchmark που έγινε στα SDPS χρησιμοποιώντας aggregation query για δεδομένα εισόδου διακριτούς αριθμούς διαφορετικών κατανομών.

Ειδικότερα παρατηρήσαμε τα εξής:

- **Στην Uniform κατανομή το Spark αποδίδει χειρότερα από το Flink.**

  Το πλήθος των διαφορετικών αριθμών ενός dataset εξαρτάται άμεσα από το μέγεθος batch Interval που χρησιμοποιούμε για να διαβάσει το Spark τα δεδομένα. Μεγαλύτερο cardinality συνεπάγεται μικρότερο throughput, ώστε το σύστημα να αντεπεξέλθει στον

ρυθμό τροφοδοσίας των δεδομένων. Επίσης, ο λόγος που το Spark αποδίδει χειρότερα, οφείλεται στο γεγονός ότι βασίζεται στην αρχιτεκτονική των μικρών batch, δηλαδή σπάει το αρχικό query σε επιμέρους query, καθένα από τα οποία εκτελείται σε διαφορετική δουλειά. Για παράδειγμα, ένα βασικό operation είναι reduceByKey που χρησιμοποποιείται για να παραλληλοποιήσει τα στάδια ενός mini-batch μέσα σε ένα συγκεκριμένο window. Το Spark το μεταμορφώνει σε 2 επιμέρους RDDS, πρώτα το ShuffledRDD και στη συνέχεια στο MapPartitionsRDD. Σε αντίθεση, το Flink το αντιμετωπίζει σαν ένα βήμα χωρίς να το διασπά.

- **Στην zipf κατανομή το Spark αποδίδει καλύτερα από το Flink.**

  Ο βασικότερος λόγος είναι ο τρόπος που τα συστήματα υπολογίζουν το aggregation. Το Flink χρησιμοποιεί ένα slot ανά στιγμιότυπο του operator με αποτέλεσμα όταν τα δεδομένα είναι συγκεντωμένα στο ένα άκρο (zipf) να μην ολοκληρώνει τους υπολογισμούς γρήγορα. Το Spark επιβάλλει όλα τα partition να στείλουν τα αποτελέσματά τους μετά από ένα reduce operation σε ένα υπολογιστικό slot.

- **Η απόδοση του Spark μειώνεται όσο αυξάνεται το window.**

  Ό λόγος που συμβαίνει αυτό στο Spark είναι λόγω της ιδιότητας να αποθηκεύει στην μνήμη cache. Ειδικότερα, το Spark αποθηκεύει στην μνήμη τα αποτελέσματα ενός window operation, με σκοπό να τα εκμεταλλευτεί αργότερα. Η συμπεριφορά αυτή οδηγεί στην γρήγορη κατανάλωση της μνήμης. Σε αντίθεση, το Flink δεν μοιράζεται τα αποτελέσματα ενός aggregation query μεταξύ διαφορετικών window. Το Flink εκτελεί το aggregation αμέσως, χωρίς να περιμένει να κλείσει το παράθυρο.

Γενικότερα, παρατηρήσαμε πως το Apache Flink αποδίδει καλύτερα σε σχέση με το Apache Spark εμφανίζοντας καλύτερα αποτελέσματα στην πλειοψηφία των πειραμάτων.

Κεφάλαιο 5

Αλγόριθμος πρόβλεψης SDPS

Στο κάτωθι κεφάλαιο περιγράφουμε την μεθοδολογία ανάπτυξης ενός αλγορίθμου machine learning με χρήση του Random Forest αλγόριθμου, προκειμένου να μπορέσουμε να προβλέψουμε ποιο streaming data processing system είναι πιο αποδοτικό για να εκτελέσει το aggregation operation μας γραμμένο στο Apache Beam ανάλογα με την μορφολογία των δεδομένων.

5.1 Διατύπωση προβλήματος

Στο προηγούμενο κεφάλαιο μελετήσαμε την απόδοση του Apache Spark και του Apache Flink στην περίπτωση που το πρόγραμμά μας είναι ένα aggregation operation. Σαν δεδομένα εισόδου είχαμε διακριτούς αριθμούς που ακολουθούν διαφορετικές κατανομές (ομοιόμορφη και zipf).

Σενάριο: Ένας χρήστης έχει δημιουργήσει ένα πρόγραμμα, χρησιμοποιώντας την Java SDK του Apache Beam, που υπολογίζει την συχνότητα εμφάνισης των δεδομένων που λαμβάνονται σε πραγματικό χρόνο. Το πρόγραμμά του χαρακτηρίζεται από ένα συγκεκριμένο window. Ωστόσο, δεν γνωρίζει ποιο σύστημα είναι βέλτιστο για την περίπτωσή του. Γνωρίζει ότι τα δεδομένα εισόδου είναι διακριτοί αριθμοί και η κατανομή που ακολουθούν είναι είτε ομοιόμορφη είτε zipf.

Στόχος είναι η δημιουργία ενός predictor που θα μπορεί να προβλέπει το βέλτιστο σύστημα χρησιμοποιώντας ένα μικρό δείγμα των δεδομένων που λαμβάνονται σε πραγματικό χρόνο. Αφού η τελική πρόβλεψη θα είναι είτε Spark είτε Flink πρόκειται για πρόβλημα classification.

5.2 Predictor

Για την παραγωγή του predictor χρησιμοποιήσαμε τον αλγόριθμο Random Forest με την βοήθεια της βιβλιοθήκης sklearn της python.

Αρχικά, έπρεπε να συνδυάσουμε τα αποτελέσματα του benchmark για να δημιουργήσουμε τα features. Συνολικά δημιουργήσαμε 20 αρχεία διακριτών αριθμών και μελετήσαμε 3 διαφορετικές

τιμές window. Κάθε αρχείο το τροφοδοτήσαμε στο Apache Spark και στο Apache Flink. Το σύστημα που σημείωσε χαμηλότερο event-time latency απέδωσε καλύτερα. Επειδή, ουσιαστικά κάθε αρχείο το μελετήσαμε 3 φορές, ανάλογα με το μέγεθος του window, προέκυψαν συνολικά 60 περιπτώσεις.

Στη συνέχεια, συνδυάσαμε τις κατανομές με τα αποτελέσμα και δημιουργήσαμε ένα νέο αρχείο σε κατάλληλη μορφή ώστε να μπορέσουμε να εξάγουμε τα features. Η μορφή του αρχείου φαίνεται στον πίνακα 5.1.

Cardinality	Power Law	Distribution	Window	Framework
10	NaN	Uniform	5	Spark
1424	2	Zipf	5	Flink

Πίνακας 5.1: Μορφολογία συνδυαστικού αρχείου αποτελεσμάτων

Σαν features (inputs) για την εκπαίδευση του αλγόριθμου χρησιμοποιούμε και αριθμητικά και κατηγοροιοποιημένα δεδομένα. Για το λόγο αυτό τα αριθμητικά δεδομένα πρέπει να έχουν το ιδιο μέγεθος (magnitude) ενώ τα κατηγοριοποιημένα πρέπει να μετατραπούν σε αριθμητικά ώστε να γίνουν κατανοητά απο τον αλγόριθμο.

Παράμετροι που χρησιμοποιήθηκαν:

- **Number of features:** Ο αριθμός των attributes που χρησιμοποιεί ο αλγοριθμος.
- **Max_Depth:** Μέγιστο βάθος των δέντρων που μπορεί να φτάσει ο αλγόριθμος.
- **Min_samples_leaf:** Ο ελάχιστος αριθμός δειγμάτων που πρέπει να βρίσκονται σε έναν κόμβο φύλλων. Σημείο διαχωρισμού θεωρείται εκείνο που αφήνει Min_samples_leaf αριστερά και δεξιά κλαδιά του δέντρου.
- **Min_samples_split:** Ο ελάχιστος αριθμός δειγμάτων που απαιτούνται για τη διάσπαση ενός εσωτερικού κόμβου.
- **n_estimators:** Ο αριθμός των δέντρων που υπάρχουν μέσα στον αλγόριθμο random forest.
- **Bootstrap:** Εάν χρησιμοποιούμε τεχνική bootstrapping όταν δημιουργούμε τον αλγόριθμο. Βάζοντας False όλο το dataset λαμβάνεται υπόψη για την δημιουργία των δέντρων.

Έπειτα, χρησιμοποιήσαμε τη βιβλιοθήκη GridSearchCV. Η GridSearchCV είναι μια βοηθητική βιβλιοθήκη ώστε να βρούμε προκαθορισμένες υπερπαραμέτρους και να προσαρμόσουμε μοντέλο στο εκπαιδευτικό σετ. Επιπλέον, καθορίζει τον αριθμό των φορών για τη διασταυρούμενη επικύρωση (cross-validation) για κάθε σύνολο υπερπαραμέτρων.

Από το αρχικό dataset κρατήσαμε το 20% σαν test για να δοκιμάσουμε την απόδοση του αλγορίθμου. Το υπόλοιπο 80% αποτέλεσε το training dataset. Αφού τρέξαμε την

GridSearchCV πραγματοποιήσαμε πρόβλεψη στο δείγμα test και η απόδοση που προέκυψε είναι 91.4%.

Όπως ανφέρθηκε και στο παραπάνω σενάριο, τα δεδομένα λαμβάνονται σε πραγματικό χρόνο. Ένα ρεαλιστικό μοντέλο είναι μολις συμπληρωθούν 10,000 αριθμοί να πραγματοποιήσουμε στατιστικη ανάλυση σε αυτό το δείγμα, προκειμένου να προκύψει το είδος της κατανομής και οι παράμετροι που χρειάζεται ο predictor για να προβλέψει το βέλτιστο σύστημα.

Κεφάλαιο 6

Συμπεράσματα και μελλοντική δουλειά

Αυτή η εργασία περιγράφει έναν αλγόριθμο για την δυναμική δρομολόγηση κατανεμημένων ροών εργασιών. Αρχικά, περιγράφει τα χαρακτηριστικά δύο DSPSs, ειδικά του Apache Flink και του Apache Spark Streaming, καθώς και το επίπεδο αφαίρεσης Apache Beam για την εφαρμογή προγραμμάτων επεξεργασίας ροής.

Για να μελετήσουμε το αντίκτυπο της απόδοσης του Apache Beam, προτείνουμε μια ελαφριά αρχιτεκτονική που χρησιμοποιεί διακριτές κατανομές αριθμών και μια προσέγγιση για τη μέτρηση του χρόνου latency. Σε όλα τα σημεία αναφοράς παρέχονται λεπτομερής αναφορές προκειμένου να διασφαλιστεί η αναπαραγωγιμότητα.

Τα αποτελέσματα του benchmark δείχνουν πως το Apache Flink υπερισχύει στην πλειοψηφία των μετρήσεων. Ωστόσο, αξίζει να σημειωθεί πως η χρήση της τεχνολογίας Apache Beam, και συγκεκριμένα των SDKs που παρέχει, για τη δημιουργία ενός προγράμματος με τη δυνατότητα να εκτελεστεί σε διαφορερικά streaming frameworks επιφέρει παραπάνω latency κατα την εκτέλεση. Αν για παράδειγμα το ίδιο pipeline είχε γραφτεί απευθείας για το Apache Spark τα αποτελέσματα θα ήταν καλύτερα, καθώς θα μπορούσαμε να εισάγωγουμε βελτιστοποιήσεις μέσω του κώδικα. Το Apache Beam είναι ένα νέο project και η μετάφραση των operators στην αντίστοιχη μορφή για κάθε streaming framework δεν είναι ακόμα η βέλτιστη.

Επιπρόσθετα, αν και το μέγεθος του cluster που χρησιμοποιήσαμε είναι μικρό, τα αποτελέσματα αναδεικνύουν σε ικανοποιητικό βαθμό τις διαφορές των SDPSs, οι οποίες διαφαίνονται μελετώντας τις διαφορές στις αρχιτεκτονικές τους.

Ο αλγόριθμος του predictor παρουσιάζει ιδιαίτερο ενδιαφέρον, καθώς θα μπορούσε να χρησιμοποιηθεί σε ένα web service που θα διαβάζει τη ροή δεδομένων π.χ. από κάποιο kafka topic, θα προβλέπει για ένα δείγμα/υποσύνολο των δεδομένων το βέλτιστο σύστημα επεξεργασίας και θα ξεκινάει το αντίστοιχο framework με τις κατάλληλες παραμέτρους που θα αναλάβει να επεξεργαστεί τα streaming δεδομένα. Το γεγονός ότι θα ξεκινάει το αντίστοιχο framework προσθέτει μια καθυστέρηση, αλλά σε βάθος χρόνου η προσέγγιση

είναι αποδοτική, αφού θα έχει επιλεγεί το βέλτιστο σύστημα και το συνολικό latency θα είναι λιγότερο, σε σχέση με το να ξεκινούσε ο χρήστης το λάθος σύστημα, όπου το latency θα ήταν μεγαλύτερο.

Το συνολικό μέγεθος του dataset που χρησιμοποιήθηκε για την εκπαίδευση του classifier δεν είναι αρκετά μεγάλο για προσφέρει μεγάλο ποσοστό επιτυχίας, ειδικά σε νέα δεδομένα. Γιάυτό θα πρέπει να γίνουν περισσότερες συγκρίσεις και για άλλες τιμές του cardinality, του power law και window.

Η μελλοντική δουλειά περιλαμβάνει τα εξής:

- Ανάλυση του Apache Beam, ώστε να εντοπιστούν και έπειτα να αποφευχθούν οι λειτουργίες του κώδικα που δεν μεταφράζονται βέλτιστα στα streaming frameworks. Βέβαια, είναι πολύ πιθανό σε μελλοντικές εκδόσεις του Apache Beam να υπάρξουν βελτιστοποιήσεις στο κομμάτι αυτό.

- Η εκτέλεση του benchmark, μελλοντικά, θα πρέπει να γίνει και σε διαφορετικά μεγέθη cluster, προκειμένου να δούμε και πως αποδίδει ένα σύστημα στην περίπτωση που οι υπολογιστικοί πόροι διαφέρουν.

- Ιδιαίτερο ενδιαφέρον παρουσιάζει η μελέτη της απόδοσης των συστημάτων όταν έχουμε διαφορετικό operator, π.χ. join.

- Οι παραμετροποιήσεις που αναφέρθηκαν παραπάνω θα δημιουργήσουν περισσότερα features στον classifier. Ως εκ τούτου, θα αυξηθεί το μέγεθος του training dataset και ο predictor θα είναι πιο αποτελεσματικός.

Συνοψίζοντας, η εργασία που υλοποιήσαμε, αποσκοπούσε στο να δώσει μια προσέγγιση για το πως θα μπορούσε να γίνει δυναμική δρομολόγηση κατανεμημένων ροών εργασιών με χρήση τεχνικών μηχανικής μάθησης. Αυτό που πρέπει να κρατήσει κάποιος διαβάζοντας την παρούσα εργασία, είναι ότι η απόδοση των streaming frameworks επηρεάζεται σε μεγάλο βαθμό από το μέγεθος του cluster και από την τοπολογία και ενώ δεν υπάρχει εύκολη λύση που μπορεί να εφαρμοστεί σε προκειμένου να επιτύχουμε τα βέλτιστα δυνατά αποτελέσματα, υπάρχει τρόπος να αυξηθεί σημαντικά η απόδοση, επιλέγοντας ένα σύστημα με αρκετά μεγάλη υπολογιστική ισχύ, γεγονός που αυξάνει δραστικά τον παραλληλισμό των συστημάτων.

Παράρτημα Α΄

Αρκτικόλεξο

ΛΣ Λειτουργικό σύστημα
SDPSs Distributed Stream Data Processing Systems
KVPs Key-Value Pairs
RDD Resilient Distributed Dataset
GB Giga Byte
D-Streams Discretized Streams
SEC Second
OOB Out-of-Bag
CPU Central Proccesing Unit
API Application Programming Interface
KVM Kernel-based Virtual Machine
SUT System Under Test
CSlab Computer Systems Laboratory
NFS Network File System

Παράρτημα Β΄

Hadoop-Yarn Configuration Files

Για την εγκατάσταση του Hadoop-Yarn πάνω σε κόμβους που χρησιμοποιούν σύστημα αποθήκευσης NFS πρέπει να γίνει παραμετροποίηση των αρχείων XML ώστε να αποφευχθεί η ταυτόχρονη πρόσβαση στο ίδιο directory. Για αυτό το λόγο έχουν δημιουργηθεί 3 διαφορετικά directory, ένα για τον namenode και υπόλοιπα για τους datanodes. Όπως φαίνεται και από τα παρακάτων xml αλλάξαμε το tmp directories, διότι το tmp/ directory ήταν γεμάτο και το Hadoop δεν μπορούσε να δημιουργήσει προσωρινά αρχεία. Επειδή, στα μηχανήματα είχαμε απλά δικαιώματα πρόσβασης δεν ήταν εφικτό να διαγραφούν δεδομένα από το tmp/ χωρίς root δικαιώματα, οπότε καταλήξαμε στην παραπάνω λύση.

Ακολουθούν οι προσθήκες στα XML αρχεία που αφορούν τον Namenode και τον ResourceManager.

core-site.xml

```
1  <configuration>
2          <property>
3                  <name>fs.defaultFS</name>
4                  <value>hdfs://clone13:9000</value>
5          </property>
6
7          <property>
8                  <name>hadoop.tmp.dir</name>
9                  <value>/home/users/nkarav/hadoop_temp/hadoop-\${user.name}<value>
10         </property>
11 </configuration>
```

mapred-site.xml

```
1  <configuration>
2      <property>
3          <name>mapreduce.framework.name</name>
4          <value>yarn</value>
```

```
    </property>
</configuration>
```

yarn-site.xml

```
<configuration>

    <property>
        <name>yarn.resourcemanager.address</name>
        <value>clone13:8032</value>
    </property>

    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>clone13:8030</value>
    </property>

    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>clone13:8031</value>
    </property>

    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <property>
        <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>

    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>

    <property>
        <name>yarn.nodemanager.vmem-check-enabled</name>
        <value>false</value>
        <description>Whether virtual memory limits will be enforced for containers</description>
    </property>

    <property>
        <name>yarn.nodemanager.vmem-pmem-ratio</name>
        <value>4</value>
        <description>Ratio between virtual memory to physical memory when setting memory limits for containers
    </property>

</configuration>
```

hdfs-site.xml

```xml
<configuration>
    <property>
        <name>dfs.namenode.http-address</name>
        <value>clone13:50070</value>
    </property>

    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/home/users/nkarav/data/namenode</value>
    </property>

    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>

    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>clone13:50090</value>
    </property>

    <property>
        <name>dfs.namenode.secondary.https-address</name>
        <value>clone13:50091</value>
    </property>

    <property>
        <name>dfs.http.address</name>
        <value>clone13:50070</value>
        <description>Enter your Primary NameNode hostname for http access.</description>
    </property>
</configuration>
```

Ακολουθούν τα αντίστοιχα αρχεία για ένα εκ των δύο datanode-nodemanager. Η ίδια παραμετροποίηση έγινε και για το άλλο datanode-nodemanager με ελάχιστες διαφορές.

hdfs-site.xml

```xml
// Same as namenode except we add:

<property>
        <name>dfs.datanode.data.dir</name>
        <value>/home/users/nkarav/datanode1</value>
</property>
```

core-site.xml

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://clone13:9000</value>
    </property>

    <property>
        <name>hadoop.tmp.dir</name>
        <value>/home/users/nkarav/hadoop_temp_clone14/hadoop-\${user.name}</value>
    </property>

</configuration>
```

yarn-site.xml

```
// Same as namenode except we add:
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>clone13</value>
    </property>

    <property>
        <name>yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage</name>
        <value>99.5</value>
    </property>
```

Βιβλιογραφία

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter", in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 147–156.

[2] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters", in *4th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems", in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002, pp. 1–16.

[5] L. Golab and M. T. Özsu, "Issues in data stream management", *ACM Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[6] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud", in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 1–14.

[7] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time url spam filtering service", in *2011 IEEE symposium on security and privacy*, IEEE, 2011, pp. 447–462.

[8] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye, "Building linkedin's real-time activity data pipeline.", *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, 2012.

[9] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing", in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[10] *Apache beam overview*, https://beam.apache.org/get-started/beam-overview/.

[11] *Apache beam*, https://github.com/apache/beam.

[12] *Apache beam programming guide*,
https://beam.apache.org/documentation/programming-guide/.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing", 2015.

[14] *Beam capability matrix*,
https://beam.apache.org/documentation/runners/capability-matrix/.

[15] *What is samza?*,
https://samza.apache.org.

[16] *Ibm streams*,
https://www.ibm.com/cloud/streaming-analytics.

[17] *Mapreduce tutorial,*
https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/.

[18] *Cloud dataflow - simplified stream and batch data processing, with equal reliability and expressiveness,*
https://cloud.google.com/dataflow/.

[19] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: A unified engine for big data processing", *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[20] *Mspark streaming programming guide,*
https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.", in *NSDI*, vol. 11, 2011, pp. 22–22.

[22] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator", in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[23] E. A. Brewer, "Kubernetes and the path to cloud native", in *Proceedings of the sixth ACM symposium on cloud computing*, 2015, pp. 167–167.

[24] *Apache spark - cluster mode overview,*
https://spark.apache.org/docs/2.4.7/cluster-overview.html.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale", in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

[26] *Flink - distributed runtime environment,* https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html.

[27] G. Biau, “Analysis of a random forests model”, *The Journal of Machine Learning Research*, vol. 13, pp. 1063–1095, 2012.

[28] L. Breiman, “Random forests”, *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[29] A. Criminisi, J. Shotton, E. Konukoglu, *et al.*, “Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning”, *Foundations and trends® in computer graphics and vision*, vol. 7, no. 2–3, pp. 81–227, 2012.

[30] U. Nair, P. Sankaran, and N. Balakrishnan, *Reliability modelling and analysis in discrete time*. Academic Press, 2018.

[31] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.

[32] *Confluent kafka python api,* https://github.com/confluentinc/confluent-kafka-python.

[33] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “Millwheel: Fault-tolerant stream processing at internet scale”, *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.