Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Εργαστηριο Συστηματων Βασεων Γνωσεων και Δεδομενων

High-Performace Data Analytics with Ray

Diploma Thesis

by

Charidimos Georgiou

Επιβλέπων: Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2024

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Συστημάτων Βάσεων Γνώσεων και Δεδομένων

High-Performace Data Analytics with Ray

Diploma Thesis

by

Charidimos Georgiou

Επιβλέπων: Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιουλίου, 2024.

.....	.....	.....
Δημήτριος Τσουμάκος	Στάμου Γεώργιος	Γκούμας Γεώργιος
Αναπληρωτής Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2024

.....

Γεωργιου Χαριδημος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η αυξανόμενη ζήτηση για την ανάπτυξη αποτελεσματικών εργαλείων επεξεργασίας και ανάλυσης μπορεί να αποδοθεί στον σημαντικό ρόλο που έχουν αποκτήσει στις μέρες μας η ανάλυση μεγάλων όγκων δεδομένων και η μηχανική μάθηση σε διάφορους τομείς. Προκειμένου να γίνεται αποτελεσματικά η εκτέλεση πολύπλοκων υπολογισμών και η διαχείριση τεραστίων όγκων δεδομένων που οδηγούν στην εξαγωγή ουσιαστικών συμπερασμάτων, προέκυψε η ανάγκη για υπολογιστικά πλαίσια κατανεμημένων υπολογισμών υψηλών δυνατοτήτων. Μέσω της ανάπτυξης πλαισίων, όπως το Ray, και των δυνατοτήτων που αυτά παρέχουν, οι υπολογιστικές απαιτήσεις της ανάλυσης μεγάλων δεδομένων και των εργασιών μηχανικής μάθησης καθίστανται αντιμετωπίσιμες.

Το Ray, εμπλουτισμένο με APIs που παραλληλοποιούν κώδικα της Python και όχι μόνο, αποτελεί ένα ισχυρό εργαλείο όσον αφορά τον κατανεμημένο υπολογισμό για τους χρήστες της προγραμματιστικής αυτής γλώσσας. Σκοπός της εργασίας είναι η ανάλυση της απόδοσης του Ray σε διαφόρων ειδών εφαρμογές, εμβαθύνοντας σε λειτουργίες ETL, στην επεξεργασία γράφων και στην κατανεμημένη εκπαίδευση και εύρεση βέλτιστων υπερπαραμέτρων μοντέλων μηχανικής μάθησης. Ως μέτρο σύγκρισης χρησιμοποιήθηκε το Apache Spark, ένα από τα πιο διαδεδομένα πλαίσια υπολογισμών σε κατανεμημένα περιβάλλοντα σήμερα, αναγνωρισμένο για τις δυνατότητές του στην επεξεργασία δεδομένων και την εκτεταμένη υποστήριξη βιβλιοθηκών. Στα πειράματα ελήφθησαν υπόψη διάφορες παράμετροι, όπως ο χρόνος εκτέλεσης, ο συνολικός χρόνος CPU, καθώς και οι απαιτήσεις μνήμης των 2 πλαισίων για διάφορα μεγέθη συνόλων δεδομένων και διατάξεις του cluster που αξιοποιήθηκε. Σύμφωνα με τις παρατηρήσεις, το Spark υπερτερεί του Ray όσον αφορά τις λειτουργίες ETL και την ανάλυση γράφων, καθώς αποτελεί ένα ωριμότερο οικοσύστημα με ποικίλες βελτιστοποιήσεις και χρησιμοποιεί την μνήμη του συστήματος αποδοτικότερα. Ωστόσο, παρατηρήθηκε ότι το Ray υπερέχει του Spark στις εργασίες της μηχανικής μάθησης, γεγονός που αναδεικνύει τις σημαντικές δυνατότητες παράλληλης επεξεργασίας του.

Λέξεις-κλειδιά: Κατανεμημένοι υπολογισμοί, Ray, Apache Spark, Ανάλυση Δεδομένων, Ανάλυση Γράφων, Μηχανική Μάθηση.

Abstract

The increasing demand for the development of efficient processing and analyzing tools, can be attributed to the significant role that Big Data analysis and Machine Learning have acquired in our days across various fields. In order for complex computations to be effectively performed, and vast amounts of data to be handled adequately and lead to the extraction of meaningful insights, the need for distributed computing frameworks with such capabilities has emerged. Through the development frameworks, such as Ray, the computational demands of Big Data analytics and ML tasks are addressed, due to a certain set of capabilities they provide.

Ray is enriched with APIs that parallelize Python code, and can therefore be characterized as a powerful tool regarding distributed computing. The objective of this thesis is to analyze Ray's performance on various applications, delving into ETL operations, graph processing, distributed ML training, and hyperparameter tuning. Apache Spark is another similar framework. Being widely used today and recognized for its powerful data-processing properties and extensive library support, Spark's performance is used as a point of reference in this work. The experiment was carried out on a cluster setup, taking into consideration various parameters including the time of execution, CPU time, as well as memory usage throughout different data sizes and node configurations. According to the results, it was demonstrated that Spark is superior to Ray regarding ETL and graph operations since it comprises a more mature ecosystem and exhibits efficient memory usage. It was nevertheless observed that Ray was outperforming Spark as far as ML training and hyperparameter tuning were involved, which showcases its significant parallel processing capabilities.

Keywords: Distributed Computing, Ray, Apache Spark, Data Analytics, Graph Analytics, Machine Learning.

Ευχαριστίες

Η παρούσα εργασία πραγματοποιήθηκε στο πλαίσιο του προπτυχιακού κύκλου σπουδών της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών το ακαδημαϊκό έτος 2023-24. Αποτελεί το τελευταίο βήμα για την απόκτηση του διπλώματός μου και θα ήθελα να εκφράσω την ειλικρινή μου ευγνωμοσύνη σε όλους εκείνους που συνέβαλαν στην πορεία μου μέχρι εδώ.

Πρώτα από όλα, θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή της εργασίας μου, κ. Δημήτριο Τσουμάκο, για την πολύτιμη καθοδήγηση και υποστήριξή του.

Ένα μεγάλο ευχαριστώ οφείλω σε όλους τους φίλους και συμφοιτητές μου για την ψυχική στήριξη και την ενθάρρυνση που μου παρείχαν καθ' όλη τη διάρκεια αυτής της προσπάθειας. Η αγάπη και η υποστήριξή τους ήταν ανεκτίμητες για μένα.

Το μεγαλύτερο «ευχαριστώ» στην οικογένειά μου και στην κοπέλα μου, που με στήριξαν όλα τα χρόνια και μου συμπαραστάθηκαν με κάθε δυνατό τρόπο σε όλες τις δυσκολίες συμβάλλοντας καθοριστικά στην επιτυχία μου.

Γεωργίου Χαρίδημος, Ιούλιος 2024

Contents

List of Figures

List of Tables

Chapter 1

Εκτεταμένη Περίληψη στα Ελληνικά

1.1 Εισαγωγή

Στην εποχή των big data, παράγονται τεράστιοι όγκοι δεδομένων με πρωτοφανείς ταχύτητες. Τα δεδομένα αυτά είναι πολύτιμα για τη βελτίωση των λειτουργιών, την προώθηση της καινοτομίας και την ενημέρωση για τη λήψη αποφάσεων, αλλά παρουσιάζουν προκλήσεις όσον αφορά την αποθήκευση, την επεξεργασία και την ανάλυση. Για την αντιμετώπιση αυτών των προκλήσεων έχουν αναπτυχθεί διάφορα πλαίσια μεγάλων δεδομένων, το καθένα με μοναδικά χαρακτηριστικά και επιδόσεις.

Το Ray και το Apache Spark είναι δύο γνωστά πλαίσια για κατανεμημένη επεξεργασία δεδομένων. Το Ray, που έχει σχεδιαστεί για να καλύπτει τις ανάγκες των εφαρμογών τεχνητής νοημοσύνης, προσφέρει ένα ευέλικτο πλαίσιο για κατανεμημένη επεξεργασία, υποστηρίζοντας ποικίλες εφαρμογές σχετικές με την ανάλυση δεδομένων, την τεχνητή νοημοσύνη και την μηχανική μάθηση. Το Apache Spark είναι γνωστό για την ταχύτητα, τη φιλικότητα προς το χρήστη και την εκτεταμένη υποστήριξη βιβλιοθηκών για εργασίες ανάλυσης δεδομένων, μηχανικής μάθησης, batch processing και stream processing.

Η παρούσα διπλωματική αποσκοπεί στην παροχή μιας ολοκληρωμένης σύγκρισης αυτών των 2 τεχνολογιών, εστιάζοντας στις επιδόσεις τους σε αναλυτικές λειτουργίες, ανάλυση γράφων και εργασίες μηχανικής μάθησης. Η μελέτη είναι σημαντική για τον προσδιορισμό του καταλληλότερου μεταξύ των Ray και Spark με βάση τις εκάστοτε απαιτήσεις ενός έργου.

Η έρευνα συνδυάζει αφενός μια θεωρητική ανάλυση και αφετέρου την εμπειρική αξιολόγηση, που αποτελεί τη βάση για το πλαίσιο σύγκρισης. Η εμπειρική αξιολόγηση περιλαμβάνει πειράματα για τη μέτρηση των επιδόσεων των Ray και Spark σε διάφορες εργασίες επεξεργασίας δεδομένων, σχεδιασμένες έτσι ώστε να αντικατοπτρίζουν σενάρια του πραγματικού κόσμου, με μετρικές όπως ο χρόνος εκτέλεσης, η κατανάλωση υπολογιστικών πόρων, η επεκτασιμότητα και η ανοχή σε σφάλματα, με σκοπό τον εντοπισμό των δυνατών και αδύνατων σημείων κάθε πλαισίου.

1.2 Επισκόπηση του Ray

Το Ray είναι ένα πλαίσιο ανοικτού κώδικα που αναπτύχθηκε από το RISELab του UC Berkeley για κλιμακούμενους και κατανεμημένους υπολογισμούς. Διαθέτει ένα φιλικό προς το χρήστη API που υποστηρίζει διάφορες εφαρμογές όπως είναι η παράλληλη επεξεργασία δεδομένων, η κατανεμημένη εκπαίδευση μοντέλων και η ρύθμιση των υπερπαραμέτρων τους, καθιστώντας το ευέλικτο για εργασίες μεγάλων δεδομένων και μηχανικής μάθησης. Η οριζόντια επεκτασιμότητα του Ray σε έναν cluster το καθιστά κατάλληλο για εργασίες υπολογισμού υψηλών επιδόσεων, δίνοντας έμφαση στην απλότητα και την ευελιξία [15].

Η αρχιτεκτονική του Ray αποτελείται από δύο κύρια επίπεδα: το επίπεδο εφαρμογής και το επίπεδο συστήματος [17].

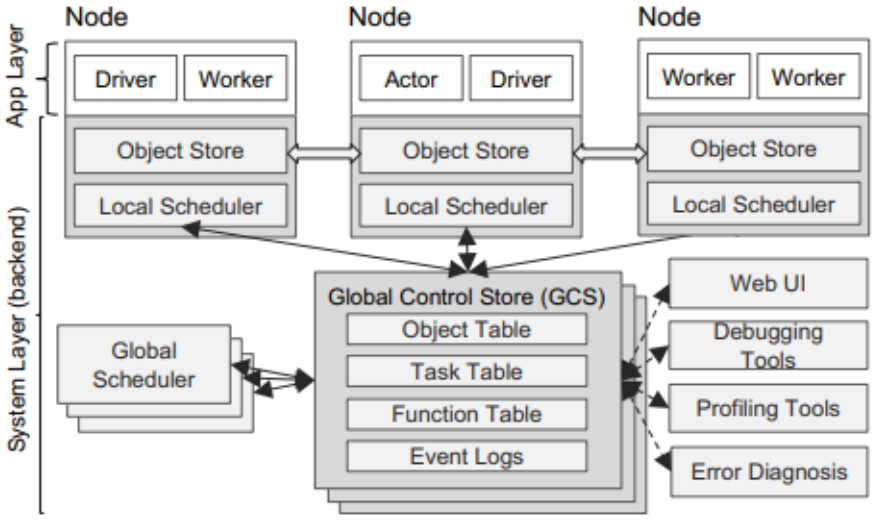

Figure 1.2.1: Αρχιτεκτονική του Ray [17]

Το επίπεδο εφαρμογής περιλαμβάνει:

- **Drivers**: Προγράμματα που εκτελούν τις κύριες συναρτήσεις και προγραμματίζουν τα tasks και τους actors.
- **Tasks**: Συναρτήσεις που δεν διατηρούν την κατάστασή τους από την τελευταία κλήση τους (stateless) και οι οποίες μπορούν να εκτελεστούν παράλληλα και ασύγχρονα.
- **Actors**: Οντότητες που διατηρούν την κατάστασή τους μεταξύ πολλαπλών κλήσεών τους (stateful).

Το επίπεδο συστήματος περιλαμβάνει τα:

- **Global Control Store (GCS)**: Ένας κατανεμημένος αποθηκευτικός χώρος κλειδιών-τιμών που διαχειρίζεται την κατάσταση ελέγχου του συστήματος. Το GCS εξασφαλίζει επεκτασιμότητα και ανοχή σε σφάλματα και επιτρέπει τη δυναμική διαχείριση εκατομμυρίων εργασιών ανά δευτερόλεπτο.
- **Bottom-Up Distributed Scheduler**: Ένα ιεραρχικό μοντέλο δύο επιπέδων με τοπικούς χρονοπρογραμματιστές σε κάθε κόμβο και έναν κεντρικό χρονοπρογραμματιστή. Οι τοπικοί χρονοπρογραμματιστές διαχειρίζονται την εκτέλεση εργασιών που γεννιούνται στον εκάστοτε κόμβο εκτός αν αυτός δεν έχει διαθέσιμους υπολογιστικούς πόρους, οπότε αναλαμβάνει την κατανομή των εργασιών ο κεντρικός χρονοπρογραμματιστής.
- **In-Memory Distributed Object Store**: Χρησιμοποιεί κοινή μνήμη για να επιτύχει πολιτική μηδενικής αντιγραφής διαμοιρασμένων δεδομένων, μειώνοντας την καθυστέρηση των εργασιών και αυξάνοντας την απόδοση. Σε περίπτωση αποτυχίας ενός κόμβου, το Ray ανακατασκευάζει αντικείμενα χρησιμοποιώντας πληροφορίες από το GCS.

Το Ray διαθέτει ένα ισχυρό οικοσύστημα βιβλιοθηκών. Αυτές είναι οι:

- **Ray Data**: Διαχειρίζεται σύνθετες εργασίες επεξεργασίας δεδομένων.
- **Ray Train**: Απλοποιεί την κατανεμημένη εκπαίδευση μοντέλων μηχανικής μάθησης, υποστηρίζοντας πλαίσια όπως το PyTorch και το TensorFlow.
- **Ray Tune**: Παρέχει εργαλεία για την εύρεση των βέλτιστων υπερπαραμέτρων για μοντέλα μηχανικής μάθησης. Υποστηρίζει πολλαπλές τεχνικές αναζήτησης.
- **Ray RLlib**: Ένα επεκτάσιμο πακέτο ενισχυτικής μάθησης που υποστηρίζει διάφορους αλγορίθμους και περιβάλλοντα προσομοίωσης [14].

- **Ray Serve**: Διευκολύνει την ανάπτυξη μοντέλων μηχανικής μάθησης σε περιβάλλοντα πραγματικού κόσμου με χαρακτηριστικά όπως η έκδοση μοντέλων και η αυτόματη κλιμάκωση [3].

Κάθε βιβλιοθήκη του οικοσυστήματος Ray ενσωματώνεται απρόσκοπτα με το Ray Core, το κύριο API του Ray, δημιουργώντας ένα ισχυρό πλαίσιο για την κατασκευή σύνθετων κατανεμημένων εφαρμογών, εξασφαλίζοντας ευελιξία και επεκτασιμότητα για ποικίλες περιπτώσεις χρήσης.

1.3 Επισκόπηση του Apache Spark

Το Apache Spark είναι ένα πλαίσιο ανοικτού κώδικα για την ανάλυση τεράστιων όγκων δεδομένων. Το Spark αναπτύχθηκε αρχικά στο AMPLab του UC Berkeley το 2009 και προσφέρει, όπως και το Ray μια διεπαφή για τον παράλληλο προγραμματισμό σε clusters με ανοχή σε σφάλματα [27].

Το Apache Spark χρησιμοποιεί μια αρχιτεκτονική master-slave που αποτελείται από executors σε κόμβους εργασίας και έναν driver που λειτουργεί ως κύριος κόμβος.

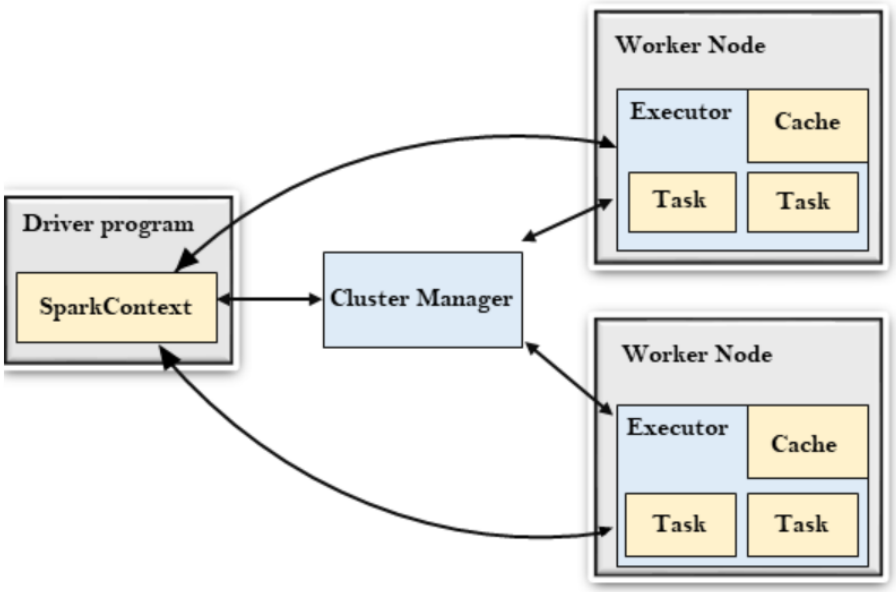

Figure 1.3.1: Αρχιτεκτονική του Spark [13]

- **Cluster Manager**: Ο Cluster Manager κατανέμει πόρους και διαχειρίζεται τον cluster. Το Spark υποστηρίζει διάφορους cluster managers, όπως ο ενσωματωμένος Standalone Cluster Manager και το Apache Hadoop YARN.
- **Spark Driver**: Ο driver διαχειρίζεται την εκτέλεση της εφαρμογής, εκτελεί την κύρια συνάρτηση και δημιουργεί το SparkContext, συνδεόμενος με τον Cluster Manager. Περιλαμβάνει μικροτερα στοιχεία στοιχεία, όπως ο Block Manager, ο DAG Scheduler, ο Task Scheduler και ο Backend Scheduler.
- **Spark Executors**: Οι executors εκτελούν εργασίες, αλληλεπιδρούν με τον Cluster Manager και τον Driver και αποθηκεύουν δεδομένα στη μνήμη ή στο δίσκο για μετέπειτα χρήση.
- **SparkContext**: Το SparkContext είναι το σημείο εκκίνησης όλων των λειτουργιών του Spark, χρησιμοποιείται για τη δημιουργία μεταβλητών εκπομπής broadcast variables, συσσωρευτών και RDDs και συντονίζει την εκτέλεση εργασιών.
- **Task**: Τα tasks είναι οι μικρότερες μονάδες εργασίας, που αντιπροσωπεύουν υπολογισμούς σε μεμονωμένα τμήματα δεδομένων. Μια εργασία Spark χωρίζεται σε tasks από τον driver, ο οποίος τις αναθέτει σε executors.

Όπως το Ray, έτσι και το Spark διαθέτει ένα ισχυρό οικοσύστημα βιβλιοθηκών, μερικές από τις οποίες είναι:

- **Spark SQL**: Για την επεξεργασία δομημένων δεδομένων και την υποβολή ερωτημάτων.
- **Spark Streaming**: Για επεξεργασία ροής δεδομένων σε πραγματικό χρόνο.
- **MLlib**: Μια κλιμακούμενη βιβλιοθήκη μηχανικής μάθησης.
- **GraphX**: Για παράλληλους υπολογισμούς σε γράφους.
- **SparkR**: Ενσωματώνει τις δυνατότητες του Spark στο περιβάλλον της R.

1.4 Αλγόριθμοι γράφων

1.4.1 Pagerank

Ο PageRank είναι ένας αλγόριθμος που αναπτύχθηκε αρχικά από την Google για την κατάταξη των ιστοσελίδων στα αποτελέσματα των μηχανών αναζήτησης. Αξιολογεί τη σημασία κάθε κόμβου σε έναν γράφο με βάση τη δομή των εισερχόμενων συνδέσμων, με το PageRank ενός κόμβου να ορίζεται αναδρομικά από το PageRank όλων των κόμβων που συνδέονται με αυτόν. Ο αλγόριθμος αυτός χρησιμοποιείται ευρέως στην κατάταξη ιστοσελίδων, στην ανάλυση κοινωνικών δικτύων και σε συστήματα συστάσεων.

Μαθηματική Διατύπωση

Ο PageRank $PR(i)$ ενός κόμβου i δίνεται από τη σχέση:

$$PR(i) = \frac{1-d}{N} + d \sum_{j \in M(i)} \frac{PR(j)}{L(j)}$$

Όπου:

- $PR(i)$ είναι ο PageRank του κόμβου i.
- d είναι ο παράγοντας απόσβεσης, συνήθως ορίζεται σε 0.85.
- N είναι το συνολικό πλήθος των κόμβων.
- $M(i)$ είναι το σύνολο των κόμβων-γειτόνων του i.
- $L(j)$ είναι ο αριθμός των εξερχόμενων συνδέσμων από τον κόμβο j.

Αλγόριθμος Επαναληπτικού Υπολογισμού

Ο αλγόριθμος για τον επαναληπτικό υπολογισμό του PageRank περιλαμβάνει τα εξής βήματα:

1. **Αρχικοποίηση**: Εκχώρηση αρχικής τιμής PageRank $\frac{1}{N}$ σε κάθε κόμβο.
2. **Επανάληψη**: Ενημέρωση των τιμών PageRank βάσει των τιμών της προηγούμενης επανάληψης χρησιμοποιώντας τη σχέση:

$$PR(i)^{(k+1)} = \frac{1-d}{N} + d \sum_{j \in M(i)} \frac{PR(j)^{(k)}}{L(j)}$$

3. **Σύγκλιση**: Επανάληψη των βημάτων μέχρι οι τιμές PageRank να συγκλίνουν, δηλαδή η αλλαγή στις τιμές PageRank μεταξύ των επαναλήψεων να είναι κάτω από ένα προκαθορισμένο όριο.

1.4.2 Καταμέτρηση τριγώνων σε γράφο

Ένα τρίγωνο σε έναν γράφο είναι ένα σύνολο από τρεις κόμβους που είναι αμοιβαία συνδεδεμένοι. Η καταμέτρηση των τριγώνων σε έναν γράφο βοηθά στην κατανόηση της δομής και της δυναμικής του δικτύου. Χρησιμοποιείται για τη μέτρηση του συντελεστή συμπλέγματος, ο οποίος δείχνει την πιθανότητα δύο κόμβοι που συνδέονται με έναν κοινό κόμβο να είναι επίσης συνδεδεμένοι μεταξύ τους. Υψηλοί συντελεστές συμπλέγματος συχνά υποδηλώνουν την παρουσία στενά συνδεδεμένων κοινοτήτων.

Μαθηματική Διατύπωση

Ο αριθμός των τριγώνων $T(i)$ στα οποία συμμετέχει ένας κόμβος i μπορεί να υπολογιστεί ως:

$$T(i) = \frac{1}{2} \sum_{j,k \in N(i)} \delta(i,j,k)$$

όπου $\delta(i,j,k)$ είναι μια συνάρτηση ένδειξης που είναι 1 αν υπάρχει ακμή μεταξύ των κόμβων i, j και k, και 0 διαφορετικά.

Αλγόριθμος Καταμέτρησης Τριγώνων

Ο αλγόριθμος για την καταμέτρηση τριγώνων μπορεί να περιγραφεί ως εξής:

1. Για κάθε κόμβο i, βρες όλα τα ζεύγη γειτόνων (j,k).
2. Για κάθε ζεύγος (j,k), έλεγξε αν υπάρχει ακμή μεταξύ των j και k.
3. Μέτρησε κάθε τέτοια τριάδα (i,j,k) ως τρίγωνο.

1.5 Μηχανική Μάθηση

1.5.1 Perceptron Πολλών Επιπέδων - Multilayer Perceptron (MLP)

Νευρώνας (Perceptron)

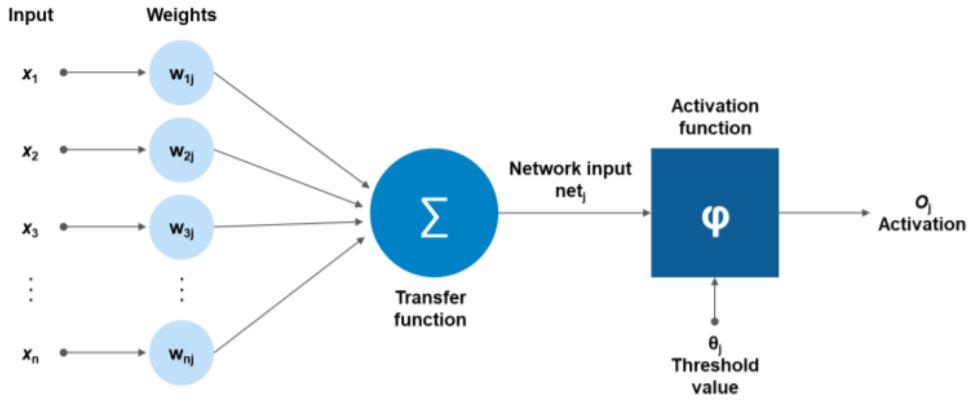

Figure 1.5.1: Ένας τεχνητός νευρώνας

Ο τεχνητός νευρώνας είναι η βασική υπολογιστική μονάδα σε ένα νευρωνικό δίκτυο. Αποτελείται από συνάψεις με αντίστοιχα βάρη, έναν αθροιστή σημάτων εισόδου, μια συνάρτηση ενεργοποίησης, και μια εξωτερική πόλωση (bias). Η συνάρτηση του νευρώνα δίνεται από τον τύπο:

$$y = \phi \left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j \right)$$

όπου w_{ij} είναι τα βάρη, x_i οι είσοδοι, b_j η πόλωση και ϕ η συνάρτηση ενεργοποίησης. Η συνάρτηση ενεργοποίησης μπορεί να είναι γραμμική, ReLU, ELU, ή sigmoid, ανάλογα με την εφαρμογή.

Perceptron Πολλών Επιπέδων

Τα πολυεπίπεδα δίκτυα Perceptron (MLP) αποτελούνται από πολλαπλά στρώματα υπολογιστικών μονάδων, συνήθως διασυνδεδεμένα με τρόπο πρόσθιας τροφοδότησης (Feedforward Networks). Κάθε νευρώνας σε ένα επίπεδο συνδέεται με όλους τους νευρώνες του επόμενου στρώματος.

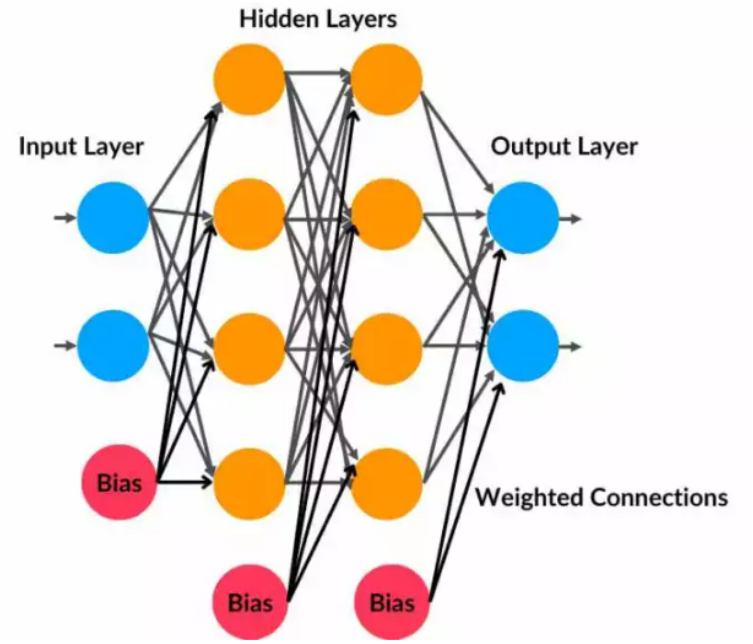

Figure 1.5.2: Multilayer Perceptron

Οπίσθια Διάδοση Σφάλματος - Backpropagation

Τα δίκτυα πολλαπλών επιπέδων χρησιμοποιούν μια ποικιλία τεχνικών μάθησης. Η πιο δημοφιλής είναι η οπίσθια διάδοση. Εδώ, οι τιμές εξόδου συγκρίνονται με τη σωστή απάντηση για να υπολογιστεί η τιμή κάποιας προκαθορισμένης συνάρτησης κόστους (Cost Function). Με διάφορες τεχνικές, το σφάλμα διαδίδεται στην αντίθετη κατεύθυνση, προς την είσοδο. Χρησιμοποιώντας αυτές τις πληροφορίες, ο αλγόριθμος προσαρμόζει τα βάρη κάθε σύνδεσης για να μειώσει την τιμή της συνάρτησης κόστους κατά κάποιο μικρό μέγεθος (ρυθμός μάθησης - learning rate). Μετά την επανάληψη αυτής της διαδικασίας, το δίκτυο συνήθως θα συγκλίνει σε κάποια κατάσταση όπου το σφάλμα των υπολογισμών είναι μικρό. Σε αυτή την περίπτωση, θα λέγαμε ότι το δίκτυο έχει εκπαιδευτεί σε έναν στόχο.

Κατάβαση Κλίσης - Gradient Descent

Για τη σωστή ρύθμιση των βαρών, εφαρμόζεται μια γενική μέθοδος για τη μη γραμμική βελτιστοποίηση που ονομάζεται κατάβαση κλίσης (gradient descent). Για το σκοπό αυτό, το δίκτυο υπολογίζει το παράγωγο της συνάρτησης σφάλματος σε σχέση με τα βάρη του δικτύου και αλλάζει τα βάρη έτσι ώστε το σφάλμα να μειώνεται (συνεπώς να κατεβαίνει προς τα κάτω στην επιφάνεια της συνάρτησης σφάλματος). Η κατάβαση γίνεται με συγκεκριμένο βήμα. Για το λόγο αυτό, η οπίσθια διάδοση μπορεί να εφαρμοστεί μόνο σε δίκτυα με διαφοροποιήσιμες λειτουργίες ενεργοποίησης. Το πιο σύνηθες πρόβλημα που εμφανίζεται με την εκμάθηση μέσω του αλγορίθμου Gradient Descent είναι ότι ο αλγόριθμος μπορεί να παγιδευτεί στα τοπικά ελάχιστα στις μη-κυρτές καμπύλες και δεν βρίσκει την βέλτιστη λύση (ολικό ελάχιστο).

1.5.2 XGBoost

Το XGBoost (Extreme Gradient Boosting) είναι ένας αλγόριθμος μηχανικής μάθησης που χρησιμοποιείται για εποπτευόμενες εργασίες μάθησης, όπως η ταξινόμηση και η παλινδρόμηση. Πρόκειται για μια υλοποίηση των δέντρων απόφασης ενισχυμένης κλίσης, σχεδιασμένη για βέλτιστη ταχύτητα και απόδοση, καθώς και για παραλληλοποίηση [2].

Βασικές Έννοιες Το XGBoost βασίζεται σε τρεις κύριες έννοιες:

- **Ενίσχυση (Boosting)**: Μια τεχνική που βελτιώνει την απόδοση συνδυάζοντας τις προβλέψεις πολλών βασικών μοντέλων. Σε κάθε βήμα, το νέο μοντέλο επικεντρώνεται στη διόρθωση των σφαλμάτων των προηγούμενων μοντέλων.
- **Δέντρα Απόφασης (Decision Trees)**: Χρησιμοποιούνται ως τα βασικά μοντέλα μάθησης, κατασκευάζονται για να ελαχιστοποιούν μια συνάρτηση απώλειας με απληστία.
- **Κατάβαση Κλίσης (Gradient Descent)**: Όπως και τα MLP.

Χαρακτηριστικά

Το XGBoost διαθέτει πολλές δυνατότητες που ενισχύουν την απόδοση και την χρηστικότητά του:

- **Παράλληλη Επεξεργασία (Parallel Processing)**: Εκτελείται σε παράλληλα νήματα, γεγονός που το καθιστά πολύ αποδοτικό και επεκτάσιμο.
- **Διαχείριση Απουσιών Τιμών (Handling Missing Values)**: Μπορεί να διαχειριστεί αυτόματα τις ελλιπείς τιμές κατά την εκπαίδευση, βελτιστοποιώντας τον τρόπο χειρισμού τους.
- **Κλάδεμα Δέντρων (Tree Pruning)**: Χρησιμοποιεί παραμέτρους για τον περιορισμό του βάθους των δέντρων και τεχνικές κλαδέματος για την απομάκρυνση κόμβων που δεν βελτιώνουν την απόδοση.
- **Ευαισθησία στην Αραιότητα (Sparsity Awareness)**: Είναι σχεδιασμένο για να διαχειρίζεται αποτελεσματικά αραιά δεδομένα.

1.5.3 Βελτιστοποίηση Υπερπαραμέτρων

Η βελτιστοποίηση υπερπαραμέτρων (hyperparameter tuning) είναι κρίσιμη στη μηχανική μάθηση, καθώς αφορά την εύρεση των καλύτερων υπερπαραμέτρων για ένα μοντέλο. Οι υπερπαράμετροι είναι ρυθμίσεις που διαμορφώνουν το μοντέλο και τη διαδικασία εκπαίδευσης και πρέπει να οριστούν πριν την έναρξη της μάθησης [7].

Σημασία

Η σωστή ρύθμιση υπερπαραμέτρων μπορεί να:

- Βελτιώσει την απόδοση του μοντέλου.
- Αποφύγει την υπερεκπαίδευση ή υποεκπαίδευση.
- Εξοικονομήσει υπολογιστικούς πόρους.

Μέθοδοι Βελτιστοποίησης

Κοινές μέθοδοι περιλαμβάνουν:

- **Αναζήτηση πλέγματος (Grid Search)**: Εξαντλητική αναζήτηση σε καθορισμένο υποσύνολο υπερπαραμέτρων.
- **Τυχαία αναζήτηση (Random Search)**: Τυχαία δειγματοληψία υπερπαραμέτρων από καθορισμένη κατανομή.
- **Μπεϋζιανή βελτιστοποίηση (Bayesian Optimization)**: Προβλέπει την απόδοση υπερπαραμέτρων χρησιμοποιώντας πιθανοτικό μοντέλο.

- **Hyperband και ASHAScheduler**: Συνδυασμός τυχαίας αναζήτησης και πρώιμης διακοπής με σκοπό την παροχή περισσότερων πόρων στους πιο υποσχόμενους συνδυασμούς υπερπαραμέτρων.

Κύριες Υπερπαράμετροι

- **Νευρωνικά Δίκτυα**: Ρυθμός μάθησης, μέγεθος παρτίδας, αριθμός εποχών, αριθμός επιπέδων.
- **Μοντέλα Βασισμένα σε Δέντρα**: Αριθμός δέντρων, μέγιστο βάθος, ελάχιστα δείγματα ανά φύλλο, ρυθμός μάθησης.

Προκλήσεις

Η βελτιστοποίηση υπερπαραμέτρων αντιμετωπίζει προκλήσεις όπως:

- Υψηλό υπολογιστικό κόστος.
- Μεγάλο μέγεθος χώρου αναζήτησης.
- Αλληλεξαρτήσεις υπερπαραμέτρων.
- Κίνδυνος υπερεκπαίδευσης στο σετ επικύρωσης.

1.6 Σύνοψη Πειραμάτων

1.6.1 Ρύθμιση Συστήματος και Πειραματική Διαδικασία

Σύστημα

Τα πειράματα πραγματοποιήθηκαν σε σύστημα 3 Εικονικών Μηχανών (EM) με τις ακόλουθες προδιαγραφές: Λειτουργικό σύστημα Ubuntu Server LTS 16.04, 4 CPUs, 8GB RAM και 30GB χωρητικότητα στον δίσκο.

Για την αποθήκευση των συνόλων δεδομένων που χρησιμοποιήθηκαν για τα πειράματα εγκαταστήθηκε στον cluster των 3 EM ένα σύστημα HDFS, με σκοπό την παροχή κλιμακούμενης και ανθεκτικής αποθήκευσης.

Τα Ray και Apache Spark εγκαταστάθηκαν ανεξάρτητα στον ίδιο cluster, με τη διαχείριση και την κατανομή πόρων του Spark να γίνεται από το YARN.

Σύνολα δεδομένων και πειραματική διαδικασία

Για πειράματα ETL και ML χρησιμοποιήθηκαν αυτοσχέδια σύνολα δεδομένων αποθηκευμένα σε αρχεία CSV συνολικού μεγέθους 8GB. Η μορφή των συνόλων δεδομένων ήταν η ακόλουθη:

float1	**float2**	**float3**	**int1**	**int2**	**label**
float64	float64	float64	int	int	{0, 1}

Για τα πειράματα γραφημάτων χρησιμοποίησαν πραγματικοί γράφοι από το Stanford Network Analysis Project (SNAP) [23].

Κάθε πείραμα πραγματοποιήθηκε τρεις φορές και παρουσιάζονται οι μέσες τιμές αυτών των εκτελέσεων. Τα προγράμματα Ray και Spark εκτελέστηκαν στο ίδιο περιβάλλον χρόνου εκτέλεσης για να εξασφαλιστούν σταθερές συνθήκες. Οι μετρικές που εξετάστηκαν ήταν ο χρόνος εκτέλεσης (ET), ο συνολικός χρόνος CPU (CPUT) και η μέγιστη τιμή που έλαβε το μέγιστο μέγεθος της μνήμης σωρού που χρησιμοποιήθηκε ανά εκτέλεση (max PHM).

1.6.2 Πειράματα ETL

Σε αυτό το στάδιο εκτελέστηκαν πειράματα που αφορούν βασικές λειτουργίες ETL.

Map

Σε αυτό το πείραμα εξετάστηκε η λειτουργία `map`. Συγκεκριμένα, μέσω της `map` εφαρμόστηκε ο παρακάτω μετασχηματισμός στα σύνολα δεδομένων:

$$\text{float1} = \text{float1} \times \text{int1} + \text{float2}^2$$

Filter

Σε αυτό το πείραμα εξετάστηκε η λειτουργία `filter`. Συγκεκριμένα, μέσω της `filter` αφαιρέθηκαν από τα σύνολα δεδομένων οι σειρές στις οποίες η τιμή του πεδίου `int1` ήταν μικρότερες του 500.

Group by και Sum

Σε αυτό το πείραμα εξετάστηκε η δυνατότητα των 2 frameworks να ομαδοποιούν δεδομένα και να αθροίζουν επιμέρους πεδία. Συγκεκριμένα, τα δεδομένα ομαδοποιήθηκαν με βάση τη στήλη `int1` και στη συνέχεια βρέθηκε το άθροισμα των τιμών της στήλης `float1` ανά ομάδα.

Ενέργεια Sort

Σε αυτό το πείραμα τα δεδομένα ταξινομήθηκαν σε αύξουσα σειρά σύμφωνα με τη στήλη `int1`.

Αποτελέσματα: Το γενικό συμπέρασμα που προέκυψε είναι ότι το Spark παρέχει γενικά υψηλότερες επιδόσεις και επιτυγχάνει μεγαλύτερη επεκτασιμότητα στις περισσότερες λειτουργίες ETL σε σύγκριση με το Ray. Παράλληλα, είναι πολύ πιο αποτελεσματικό στη διαχείρηση της μνήμης. Το Ray, ενώ παρουσιάζει δυνατότητες σε ορισμένες λειτουργίες, κάνει κακή διαχείρηση της μνήμης σωρού, ιδίως όταν οι λειτουργίες απαιτούν υψηλή χρήση μνήμης ή αφορούν μεγάλα σύνολα δεδομένων. Αυτό το γεγονός δυσχεραίνει και την επεκτασιμότητά του σε μεγαλύτερα μεγέθη δεδομένων εισόδου. Τα πλεονεκτήματά του εμφανίζονται σε μικρότερα μεγέθη εισόδου. Όσον αφορά, λοιπόν, λειτουργίες ETL, το Spark αποτελεί την ενδεδειγμένη επιλογή.

1.6.3 Πειράματα Γραφημάτων

Σε αυτό το στάδιο εκτελέστηκαν πειράματα που αφορούν την ανάλυση γραφημάτων, και συγκεκριμένα υλοποιήθηκαν προγράμματα υπολογισμού του pagerank των κόμβων ενός γράφου και του πλήθους των τριγώνων.

PageRank

Σε αυτό το πείραμα, ο αλγόριθμος PageRank υλοποιηθήκε παράλληλα τόσο στο Ray, χρησιμοποιώντας το Ray Data API, όσο και στο Spark, με χρήση των Spark Dataframes. Οι υλοποίησεις ήταν πολύ παρόμοιες με σκοπό να εξεταστεί πώς αποδίδει το Ray σε σχέση με το Spark σε όμοιες εργασίες. Παρουσιάστηκε και μια ταχύτερη υλοποίηση στο Spark, με χρήση της GraphX, για να αναδειχτούν οι δυνατότητες του Spark σε ό,τι αφορά την ανάλυση γράφων

Triangle Counting

Σε αυτό το πείραμα έγινε μέτρηση των τριγώνων ενός γράφου. Για τον σκοπό αυτό, στο Ray παραλληλοποιήθηκε η υλοποίηση του αλγορίθμου από την βιβλιοθήκη `networkX`, ενώ στο Spark χρησιμοποιήθηκε η σχετική δυνατότητα που παρέχεται από την GraphX.

Αποτελέσματα: Για άλλη μια φορά το Spark υπερείχε του Ray σε χρόνο εκτέλεσης σχεδόν για όλους τους γράφους και τις διατάξεις του cluster που εξετάστηκαν. Αντίστοιχα, ο συνολικός χρόνος CPU για το Spark ήταν σημαντικά χαμηλότερος από ότι για το Ray, αποεικνύοντας ότι το Spark είναι ταχύτερο για αυτές τις λειτουργίες. Επίσης, το χάσμα επιδόσεων μεταξύ Ray και Spark γίνεται πιο έντονο σε μεγαλύτερα γραφήματα υποδεικνύοντας ότι η υλοποίηση στο Ray κλιμακώνεται χειρότερα με την αύξηση του μεγέθους εισόδου. Φάνηκε, βέβαια, ότι και τα δύο πλαίσια επωφελήθηκαν από από την προσθήκη περισσότερων κόμβων στον cluster, ωστόσο το Spark παρουσιάζει πιο σταθερή και σημαντική επιτάχυνση. Πρέπει να σημειωθεί εκ νέου ότι για το Ray είχε πολύ μεγαλύτερες απαιτήσεις μνήμης σε σχέση με το Spark.

1.6.4 Πειράματα Εκπαίδευσης και Βελτιστοποίησης Υπερπαραμέτρων σε Μοντέλα Μηχανικής Μάθησης

Σε αυτό το στάδιο μελετήθηκε η επίδοση των 2 πλαισίων όσον αφορά την εκπαίδευση μοντέλων μηχανικής μάθησης και την εύρεση του βέλτιστου συνδυασμού των υπερπαραμέτρων τους.

Εκπαίδευση MLP

Σε αυτό το πείραμα εκπαιδεύτηκε ένα Perceptron πολλών επιπέδων (MLP) με τα εξής χαρακτηριστικά:

- **Επίπεδο εισόδου**: 4 χαρακτηριστικά
- **1ο κρυφό επίπεδο**: 128 νευρώνες, ReLU συνάρτηση ενεργοποίησης
- **Hidden Layer 2**: 128 νευρώνες, ReLU συνάρτηση ενεργοποίησης
- **Output Layer**: 1 νευρώνας, Sigmoid συνάρτηση ενεργοποίησης

Για την εκπαίδευση έγινε χρήση συναρτήσεων από τις βιβλιοθήκες Ray Train (Ray) και mllib (Spark). Τα χαρακτηριστικά του ήταν τα εξής:

Εκπαίδευση XGBoost

Σε αυτό το πείραμα εκπαιδεύτηκε ένα μοντέλο XGBoost με τα εξής χαρακτηριστικά:

- Ρυθμός εκμάθησης: 0.2
- Μέγιστο βάθος: 5
- Αριθμός παράλληλων δέντρων: 50
- Αριθμός γύρων για boosting: 5

Βελτιστοποίηση Υπερπαραμέτρων στο MLP

Σε αυτό το πείραμα εξετάστηκαν 8 συνδυασμοί υπερπαραμέτρων για το MLP με σκοπό να βρεθεί ο καταλληλότερος (αυτός που κάνει πιο ακριβείς προβλέψεις). Οι συνδυασμοί αυτοί ήταν οι εξής:

- **Μέγεθος παρτίδας**: [512, 1024]
- **Αριθμός νευρώνων στο 1ο κρυφό επίπεδο**: [64, 128]
- **Αριθμός νευρώνων στο 2ο κρυφό επίπεδο**: [64, 128]

Βελτιστοποίηση Υπερπαραμέτρων στο XGBoost

Σε αυτό το πείραμα εξετάστηκαν 8 συνδυασμοί υπερπαραμέτρων για το μοντέλο XGBoost με σκοπό να βρεθεί ο καταλληλότερος. Οι συνδυασμοί αυτοί ήταν οι εξής:

- **Ρυθμός εκμάθησης**: [0.1, 0.2]
- **Μέγιστο βάθος**: [5, 7]
- **Αριθμός παράλληλων δέντρων**: [15, 30]

Αποτελέσματα: Στα πειράματα εκπαίδευσης μοντέλων και εύρεσης των βέλτιστων υπερπαραμέτρων τους, το Ray ήταν σχεδόν πάντα ταχύτερο κατά πολύ του Spark. Τα Ray Tune και Ray Train είναι εξαιρετικά βελτιστοποιημένα. Το μόνο αρνητικό, όπως υπογραμμίστηκε και νωρίτερα, είναι ότι το Ray έχει υψηλότερες απαιτήσεις μνήμης από το Spark με αποτέλεσμα όταν τα δεδομένα εισόδου είναι μεγάλα να μην μπορεί να επιταχύνει περαιτέρω την εκτέλεση των προγραμμάτων λόγω έλλειψης παραπάνω μνήμης. Συνολικά, το Ray αποτελεί την πλέον κατάλληλη επιλογή για τέτοιου είδους εργασίες.

Chapter 2

Introduction

2.1 Background and Motivation

In the big data age, organizations in a wide range of industries are producing and gathering enormous volumes of data at previously unheard-of speeds. This data is extremely valuable because it can offer insights that improve operations, stimulate innovation, and guide decision-making. However, there are significant difficulties with storage, processing, and analysis due to the sheer volume, variety, and velocity of this data. Many large data frameworks have been developed to address these issues, and each one has its own set of features and performance characteristics [4].

Ray and Apache Spark are two of these frameworks that have become well-known solutions for distributed data processing. More recently, Ray was introduced with the goal of satisfying the needs of developing AI applications. It provides a scalable and adaptable framework for distributed and parallel computing, supporting a broad range of applications from ETL processes to machine learning pipelines. Since its initial release in 2014, Apache Spark has grown to become a key component of the big data ecosystem because of its quickness, user-friendliness, and extensive library. It is extensively used for tasks including machine learning, batch processing, and stream processing.

These frameworks target diverse use cases and have different design philosophies, which makes them necessary to compare. Ray is commended for its adaptability and capacity to manage dynamic, fine-grained workloads, whereas Spark is well-known for its strong support for large-scale data processing and its developed ecosystem. It is imperative for practitioners and researchers to comprehend the advantages and disadvantages of each framework in order to effectively utilize them for their unique requirements.

With an emphasis on how well Apache Spark and Ray perform in analytical operations, graph processing, and machine learning pipelines, this thesis seeks to present a thorough comparison of both technologies. This work aims to provide insights that can direct the selection of the suitable framework based on particular requirements by carrying out a series of experiments and assessing the outcomes.

2.2 Objectives of the Thesis

The main objectives of this thesis are:

1. **Overview of Frameworks:** To provide an in-depth overview of Ray and, secondarily, Apache Spark including their architectures, key features, and common use cases.
2. **Comparison Framework:** to create a methodical methodology for evaluating Ray and Spark's performance in various kinds of data processing jobs.
3. **Performance Evaluation:** To conduct experiments evaluating the performance of Spark and Ray in analytical operations, graph processing, and machine learning pipelines.

4. **Analysis and Discussion:** To analyze and discuss the results, highlighting the scenarios where each framework excels.
5. **Recommendations:** To provide practitioners with recommendations regarding the selection of Spark or Ray in accordance with their unique needs.

2.3 Structure of the Thesis

This thesis is structured as follows:

- **Chapter 2:** Provides an overview of Ray, describing its design, salient characteristics, and applications.
- **Chapter 3:** Focuses on Apache Spark, highlighting its key features, design, and potential use cases.
- **Chapter 4:** Outlines the comparison framework and methodology used for the performance evaluation.
- **Chapter 5:** Outlines the performance comparison's findings and covering analytical processes, graph processing and machine learning pipelines.
- **Chapter 6:** Discusses the findings, comparing the strengths and weaknesses of Spark and Ray, and their suitability for different use cases.
- **Chapter 7:** Concludes the thesis, summarizing the work and suggesting directions for future research.

2.4 Significance of the Study

This study is important because it can help determine which big data frameworks are best for certain applications. Processing and analyzing huge datasets quickly becomes essential as businesses depend more and more on data-driven decision-making. This thesis contributes to the understanding of Spark and Ray's capabilities and performance by offering a thorough comparison, which helps practitioners make well-informed judgments.

Furthermore, the results of this study have ramifications for the larger field of distributed computing and big data analytics. As new frameworks arise, comparative studies like this one help to advance the discipline by emphasizing the benefits and limitations of various methods. This paper also suggests potential areas of improvement and future research, encouraging the creation of more efficient and effective tools for big data processing.

2.5 Research Methodology

The research methodology adopted in this thesis involves both theoretical analysis and empirical evaluation. The theoretical analysis includes a comprehensive review of the literature on Apache Spark and Ray, focusing on their architectures, key features, and use cases. This review provides the foundation for developing the comparison framework.

The empirical evaluation involves conducting a series of experiments to measure the performance of Spark and Ray in different types of data processing tasks. These tasks include analytical operations, graph processing, and machine learning pipelines. The experiments are designed to mimic real-world scenarios, ensuring that the results are relevant and applicable to practical use cases.

The performance metrics considered in the evaluation were execution time, resource consumption, scalability, and fault tolerance. The results are studied and compared to determine each framework's strengths and flaws.

Chapter 3

Distributed Computing Frameworks

3.1 Overview of Ray

Ray is an open-source framework designed for scalable and distributed computing, originally developed at the University of California, Berkeley's RISELab. It employs a user-friendly, flexible API that supports a wide variety of applications, such as parallel processing, distributed training, and hyperparameter tuning. Ray can be considered a versatile tool for developers working with big data and machine learning applications, since it facilitates scaling Python code from a single machine to a large cluster [17].

Ray's ability to scale horizontally across a cluster renders it a suitable choice for high-performance computing tasks. The idea behind its design emphasizes simplicity and flexibility, allowing developers to concentrate their attention on writing their application logic without being concerned about the underlying distributed system complexities. This quality has established Ray as a preferred option for both academic research and industry applications [15].

3.1.1 Architecture and Components

Ray's architecture can be divided in two main layers: the application layer and the system layer, each one being responsible for carrying out different functions to support the execution of distributed computations and AI applications.

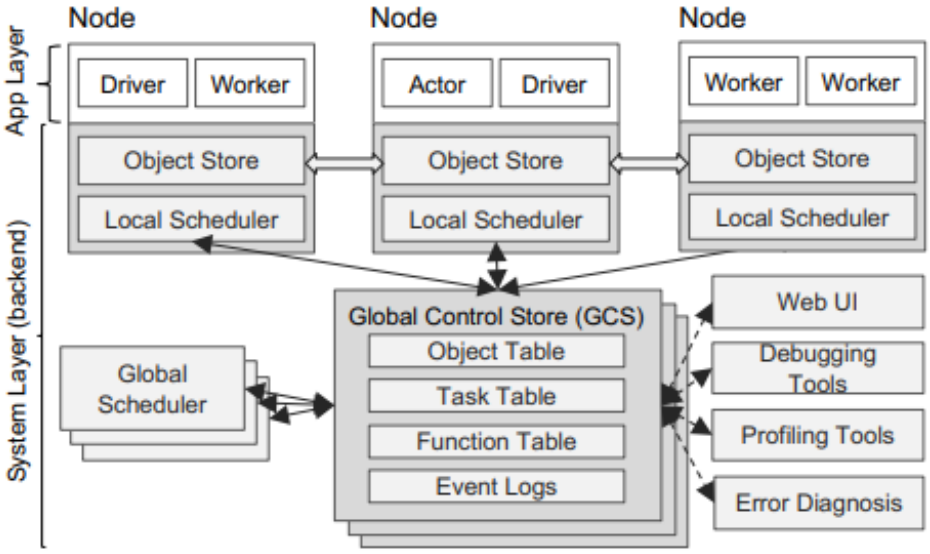

Figure 3.1.1: Ray's Architecture [17]

Application Layer

The application layer implements the API that qualifies users to interact with the Ray framework and express task-parallel and actor-based computations.

The application layer consists of three types of processes:

- **Drivers**: A driver is a program that runs the main function and schedules tasks and actors. It may be viewed as the orchestrator of the application, coordinating the execution of distributed tasks by interacting with Ray's API. The driver submits tasks and creates actors, which are consequently scheduled and executed across the cluster by Ray's scheduling system.
- **Tasks**: Tasks are stateless functions that may be executed in parallel across a cluster. These functions are invoked asynchronously, permitting for large-scale parallelism and efficient utilization of resources. Tasks are suitable for fine-grained parallelism and are particularly useful for applications that can be decomposed into many small, independent units of work.
- **Actors**: Actors, on the other hand, are stateful entities that maintain their state across multiple invocations. They are applicable for scenarios where stateful computations are required, including simulations, online learning, and real-time data processing. Actors may interact with one another through message passing, providing a flexible model for building complex, stateful applications.

Ray's System Layer

The system layer in Ray is designed to accommodate efficient and fault-tolerant execution of distributed AI applications. Three main components are included: the Global Control Store (GCS), a distributed scheduler, and an in-memory distributed object store. Each component is horizontally scalable and contributes to the overall robustness of the system.

- **Global Control Store (GCS)**

  The GCS is a key-value store with pub-sub functionality that maintains the control state of the entire system. ensures scalability and fault tolerance via sharding and chain replication. The GCS permits Ray for dynamic management of millions of tasks per second, decoupling durable lineage storage from other system components and simplifying fault tolerance and scalability.

- **Bottom-Up Distributed Scheduler**

  Ray's scheduler utilizes a two-level hierarchical model with a global scheduler and local schedulers on each node. This bottom-up approach helps manage the dynamic scheduling of tasks with minimal latency. Tasks are carried out by local schedulers unless overloaded or lacking resources, in which case the global scheduler intervenes to delegate tasks according to resource availability and task constraints. This architecture achieves efficient resource allocation and scalability.

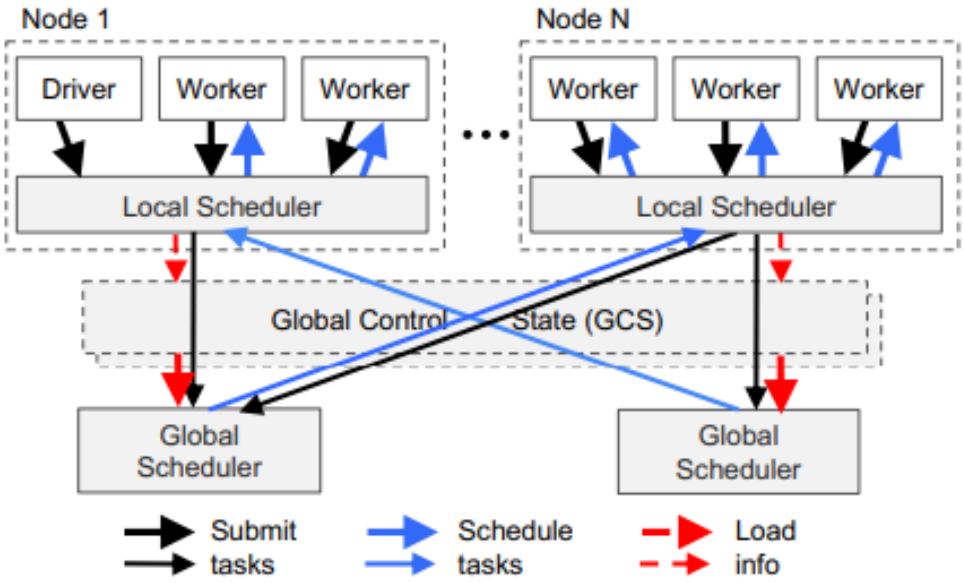

Figure 3.1.2: Bottom-Up Distributed Scheduler in Ray [17]

In this model, each node in the cluster contains a local scheduler that first attempts to handle task scheduling locally. Tasks created at a node are submitted to its local scheduler, which manages the execution unless the node is overloaded or lacks the necessary resources (e.g. a GPU). If it is not possible for a task to be scheduled locally, it is forwarded to the global scheduler.

The global scheduler, subsequently, estimates the load and resource availability of each node to reach scheduling decisions. It detects nodes with sufficient resources and selects the node with the lowest estimated waiting time for task execution. The task's queue size and the transfer time of remote inputs are considered for this estimation. Up-to-date information regarding node load and task requirements are held by the global scheduler through heartbeats and data from the GCS.

- **In-Memory Distributed Object Store**

  The in-memory object store on each node utilizes shared memory for zero-copy data sharing, reducing task latency and increasing throughput. It handles immutable data, simplifying consistency and fault tolerance protocols. In the instance of node failure, Ray reconstructs objects using lineage information stored in the GCS.

Implementation

Ray is implemented with around 40,000 lines of code, primarily in C++ for the system layer and Python for the application layer. The GCS employs Redis for key-value storage, along with sharded and chain-replicated tables for fault tolerance. Local and global schedulers are event-driven and single-threaded, with local schedulers managing cached states for efficient task dispatch and data transfer.

3.1.2 Key Features

Ray encompasses multiple key features that render it a powerful framework for distributed computing:

Dynamic Task Scheduling

Ray's dynamic task scheduling mechanism enhances resource utilization and adaptation to changing workloads effectively. Ray implements an approach of increased responsiveness and adaptiveness, aiming at the optimalization of performance and resource use, in contrast to traditional static scheduling methods.

Dynamic task scheduling in Ray leverages both local and global scheduling strategies, as previously analysed, to achieve high throughput and low latency. Tasks are initially handled by local schedulers on each node, making quick decisions based on local resource availability, therefore minimizing the latency associated with task dispatch and execution.

In case local resources are deemed insufficient, tasks are passed on to the global scheduler, which has a comprehensive view of the entire cluster. Through this approach, efficient resource allocation across the cluster, reducing bottlenecks and enhancing overall throughput is achieved [17].

Additionally, advanced features like speculative execution and straggler mitigation are supported by Ray's dynamic scheduling. By continuously monitoring task progress and resource utilization, Ray can pre-emptively schedule redundant executions of slow-moving tasks, therefore improving fault tolerance and guaranteeing timely completion of workloads.

Fault Tolerance

Ray includes built-in fault tolerance mechanisms, ascertaining the recovery of applications from node failures without losing progress. This is achieved through automatic task retry and lineage reconstruction, which empowers the system to rebuild lost state and carry on with processing.

Fault tolerance in Ray is designed to handle both transient and permanent failures. In case a node fails, the failure is detected by Ray's scheduler and the tasks are reassigned to other available nodes. The system monitors task dependencies and execution history, enabling it to reconstruct the state of incomplete tasks and ensure continuity.

Actor Model

Ray's actor model provides a simple way to manage stateful computations, making it easy to build applications that require long-lived processes. Actors may maintain state across multiple tasks, permitting more complicated workflows and decreasing the need for global state management [17].

The actor model in Ray provides developers with the ability to create stateful objects that can execute methods asynchronously. This model is particularly useful for applications that require persistent state, such as simulation environments, online learning systems, and real-time data processing pipelines [17].

Library Ecosystem

Apart from its fundamental capabilities, Ray includes a robust ecosystem of libraries that streamline a variety of distributed computing activities and render it an adaptable option for a broad range of applications. These libraries are:

Ray Data is designed to manage complex data processing jobs. It offers a high-level API for distributed data processing, making it possible to integrate with other Ray libraries seamlessly and perform effective ETL (Extract, Transform, Load) operations. Ray Data is particularly applicable for batch processing at scale, preparing big datasets, and transforming data for machine learning workflows. This library is a crucial tool in the handling of large datasets in remote situations, as it guarantees scalable and efficient data management.

Ray Train simplifies the distributed training of machine learning models. It supports widely-used frameworks such as PyTorch, XGBoost, and TensorFlow, and abstracts away the difficulties involved in distributing training workloads across numerous nodes. Ray Train implements the training of models in a scalable manner, thus assisting developers in making the most of distributed computing resources.

Ray Tune offers an extensive toolkit to experiment with multiple model setups, therefore rendering hyper-parameter tuning easier. Machine learning models may be efficiently optimized because of its support for multiple search techniques [15].

Ray RLlib is a scalable reinforcement learning package that hides the difficulties and complexities involved in creating and implementing RL models. It is suitable for both research and production use cases since it supports a large variety of reinforcement learning algorithms and interfaces with well-known simulation environments, such as OpenAI Gym and Unity. [14].

Ray Serve simplifies the management and implementation of machine learning models in real-world settings. With capabilities like model versioning, auto-scaling, and request batching, developers can create scalable, low-latency serving pipelines with its support for both batch and real-time inference [3].

Each library in the Ray ecosystem integrates seamlessly with Ray Core, the main API of Ray, creating a cohesive framework for building complex distributed applications. This modular design allows developers to select only the necessary components, ensuring flexibility and scalability for various use cases. As a result, tasks varying from data processing and training to serving and tuning can be effectively managed within a single, unified framework.

3.1.3 Industry Adoption of Ray

Many businesses use Ray to help them get past their present scalability issues and use it as a basis for developing AI applications in the future. More particularly, Ray opens up a vast array of practical applications and use cases.

A recent example of utilizing Ray's versatility is the development of a large-scale project titled Alpa [30]. Alpa is built by researchers from Google, AWS, UC Berkeley, Duke, and CMU and is designed to make training big deep learning models easier. Scaling entails dividing a computation network of operators among numerous devices. These operators are stateful, operate in parallel, and carry out various calculations (because they implement several neural network components). To be more precise, there are two kinds of parallelism: intra-operator parallelism, in which the same operator is divided among several devices, and inter-operator parallelism, in which distinct operators are allocated to various devices. Data parallelism (Deepspeed-Zero), operator parallelism (Megatron-LM), and mixture-of-experts parallelism (GShard-MoE) are a few examples of intra-operator parallelism. Through its automatic identification and execution of the optimal intra- and inter-operator parallelism patterns in the computation graph, Alpa is the first tool to unite various parallelism techniques. In this manner, Alpa uses hundreds of GPUs to autonomously split, schedule, and carry out the training computation of very large deep learning models. Because Ray can handle these very flexible parallelism patterns and mappings between operations and devices, Alpa engineers choose Ray as the distributed execution framework to support these unified parallel patterns.

Furthermore, IBM uses Ray because it allows them to efficiently handle and scale their AI and machine learning tasks across many machines [12]. Ray's ability to distribute computing tasks makes it perfect for managing the large-scale data and computational needs of AI models. Furthermore, Ray integrates well with Kubernetes through KubeRay, which simplifies deploying and managing applications in enterprise settings. This integration is crucial for IBM as it supports the advanced requirements of AI foundation models and large language models (LLMs), especially within IBM's watsonx data and AI platform.

Another example of Ray being integrated in the industry is Uber's production deep learning pipeline [19]. When compared to the legacy design of 16 GPU nodes, a heterogeneous setup with Ray, consisting of 8 GPU nodes and 9 CPU nodes, improves pipeline throughput by 50% and significantly reduces capital cost.

Lastly, the framework is also being used by other major tech companies, such as Amazon and OpenAI to scale their machine learning workloads.

3.2 Overview of Apache Spark

Apache Spark is an open-source unified analytics engine for analyzing massive amounts of data. Spark, which was initially developed at AMPLab at UC Berkeley in 2009, offers an interface for programming complete

clusters with fault tolerance as well as implicit data parallelism. Spark's widespread appeal can be justified by its swiftness, user-friendliness, and versatility in managing diverse big data tasks such as stream processing, machine learning, and batch processing [27].

Since its launch, Apache Spark has evolved into a crucial part of the big data ecosystem. It is widely recognized for its broad library ecosystem that supports a variety of data processing and analytics jobs, in addition to its ability to handle huge datasets quickly through the utilization of in-memory computing [28].

3.2.1 Architecture and Components

Apache Spark uses a master-slave architecture which consists of several executors that operate across worker nodes in the cluster and a driver that functions as a master node. This architecture helps Spark to efficiently manage and complete distributed data processing tasks. The basic architectural scheme of Spark is illustrated below:

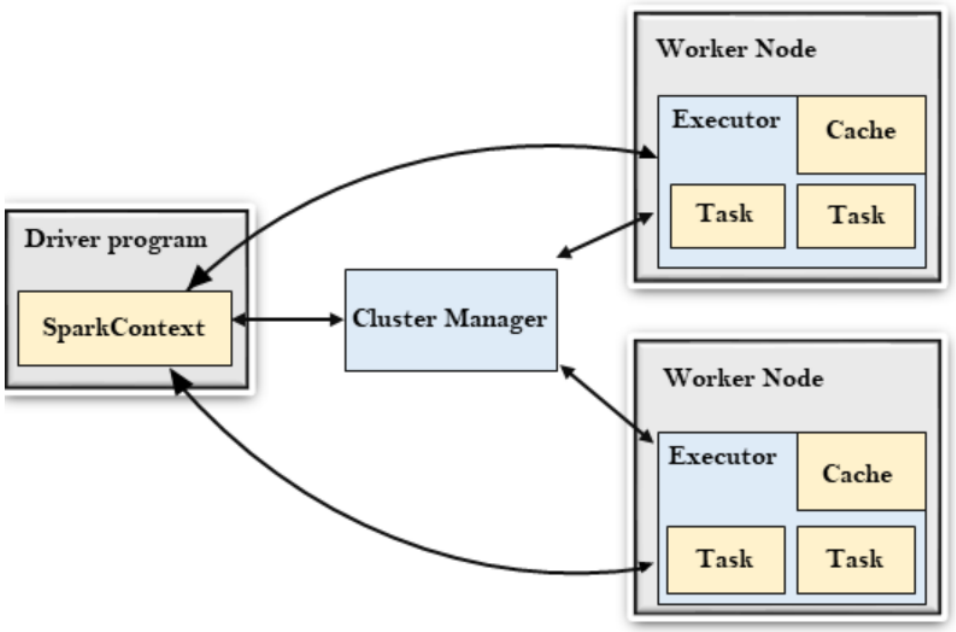

Figure 3.2.1: Spark's Architecture [13]

Delving into the architectural components shown above:

Cluster Manager

The cluster manager is in charge of allocating resources and managing the cluster on which the Spark application runs. Spark encompasses a basic built-in cluster manager called the Standalone Cluster Manager that comes with its installation. It is among the choices for cluster resource management and Spark application deployment. Nevertheless, various other cluster managers such as Apache Hadoop YARN, which we will be using in our system, are also supported by Spark.

SparkContext

SparkContext is the starting point of all Spark functionalities. It can be used to build broadcast variables, accumulators, and RDDs (Resilient Distributed Datasets). SparkContext is additionally responsible for the coordination of the execution of tasks by interacting with the cluster manager. All in all, It represents the connection to a Spark cluster.

Spark Driver

The driver is the program or process responsible for managing the Spark application's execution. It runs the main function and creates the SparkContext, which connects to the cluster manager. Furthermore, the driver is comprised of multiple parts, including the Block Manager, DAG Scheduler, Task Scheduler, and Backend Scheduler, which all cooperate to convert user-written code into jobs that run on the cluster.

Spark Executors

Executors are worker processes accountable for carrying out tasks in Spark applications. They interact with the cluster management and driver application after being launched on worker nodes. In order to perform concurrent tasks, executors cache and store data in memory or on disk for later use.

Task

Tasks are the smallest units of work in Spark and they represent a computation that may be completed on a single data partition. A Spark job is divided into tasks by the driver, which then designates the executor nodes to handle each task.

Resilient Distributed Dataset (RDD)

RDDs are the primary abstraction Spark's architecture is based on. An RDD is an immutable collection of elements that may be worked on in parallel across a cluster. Each dataset in an RDD can be divided into logical partitions, which are then executed on different nodes of the cluster. RDDs provide fault tolerance by using lineage information that facilitates their recomputation in the event of a node failure.

3.2.2 Key Features

Apache Spark is a strong and adaptable big data processing framework due to a number of important capabilities it provides [24]:

Unified Engine

Batch processing, stream processing, and interactive queries are among the numerous data processing tasks that Spark's unified engine can manage. Big data application development and deployment are simplified through this unified approach, which enables developers to use a single framework for a variety of use cases.

Batch and Streaming Data

Spark unifies the processing of data in batches and real-time streaming, making it possible for developers to use their preferred languages such as Python, SQL, Scala, Java, or R for both batch and streaming data processing.

SQL Analytics

Spark enables fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. It has efficient SQL analytics capabilities and operates more quickly than most data warehouses [1].

Data Science at Scale

Spark allows performing Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling. As a result, data scientists can work with large datasets adeptly.

Machine Learning

Spark provides the opportunity to train machine learning algorithms on a laptop and scale the same code to fault-tolerant clusters of thousands of machines. Such flexibility makes Spark a powerful tool for machine learning tasks.

Library Ecosystem

All the previously mentioned capabilities are being provided due to Spark's extensive ecosystem of libraries and modules that minimize the complexities of several data processing tasks and render it a suitable option for an extensive array of uses. Among these are:

Spark SQL Spark SQL is a module for structured data processing. It makes querying data via SQL possible and integrates with multiple data sources. Moreover, the DataFrame API, provided by Spark SQL, renders interaction with structured data simpler and more efficient for developers.

Spark Streaming Spark Streaming is a module for real-time data stream processing. It facilitates fault-tolerant, high-throughput stream processing of real-time data streams possible. High-level functions including map, reduce, join, and window may be utilized for the expression of complex algorithms that process data ingested from sources like Flume, HDFS, and Kafka [29].

MLlib (Machine Learning Library) MLlib is Spark's scalable machine learning library. It provides a vast range of machine learning algorithms and utilities, such as classification, regression, clustering, collaborative filtering, and dimensionality reduction [16].

GraphX GraphX is a module for graph-parallel computations. It supplies an effective API for graph operations and supports a set of graph algorithms, as for example PageRank and triangle counting. It provides users with the opportunity to create, modify, and query graphs in a distributed setting. [26].

SparkR SparkR is an R package that lays out a lightweight frontend to use Apache Spark from R. It incorporates Spark's capacities into the R environment and therefore data scientists can analyze massive datasets using familiar R syntax.

3.3 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a scalable and reliable storage system designed to handle large datasets across multiple machines. It is a core component of the Hadoop ecosystem, providing high-throughput access to data and fault tolerance through replication HDFS is optimized for large files, streaming data access, and cases where scaling out from a few nodes to thousands is necessary.

3.3.1 Architecture of HDFS

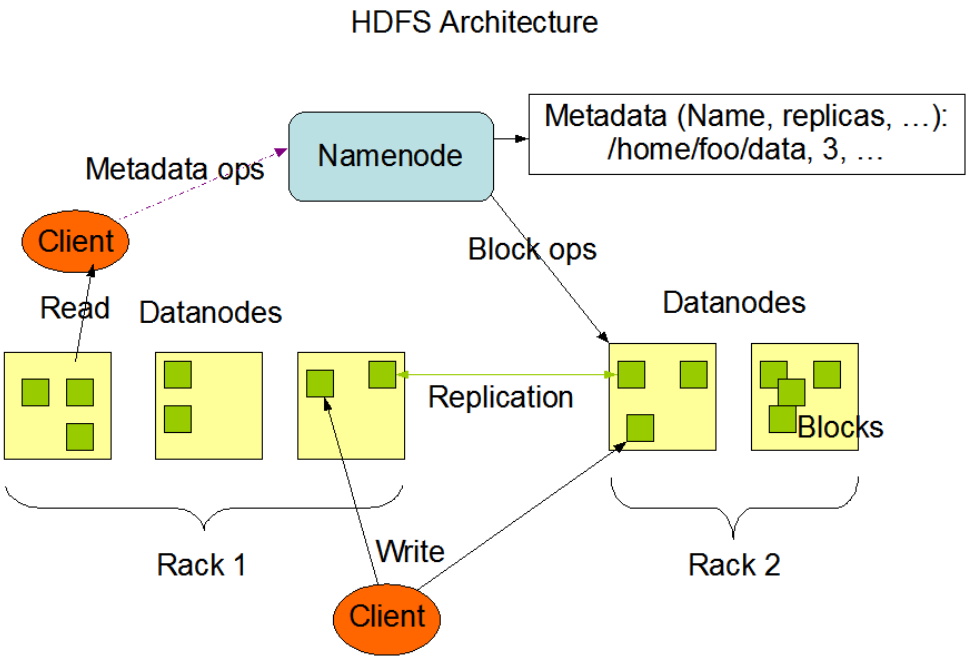

Figure 3.3.1: Architecture of HDFS [10]

NameNode and DataNodes

HDFS follows a master-slave architecture with a single NameNode in the role of the master server that manages the file system namespace and controls access to files by clients [10]. The actual data is stored on multiple DataNodes, responsible for handling read and write requests from the file system's clients.

NameNode The NameNode is the focal point of the HDFS architecture. It maintains the directory tree of all files in the file system and tracks the metadata about files such as file permissions, replication factors, and block locations. The NameNode stores the metadata and the file system tree rather than the actual data.

DataNodes DataNodes are in charge of storing the actual data in HDFS. Each file is separated into blocks, which are then stored across various DataNodes. DataNodes perform read and write operations as per client request and periodically send block reports to the NameNode to keep it updated with the current status of the data block.

HDFS Client

The HDFS client is a software component that facilitates the applications' interaction with HDFS. It provides APIs for reading from and writing to HDFS, and manages the communication among the client application and the HDFS cluster [22].

Replication and Fault Tolerance

HDFS achieves fault tolerance through the replication of data blocks across multiple DataNodes. By default, every data block is replicated three times: once on the local DataNode, once on a DataNode in the same rack, and finally, once on a DataNode in a different rack. This replication strategy ensures data availability even if a node or a rack fails.

3.3.2 Data Flow in HDFS

The data flow in HDFS involves several steps for reading and writing files, leading to efficient data transfer and robust fault tolerance.

Writing Data to HDFS When a client wants to write a file to HDFS, the following steps are executed:

1. The client communicates with the NameNode in order to create a new file entry in the metadata.
2. The NameNode checks for file name conflicts and makes sure that the client has the necessary permissions.
3. Once the file is created, the client initiates writing data to HDFS. The data is divided into blocks, and the client requests block allocation from the NameNode.
4. The NameNode reports the DataNode addresses where the blocks will be stored.
5. The client writes the data blocks to the first DataNode. The data is then pipelined by this DataNode to the following one in the replication chain, and so forth, until writing of all replicas is completed.
6. After all blocks are written and replicated, the client closes the file, and the NameNode updates the metadata to reflect the new file's status.

Reading Data from HDFS In case a client wants to read a file from HDFS, the following steps are executed:

1. The client comes in contact with the NameNode to retrieve the metadata and block locations for the file.
2. The NameNode responds with the list of DataNodes that hold the replicas of each block of the file.
3. The client then directly contacts the appropriate DataNodes to read the blocks. The data is streamed from the DataNodes to the client, reassembling the file as it is read.
4. If a DataNode is not available, the client may retrieve the data from a different DataNode that holds a copy of the block, achieving fault tolerance.

3.3.3 HDFS in Apache Spark and Ray

HDFS can be used with multiple data processing frameworks, including Apache Spark and Ray.

HDFS in Apache Spark Apache Spark can use HDFS as a distributed storage system, taking advantage of its scalability and fault tolerance. HDFS is a core component of the big data processing pipeline due to the fact that Spark jobs can read and write data to and from it. Spark is capable of processing massive datasets stored across several cluster nodes quickly by using HDFS.

HDFS in Ray Ray may as well take advantage of HDFS's powerful data management and fault tolerance features by using it as distributed storage too. Ray applications can acquire access to massive amounts of data stored throughout a cluster by integrating with HDFS, thus rendering scalable and effective data processing activities possible. Applications in machine learning and artificial intelligence that need access to large datasets may find this connection very helpful.

3.4 Yet Another Resource Negotiator (YARN)

Yet Another Resource Negotiator (YARN) is the resource management layer of the Hadoop ecosystem. Numerous data processing engines, such as batch, stream, and interactive querying, are capable of processing data stored in HDFS. YARN makes it possible for multiple apps to share resources efficiently by managing cluster resources and scheduling jobs. [11].

3.4.1 Architecture of YARN

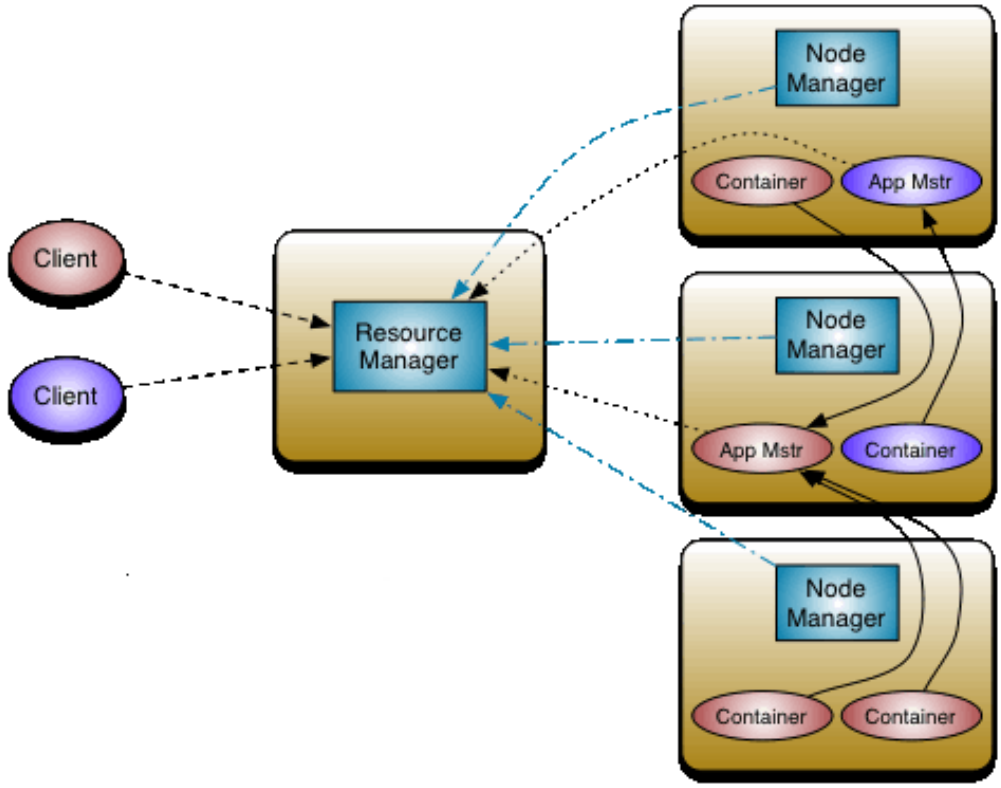

Figure 3.4.1: Architecture of YARN [11]

YARN follows a master-slave architecture as well, with per-node NodeManagers and a global ResourceManager. While NodeManagers oversee the job execution on specific nodes, ResourceManagers are responsible for scheduling and resource allocation. [25].

ResourceManager The ResourceManager is the primary mediator that arbitrates resources among all applications in the system. It consists of two main components:

- **Scheduler**: Allocates resources according to resource requirements.
- **ApplicationsManager**: Manages application lifecycles and provides services for starting and monitoring applications.

NodeManagers Each cluster node has a NodeManager responsible for managing containers, keeping track of how much CPU, memory, disk space, and network resources are being used, and reporting these findings to the ResourceManager. NodeManagers additionally manage logs and localize data, among other things.

ApplicationMaster Each application running on YARN has an ApplicationMaster, responsible for negotiating resources with the ResourceManager and working with NodeManagers to carry out and monitor tasks. The ApplicationMaster is specific to each application and handles the application's needs, as for example fault tolerance and job completion.

Containers Containers are the basic processing units. A set amount of resources (CPU, memory, etc.) are encapsulated in them and are distributed to NodeManagers by the ResourceManager for application tasks to be executed. Each application can possibly run numerous containers on various NodeManagers.

3.4.2 Resource Allocation and Scheduling

YARN manages cluster resources in an efficient manner by allocating and utilizing them through the usage of containers.

Resource Requests and Allocation The ResourceManager accepts resource requests from the ApplicationMasters, which include any location preferences and the necessary resources (memory, CPU, etc.). The ResourceManager's Scheduler employs policies such as Capacity Scheduler, Fair Scheduler, or FIFO Scheduler to allocate resources depending on availability and application requirements.

Container Allocation Once resources have been assigned, NodeManagers are given containers by the ResourceManager, who then launches and oversees the containers. The AM oversees how the tasks are carried out inside these containers.

Monitoring and Reallocation NodeManagers constantly monitor resource usage and report to the ResourceManager. If deemed necessary, the ResourceManager may preempt lower-priority containers aiming to reallocate resources and maintain optimal cluster usage.

3.4.3 YARN in Apache Spark

YARN is compatible with Apache Spark, and by taking advantage of its resource management features, Spark jobs can be properly scheduled and carried out. Spark can coexist with various data processing frameworks thanks to YARN, which gives big data applications flexibility and scalability. By permitting resource sharing and fine-grained resource allocation, Spark running on YARN increases cluster performance and utilization.

Chapter 4

Algorithms and Machine Learning Background

4.1 Graph Algorithms

Graph algorithms are fundamental in processing and analyzing graph-structured data. In this thesis, we focus on two broadly used graph algorithms: PageRank and Triangle Counting.

4.1.1 PageRank

PageRank is an algorithm initially used by Google Search to rank web pages in their search engine results. It estimates the importance of each node in a graph based on the structure of incoming links. The PageRank of a node is defined recursively, taking into consideration the PageRank of all nodes linking to it. This algorithm is used widely in web page ranking, social network analysis, and recommendation systems [18].

Mathematical Formulation The PageRank of a node i is given by:

$$PR(i) = \frac{1-d}{N} + d \sum_{j \in M(i)} \frac{PR(j)}{L(j)}$$

where:

- $PR(i)$ is the PageRank of node i.
- d is the damping factor, typically set to 0.85.
- N is the total number of nodes.
- $M(i)$ is the set of nodes that link to i.
- $L(j)$ is the number of outbound links from node j.

Iterative Computation Algorithm The iterative algorithm to compute PageRank involves the following steps:

1. **Initialization**: Assign an initial PageRank value to each node. Typically, each node is assigned a value of $\frac{1}{N}$.
2. **Iteration**: Update the PageRank values of all nodes based on the PageRank values from the previous iteration using the formula:

$$PR(i)^{(k+1)} = \frac{1-d}{N} + d \sum_{j \in M(i)} \frac{PR(j)^{(k)}}{L(j)}$$

where $PR(i)^{(k+1)}$ is the PageRank of node i at iteration $k+1$, and $PR(j)^{(k)}$ is the PageRank of node j at iteration k.

3. **Convergence**: Repeat the iteration step until the PageRank values converge, meaning that the change in PageRank values between iterations drops below a predetermined threshold.

The iterative computation of PageRank is illustrated by the following pseudocode:

```
Initialize PR(i) = 1/N for all nodes i
Set d = 0.85
Set convergence_threshold = 0.0001
Set max_iterations = 100

for k = 1 to max_iterations do
    PR_new(i) = (1 - d) / N for all nodes i

    for each node i do
        for each node j in M(i) do
            PR_new(i) += d * PR(j) / L(j)

    if |PR_new(i) - PR(i)| < convergence_threshold for all nodes i then
        break

    PR(i) = PR_new(i) for all nodes i
end for
```

Convergence and Complexity The PageRank algorithm typically converges within 50-100 iterations for most graphs, depending on the graph's size and structure. The computational complexity of each iteration is $O(E)$, where E represents the number of edges in the graph. This makes the algorithm efficient for large-scale graphs.

Applications PageRank has a wide range of applications beyond web page ranking, including:

- Ranking scientific papers according to citation networks.
- Recognizing influential users in social networks.
- Recommending products or content based on user interaction graphs.

4.1.2 Triangle Counting

Triangle Counting is a graph algorithm used to count the number of triangles in a graph. A triangle in a graph is a set of three nodes that are mutually connected.

Significance Counting triangles in a graph helps in understanding the structure and dynamics of the network. It is used to measure the clustering coefficient, which showcases the possibility that two nodes connected to a common node are also connected to each other. High clustering coefficients often indicate the presence of tightly-knit communities [21].

Mathematical Formulation The number of triangles $T(i)$ that a node i participates in can be calculated as:

$$T(i) = \frac{1}{2} \sum_{j,k \in N(i)} \delta(i,j,k)$$

where $\delta(i,j,k)$ is an indicator function that is 1 if there is an edge between nodes i, j, and k, and 0 otherwise.

Iterative Computation Algorithm The algorithm for counting triangles can be described as follows:

1. For each node i, find all pairs of neighbors (j, k).
2. For each pair (j, k), check if there is an edge between j and k.
3. Count each such triplet (i, j, k) as a triangle.

Applications Triangle counting is used in various applications such as:

- Community detection in social networks.
- Network robustness analysis.
- Fraud detection.

4.2 Machine Learning Background

Machine learning models are essential for predictive analytics and pattern recognition in data. In this thesis, we focus on two models: Multilayer Perceptron (MLP) and XGBoost.

4.2.1 Perceptron

In order to gain an understanding of the way the MLP works we first have to discuss the Perceptron. The Perceptron is a type of artificial neuron used in machine learning, which serves as a building block for neural networks [20].

Biological Neuron Human functions comprise a source of inspiration for machine learning techniques and algorithms. The human nervous system, with the neuron being the most basic unit, is the foundation for the biological relevance of the neural network. In a biological neural network, a neuron's function is to receive signals from neighboring neurons, process them as necessary, and then pass the signals to other neurons for additional processing. In this context, axons are the long projections that carry electrical signals away from the neuron's cell body, while dendrites are the branching structures that receive these signals from other neurons, facilitating the overall communication process within the network. As a result, a signal travels through numerous neurons. The junctions between axons and dendrites from different neurons are termed Synapses. Synapses are highly sophisticated structures that facilitate various functions during the transmission of signals. Synapses are in a state of dynamic equilibrium throughout an organism's existence, with new synapses being formed and old ones being destroyed as the brain learns, identifies, and comprehends more of its surroundings.

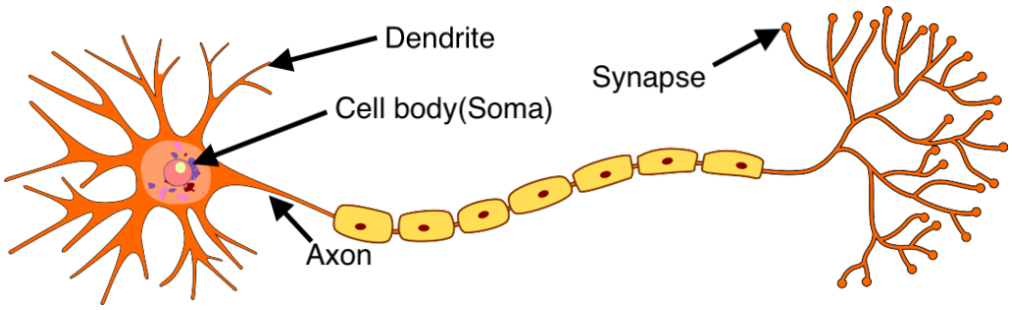

Figure 4.2.1: Biological Neuron

Artificial Neuron Early scientists managed to simulate the behavior of rudimentary artificial neurons by utilizing the fundamental discoveries regarding the functioning of real neurons. Once inputs from the environment are combined to create a "net" input, an artificial processing neuron sends the output signal y to another neuron or the environment after passing it via a linear or nonlinear activation function. An extensive

family of neural networks is designed using artificial neurons, which are comprised of four fundamental components:

1. **Nodes (Synapses)**, each characterized by its own weight, denoted by w. In contrast to the weight of a synapse in the human brain, the weight of an artificial neuron can accept both negative and positive values.
2. **An adder**, which sums the input signals, weighted by the corresponding synaptic weights of the neuron.
3. **An activation function** ϕ to limit the amplitude of a neuron's output system.
4. **An externally applied bias** b_k or θ_j, which induces a positive or negative bias in the result of the activation function.

The connections between nodes in an artificial neuron are analogous to axons and dendrites in a biological neuron; connection weights are analogous to synapses; and the threshold is analogous to the activity in the body. The function of the neuron is given by:

$$y = \phi\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right)$$

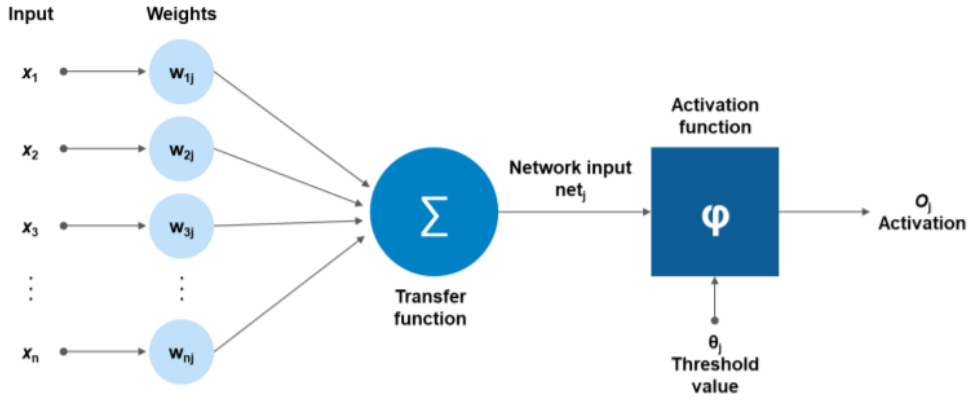

Figure 4.2.2: Artificial Neuron

4.2.2 Activation Functions

Activation functions compute a weighted sum and add bias to dictate if a neuron should be activated or not. Some of the most typical activation functions include:

Step Function Returns the neuron's state, whether activated or not.

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Linear Activation Function Returns the input without any change.

$$f(x) = x$$

Rectified Linear Unit (ReLU) Activates the neuron if the input is positive. If negative, it keeps it inactive.

$$f(x) = \max(0, x)$$

Exponential Linear Unit (ELU) Behaves similarly to ReLU for positive input values but differs for negative input values by being smooth and non-zero.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha \cdot (e^x - 1) & \text{if } x \leq 0 \end{cases}$$

where α is a positive parameter determining the rate of convergence to zero for a negative input.

Sigmoid Function Often used in binary classification problems.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Its derivative with respect to the input z is $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

4.2.3 Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) consists of several layers of feedforward-connected computational units. Every neuron in a layer directs connections to all neurons in the following layer [8].

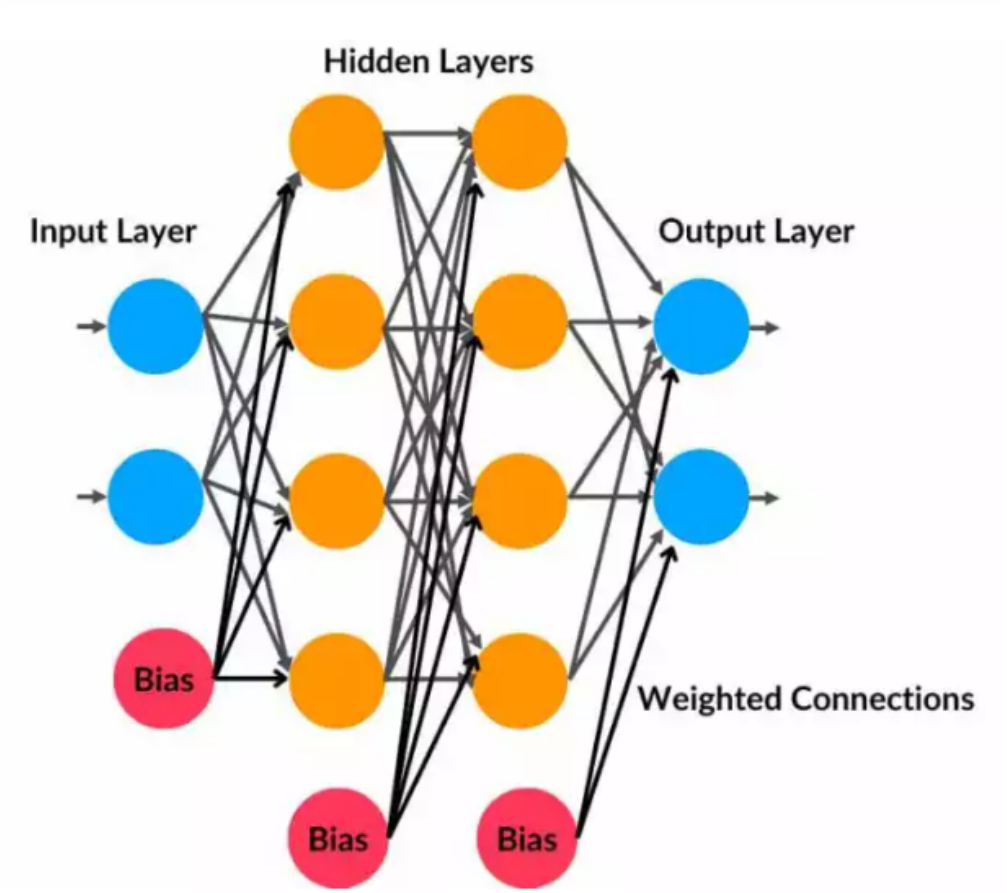

Figure 4.2.3: Multilayer Perceptron

Backpropagation MLPs utilize multiple learning techniques, with backpropagation being the most extensively used. Backpropagation establishes the value of a predetermined cost function through the comparison of output values with the correct responses. The fault is propagated backward towards the input using different approaches. The algorithm uses this data to adjust the weights of each connection decrease the value

of the cost function by a minimal amount (learning rate). Upon repetition of this procedure, the network eventually converges to a state where the calculation error is minor, indicating that it has acquired knowledge regarding a particular target function.

Gradient Descent Gradient Descent, a general nonlinear optimization technique, is applied for proper adjustment of weights. Aiming reduce the error and move towards the bottom of the error function's surface, the network determines the derivative of the error function with respect to the network weights. Backpropagation is restricted to networks with differentiable activation functions as the descent occurs in specific steps. There are three main variants of the Gradient Descent algorithm, which differ based on the amount of data used to compute the gradient of the objective function:

- **Batch Gradient Descent**: Calculates the gradient of the cost function concerning parameters using the entire training set. It is efficient but can take more time for large datasets since it requires computing the gradient on all data in each iteration.

  $$\theta = \theta - \eta \cdot \nabla J(\theta)$$

  where θ are the model parameters, η is the learning rate, and $\nabla J(\theta)$ is the gradient of the cost function.

- **Stochastic Gradient Descent (SGD)**: Calculates the gradient using solely one randomly selected sample from the training set in each iteration. It is quicker for large datasets and can avoid local minima in non-convex surfaces.

  $$\theta = \theta - \eta \cdot \nabla J(\theta, x_i, y_i)$$

- **Mini-Batch Gradient Descent**: Incorporates both approaches by using a small subset of the training data (batch) in each iteration. This method combines the speed of SGD with the stability of Batch Gradient Descent.

  $$\theta = \theta - \eta \cdot \nabla J(\theta; x^{(i:i+b)}, y^{(i:i+b)})$$

  Mini-Batch Gradient Descent reduces the chances of the model getting stuck in a local minimum, rendering it the most well-known algorithm in Deep Learning libraries.

4.2.4 XGBoost

XGBoost (Extreme Gradient Boosting) is a powerful and scalable machine learning system used for supervised learning tasks, such as classification and regression. It is an implementation of gradient boosted decision trees designed for optimal speed and performance and parallelization [2].

Key Concepts XGBoost builds an ensemble of trees sequentially, where each tree aims to correct the errors of the previous one. It uses gradient boosting, which combines the predictions of multiple weak models (usually decision trees) to produce a strong model. The main concepts in XGBoost are:

- **Boosting**: Boosting is an ensemble technique that merges the predictions of several base models to enhance the overall performance. In gradient boosting, every new model focuses on reducing the errors made by the previous models.
- **Decision Trees**: XGBoost uses decision trees as the base learners. These are constructed in a greedy manner to minimize a loss function.
- **Gradient Descent**: XGBoost uses gradient descent to optimize the loss function by adjusting the model parameters. Each new tree is fitted to the negative gradient of the loss function with respect to the current model predictions.

Mathematical Formulation The objective function in XGBoost encompasses a training loss term and a regularization term to control model complexity:

$$\text{Obj}(\theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k} \Omega(f_k)$$

where:

- l is the loss function (e.g., mean squared error for regression).
- f_k represents the k-th decision tree in the ensemble.
- $\Omega(f_k)$ is the regularization term for the k-th tree, controlling the complexity of the tree.
- θ represents the model parameters.

Algorithm The algorithm for XGBoost can be encapsulated in the following steps:

1. **Initialization**: Begin with an initial prediction (e.g., mean of the target values for regression).
2. **Compute Residuals**: Calculate the residuals (errors) based on the current model predictions.
3. **Fit a Tree**: Fit a decision tree to the residuals.
4. **Update Predictions**: Update the model predictions by adding the new tree's predictions, scaled by a learning rate.
5. **Repeat**: Repeat the steps of computing residuals, fitting trees, and updating predictions until the specified number of trees is reached or the model performance stops improving.

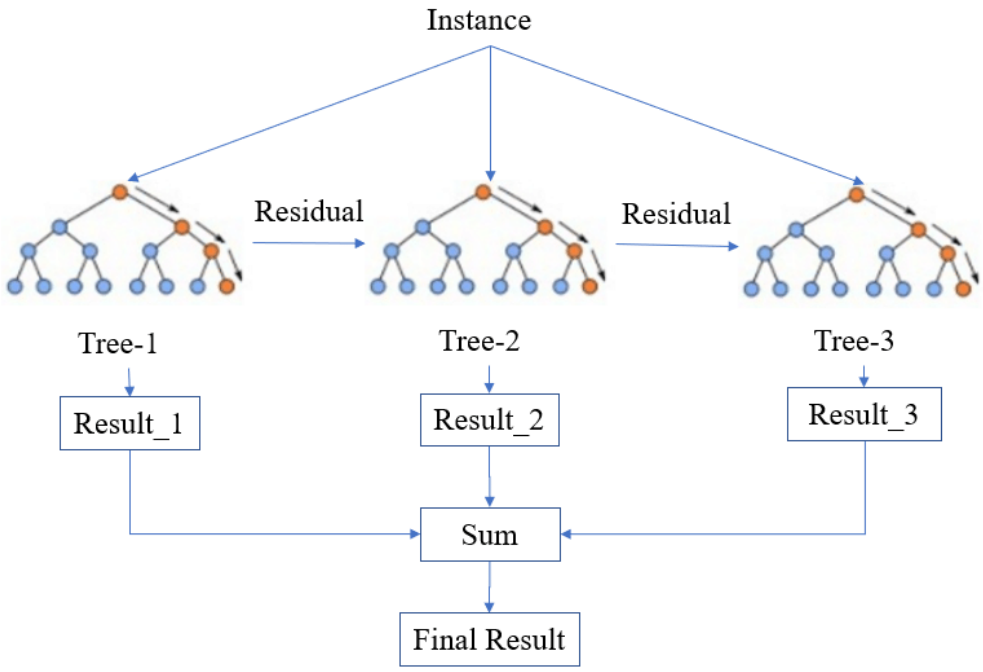

Figure 4.2.4: XGBoost workflow

Regularization XGBoost utilizes regularization techniques such as Lasso (L1) and Ridge (L2) regression to prevent overfitting and enhance generalization:

Lasso Regression Least Absolute Shrinkage and Selection Operator Regression (Lasso) regression adds a penalty proportional to the absolute value of the coefficients. The singling out of key features and the elimination of less significant ones, simplifies the model. The optimization objective for Lasso is:

$$\min \left(\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^{p} |w_j| \right)$$

where α is the regularization parameter controlling the strength of the penalty.

Ridge Regression Ridge regression, on the other hand, adds a penalty proportional to the square of the coefficients. This helps in handling multicollinearity (correlation between features) by shrinking the coefficients. The optimization objective for Ridge is:

$$\min \left(\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^{p} w_j^2 \right)$$

where α is the regularization parameter controlling the strength of the penalty.

Features XGBoost provides various features that enhance its performance and usability:

- **Parallel Processing**: XGBoost can run in parallel, making it highly efficient and scalable.
- **Handling Missing Values**: During training, XGBoost may automatically manage missing values by figuring out the favourable way to handle them.
- **Tree Pruning**: XGBoost uses a maximum depth parameter to limit the depth of trees and prunes trees using a post-pruning technique to eliminate splits that do not contribute to the performance's enhancement.
- **Sparsity Awareness**: XGBoost is optimized to handle sparse data effectively.

Applications XGBoost is widely used in various applications, including:

- **Classification**: Used in tasks such as spam or fraud detection, and image recognition.
- **Regression**: Utilized in predicting house prices, stock prices, and customer lifetime value.
- **Ranking**: Applied in recommendation systems, search engines, and ranking problems.

4.3 Hyperparameter Tuning

Hyperparameter tuning is a critical process in machine learning that involves finding the best set of hyperparameters for a model. Hyperparameters are configuration settings used to structure the model and training process, and they need to be set prior to the initiation of the learning process. Unlike model parameters, which are learned from the training data, hyperparameters need to be set either by hand or via an optimization process [7].

4.3.1 Importance of Hyperparameter Tuning

Many reasons make hyperparameter tuning necessary:

- **Model Performance**: A well-tuned set of hyperparameters can greatly improve the model's performance, improving its accuracy and generalizability to new data.
- **Avoiding Overfitting/Underfitting**: Correct tuning helps in equalizing the bias-variance tradeoff, avoiding situations where the model is rather simple (underfitting) or too complicated (overfitting).
- **Resource Efficiency**: Identifying the most promising hyperparameter configurations early on in the training process, can lead to efficient hyperparameter tweaking and hence cut down both training time and computational resources.

4.3.2 Common Hyperparameter Tuning Methods

Hyperparameter tuning can be done using several methods, most common of which include:

- **Grid Search**: This method involves an exhaustive search over a manually specified subset of the hyperparameter space. Even though it is simple and easy to understand, it is time-consuming and can be considered impractical when we work with large loads or complex models.

- **Random Search**: It is more efficient than grid search and can often find proper hyperparameters with fewer iterations, since instead of going through all possible combinations, random search samples hyperparameters from a specified distribution.
- **Bayesian Optimization**: This technique uses a probabilistic model to predict the performance of hyperparameter configurations and choose the most promising ones. It aims to find balance between exploration and exploitation, making it more efficient than random search.
- **Hyperband and ASHAScheduler**: These methods merge random search with early stopping so that resources are allocated to the most promising hyperparameter configurations properly. These approaches are of great assistance for tuning models with long training times.

4.3.3 Key Hyperparameters in Machine Learning Models

Different machine learning models have different hyperparameters. Here are some key hyperparameters for common models:

- **Neural Networks**: Learning rate, batch size, number of epochs, number of layers, number of units per layer, activation functions, dropout rate.
- **Tree-based Models (such as XGBoost)**: Number of trees, maximum depth, minimum samples per leaf, learning rate (for boosting methods), subsample ratio.

4.3.4 Challenges in Hyperparameter Tuning

Although hyperparameter tuning is providing many desired capabilities, it also presents several limitations:

- **Computational Cost**: Tuning hyperparameters, especially when it comes to large datasets or complex models, can be computationally expensive and time-consuming.
- **Search Space Size**: The hyperparameter space can be vast, and consequently hard to explore efficiently.
- **Interdependencies**: Hyperparameters can be dependent on one another, meaning that the value of one hyperparameter can be optimal only if specific hyperparameters have certain values.
- **Overfitting to Validation Set**: By excessive tuning on a validation we run the risk of overfitting, in which case the model would perform adequately on validation data but poorly on unseen test data.

Chapter 5

Experiments

5.1 Setup

5.1.1 Hardware

The cluster was built on 3 Virtual Machines provided by Okeanos-Knossos [9]. Their specifications:

- **Operating System**: Ubuntu Server LTS 16.04
- **Processors**: 4 CPUs
- **Memory**: 8GB RAM
- **Storage**: 30GB disk capacity

5.1.2 HDFS

For the experiments, a Hadoop Distributed File System (HDFS) was deployed across the 3-VM cluster. HDFS was selected due to its scalability and fault tolerance, providing reliable storage and quick data access across the distributed nodes. In addition to serving as the NameNode, the master node was also configured as a DataNode, storing actual data blocks and handling file system information. The remaining 2 VMs served as additional DataNodes, ensuring high availability and redundancy by replicating data across multiple nodes in case of hardware failures.

5.1.3 Ray

For each experiment, the Ray cluster was started by initializing the head node on the master VM and connecting the remaining VMs as worker nodes.

5.1.4 Spark and YARN

YARN (Yet Another Resource Negotiator) was specifically configured to act as the resource manager for Spark, guaranteeing effective job scheduling and resource allocation. Once more, the master node was assigned the dual role of a NodeManager and ResourceManager, with the other virtual machines acting as extra NodeManagers. It's important to note that Ray was not managed by YARN and operated independently on the same cluster.

5.1.5 Datasets

For the ETL and ML scripts, a set of CSV files with a size of 1GB each was used. These files were produced by a custom script. The names and the type of the dataset columns are shown below:

float1	float2	float3	int1	int2	label
float64	float64	float64	int	int	{0, 1}

Table 5.1: Dataset Schema

Columns float1, float2, float3 and label are produced by the `make_classification` function of sklearn which generates a dataset used for a random n-class classification problem (in our case 2-class). They were used for the training and tuning scripts.

There were 8 such files produced so that, when used in the same execution, they would have a total size of 8GB and they would not be able to fit into main memory.

For the Graph operations scripts, real-world graphs were used provided by the Stanford Network Analysis Project (SNAP) [23]. The graphs selected were:

- **p2p-Gnutella31**: a series of snapshots taken from the Gnutella peer-to-peer file-sharing network. In the Gnutella network topology, hosts are represented by nodes, and connections between Gnutella hosts are represented by edges.
- **email-EuAll**: The network was created using email data from a large European research institution, covering 18 months from October 2003 to May 2005. The dataset includes information on all incoming and outgoing emails within the institution. In the network, each node represents an email address, and a directed edge is created from node i to node j if i sent at least one email to j.
- **amazon0302, amazon0505**: These networks were collected by crawling the Amazon website. They are based on the "Customers Who Bought This Item Also Bought" feature of the Amazon website. The graph has a directed edge from product i to product j if product j is regularly purchased after i.
- **web-NotreDame**: Nodes represent pages from the University of Notre Dame. Links between the University's pages (www.nd.edu) are represented by directed edges.
- **wiki-Talk**: Each registered Wikipedia user has a talk page, that can be edited by them or other users in order to communicate and discuss modifications to various articles on Wikipedia. Wikipedia users are represented by nodes in the network, and a directed edge from node i to node j indicates that user i has at least once updated the talk page of user j.
- **web-BerkStan**: Nodes represent pages from berkely.edu and stanford.edu domains and directed edges represent hyperlinks between them.

Table 4.3 displays the number of edges and nodes in each of the graphs that were used:

graph	# of nodes	# of edges
p2p-Gnutella31	62,586	147,892
email-EuAll	265,241	420,045
amazon0302	262,111	1,234,877
web-NotreDame	325,729	1,497,134
amazon0505	410,236	3,356,824
wiki-Talk	2,394,385	5,021,410
web-BerkStan	685,230	7,600,595

Table 5.2: Dataset Schema

The graph data were saved in .txt files in the form of a list of node pairs that represent the beginning and the ending of each edge.

5.2 Experimental Procedure

- Each experiment was conducted 3 times, and the results presented are the average values of these five runs.

- Additionally, Ray and Spark scripts for the same experiments were performed in the same runtime environment. This approach ensured that the traffic in the Okeanos-Knossos VM system remained consistent, providing a fair comparison of the system's performance under similar conditions.
- System's cache was cleaned after every run so that neither Ray nor Spark gained advantages due to cached memory blocks.

The examined metrics were Execution Time, Total CPU Time, and maximum (1 node) Peak Heap Memory:

- Execution Time (ET): Duration of the operation's execution
- Total CPU Time (CPUT): Total time the CPUs of the system were occupied by performing the operation
- max value of Peak Heap Memory (max PHM) in 1 node of the cluster during the operation

5.3 ETL Experiments

In this section, the results of the ETL (Extract, Transform, Load) experiments are analyzed. 4 operators were tested: `map`, `filter`, `group_by/sum`, and `sort`. Each operator was tested using separate scripts in Ray and Spark.

5.3.1 Map Operation

The `map` operation is used to transform each element according to a function. In this experiment, the `map` operation was applied to perform this transformation on the float1 column:

$$\text{float1} = \text{float1} \times \text{int1} + \text{float2}^2$$

It should be noted that, in Ray, after experimentation it was figured out that when used with a transformation that is vectorizable (it can operate on a set of elements instead of just one element at a time), `map_batches` with pandas batch format (that is vectorized) should be preferred to `map` as it was found to be around 1.3-1.7 times faster in general and so it was selected for this experiment.

Results

The results acquired from running the map operation on ray and spark using 1, 2, and 3 worker nodes are shown in the tables below:

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	**28.3**	229.2	1254	38.1	232.8	233
2	**48.7**	355.5	1546	52.8	356.7	219
3	**69.2**	544.1	1747	71.1	512.7	245
4	93.3	721.9	1992	**88.2**	653.1	210
5	112.4	897.5	1879	**100.2**	802.5	224
6	136.3	1083.4	1956	**114.5**	934.0	235
7	159.7	1254.7	2153	**130.4**	1083.7	217
8	185.8	1411.0	1958	**146.4**	1215.1	240

Table 5.3: Map results on 3 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	52.3	226.1	1315	**47.8**	229.7	245
2	82.4	350.3	1522	**73.5**	352.9	254
3	114.5	537.1	1831	**104.3**	516.3	260
4	149.8	722.4	1905	**124.6**	651.5	239
5	191.7	888.9	1993	**143.3**	798.3	267
6	227.1	1074.7	2045	**160.8**	937.1	228
7	261.7	1241.7	1937	**181.4**	1076.7	263
8	291.0	1397.8	2023	**202.8**	1211.1	245

Table 5.4: Map results on 2 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	85.3	216.2	1276	**67.8**	212.7	217
2	127.7	341.5	1563	**104.0**	335.4	225
3	191.7	523.1	1785	**156.4**	508.3	233
4	243.6	709.7	1957	**195.5**	647.7	245
5	289.7	869.4	1914	**238.1**	792.9	212
6	342.0	1051.8	2203	**280.4**	939.5	231
7	396.1	1212.7	2025	**320.3**	1082.8	240
8	441.4	1355.9	1986	**361.9**	1220.0	250

Table 5.5: Map results on 1 node

Visualizing the operation's scalability as the input size increases on a 3-node cluster:

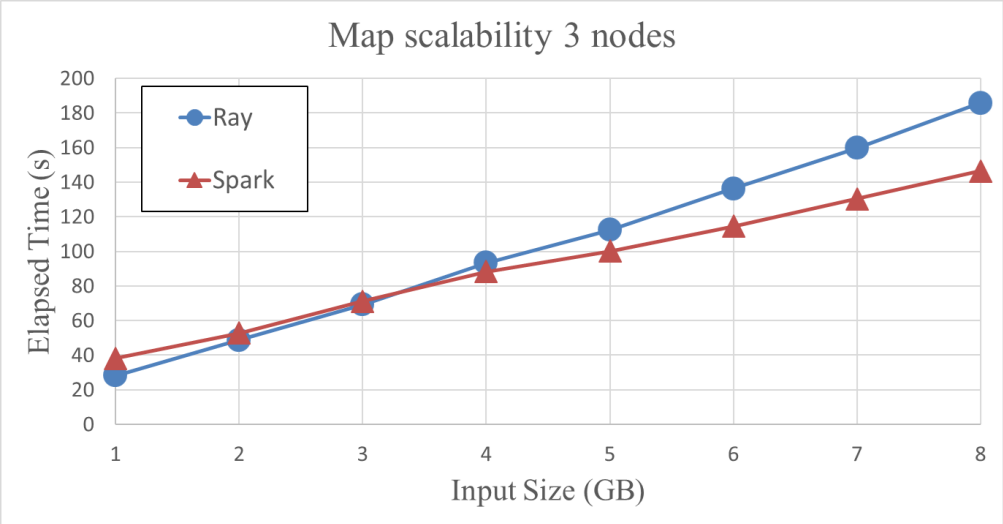

Figure 5.3.1: Map operation's scalability

Visualizing the operation's speedup (elapsed time reduction) as the number of nodes increases for a given input size:

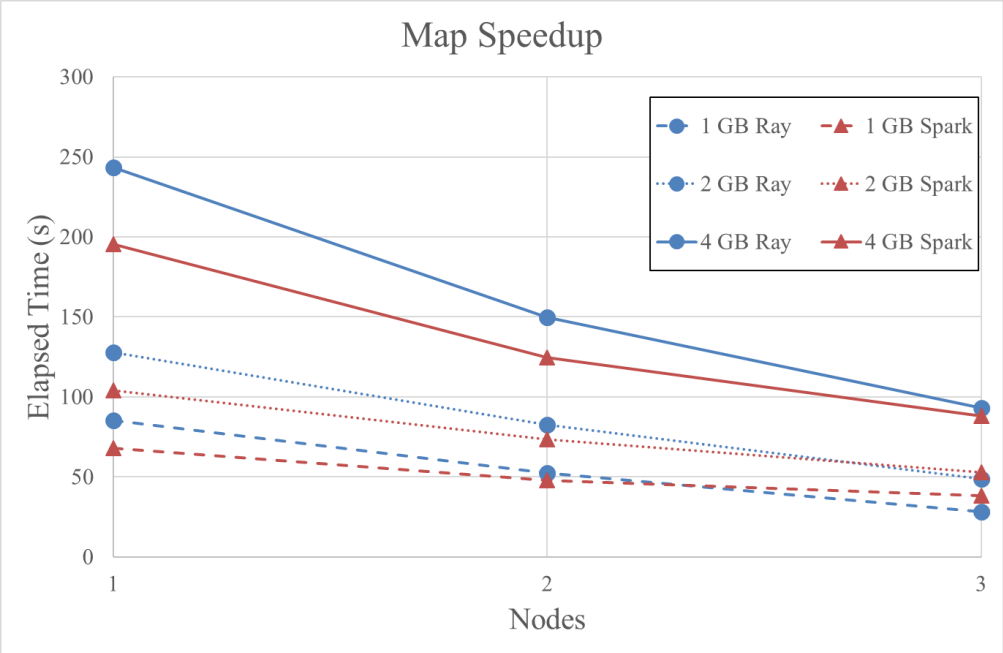

Figure 5.3.2: Map operation's speedup

Comparison

Spark generally shows faster execution times compared to Ray and scales better as the input size increases. Ray, on the other hand, even if it is generally slower, shows higher speedup with the increase of the cluster's nodes which is really evident when it comes to lower input sizes and to the transition from 2 to 3 nodes (from 1 to 2 nodes the speedup is almost identical). For the largest datasets, it seems Spark performs faster and achieves similar or higher speedups. This could be attributed to Ray's much higher (5-10 times) peak memory usage that could slow down the execution for larger datasets.

5.3.2 Filter Operation

The `filter` operation is used to select rows of a dataset that satisfy a given condition. In this experiment, the `filter` operation was applied to select the columns where `int1` column was lower than 500. Although the `filter` function is also vectorized in pandas, this time Ray's `filter` performed better than the combination of `map_batches` and pandas.

Results

The results acquired from running the filter operation on ray and spark using 1, 2, and 3 worker nodes are shown in the table below:

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	**17.5**	134.8	851	17.8	110.4	200
2	26.0	206.4	1291	**24.1**	154.6	233
3	41.9	339.5	1355	**35.5**	237.8	189
4	59.6	464.8	1606	**42.2**	296.7	221
5	69.0	566.5	1914	**53.2**	393.7	186
6	84.5	693.3	1963	**63.2**	482.0	213
7	93.7	763.0	1858	**69.4**	561.0	231
8	103.5	879.9	1919	**77.6**	644.2	248

Table 5.6: Filter results on 3 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	30.5	130.0	942	**24.5**	112.9	244
2	47.6	202.5	1267	**34.4**	161.0	256
3	79.3	341.7	1476	**51.2**	295.4	270
4	102.3	452.6	1805	**74.2**	397.8	232
5	128.5	571.4	2043	**92.6**	464.6	219
6	150.6	682.3	1977	**105.3**	554.3	263
7	165.2	745.2	1937	**110.7**	609.1	211
8	190.8	881.8	1965	**121.0**	738.5	279

Table 5.7: Filter results on 2 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	47.1	127.3	913	**32.8**	105.3	236
2	70.8	198.5	1371	**49.5**	158.4	217
3	115.2	335.6	1566	**89.2**	287.1	238
4	149.5	447.1	1780	**119.1**	387.2	232
5	185.5	563.9	1967	**140.2**	459.9	247
6	222.7	671.0	1882	**165.7**	546.8	272
7	241.8	742.7	2167	**180.4**	601.0	251
8	278.7	861.9	2345	**216.4**	726.7	268

Table 5.8: Filter results on 1 node

Visualizing the operation's scalability as the input size increases on a 3-node cluster:

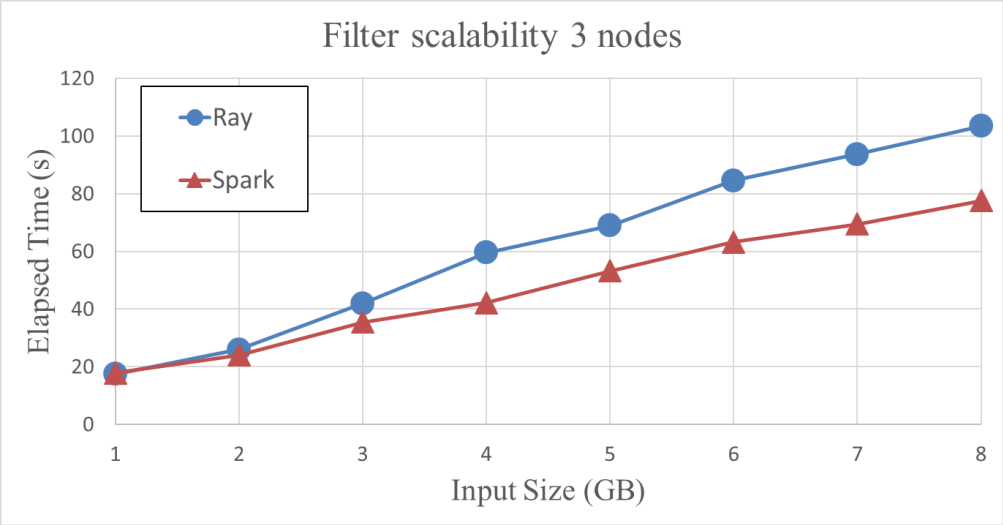

Figure 5.3.3: Filter operation's scalability

Visualizing the operation's speedup (elapsed time reduction) as the number of nodes increases for a given input size:

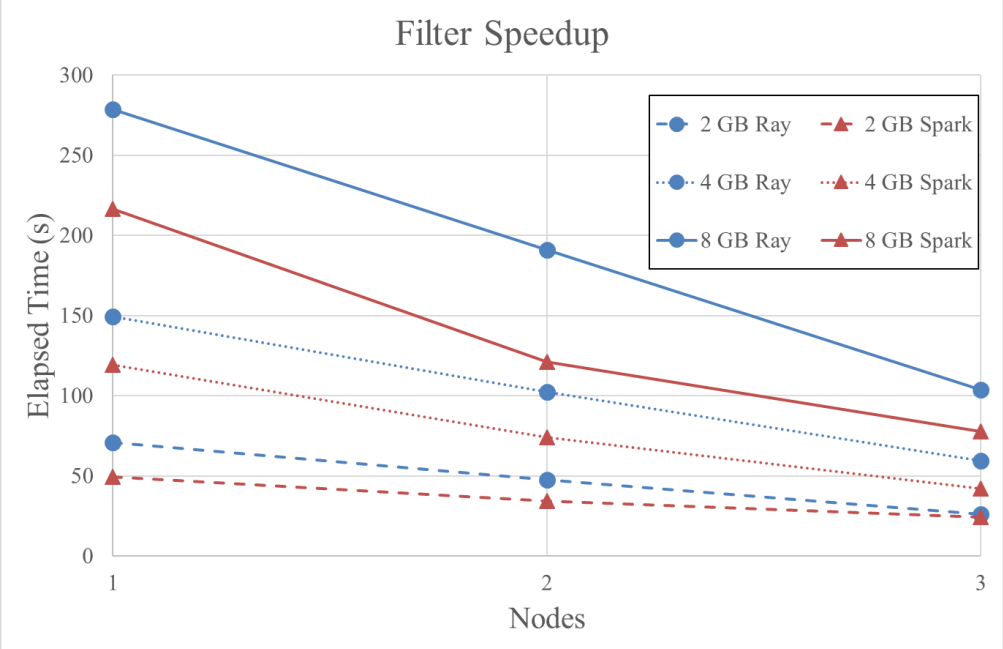

Figure 5.3.4: Filter operation's speedup

Comparison

This is another transforming operation and the results are similar to the previous ones. Spark achieves in general lower execution times but it seems Ray could catch up as more nodes are added to the cluster (and it even does for lower datasets in the 3-node cluster. Again it seems that Ray's advantage when it comes to speeding up by adding more nodes is clearly observable when the 3rd node is added to the cluster. It should also be noted that once again Ray's memory utilization is a lot worse as its maximum peak heap memory is consistently 5-10 times higher than Spark's.

5.3.3 Group by and Sum Operation

In this experiment, the `groupby` operator was applied to group the data by the `int1` column, and the `sum` operator was used to compute the total value for the `float1` column within each category.

As the group by operation required a shuffle operation, meaning that data is shuffled from all of the input partitions to all of the output partitions, for the Ray script, the environment variable `RAY_DATA_PUSH_BASED_SHUFFLE` was set as it enhanced the performance of the operation. As it is stated in Ray's documentation [6], scaling shuffling to big data sizes and clusters can be difficult, particularly if the size of the total dataset is too big to fit in memory, as it happens in this case.

Results

The results acquired from running the group by and sum operations on ray and spark using 1, 2, and 3 worker nodes are shown in the tables below:

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	35.6	293.7	1963	**12.3**	76.3	257
2	61.2	504.9	2268	**17.3**	110.7	294
3	84.8	653.0	2574	**23.8**	166.6	308
4	86.6	760.3	2856	**30.8**	227.9	286
5	124.7	1000.9	2968	**41.6**	307.0	314
6	117.2	1073.0	3187	**45.8**	372.4	328
7	180.5	1435.0	3324	**51.7**	426.4	372
8	209.0	1688.7	3584	**58.6**	486.4	349

Table 5.9: Group by - Sum results on 3 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	53.0	291.6	1754	**16.7**	80.2	287
2	91.3	501.3	1914	**23.5**	112.8	243
3	126.3	658.2	2356	**29.7**	154.7	290
4	163.8	802.7	2571	**45.2**	218.8	271
5	192.7	987.5	2489	**61.1**	305.8	368
6	234.5	1265.9	2790	**66.9**	364.6	305
7	262.6	1432.1	3102	**75.3**	427.3	333
8	301.7	1676.5	3277	**83.2**	497.9	352

Table 5.10: Group by - Sum results on 2 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	96.8	284.2	1843	**23.2**	74.5	213
2	167.4	493.5	2056	**34.6**	111.4	247
3	213.6	643.9	2467	**49.6**	158.7	286
4	267.0	786.8	2233	**65.0**	208.0	250
5	315.0	951.3	2392	**88.2**	286.7	268
6	387.3	1226.0	2504	**115.6**	378.2	312
7	450.2	1404.1	2828	**128.9**	434.4	295
8	529.4	1625.3	3040	**139.6**	471.9	347

Table 5.11: Group by - Sum results on 1 node

Visualizing the operation's scalability as the input size increases on a 3-node cluster:

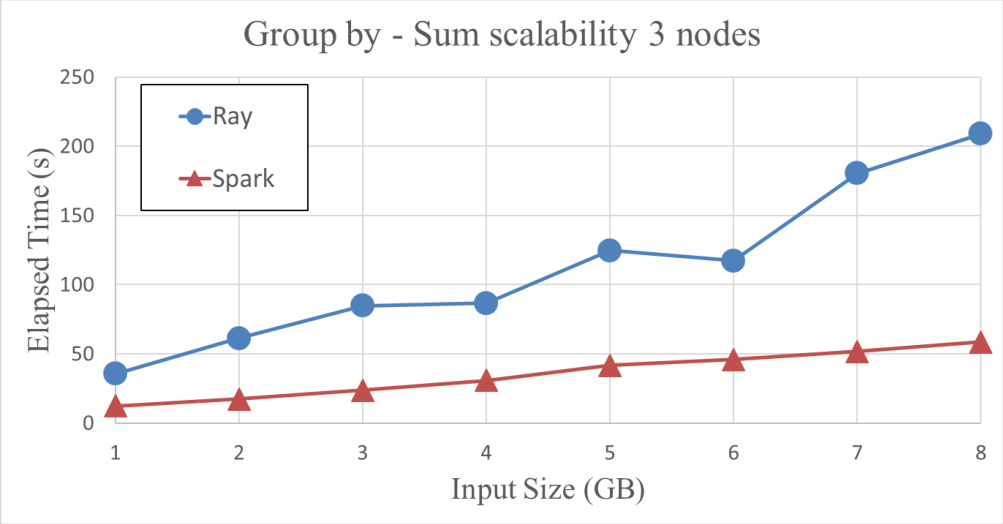

Figure 5.3.5: Scalability for Group by - Sum

Visualizing the operation's speedup (elapsed time reduction) as the number of nodes increases for a given input size:

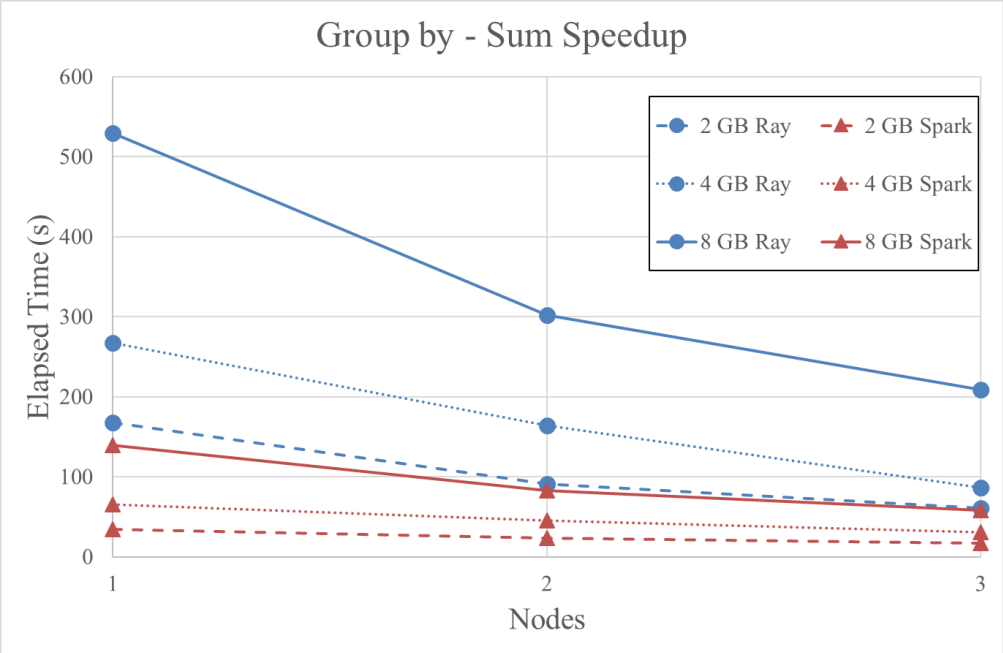

Figure 5.3.6: Speedup for Group by - Sum

Comparison

It seems the 2 frameworks are incomparable in this experiment as Spark is much more suited for this task. It performs the operations a lot faster and scales well with increasing input size. It is also noteworthy that Ray almost used the whole memory acquired for the shared-memory object store on 1 of the 3 nodes of the cluster.

5.3.4 Sort Operation

In this experiment, the `sort` operator was applied to sort the data by the `int1` column.

Results

The results acquired from running the sort operation on ray and spark using 1, 2, and 3 worker nodes are shown in the tables below:

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	29.2	230.9	1653	**25.3**	211.7	247
2	52.1	422.2	1964	**44.0**	389.7	238
3	72.8	545.8	2346	**61.0**	461.4	209
4	103.5	751.8	2622	**79.9**	611.3	228
5	128.9	966.7	2965	**107.3**	815.5	279
6	158.6	1205.0	3409	**131.5**	1045.4	305

Table 5.12: Sort results on 3 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	**42.3**	236.7	1758	45.0	215.9	273
2	**73.8**	435.6	2035	79.3	393.2	217
3	98.0	561.9	2417	**94.0**	457.3	289
4	124.4	742.8	2701	**123.3**	602.0	321
5	165.1	963.0	2934	**161.8**	819.2	266

Table 5.13: Sort results on 2 nodes

	Ray			Spark		
Input Size (GB)	**ET (s)**	**CPUT (s)**	**max PHM (MB)**	**ET (s)**	**CPUT (s)**	**max PHM (MB)**
1	72.6	216.3	1634	**64.9**	223.7	184
2	**103.4**	383.5	1982	105.5	365.6	236
3	161.1	513.9	2225	**135.2**	468.0	255
4	181.3	653.0	2583	**176.9**	607.9	221
5	245.7	858.3	2706	**233.0**	805.3	279

Table 5.14: Sort results on 1 node

Visualizing the operation's scalability as the input size increases on a 1-node cluster:

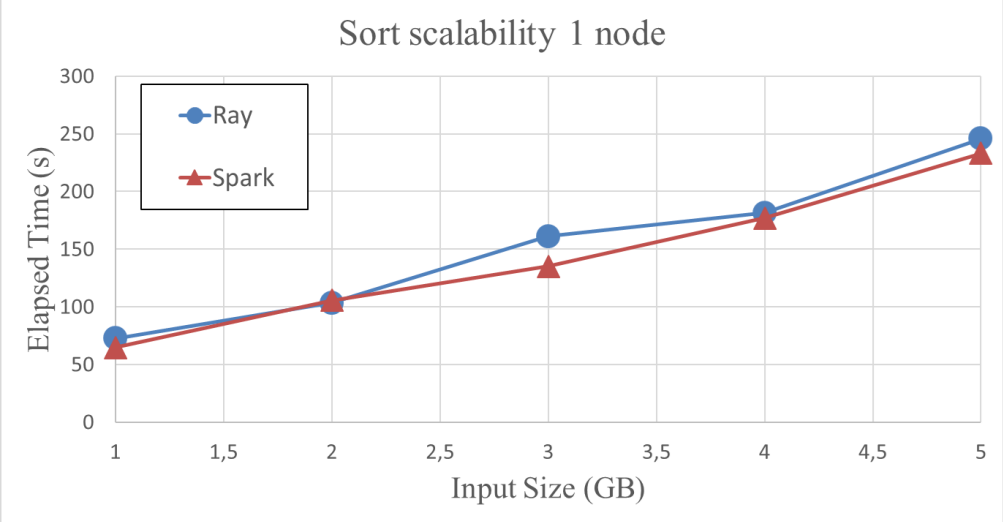

Figure 5.3.7: Sort operation's scalability

Visualizing the operation's speedup (elapsed time reduction) as the number of nodes increases for a given input size:

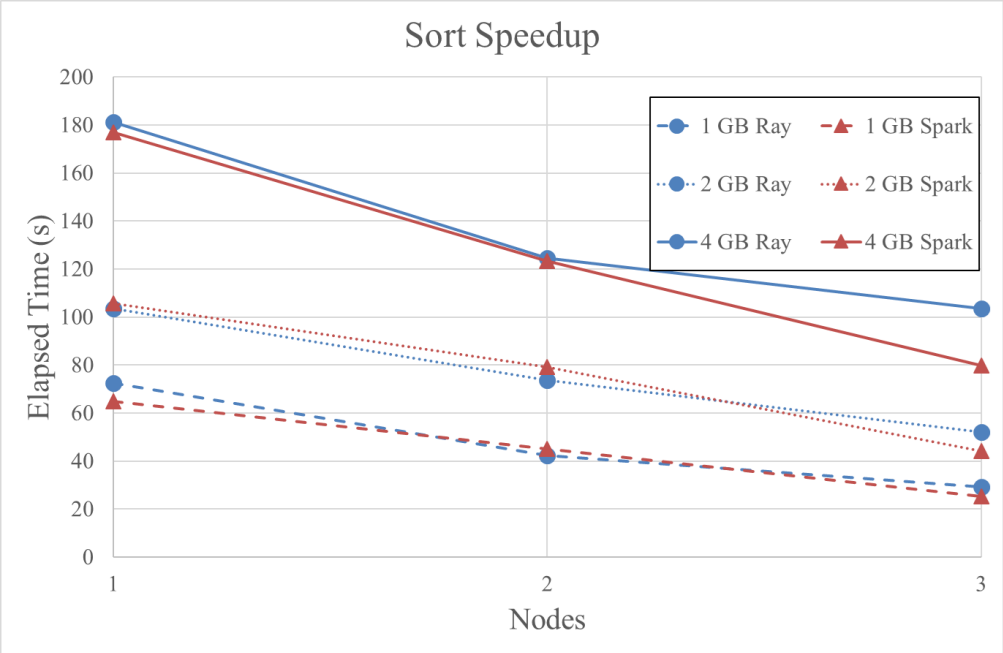

Figure 5.3.8: Sort operation's speedup

Comparison

On this operation, regarding execution time, the 2 frameworks perform quite similarly. They seem to scale with increasing input size and speed up with raising the number of cluster nodes in a similar manner.

During the sort operation, data is reordered between blocks. Therefore shuffling across partitions is required and so Ray's environment variable `RAY_DATA_PUSH_BASED_SHUFFLE` was set again. What should be noted is that Ray was extremely prone to "extreme memory (or disk, if spilling was necessary) usage" errors and could not sort the largest datasets it was provided with. This is not surprising as sorting is an all-to-all operation and so it needs higher memory usage to be successfully carried out. Ray's inability to sort the largest datasets is explained in its own documentation [6] where it is stated that shuffling can be challenging to scale to large data sizes, especially when the total dataset size can't fit into memory, which is our case.

5.3.5 Memory Usage

It is clear that Ray constantly shows higher peak heap memory usage across all workloads when comparing memory consumption between Spark and Ray. It is also important to note that, in many cases in the 3-node Ray cluster, it was observed that the memory of 1 of the 3 nodes (not a specific one) was utilized disproportionately more than the memory of the other 2 nodes. As a result, Ray showed a higher vulnerability to Out-of-Memory errors and required higher disk spillage. Taking these into consideration, we can conclude that while performing ETL tasks, Ray uses more memory resources than Spark and also allocates them more unevenly hurting the system's performance. This is not surprising since, as it is stated on the official Ray documentation [5], improving heap memory usage in Ray Data is an active area of development and 2 known occasions where heap memory management may be very high are reading large datasets (over 1GB) or transforming such datasets. This could be attributed to Ray's scheduler prioritizing data locality when assigning tasks to nodes.

Lastly, in subsequent tests that restricted the number of CPUs in the 3-node Ray cluster (instead of the workers), no significant decrease in performance was observed. This implies that the main constraint for a large number of Ray tasks was the accessible memory and not the CPU capacity of the cluster.

5.3.6 ETL Performance Evaluation

The ETL experiments reveal that Apache Spark generally provides better performance and scalability across most operations compared to Ray. Spark's advantages include:

- Faster execution times for map and group by/sum operations.
- Better handling of large-scale shuffling and sorting tasks.
- Lower and more efficient memory utilization.

Ray, while showing potential in certain operations (e.g. filter, sort), struggles with memory-intensive tasks and large datasets. Its strengths lie in smaller input sizes and simpler operations where it can leverage its vectorized processing capabilities. However, Spark and its RDDs remain the more robust and scalable option for comprehensive ETL workloads.

5.4 Graph Experiments

5.4.1 PageRank

In this experiment, the PageRank algorithm was applied to each of the graphs to identify the most influential nodes. For both frameworks a basic, but almost identical (logically), implementation of the algorithm as it was described in chapter 3.1.1 was tested using Ray Datasets and Spark Dataframes. It should be noted that due to Ray Datasets's lack of a join or merge operator, some different customizations were tried and the substitution of the join by sorting and zipping was found to be more efficient (but not enough) than transforming the ray datasets into pandas or dask dataframes. To illustrate Spark's superiority in this type of tasks another implementation using GraphX's PageRank was tested.

The results acquired from running this experiment on ray and spark using 1, 2, and 3 worker nodes are shown in the tables below:

	Ray		**Spark**		**GraphX**	
Graph	**ET (s)**	**CPUT (s)**	**ET (s)**	**CPUT (s)**	**ET (s)**	**CPUT (s)**
p2p-Gnutella31	41.2	315.1	**10.5**	79.2	5.9	42.1
email-EuAll	175.4	1347.7	**42.7**	301.9	17.9	155.7
amazon0302	274.9	2113.8	**66.8**	551.3	31.7	245.8
web-NotreDame	687.2	5528.8	**123.4**	993.2	51.6	419.8
amazon0505	647.1	5316.1	**138.3**	1252.4	92.2	750.1
wiki-Talk	1935.6	15533.0	**389.2**	3313.6	161.0	1327.2
web-BerkStan	2406.5	19208.2	**537.6**	4513.8	223.5	1967.1

Table 5.15: Pagerank results on 3 nodes

	Ray	**Spark**	**GraphX**
Graph	**ET (s)**	**ET (s)**	**ET (s)**
email-EuAll	306.9	**66.5**	28.5
web-NotreDame	921.3	**213.2**	73.3
amazon0505	1223.1	**292.0**	132.8
web-BerkStan	3738.7	**736.9**	320.1

Table 5.16: Pagerank results on 2 nodes

	Ray	Spark	GraphX
Graph	**ET (s)**	**ET (s)**	**ET (s)**
email-EuAll	487.1	**104.2**	51.6
web-NotreDame	1501.4	**307.8**	142.7
amazon0505	2154.0	**451.4**	240.0
web-BerkStan	6089.4	**1390.1**	683.9

Table 5.17: Pagerank results on 1 node

Visualizing the scalability of the 2 frameworks as the input size increases on a 3-node cluster:

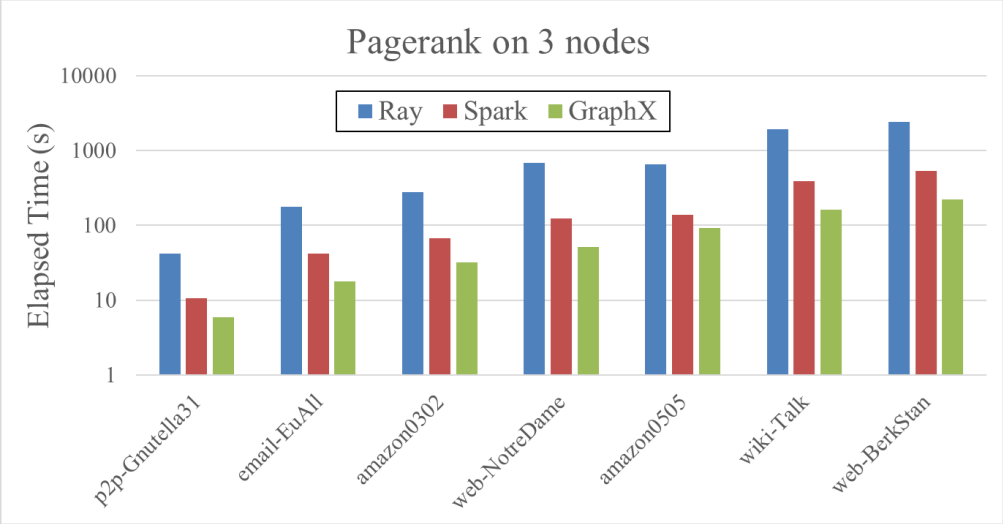

Figure 5.4.1: Scalability for the pagerank implementations

Visualizing the speedup (elapsed time reduction) of the 2 frameworks as the number of nodes increases for a given input size:

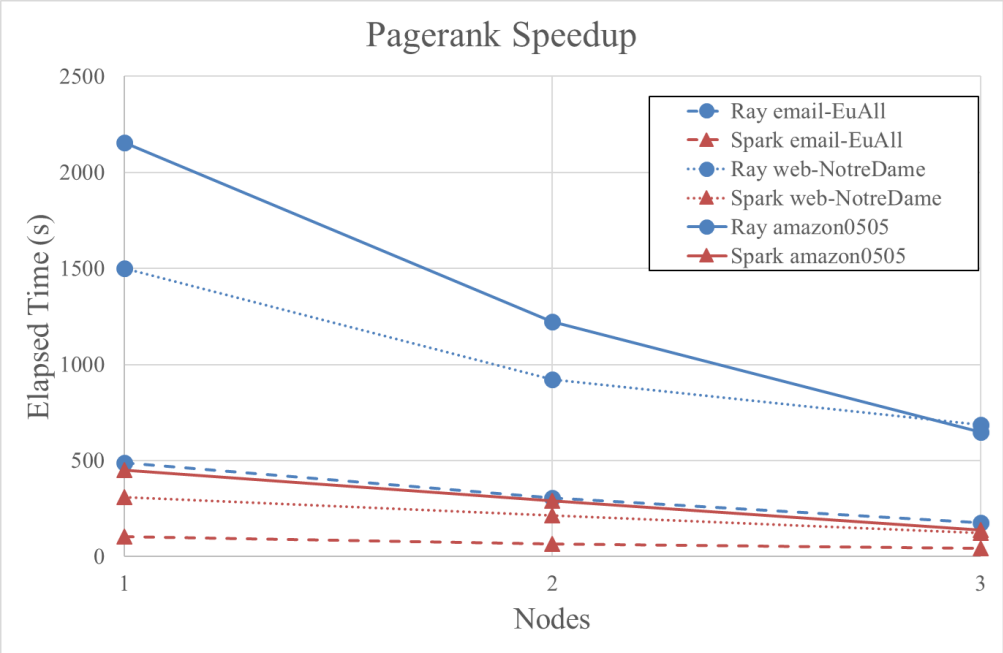

Figure 5.4.2: Speedup for the pagerank implementations

Comparison

The PageRank experiment results clearly indicate that Spark's "naive" implementation outperforms Ray in terms of both execution time, CPU time, scalability and speedup. Across all graphs, Spark shows significantly faster execution times compared to Ray, even being 5 times quicker in some cases. The memory usage on Ray was once again really high, with the maximum single-node peak heap memory usage being around 3GB for the largest graphs tested as opposed to Spark's 300MB for the same tests. It should also be noted that experiments on bigger graphs could not be executed as the Ray experiment threw the system out of memory. It was noticed again that one node's memory usage was 2 to 3 times higher than the 2 other nodes on the 3-node cluster.

GraphX's implementation is superior in every aspect to the other ones, as expected. It consistently shows the lowest execution and Total CPU times, while only using around 500MB of maximum peak Heap Memory, which is higher than Spark's simple implementation but it is counterbalanced by the higher efficiency GraphX offers.

5.4.2 Triangle Counting Experiment

As Ray does not provide a graph processing library, for this experiment triangle counting is implemented on Ray by parallelizing networkX's `triangles` function. Another implementation using the Ray Data API was tested (in a similar way pagerank was implemented), but parallelizing networkX's implementation was faster. In Spark, triangle counting is implemented using GraphX's `triangleCount`.

The results acquired from running this experiment on ray and spark using 1, 2, and 3 worker nodes are shown in the tables below:

	Ray		Spark	
Graph	**ET (s)**	**CPUT (s)**	**ET (s)**	**CPUT (s)**
p2p-Gnutella31	**5.2**	39.4	15.7	120.1
email-EuAll	38.9	278.3	**17.8**	148.6
amazon0302	33.7	267.8	**21.4**	174.1
web-NotreDame	57.6	423.2	**24.2**	196.9
amazon0505	93.9	729.4	**27.5**	219.6
wiki-Talk	115.3	962.4	**25.8**	208.7
web-BerkStan	141.2	1173.9	**29.6**	241.7

Table 5.18: Triangle counting results on 3 nodes

	Ray	Spark
Graph	**ET (s)**	**ET (s)**
email-EuAll	61.5	**32.5**
web-NotreDame	97.2	**42.4**
amazon0505	155.5	**46.9**
web-BerkStan	250.4	**56.1**

Table 5.19: Triangle counting results on 2 nodes

	Ray	Spark
Graph	**ET (s)**	**ET (s)**
email-EuAll	93.2	**52.1**
web-NotreDame	125.8	**63.0**
amazon0505	211.2	**70.4**
web-BerkStan	375.4	**78.5**

Table 5.20: Triangle counting results on 1 node

Visualizing the scalability of the 2 frameworks as the input size increases on a 3-node cluster:

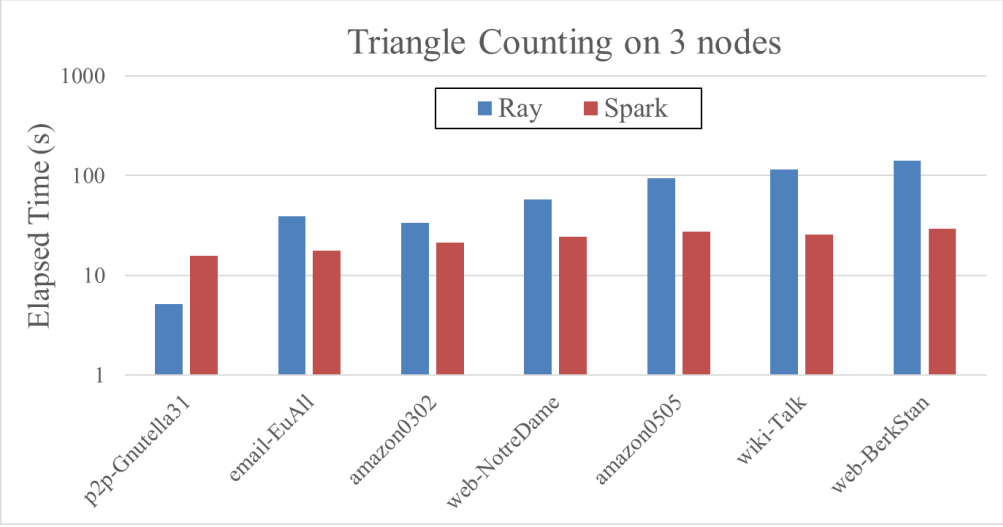

Figure 5.4.3: Scalability for triangle counting implementations

Visualizing the speedup (elapsed time reduction) of the 2 frameworks as the number of nodes increases for a

given input size:

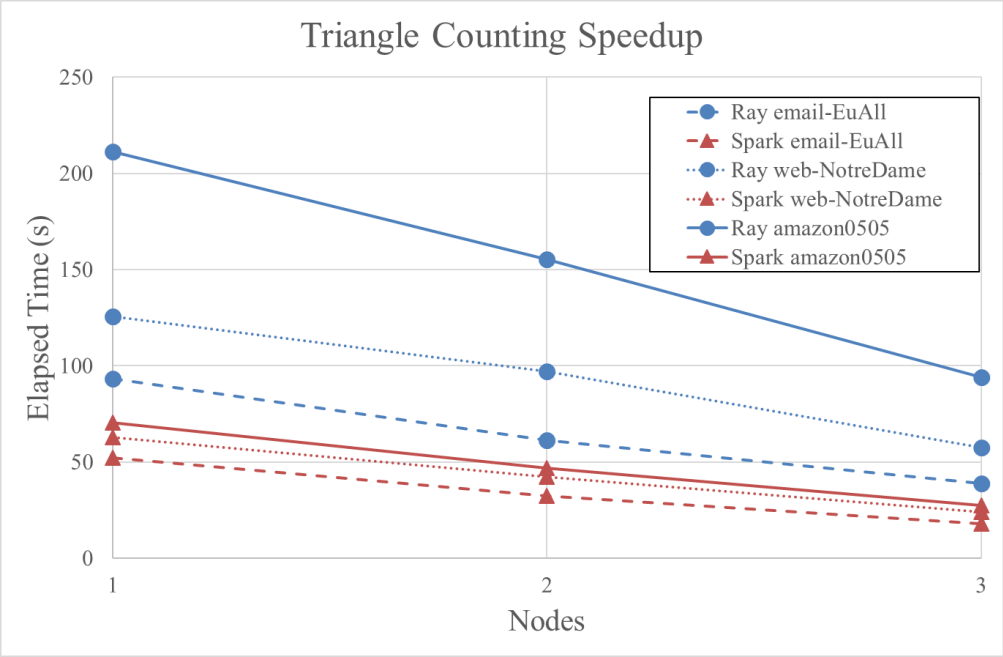

Figure 5.4.4: Speedup for triangle counting implementations

Comparison

Once again Spark outperformed Ray in terms of execution time across all datasets and configurations. Similarly, the CPU time for Spark was significantly lower than for Ray.

The performance gap between Ray and Spark becomes more pronounced with larger graphs indicating that Ray's implementation scales worse with higher input sizes. It also seems that both frameworks benefit from the addition of more worker nodes, but Spark shows a more consistent and significant speedup as is evident in Figure 4.12.

Lastly, it should be noted that once more Ray required higher memory usage (around 2.7GB for the larger graphs, while during GraphX's implementation, the maximum 1-node peak heap memory was around 700MB for the same graphs).

5.4.3 Overall Comparison on Graph Operations

The experiments on graph operations highlight clear performance differences between Ray and Spark. Based on the results, Spark is better suited for graph operations due to several key advantages:

- **Execution Time and CPU Time:** Across both experiments, Spark consistently outperformed Ray in terms of execution and CPU times.
- **Memory Efficiency:** Spark demonstrated significantly better memory efficiency compared to Ray. Ray's higher memory usage led to out-of-memory errors on larger graphs, while Spark managed memory more effectively, avoiding such issues.
- **Scalability:** Spark showed superior scalability with increasing input sizes and the addition of worker nodes. Generally, the performance gap between Ray and Spark widened with larger datasets, indicating that Spark scales better for large-scale graph analytics.
- **Library Support:** While parallelizing simple code in Ray is possible and easy, Spark's integrated GraphX library provided more straightforward optimized implementations for graph algorithms. This makes Spark a more robust choice for comprehensive graph analytics tasks.

5.5 Training Experiments

5.5.1 MLP Training

The MLP model that was used in this experiment was designed with two hidden layers to enhance its capacity for capturing complex patterns in the data. The structure of the MLP is as follows:

- **Input Layer**: 4 features
- **Hidden Layer 1**: 128 neurons, ReLU activation
- **Hidden Layer 2**: 128 neurons, ReLU activation
- **Output Layer**: 1 neuron, Sigmoid activation

The training process involved the following steps:

1. **Data Preparation**: The dataset was read, a filter was applied, and the 3 float columns were selected.
2. **Preprocessing**: Features were concatenated, and a MinMaxScaler was used to normalize the feature values.
3. **Training Loop**: The Binary Cross Entropy Loss function and the Adam optimizer were used to train the model. The training loop updated the model weights by iterating through the dataset for 5 epochs.
4. **Evaluation**: After each epoch, the model was evaluated on the validation set to compute the accuracy of the model.

Ray train's `TorchTrainer` was used for the training of the custom MLP class on Ray, while in Spark pyspark's `MultilayerPerceptronClassifier` class was utilized.

The results of training the MLP on Ray and Spark are presented below:

	Ray			**Spark**		
Input Size (GB)	**Pre (s)**	**Train (s)**	**Total (s)**	**Pre (s)**	**Train (s)**	**Total (s)**
1	50.4	**110.6**	161.0	20.7	367.7	388.4
2	97.3	**206.9**	304.2	30.2	712.5	742.7
4	171.0	**390.1**	561.1	42.2	1054.3	1096.5
8	313.1	**721.4**	1034.5	68.9	1870.6	1939.5

Table 5.21: MLP Training results on 3 nodes

	Ray			**Spark**		
Input Size (GB)	**Pre (s)**	**Train (s)**	**Total (s)**	**Pre (s)**	**Train (s)**	**Total (s)**
1	62.7	**125.4**	188.1	25.5	652.5	678.0
2	152.7	**253.2**	405.9	37.8	1075,6	1113,4
4	249.1	**435.3**	684.4	54.7	1663,8	1718,5
8	472.9	**785.6**	1258,5	101.0	2768,5	2869,5

Table 5.22: MLP Training results on 2 nodes

	Ray			Spark		
Input Size (GB)	**Pre (s)**	**Train (s)**	**Total (s)**	**Pre (s)**	**Train (s)**	**Total (s)**
1	97.8	**172.0**	269.8	43.8	1087.4	1131.2
2	253.8	**371.2**	625.0	61.3	1814.6	1875.9
4	361.1	**682.6**	1043.7	93.1	2790.6	2883.7
8	793.3	**1277.5**	2070.8	185.2	4356.4	4541.6

Table 5.23: MLP Training results on 1 node

Visualizing the scalability of the 2 frameworks as the input size increases on a 3-node cluster:

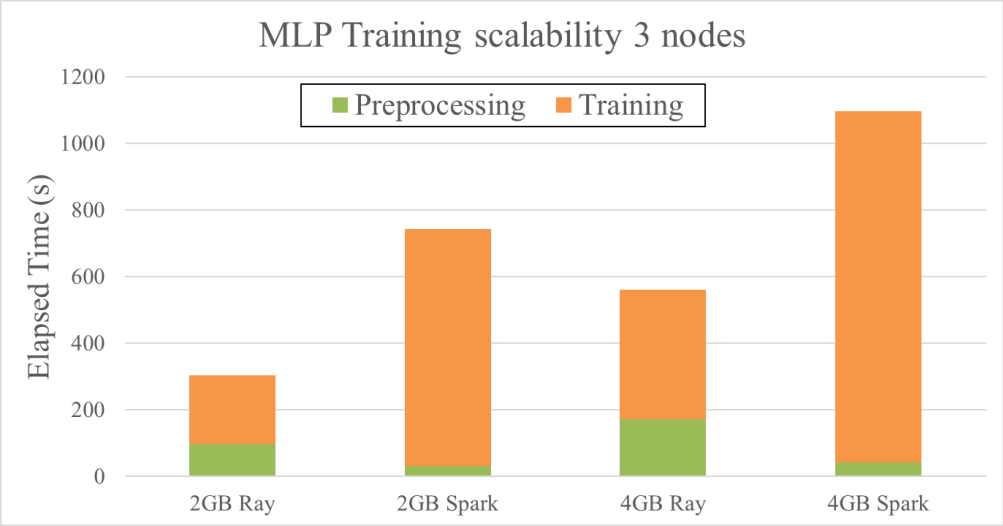

Figure 5.5.1: MLP preprocessing and training times

Visualizing the speedup (elapsed time reduction) of the 2 frameworks as the number of nodes increases for a given input size:

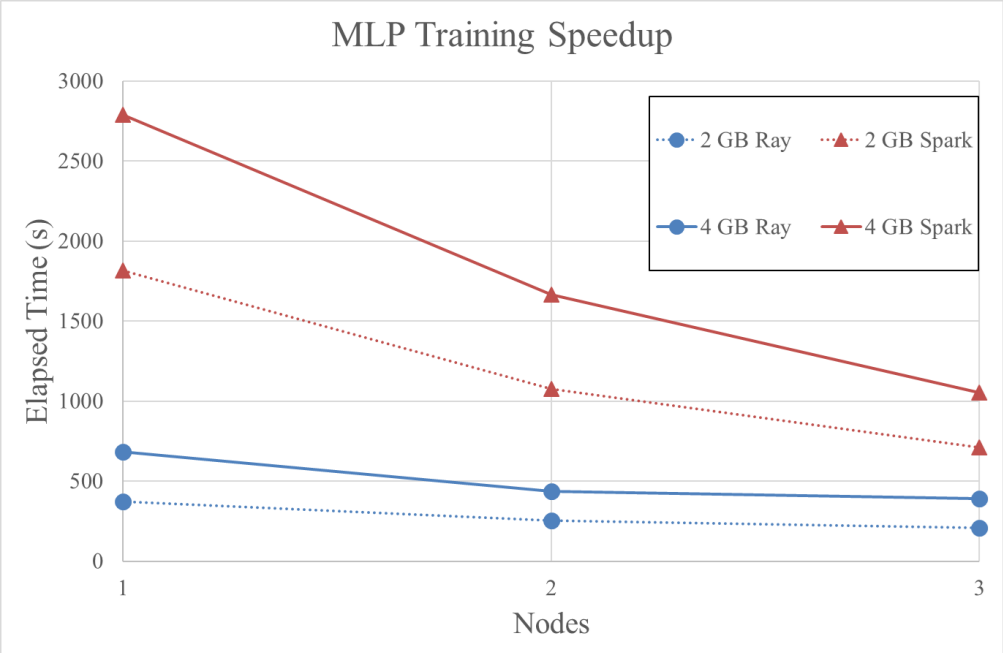

Figure 5.5.2: Speedup of MLP Training (Training times are depicted)

Comparison

As expected, Spark consistently showed faster preprocessing times compared to Ray. But Ray outperformed Spark by a lot in terms of training time across all input sizes and node configurations. This suggests that Ray's training loop and use of TorchTrainer were more efficient for the MLP training task. That is why despite Ray's longer preprocessing time, it achieved a lower total times compared to Spark. Regarding scaling, It seems Spark's scales better both with increasing input size and the addition of nodes to the cluster, but given the large starting gap between the 2 frameworks, it is obvious that Ray is the go-to choice for MLP training considering speed.

It should be mentioned that Ray exhibited the aforementioned high memory usage and the speedup for the 4GB and 8GB datasets when adding the 3rd node to the cluster was insignificant due to memory constraints. This is supported by the fact that after reducing the number of CPUs in the cluster to 10 (instead of 12), Ray took a similar amount of time to train the MLP. However, no out-of-memory errors were thrown this time, the training was just slowed down.

Lastly, as expected, both trained models performed almost the same regarding the following metrics:

- **Accuracy**- Ray: 93.1%, Spark: 92.3%
- **Precision**- Ray: 92.4%, Spark: 94.2%
- **Recall**- Ray: 93.5%, Spark: 91.7%

5.5.2 XGBoost Training

The XGBoost model was trained using a similar distributed setup. The key hyperparameters used for this XGBoost model were:

- Learning rate: 0.2
- Max depth: 5
- Number of parallel trees: 50
- Number of boosting rounds: 5

The steps involved in training the XGBoost model include:

1. **Data Preparation**: The dataset was loaded and preprocessed similarly to the MLP training.

2. **Preprocessing**: Features were concatenated into a single column, and a MinMaxScaler was used to normalize the feature values.

3. **Training and Evaluation**: The dataset was split into training and validation sets, and was trained using the Ray's `XGBoostTrainer` and Spark's `XGBoostClassifier` class from the `sparkxgb` library. The model was then evaluated on the validation set.

The results of training the XGBoost model on Ray and Spark are presented below:

	Ray			**Spark**		
Input Size (GB)	**Pre (s)**	**Train (s)**	**Total (s)**	**Pre (s)**	**Train (s)**	**Total (s)**
1	56.3	**61.4**	117.7	22.1	334.3	356.4
2	105.6	**90.6**	196.2	35.2	608.3	643.5
4	173.5	**186.2**	359.7	47.9	757.3	805.2
8	342.9	**442.9**	785.8	67.5	1462.9	1530.4

Table 5.24: XGBoost Training results on 3 nodes

	Ray			**Spark**		
Input Size (GB)	**Pre (s)**	**Train (s)**	**Total (s)**	**Pre (s)**	**Train (s)**	**Total (s)**
1	64.9	**91.9**	156.8	27.8	477.5	505.3
2	159.2	**119.1**	278.3	40.2	742.7	782.9
4	261.8	**217.2**	479.0	53.0	1067.0	1120.0
8	503.8	**459.7**	963.5	93.8	1880.1	1973.9

Table 5.25: XGBoost Training results on 2 nodes

	Ray			**Spark**		
Input Size (GB)	**Pre (s)**	**Train (s)**	**Total (s)**	**Pre (s)**	**Train (s)**	**Total (s)**
1	100.4	**150.2**	250.6	46.1	710.0	756.1
2	221.2	**189.3**	410.5	66.4	1214.1	1280.5
4	385.8	**397.9**	783.7	97.3	1874.9	1927.2
8	834.1	**613.8**	1447.9	179.4	3389.0	3568.4

Table 5.26: XGBoost Training results on 1 node

Visualizing the scalability of the 2 frameworks as the input size increases on a 2-node cluster:

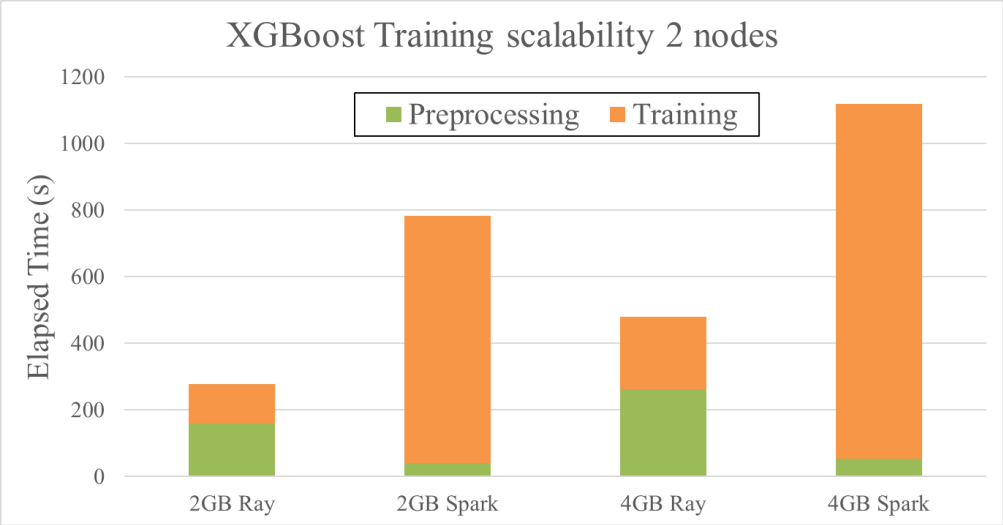

Figure 5.5.3: XGBoost preprocessing and training times

Visualizing the speedup (elapsed time reduction) of the 2 frameworks as the number of nodes increases for a given input size:

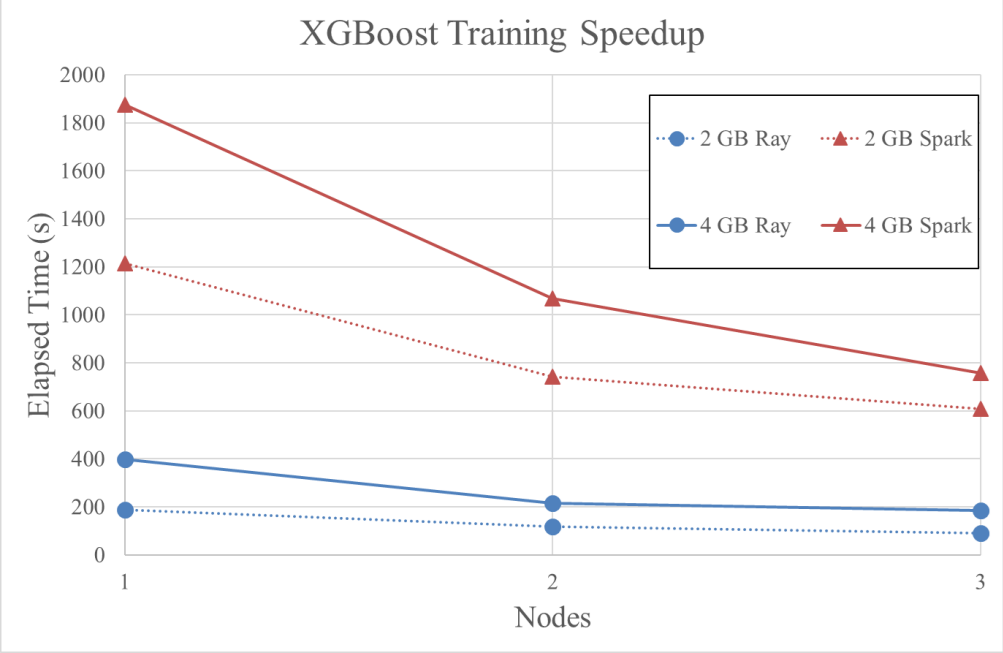

Figure 5.5.4: Speedup of XGBoost Training (Training times are depicted)

Comparison

The observations are the same as before. Ray consistently shows faster execution times for training compared to Spark across all input sizes and node configurations. Spark's strong point is its data preprocessing, while Ray excels in XGBoost's training. Although Spark's scaling ability seems better, given the difference in performance Ray also exhibits impressive scalability. The same problem with Ray's memory usage is observed here too for the largest datasets.

Both models performed almost the same regarding the following metrics:

- **Accuracy**- Ray: 95.7%, Spark: 96.9%
- **Precision**- Ray: 96.1%, Spark: 97.5%
- **Recall**- Ray: 95.5%, Spark: 96.4%

5.6 Hyperparameter Tuning Experiments

5.6.1 Hyperparameter Tuning on MLP

The values of the hyperparameters used for the tuning of the MLP are presented below:

- **Batch Size**: [512, 1024]
- **Number of Neurons in Hidden Layer 1**: [64, 128]
- **Number of Neurons in Hidden Layer 2**: [64, 128]

The results of hyperparameter tuning for the MLP on Ray and Spark are presented below.

	Ray		Spark	
Input Size (GB)	**Tuning (s)**	**max PHM (MB)**	**Tuning (s)**	**max PHM (MB)**
1	**323.3**	1731	1085.6	469
2	**452.7**	2942	1632.1	574
4	**786.1**	3291	2849.3	425
8	**1579.0**	3173	4510.7	630

Table 5.27: MLP Hyperparameter Tuning results on 3 nodes

	Ray		Spark	
Input Size (GB)	**Tuning (s)**	**max PHM (MB)**	**Tuning (s)**	**max PHM (MB)**
1	**385.2**	1962	1431.9	376
2	**657.6**	2563	2327.4	424
4	**1053.4**	3010	3970.5	671
8	**1745.9**	3258	6141.2	481

Table 5.28: MLP Hyperparameter Tuning results on 2 nodes

	Ray		Spark	
Input Size (GB)	**Tuning (s)**	**max PHM (MB)**	**Tuning (s)**	**max PHM (MB)**
1	**625.3**	2321	2673.1	534
2	**1278.8**	2785	4174.8	444
4	**1789.5**	2813	6423.3	321
8	**3127.2**	2947	10632.0	627

Table 5.29: MLP Hyperparameter Tuning results on 1 node

Visualizing the scalability of the 2 frameworks as the input size increases on a 3-node cluster:

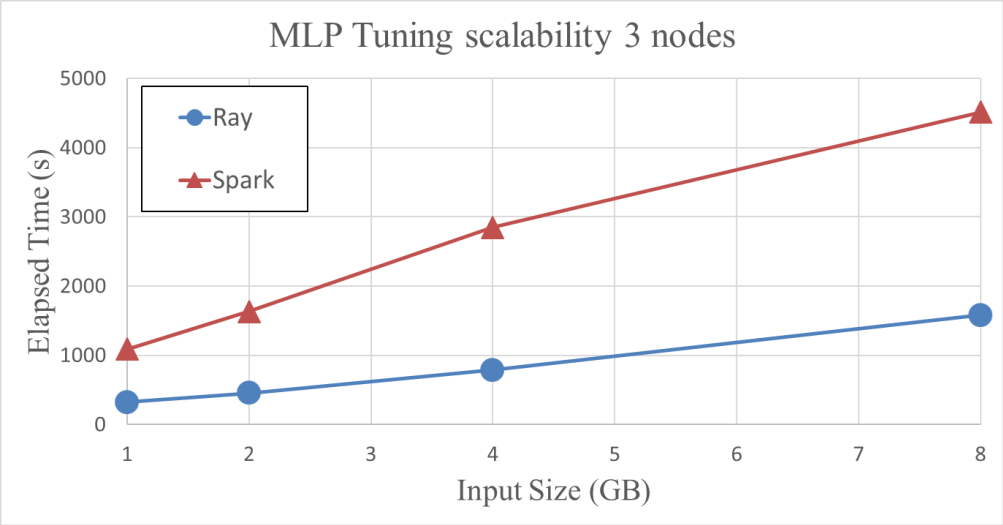

Figure 5.6.1: Scalability of MLP Tuning

Visualizing the speedup (elapsed time reduction) of the 2 frameworks as the number of nodes increases for a given input size:

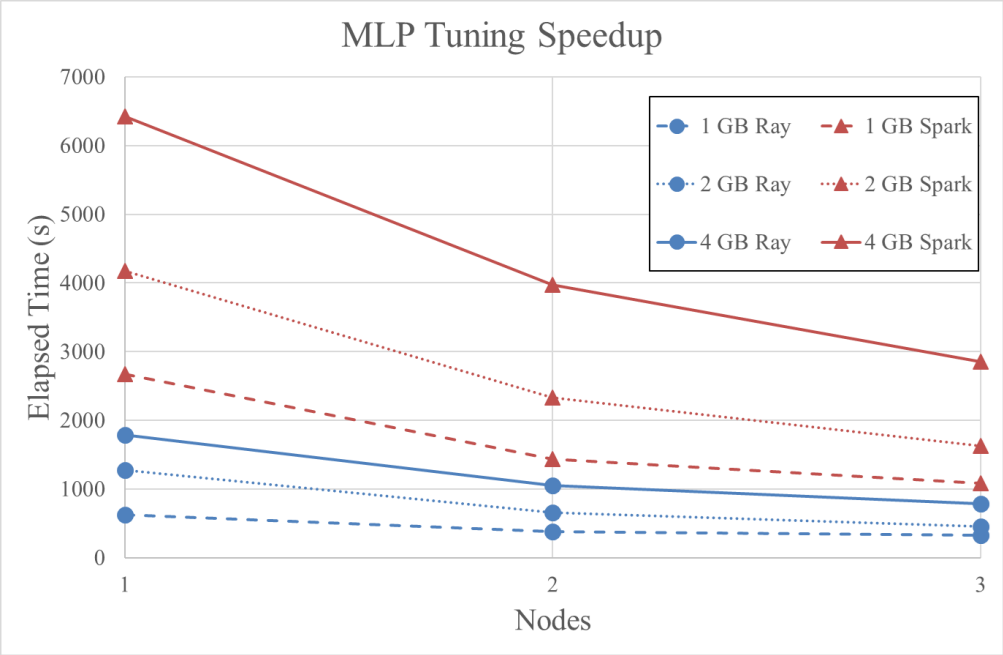

Figure 5.6.2: Speedup of MLP Tuning

Just as it happened in the training experiments, Ray overperforms Spark in tuning as well. As always it is quite more memory-hungry making it difficult to scale larger datasets. It should be noted that both programs selected 2 different combinations of hyperparameters:

- **Ray:**
  - Batch Size: 1024
  - Number of Neurons in Hidden Layer 1: 128
  - Number of Neurons in Hidden Layer 2: 128

- **Spark:**
  - Batch Size: 512
  - Number of Neurons in Hidden Layer 1: 256
  - Number of Neurons in Hidden Layer 2: 128

They achieve, however, similar accuracies (Ray: 95.7%, Spark: 97.1%).

5.6.2 Hyperparameter Tuning on XGBoost

The values of the hyperparameters used for the tuning of the XGBoost model are presented below:

- **Learning Rate**: [0.1, 0.2]
- **Maximum Depth**: [5, 7]
- **Number of Parallel Trees**: [15, 30]

The results of hyperparameter tuning for the XGBoost on Ray and Spark are presented below.

	Ray		Spark	
Input Size (GB)	**Tuning (s)**	**max PHM (MB)**	**Tuning (s)**	**max PHM (MB)**
1	**205.1**	2037	857.3	286
2	**320.7**	2589	1363.2	362
4	**550.2**	2784	2298.6	321
8	**1008.9**	2690	2748.0	431

Table 5.30: XGBoost Hyperparameter Tuning results on 3 nodes

	Ray		Spark	
Input Size (GB)	**Tuning (s)**	**max PHM (MB)**	**Tuning (s)**	**max PHM (MB)**
1	**270.7**	1675	1232.8	371
2	**415.1**	2385	1989.2	345
4	**837.4**	2634	2992.4	267
8	**1263.8**	3023	4678.9	493

Table 5.31: XGBoost Hyperparameter Tuning results on 2 nodes

	Ray		Spark	
Input Size (GB)	**Tuning (s)**	**max PHM (MB)**	**Tuning (s)**	**max PHM (MB)**
1	**432.9**	2201	2049.5	342
2	**692.7**	2622	3112.0	296
4	**1355.8**	2934	5317.9	415
8	**2229.6**	3105	8234.7	473

Table 5.32: XGBoost Hyperparameter Tuning results on 1 node

Visualizing the scalability of the 2 frameworks as the input size increases on a 3-node cluster:

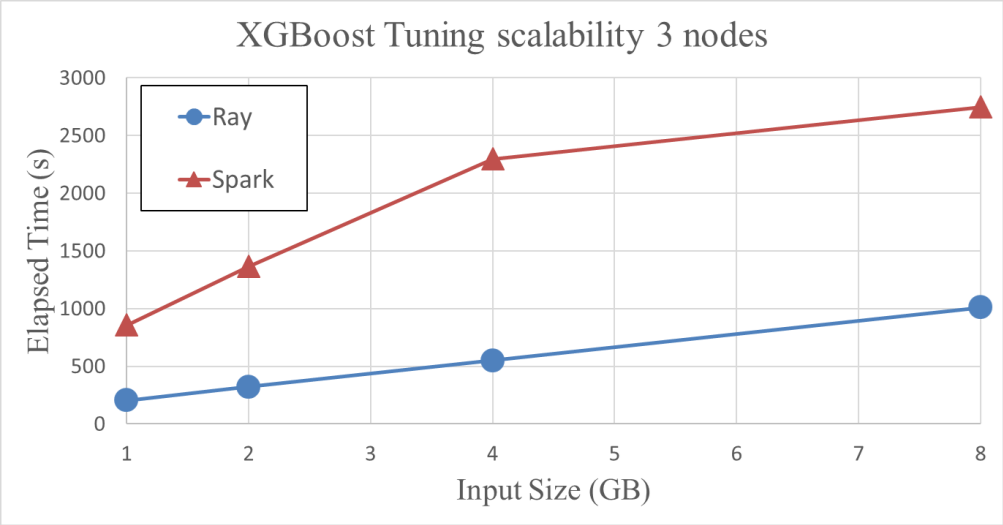

Figure 5.6.3: Scalability of XGBoost Tuning

Visualizing the speedup (elapsed time reduction) of the 2 frameworks as the number of nodes increases for a given input size:

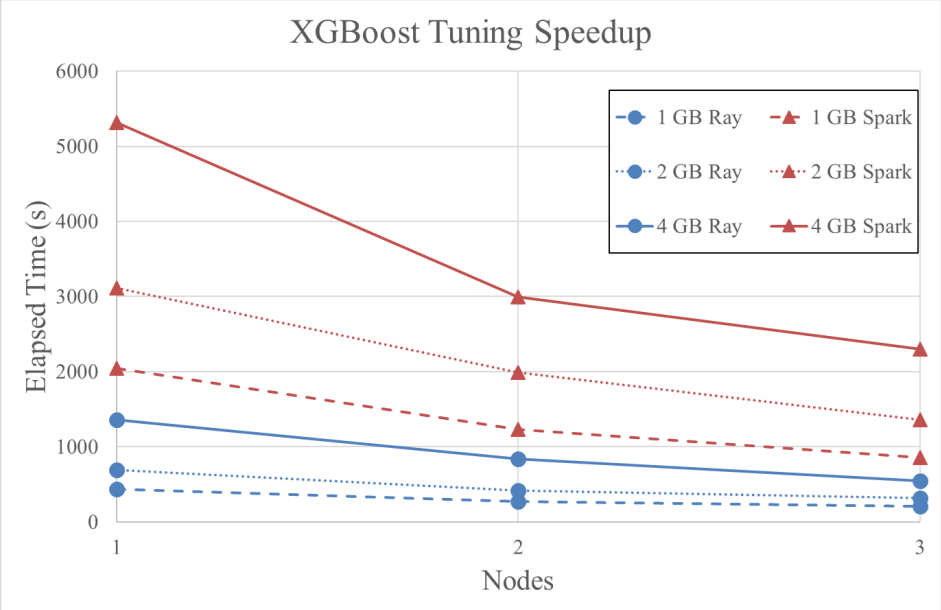

Figure 5.6.4: Scalability of XGBoost Tuning

The conclusion is the same as the one in the previous experiment. Nevertheless, in this experiment the 2 programs choose the same combination of hyperparameters:

- Learning Rate: 0.1
- Maximum Depth: 7
- Number of Parallel Trees: 30

They achieve comparable accuracies:

- Ray: 97.2%

- Spark: 96.6%

5.6.3 Overall Comparison on Model Training and Tuning

In the model training and tuning experiments, Ray almost always outperformed Spark in terms of speed, demonstrating its efficiency. It seems Ray Tune and Ray Train are extremely well optimized, and this could be attributed to Ray being a lower-level framework and giving more freedom to developers for optimizing ML libraries, such as PyTorch and XGBoost. It is also a Python-native framework (Spark's main language is Scala). Nevertheless, as we constantly underline in this thesis, Ray utilizes a lot more memory than Spark, and training and tuning tasks are no exceptions. But overall, Ray's faster training times and efficient hyperparameter tuning make it a more suitable choice for these tasks.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

This thesis presented a comprehensive comparative study of Ray and Apache Spark, two prominent distributed computing frameworks, focusing on their performance in ETL tasks, graph operations, and machine learning training and tuning. The experimental results demonstrated distinct advantages and limitations for each framework in different contexts.

For ETL tasks, Apache Spark consistently outperformed Ray in terms of execution time, memory efficiency, and scalability. Spark's robust ecosystem and optimized libraries make it a superior choice for large-scale data processing and complex transformations. Ray, while demonstrating potential in simpler tasks and smaller datasets, struggled with memory-intensive operations, highlighting its higher memory consumption.

In graph operations, Spark's GraphX library showcased superior performance. Ray's limitations in memory management and lack of specialized graph processing libraries resulted in slower execution times and higher memory usage, making it less suitable for large-scale graph analytics.

For training of machine learning models and hyperparameter tuning, Ray demonstrated impressive capabilities. Ray's `TorchTrainer` and `XGBoostTrainer` provided efficient and scalable solutions for distributed ML tasks, outperforming Spark in training times. However, Spark's preprocessing efficiency and lower memory usage remained significant advantages.

Overall, the study revealed that Spark excels in intensive data analytics tasks, while Ray shines in distributed machine learning, making them complementary tools. Combining their strengths can leverage efficient data processing with Spark and scalable ML training with Ray, providing a powerful approach for comprehensive big data and AI projects.

6.2 Future Directions

Future work can explore several avenues to build on the findings of this thesis:

- **Integration of Ray and Spark:** Developing seamless integration between Ray and Spark can harness the strengths of both frameworks. This includes optimizing data transfer and resource management when using Spark for preprocessing and Ray for ML training within a single pipeline.
- **Memory Management Improvements:** Investigating advanced memory management techniques for Ray can address its high memory usage and enhance its performance in memory-intensive tasks. This may involve optimizing the Object Store or developing more efficient shuffling mechanisms.
- **Scalability Studies with Larger Clusters:** Extending the experiments to larger clusters with more nodes can provide deeper insights into the scalability limits and performance characteristics of both frameworks.

- **Exploring Other Distributed Frameworks:** Including other distributed computing frameworks like Dask, Flink, or Kubernetes-based solutions can provide a broader perspective and identify the best tools for specific types of big data and ML workloads.

Chapter 7

Bibliography

[1] Armbrust, M. et al. "Spark SQL: Relational data processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 1383–1394.

[2] Chen, T. and Guestrin, C. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (2016), pp. 785–794.

[3] Crankshaw, D. et al. "Ray Serve: A Scalable and Programmable Serving System for Machine Learning Models". In: *Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing*. 2020.

[4] Dean, J. and Ghemawat, S. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[5] Developers, R. *Memory Management and Out of Memory Errors*. [Online; accessed 12-July-2024]. 2023. URL:

[6] Documentation, R. *Shuffling Data*. Accessed: 2024-07-12. 2024. URL:

[7] Feurer, M. and Hutter, F. "Hyperparameter optimization". In: *Automated Machine Learning* (2019), pp. 3–33.

[8] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016.

[9] GRNET. *Okeanos-Knossos IaaS Service*. 2024. URL:

[10] Hadoop, A. *Hadoop Distributed File System*. Accessed: 2024-07-06. 2024. URL:

[11] Hadoop, A. *Yet Another Resource Negotiator (YARN)*. Accessed: 2024-07-06. 2024. URL:

[12] IBM and Project, R. *IBM and Ray Collaboration on Distributed Computing and AI Models*. Accessed: 2024-07-18.

[13] Javatpoint. *Apache Spark Architecture*. Accessed: 2024-07-06. 2024. URL:

[14] Liang, E. et al. "RLlib: Abstractions for distributed reinforcement learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. 2018, pp. 3053–3062.

[15] Liaw, R. et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018).

[16] Meng, X. et al. "MLlib: Machine learning in Apache Spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[17] Moritz, P. et al. "Ray: A distributed framework for emerging AI applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 561–577.

[18] Page, L. et al. "The PageRank citation ranking: Bringing order to the web". In: *Stanford InfoLab* 1999-66 (1999), pp. 1–17.

[19] Project, R. *Overview of Ray*. Accessed: 2024-07-18.

[20] Rosenblatt, F. "Principles of neurodynamics. perceptrons and the theory of brain mechanisms". In: *Spartan Books* (1961).

[21] Schank, T. and Wagner, D. "Finding, counting and listing all triangles in large graphs, an experimental study". In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2005, pp. 606–609.

[22] Shvachko, K. et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2010, pp. 1–10.

[23] SNAP. *Stanford Large Network Dataset Collection*. 2024.

[24] Spark, A. *Apache Spark - Unified Analytics Engine for Big Data*. Accessed: 2024-07-06. 2024. URL:
[25] Vavilapalli, V. K. et al. "Apache Hadoop YARN: yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, pp. 1–16.
[26] Xin, R. S. et al. "GraphX: A resilient distributed graph system on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. ACM. 2013, p. 2.
[27] Zaharia, M. et al. "Spark: Cluster computing with working sets". In: *HotCloud* 10.10-10 (2010), p. 95.
[28] Zaharia, M. et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. 2012, pp. 2–2.
[29] Zaharia, M. et al. "Discretized streams: Fault-tolerant streaming computation at scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 423–438.
[30] Zheng, L. et al. "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 559–578. ISBN: 978-1-939133-28-1. URL: