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

Επιβεβαιώσιμη Κατανεμημένη Επεξεργασία στο **Apache Spark**

Διπλωματικη Εργασια

του

Βουλγαρίδη Ιωάννη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εργαστηριο Επιστημης Υπολογιστων
Αθήνα, Μάρτιος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Επιστήμης Υπολογιστών

Επιβεβαιώσιμη Κατανεμημένη Επεξεργασία στο **Apache Spark**

Διπλωματικη Εργασια

του

Ιωάννη Βουλγαρίδη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Μαρτίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2022

(Υπογραφή)

.....

Ιωαννης Βουλγαριδης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της εργασίας αυτής είναι η μελέτη και η υλοποίηση ενός συστήματος κατανεμημένης επεξεργασίας δεδομένων το οποίο θα παρέχει στον ενδιαφερόμενο χρήστη πιστοποίηση ότι τα αποτελέσματα που έλαβε από την συστάδα των υπολογιστών-κόμβων είναι σωστά. Ένα τέτοιο πρωτόκολλο βρίσκει εφαρμογή σε σενάρια όπως αυτό ενός αποκεντρωμένου cluster ο οποίος συντηρείται από κάποιο ίδρυμα ή πανεπιστήμιο και επιτρέπει στον καθένα να συνεισφέρει πόρους προς εξυπηρέτηση εργασιών τρίτων- προκειμένου να πετύχει καλύτερη κλιμάκωση- παρέχοντας κάποιο κίνητρο για την συμμετοχή. Όπως είναι φυσικό, εφόσον ο καθένας μπορεί να συμμετέχει, κανείς δεν μπορεί να εγγυηθεί για το πώς θα ενεργήσει ο εκάστοτε κόμβος. Για το λόγο αυτό είναι απαραίτητο να υπάρχει μιας μορφής επαλήθευση όσων υπολογίζονται εντός της συστάδας. Η προσέγγιση που παρουσιάζεται σε αυτή την εργασία είναι η ενσωμάτωση ενός μηχανισμού consensus στο Apache Spark, ένα από τα γνωστότερα εργαλεία κατανεμημένης εργασίας ανοιχτού κώδικα.

Λέξεις Κλειδιά

Apache Spark, Κατανεμημένα Συστήματα, Επιβεβαιώσιμη Επεξεργασία

Abstract

The purpose of this thesis is to design and implement a verifiable distributed computation system which can assure any interested party that the final outcome of a computation is correct. One possible application of such system is the case of a cluster which is maintained by a central authority but allows anyone to contribute a fraction of their resources to achieve better scaling, and offers an incentive for participation. It is obvious that allowing anyone to participate in the cluster results to being unable to guarantee whether an arbitrary participant is malicious or not. Thus, it is necessary to validate any occurring results before accepting them. This thesis approaches this issue by integrating a consensus mechanism in Apache Spark, one of the most established open source frameworks for distributed programming.

Keywords

Distributed Systems, Apache Spark, Verifiable computing

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον κ. Νεκτάριο Κοζύρη που μου εμπιστεύτηκε ένα τοσο ενδιαφέρον και διδακτικό θέμα, την κα. Κατερίνα Δόκα και τον κ. Ιωάννη Μυτιλήνη για την πολύτιμη συνεργασία, την βοήθεια και τις ιδέες τους πάνω στο θέμα. Θα ήθελα επίσης να ευχαριστήσω τους γονείς μου, Ρόη και Κώστα, τον αδερφό μου Αχιλλέα, την αγαπημένη μου γιαγιά Ευανθούλα και την υπόλοιπη οικογένεια μου για την διαρκή στήριξη που μου παρείχαν όλα αυτά τα χρόνια. Τέλος, ένα μεγάλο ευχαριστώ στην Αθηνά, τη Μαριάννα, τον Κωνσταντίνο, το Γιώργο, το Δημήτρη και το Μάνο. Οι φίλοι είναι ένα πάντα σημαντικό στήριγμα, ο καθένας με τον δικό του ιδιαίτερο τρόπο.

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Η κατανεμημένη επεξεργασία είναι μια μεθοδολογία η οποία χρησιμοποιείται σε πληθώρα εφαρμογών και η οποία γίνεται ολοένα και πιο αναγκαία καθώς ο όγκος των δεδομένων προς επεξεργασία μεγαλώνει με όλο και ταχύτερους ρυθμούς. Τα κατανεμημένα συστήματα ομαδοποιούν τα διαθέσιμα μηχανήματα σε συστάδες (clusters) και κάθε εργασία που εισάγεται διαμοιράζεται σε διαφορετικά μηχανήματα. Ένα τέτοιο σύστημα πετυχαίνει εύκολη κλιμάκωση εισάγοντας νέα μηχανήματα στο δίκτυο του cluster όταν υπάρχει ανάγκη για επιπλέον πόρους.

Υπάρχουν πολλά frameworks τα οποία χρησιμοποιούνται για κατανεμημένη επεξεργασία δεδομένων. Γνωστότερα είναι το Hadoop MapReduce και το Apache Spark. Τα εργαλεία χρησιμοποιούνται κατά κύριο λόγο σε clusters οι οποίοι βρίσκονται υπό μιας κεντρικής διοίκησης η οποία μπορεί να εγγυηθεί ότι όλα τα μηχανήματα είναι τίμια και δεν προκαλούν επίτηδες λάθη, παρα μόνο κάποιες περιστασιακες αποτυχίες. Το πλαίσιο αυτό περιορίζει την κλιμακωσιμώτητα των συστημάτων. Τα ανοιχτά clusters- στα οποία συνεισφέρει όποιος το επιθυμεί- είτε εθελοντικά είτε με κάποιο σκοπό (π.χ. υπολογιστικό χρόνο στον cluster) έχουν αρχίσει να κερδίζουν έδαφος διότι δίνουν την δυνατότητα σε οργανισμούς να κλιμακώσουν την ισχύ των Clusters τους οικονομικά εισάγοντας νέους κόμβους οι οποίοι δεν τους ανήκουν και ταυτόχρονα δίνεται η επιλογή σε ενδιαφερόμενους (π.χ. data scientists) να συνεισφέρουν πόρους στον cluster ούτως ώστε να μπορέσουν μετά να τον χρησιμοποιήσουν για τις δικές τους εργασίες. Το πρόβλημα σε αυτό το σενάριο είναι ότι εφόσον συμμετέχει ελεύθερα όποιος το επιθυμεί κανείς δεν μπορεί να εγγυηθεί για την ορθή και τίμια λειτουργία του κάθε κόμβου.

Σκοπός λοιπόν της εργασίας αυτής είναι η μελέτη και υλοποίηση ενός συστήματος κατανεμημένης επεξεργασίας, το οποίο επεκτείνει το Apache Spark ώστε να μπορεί να λειτουργήσει και σε κάποιον ανοιχτό cluster, χωρίς να υπάρχει εμπιστοσύνη στους συμμετέχοντες.

1.2 Προτεινόμενη Μέθοδος

Ο τρόπος που παρουσιάζεται συνοψίζεται στην ύπαρξη ενός έμπιστου μέρους του οποίου ευθύνη είναι να μοιράζει όλα tasks σε δύο κόμβους. Κάθε κόμβος υπολογίζει το αποτέλεσμα

και ένα hash που το αντιπροσωπεύει και στην συνέχεια το αποστέλλει στο έμπιστο μέρος. Το έμπιστο μέρος συγκρίνοντας τα hashes καταλαβαίνει αν τα δύο αποτελέσματα είναι ίδια και κατ΄ επέκταση αν το αποτέλεσμα είναι έμπιστο. Επειδή το να τρέχουν διπλάσια tasks στο σύστημα είναι μια μέθοδος που μπορεί να δημιουργήσει μεγάλο φόρτο, εξετάζεται και μια μέθοδος πιθανοτική, σύμφωνα με την οποία αν έχουμε πληροφορίες ότι κάποιος κόμβος δρα συστηματικά τίμια τότε κάποια tasks μπορεί να τα αναλάβει και χωρίς να τα αναλάβει άλλος για επαλήθευση.

1.3 Διάρθρωση Εργασίας

Η παρούσα Διπλωματική Εργασία διαρθρώνεται σε 6 κεφάλαια. Στο Κεφάλαιο 1, πραγματοποιείται μια εισαγωγή στο θέμα. Στο κεφάλαιο 2, γίνεται μια εισαγωγή στην επιβεβαιώσιμη επεξεργασία, αναλύοντας κάποιους βασικούς τύπους πρωτοκόλλων, τις εφαρμογές και τους περιορισμούς τους. Στην συνέχεια, στο κεφάλαιο 3 γίνεται ανάλυση της αρχιτεκτονικής του Apache Spark, πάνω στο οποίο στηρίζεται η προτεινόμενη μέθοδος που θα παρουσιαστεί στο κεφάλαιο 4. Στο κεφάλαιο 5, γίνεται πειραματική αξιολόγηση της μεθόδου με χρήση ενός demo που υλοποιήθηκε. Τέλος, στο κεφάλαιο 6, παρουσιάζονται οι μελλοντικές επεκτάσεις της παρούσας εργασίας.

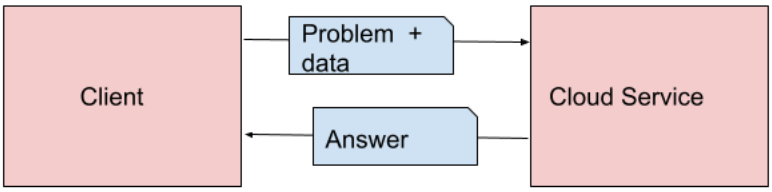
Κεφάλαιο 2

Επιβεβαιώσιμη Επεξεργασία Δεδομένων

2.1 Γενικά περί Επιβεβαιώσιμης επεξεργασίας

2.1.1 Ανάθεση Εργασίων σε υπηρεσίες τρίτων

Ένα νέο μοντέλο υπολογισμού που έκανε την εμφάνιση μαζί με την εμφάνιση του υπολογιστικού νέφους (cloud) - και μάλιστα έχει επικρατήσει σε πολλές εφαρμογές - είναι το μοντέλο της ανάθεσης εργασίων σε υπηρεσίες τρίτων (outsourcing). Σύμφωνα με το μοντέλο αυτό, ένας πελάτης επιλέγει να μην τρέξει τοπικά ένα μέρος των εργασιών του, αλλά προτιμά να τις αναθέσει σε μια έμπιστη υπηρεσία τρίτου. Ο πελάτης αρχικά πρέπει να στείλει στα μηχανήματα της outsourcing υπηρεσίας τα εκτελέσιμα και τα αρχεία εισόδου. Στα μηχανήματα της υπηρεσίας αρχίζει να τρέχει ο κώδικας του πελάτη και όταν ολοκληρωθεί, το αποτέλεσμα αποστέλεται ξανά στον πελάτη ο οποίος το καταναλώνει κάτα το δοκούν. Ας τονιστεί εδώ ότι ο πελάτης δεν έχει εικόνα τί γίνεται εντός της υπηρεσίας του παρόχου, βλέπει δηλαδή ένα μαύρο κουτί που θα του δώσει το αποτέλεσμα που αντιστοιχεί στο πρόβλημα εισόδου. Ο κώδικας του πελάτη μπορεί να τρέχει με τρόπο κατανεμημένο σε κάποιο cluster που ανήκει στον πάροχο, ή σε ένα μηχάνημα να τρέχουν παράλληλα εργασίες πολλές πελατών.



Σχήμα 2.1: Outsourcing Computation

Το παραπάνω σενάριο είναι πάρα πολύ σύνηθες, αφού εξυπηρετεί τόσο μεμονομένους (ιδιώτες) πελάτες οι οποίοι δεν έχουν τους πόρους για να τρέξουν τις εργασίες τους, όσο και

μεγάλους εταιρικούς πελάτες οι οποίο επιλέγουν να μην αγοράσουν τα δικά τους μηχανήματα διότι θεωρούν ότι θα είναι πιο φθηνό να χρησιμοποιήσουν ένα pay-as-you go μοντέλο για να εργασίες τους. Η λύση αυτή πέρα από λιγότερο κόστος (σε κάποιες περιπτώσεις) παίρνει την ευθύνη της συντήρησης των μηχανημάτων στην μεριά του παρόχου και κάνει την διαδικασία ευκολότερη για εταιρείες οι οποίες δεν έχουν τεχνογνωσία σε αυτό τον τομέα ή που θέλουν να μειώσουν το time-to-market του λογισμικού τους. Πολλές εταιρείες προσφέρουν τέτοιες υπηρεσίες με γνωστότερες το AWS και το Azzure.

2.1.2 Κατηγοριοποίηση πρωτοκόλλων επιβεβαιώσιμης επεξεργασίας

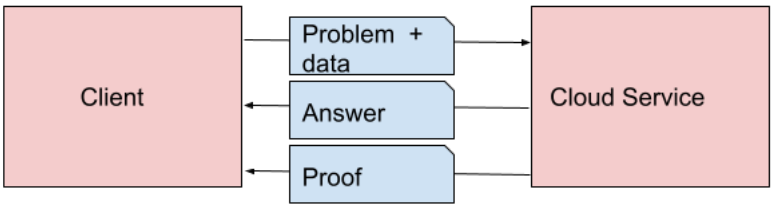
Στην παραπάνω περιγραφή του μοντέλου ανάθεσης εργασιών σε τρίτους έγινε η υπόθεση της έμπιστης υπηρεσίας η οποία αναλαμβάνει τον υπολογισμό. Εγείρεται λοιπόν η ερώτηση κατά πόσο μπορεί μια υπηρεσία τρίτου να θεωρηθεί έμπιστη. Υπάρχουν πολλά σενάρια κατά τα οποία κάποιος μπορεί να θελήσει να εξαπατήσει έναν πελάτη ο οποίος του ανέθεσε μια εργασία και τα κίνητρα για την πράξη αυτή μπορούν να κατηγοριοποιηθούν - μεταξύ άλλων - και σε οικονομικά. Τέτοια παραδείγματα είναι η περίπτωση κατά την οποία η υπηρεσία αναλαμβάνει επί πληρωμή την εργασία και ούσα κακόβουλη επιλέγει να μην τρέξει την εργασία και να δώσει στον πελάτη ένα τυχαίο αποτέλεσμα με σκοπό να μην καταναλώσει πόρους και όλο το ποσό της πληρωμής να είναι καθαρό κέρδος. Πέραν των οικονομικών κινήτρων, υπάρχουν και άλλα σενάρια όπως αυτό στο οποίο η κακόβουλη υπηρεσία επιθυμεί να δώσει σε κάποιον στοχευμένα - και ενδεχομένως στοχευμένα σε κάποιου είδους εργασίας - λανθασμένο αποτέλεσμα. Το παραπάνω πρόβλημα γίνεται καλύτερα αντιληπτό στην περίπτωση που η ανάθεση δεν γίνεται σε κάποιον πάροχο αλλά γίνεται σε κάποιο μέλος ενός peer-to-peer δικτύου. Στην περίπτωση αυτή, η ανωνυμία εντός του δικτύου ενισχύει την υιοθέτηση μιας πιο αυστηρής πολιτικής σχετικά με το ποιον μπορεί κάποιος να θεωρήσει έμπιστο.

Μια άλλη συνιστώσα η οποία κάνει ακόμα πιο σημαντικό το πρόβλημα είναι ότι κατά μεγάλη πιθανότητα, αυτός ο οποίος αναθέτει την εργασία του σε τρίτο είτε δεν έχει τους πόρους για να την εκτελέσει τοπικά είτε ο απαιτούμενος χρόνος εκτέλεσης με του ίδιους πόρους είναι ασύμφορα μεγάλος με αποτέλεσμα ο πελάτης να μην μπορεί να ελέγξει το αποτέλεσμα. Σε αυτό το σημείο εισάγεται η έννοια της επιβεβαιώσιμης επεξεργασίας. Από εδώ και στο εξής, η μη- έμπιστη πλέον υπηρεσία η οποία αναλαμβάνει τον υπολογισμό θα πρέπει να στέλνει μαζί με το αποτέλεσμα και μια απόδειξη ότι η απάντηση είναι σωστή για το πρόβλημα αυτό με την συγκεκριμένη είσοδο, όπως φαίνεται και στην παρακάτω εικόνα (1.2).

Σε όλα τα μοντέλα επιβεβαιώσιμης επεξεργασίας πρέπει να πληρούνται οι εξής προυποθέσεις:

- Μια τίμια - καλόβουλη υπηρεσία που αναλαμβάνει την εκτέλεση της εργασίας ενός πελάτη μπορεί πάντα να τον πείσει ότι το αποτέλεσμα είναι σωστό.
- Στην αντίθετη περίπτωση, ο πελάτης κατά μεγάλη πιθανότητα θα εντοπίσει την απόπειρα παραπλάνησης του.

Υπάρχουν και κάποιοι επιπλέον περιορισμοί στο μοντέλο αυτό, οι οποίοι αφορούν τους πόρους:



Σχήμα 2.2: Outsourcing verifiable computation

- Η διαδικασία κατασκευής της απόδειξης δεν θα πρέπει να προσθέτει πολύ επιπλέον υπολογιστικό κόστος πέραν του υπολογισμού του αποτελέσματος.
- Η διαδικασία ανάγνωσης της απόδειξης και απόφασης για την εγκυρότητα της δεν θα πρέπει να επιβαρύνει τον πελάτη πιο πολύ από το να διαβάσει απλά την είσοδο (σε πολυπλοκότητα).

Τα πρωτόκολλα επιβεβαιώσιμης επεξεργασίας μπορούν να καταταγούν σε αυτά που χρησιμοποιούν οικονομικά κίνητρα για την αποτροπή της εξαπάτησης, κάποια εκ των οποίων αξιοποιούν και την τεχνολογία του Blockchain και σε αυτά που χρησιμοποιούν μαθηματικούν και πιο συγκεκριμένα κρυπτογραφικούς αλγόριθμους για την πιστοποίηση των αποτελέσμάτων και την αποτροπή εξαπάτησης.

2.2 Truebit

Το Trubebit είναι ένα πρωτόκολλο επιβεβαιώσιμης επεξεργασίας το οποίο αξιοποιεί την τεχνολογία του Blockchain, και συγκεκριμένα είναι χτισμένο πάνω στο Ethereum και βασίζεται στην οικονομική παρότρυνση των συμμετεχόντων να πράττουν έντιμα. Στο πρωτόκολλο αυτό κάποιος μπορεί να συμμετάσχει ως Πελάτης, Λύτης, Επικυρωτής, Κριτής και μάλιστα μπορεί να υιοθετήσει περισσότερους από έναν ρόλο την ίδια στιγμή. Πιο αναλυτικά για κάθε ρόλο ισχύει:

- Ο Πελάτης είναι αυτός ο οποίος επιθυμεί να αναθέσει μια εργασία σε κάποιον τρίτο στον οποίο θα καταβάλλει και ένα αντίτιμο ικανό να καλύψει το κόστος των πόρων που απαιτούνται συν ένα ποσό κέρδος το οποίο να αποτελεί κίνητρο για λύτες.
- Ο λύτης είναι όποιος διαθέτει πόρους προς εξυπηρέτηση εργασιών τρίτων. Όταν ολοκληρώσει επιτυχώς και σωστά μια εργασία αμοίβεται με κάποιο ποσό το οποίο αντιστοιχεί στους πόρους που κατανάλωσε συν ένα ποσό το οποίο θεωρείται το κέρδος του. Σε περίπτωση που βρεθεί λάθος στο αποτέλεσμα του λύτη τότε αυτός είναι υποχρεωμένος να καταβάλλει αποζημίωση σε όλους όσους κατανάλωσαν επιπλέον πόρους για την ανίχνευση και διόρθωση του λάθους.

- Ο επικυρωτής είναι αυτός ο οποίος προσπαθεί να βρεί λάθη σε υπολογισμούς που έγιναν από λύτες. Αμοίβεται σε περίπτωση που όντως βρεθεί λάθος αλλιώς είναι υποχρεωμένος - όπως και ο λύτης - να καταβάλλει αποζημίωση σε όσους συμμετείχαν στην διαδικασία εξέτασης του αποτελέσματος.

- Ο Κριτής αντιπροσωπεύει το έμπιστο σώμα το οποίο αποφασίζει ποιος έχει κάνει σωστό υπολογισμό στην περίπτωση διαφωνίας του λύτη με κάποιον Επικυρωτή. Το σώμα των κριτών αποτελείται από αυτούς που συμμετέχουν στο mining του Ethereum. Οι κριτές έχουν αυστηρό περιορισμό στους διαθέσιμους πόρους, αφού ό,τι τρέχουν το τρέχουν πάνω στο Blockchain, το οποίο θέτει περιορισμούς ως προς το μέγεθος των δεδομένων και την πολυπλοκότητα. Για τον λόγο αυτό, σε περίπτωση διαφωνίας λύτη- επικυρωτή, θα πρέπει να βρεθεί η ρίζα της διαφωνίας, να περιοριστεί η διαφορά σε ένα κομμάτι τόσο μικρό (π.χ. σε ένα βήμα ενός βρόχου) το οποίο να μπορούν οι κριτές να τρέξουν on chain και να αποφανθούν για το ποιος έχει δίκιο.

  Ο τρόπος με τον οποίο περιορίζεται η διαφωνία των εμπλεκόμενων μερών αλλά και ο τρόπος με τον οποίο γίνονται οι πληρωμές θα αναληθούν αναλυτικότερα στην συνέχεια.

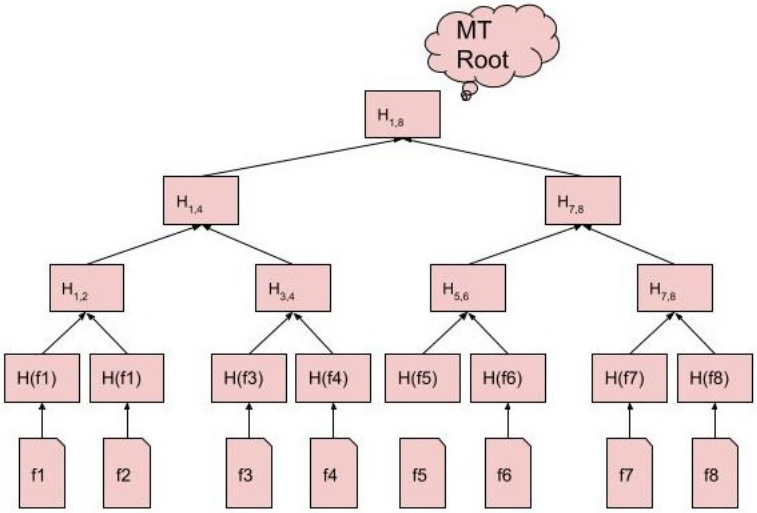
2.2.1 Χρησιμές έννοιες που χρησιμοποιούνται στο Truebit

Merkle Tree

Το δέντρο Merkle είναι μια δομή η οποία μπορεί να αναπαρασταθεί γραφικά ώς ένα κανονικό δυαδικό δέντρο. Το χαρακτηριστικό του είναι ότι όλη η πληροφορία βρίσκεται αποθηκευμένη στα φύλλα του και κάθε κόμβος ο οποίος δεν είναι φύλλο έχει αποθηκευμένο το hash των τιμών των παιδιών του. Επαγωγικά, η ρίζα του δέντρου θα έχει αποθηκευμένη μόνο μια τιμή η οποία θα έχει προκύψει με τον παραπάνω τρόπο και είναι προφανές ότι η τιμή του hash της ρίζας επηρεάζεται από όλα τα δεδομένα που βρίσκονται στα φύλλα και είναι κατά κάποιο τρόπο αντιπροσωπευτικό όλης της πληροφορίας. Από τις ιδιότητες των συναρτήσεων κατακερματισμού, είναι γνωστό ότι είναι πρακτικά αδύνατο να βρεθούν δύο τιμές $x_1 \neq x_2$ οι οποίες να έχουν τις ίδιες τιμές κατακερματισμού. Αν λοιπόν, για δυό block δεδομένων φτιαχτούν τα αντίσοιχα Merkle δέντρα, συγκριθούν τα hashes των ριζών τους και βρεθούν ίδια, τότε κατά μεγάλη -σχεδόν 1- πιθανότητα τα δύο blocks περιέχουν ακριβώς την ίδια πληροφορία. Στην περίπτωση όπου τα hashes των ριζών δεν είναι ίδια, τότε πάλι με μεγάλη πιθανότητα, η πληροφορία στα δύο δέντρα δεν είναι ίδια και με δυαδική αναζήτηση (ακολουθώντας τα διαφορετικά hashes) μπορεί να βρεθεί αποδοτικά - σε λογαριθμικό ως προς το μέγεθος των δεδομένων χρόνο - το πού διαφέρουν τα δύο blocks δεδομένων.

Τέλος, το μεγαλύτερο πλεονέκτημα των MT (το οπποίο χρησιμοποιείται και στο Truebit) είναι η δυνατότητα να επαληθευτεί η ακεραιότητα δεδομένων (π.χ. ένός αρχείου) ή διαφορετικά να αποδειχθεί ότι ένα κομμάτι δεδομένων ανήκει πραγματικά στο MT ότι είναι δηλαδή ένα φύλλο του δέντρου. Ο παραπάνω έλεγχος γίνεται σε λογαριθμικό

χρόνο και χώρο και απαιτεί την λήψη λογαριθμικών - ως προς το πλήθος φύλλων του δέντρου - Merkle Hashes πέρα από την λήψη των πραγματικών δεδομένων. Για να εξηγηθεί η παραπάνω διαδικασία θεωρείται ένα σύνολο 8 στοιχείων από τα οποία προκύπτει ένα MT με ισάριθμα (8) φύλλα, αρχικά εφαρμόζοντας μια συνάρτηση κατακερματισμού (π.χ. SHA256) στα αρχικά δεδομένα και αποθηκεύοντας τις τιμές κατακερματισμού στα φύλλα. Στην συνέχεια υπολογίζονται οι τιμές των επόμενων κόμβων του δέντρου όπως φαίνεται στην παρακάτω εικόνα 1.3:

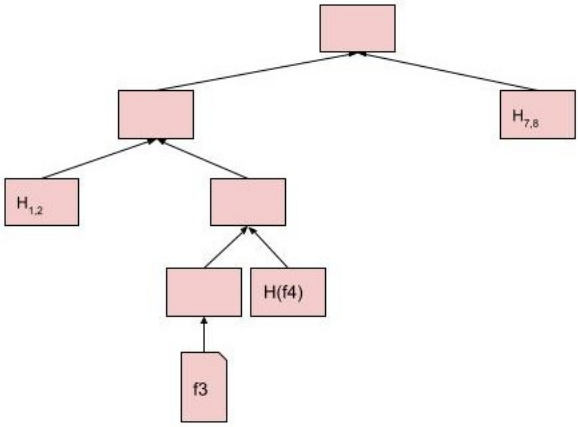


Σχήμα 2.3: Merkle Tree

όπου $H_{i,j}$ είναι το hash που περιλαμβάνει όλα τα φύλλα από το i εως το j.

Στην συνέχεια, αποθηκεύεται το MH root ενώ τα αρχεία αποστέλλονται σε άλλο μηχάνημα (outsourcing) μαζί με το MT και στην συνέχεια διαγράφονται από τον τοπικό δίσκο. Όταν κάποια στιγμή χρειαστεί λήψη ενός αρχείου (π.χ. του f3), τότε πέραν του πραγματικού αρχείου θα γίνει λήψη και μιας απόδειξης που πιστοποιεί ότι το αρχείο είναι όντως αυτό που νωρίτερα μεταφέρθηκε στην απομακρυσμένη αποθήκεθη. Η απόδειξη αυτή ονομάζεται ¨Απόδειξη Merkle' και αποτελείται από ένα ελάχιστο υποσύνολο hashes του MT το οποίο καθιστά δυνατή την ανακατασκευή του MT. Η απόδειξη Merkle αναπαριστάται γραφικά στην επόμενη εικόνα 1.4:

Έχοντας κάνει λήψη του αρχείου f3 είναι εύκολο να υπολογιστεί η τιμή $H(3) = H(f_3)$ και στην συνέχεια με χρήση των hashes που παρέχονται στην απόδειξη είναι δυνατό να υπολογιστεί μια τιμή για τη ρίζα του δέντρου. Αν η τιμή αυτή ταυτίζεται με την αυτή η οποία είχε νωρίτερα αποθηκευτεί, τότε το αρχείο που λάβαμε είναι το ιδιο με αυτό που είχαμε στείλει. Το συμπερασμα αυτό προκύπτει από τις ιδιότητες της συνάρτησης κατακερματισμού η οποία λέει ότι είναι υπολογιστικά πολύ δύσκολο να βρεθούν δύο διαφορετικά στοιχεία με ίδια τιμή κατακερματισμού. Το μέγεθος της απόδειξης αποτελείται δηλαδή από λογαριθμικά πολλές - ως προς το πλήθος των αρχείων - τιμές κατακερματι-



Σχήμα 2.4: Απόδειξη Merkle

σμού και το μεγεθός της είναι πολλές τάξεις μεγέθους μικρότερη από το μέγεθος ενός αρχείου σε πραγματικές συνθήκες.

Turing Machine

Η μηχανή Turing είναι μια συσκευή που ανήκει στην κατηγορία των αυτομάτων και υπάρχει κυρίως στην σφαίρα της θεωρητικής πληροφορικής. Για να οριστεί μια μηχανή Turing χρειάζεται αρχικά ένα πεπερασμένο αλφάβητο και ένα πεπερασμένο σύνολο καταστάσεων. Από εκεί και πέρα, 4 είναι τα μέρη τα οποία χρησιμοποιεί:

- Μια ταινία, η οποία είναι χωρισμένη σε κελιά (σαν τα κελιά μνήμης) και σε κάθε κελί μπορεί να γραφτεί ένα σύμβολο από το αλφάβητο. Η ταινία θεωρείται ότι έχει άπειρο μήκος υπό την έννοια ότι υπάρχει πάντα όση ταινία απαιτείται για τον εκάστοτε υπολογισμό.
- Μια κεφαλή η οποία δύναται να διαβάσει ή να γράψει στην ταινία και να μεταβεί στο αμέσως αριστερό είτε στο αμέσως δεξιότερο κελί αλλά μόνο μια από αυτές τις 4 πράξεις κάθε φορά.
- Έναν πίνακα καταστάσεων ο οποίος αποθηκεύει την κατάσταση του αυτόματου. Η κατάσταση αυτή ανήκει πάντα στο σύνολο των καταστάσεων που αναφέρεται παραπάνω.

Η πρώτη υπόθεση περί άφθονης ταινίας καθιστά την κατασκευή μιας τέτοιας μηχανής ανέφικτη, καθώς όλα τα συστηματα έχουν πεπερασμένη μνήμη. Ωστόσο, η μηχανή Turing αποτελεί μια καλή βάση για να εξηγηθούν απλά κάποιες λεπτομέρειες του Truebit. Πιο συγκεκριμένα, χρειάζεται να γνωρίζουμε την ακριβή κατάσταση του μηχανήματος μετά από εκτέλεση κάθε εντολής και η μηχανή Turing μέσω των μερών της (ταινία, κεφαλή) παρέχει αυτή τη δυνατότητα.

2.2.2 Επίπεδο επίλυσης διαφορών

Στο επίπεδο αυτό γίνεται η επίλυση των διαφωνιών ανάμεσα σε λύτες και επικυρωτές, για κάποιο συγκεκριμένο task. Η διαδικασία αυτή ονομάζεται διαφορετικά και παιχνίδι επιβεβαίωσης (Verification Game). Αρχικά, γίνεται η υπόθεση ενός υπολογισμού ο οποίος θα τρέξει σε t διακριτά βήματα και ότι ο χώρος που απαιτείται για αποθηκευτεί όλη η πληροφορία σαν να υπήρχε ένα Turing machine για τον υπολογισμό αυτό είναι s bits (για την αποθήκευση της κεφαλής, της ταινίας και της κατάστασης). Επιλέγεται ένας ακέραιος αριθμός c ο οποίος δηλώνει πόσες s-bit Turing καταστάσεις θα εισάγει ο λύτης στο Blockchain σε κάθε γύρο της διαδικασίας επίλυσης διαφορών. Τέλος, υπάρχει ένα άνω όριο για τον χρόνο μέσα στον οποίο πρέπει τα εμπλεκόμενα μέρη να απαντούν και αν αυτό δεν γίνει, τότε αυτομάτως θεωρούνται χαμένοι στην διαδικασία.

Έχοντας ορίσει τα παραπάνω ξεκινά η πρώτη φάση του Verification Game η οποία έχει σκοπό τον οσο γίνεται ακριβέστερο προσδιορισμό της πηγής διαφωνίας μεταξύ λύτη και επικυρωτή. Κατά την φάση αυτή γίνονται τα ακόλουθα:

1. Ο λύτης παράγει c Merkle Trees τα οποία περιέχουν την κατάσταση του μηχανήματος σε κάποιο βήμα (ταινία, κεφαλή, εσωτερική κατάσταση) και όπως αναφέρεται και πάνω, ο χώρος που απαιτείται για την αποθήκευση αυτής της κατάστασης είναι s bits και επομένως τα παραγόμενα Merkle δέντρα θα έχουν από s φύλλα. Τα βήματα στα οποία γίνεται αυτή η δειγματοληψία δεν είναι τυχαία αλλά προκύπτουν χωρίζοντας σε c ίσα διαστήματα τα t βήματα που είναι το σύνολο του υπολογισμού. Έτσι, τα δείγματα αυτά είναι στα βήματα
$$\frac{t}{c}, \frac{2t}{c} \cdots \frac{ct}{t}$$
.
2. Στην συνέχεια, ο επικυρωτής, δηλώνει στο Blockchain έναν αριθμό $i \leq c$ για να υποδείξει το πρώτο σημείο στο οποίο αποκλίνει ο υπολογισμός του από αυτόν του λύτη.
3. Στην συνέχεια, το σώμα των κριτών ελέγχει αν ο λύτης δημοσίευσε c Merkle Trees όπως ήταν συνφωνημένο και αν ο αριθμός i τον οποίο δημοσίευσε ο επικυρωτής καλύπτει την απίτηση $1 \leq i \leq c$. Σε περίπτωση που κάποιο μέρος δεν έχει δράσει όπως έπρεπε αυτομάτως χάνει το παιχνίδι της επιβεβαίωσης.
4. Ο επόμενος γύρος επαναλαμβάνει την παραπάνω διαδικασία αλλά για τα βήματα που βρίσκονται στο δίαστημα $[\frac{(i-1)t}{c}, \frac{it}{c}]$

Η παραπάνω διαδικασία συγκλίνει στο πρώτο βήμα υπολογισμού στο οποίο λύτης και επικυρωτής διαφωνούν. Η υπολογιστική ισχύ η οποία απαιτείται σε αυτό το σημείο για να διαπιστωθεί αν υπάρχει λάθος είναι πολύ μικρή οπότε μπορούν και οι κριτές να κάνουν τον υπολογισμό. Ευθύνη του λύτη είναι να στείλει στους κριτές αποδείξεις Merkle για κάποιες πληροφορίες κατά το βήμα της διαφωνίας (έστω e)καθώς και για το αμέσως προηγούμενο (e-1). Οι πληροφορίες αυτές περιέχουν:

- Την θέση της κεφαλής κατά το εκάστοτε βήμα.
- Το περιεχόμενο της ταινίας που δείχνει η κεφαλή κατά το βήμα και τα αμέσως αριστερότερα και δεξιότερα κελια.
- Την εσωτερική κατάσταση κατά το βήμα.

Οι κριτές στην συνέχεια, ελέγχουν την εγκυρότητα τον αποδείξεων επαληθεύοντας με τα ήδη δημοσιευμένα στο Blockchain. Αν κάποιο μονοπάτι δεν είναι σωστό ο λύτης χάνει αμέσως.

Τέλος, οι κριτές έχοντας τον κώδικα μηχανής, τρέχουν τον κώδικα από το βήμα e-1 στο e και αποφάνονται σχετικά με το αν έχει δίκιο ο λύτης ή ο επικυρωτής.

Αξίζει να αναφερθεί ότι παρόλο που παραπάνω γίνονται αναφορές σε στοιχεία που αφορούν Turing Machines η πρακτική υλοποίηση χρησιμοποιεί την αρχιτεκτονική Google Lanai για να προσομοιάσει τέτοια συμπεριφορά και οι αναφορές αυτές γίνονται για ευκολία συνεννόησης.

2.2.3 Επίπεδο παροχής κινήτρου

Το επίπεδο είναι πιο εξωτερικό σε σχέση με το επίπεδο επίλυσης διαφορών και σαν αναλογία μπορεί να σκεφτεί κανείς το επίπεδο παροχής κινήτρου ως μια μέθοδο η οποία εσωτερικά καλεί το επίπεδο επίλυση διαφορών όταν χρειάζεται. Όπως έχει ήδη αναφερθεί, το Truebit είναι ένα σύστημα το οποίο βασίζεται σε οικονομικά κίνητρα. Όποιος αποφασίσει να συμμετάσχει και να διαθέσει υπολογιστικό χρόνο για επίλυση ή επιβεβαίωση εργασιών τρίτων το κάνει για χρηματικό του όφελος, ενώ όποιος αποφασίζει να αναθέσει ένα υπολογιστικό πρόβλημα σε κάποιον τρίτο πρέπει να πληρώσει. Αυτός ο οποίος επιλύει ένα υπολογιστικό πρόβλημα θα πληρωθεί αν το αποτέλεσμα που προτείνει γίνει εν τέλει δεκτό. Από την άλλη, κάποιος που αναλαμβάνει να ελέγξει έναν υπολογισμό μπορεί να βρεί κάποιο πιθανό λάθος και θα πληρωθεί μόνο στην περίπτωση που όντως βρεθεί λάθος. Ο τρόπος πληρωμών αυτός παρουσιάζει κάποια προβλήματα. Αρχικά, επιτρέπει να επιβραβεύονται κάποιοι οι οποίοι σποραδικά αποφασίζουν να ελέγξουν αποτελέσματα υπολογισμών τα οποία θεωρούν ότι μπορεί να κρύβουν λάθη, με αποτέλεσμα πολλές εργασίες να μην επιβεβαιώνονται από κανέναν. Ένα άλλο πρόβλημα είναι ότι ο ρυθμός με τον οποίο προκύπτουν τα λάθη είναι ακαθόριστος ή - στις περισσότερες περιπτώσεις - είναι τόσο μικρός που δεν παρακινεί τους συμμετέχοντες να καταναλώνουν πόρους για εύρεση λαθών, όσο μεγάλο κι αν είναι το κέρδος που θα έχουν αν βρουν κάποιο. Το Truebit είναι προσανατολισμένο τόσο στον συστηματικό έλεγχο των υπολογισμών όσο και στην επιβράβευση της διαρκούς και συστηματικής σωστής συμπεριφοράς. Για το λόγο αυτό έχει αναπτύξει έναν μηχανισμό τον οποίο ακολουθούν οι λογικοί συμμετέχοντες - όσοι συμμετέχουν με σκοπό να μεγιστοποιήσουν το κέρδος τους - αν και δεν είναι υποχρεωτικό. Στα πλαίσια του μηχανισμού αυτού εισάγεται η έννοια των ¨επίτηδες λαθών¨.

Αυτό που γίνεται στην πραγματικότητα στο Truebit είναι ότι για κάθε εργασία ο λύτης ετοιμάζει μια σωστή και μια λάθος απάντηση. Στην συνέχεια, ανάλογα με μια οδηγία (σημαία) που το πρωτόκολλο του γνωστοποιεί και του επιβάλλει να ακολουθήσει, κοινοποιεί είτε την γνωστή είτε τη λάθος απάντηση. Αν είναι σε ισχύ το ¨επίτηδες λάθος¨, ο λύτης θα κληθεί να καταθέσει το σωστό αποτέλεσμα αφού πρώτα προκληθεί διαφωνία με κάποιον επικυρωτή. Επίσης, η πληρωμή του γίνεται κανονικά σαν να είχε εξ΄ αρχής καταθέσει σωστό αποτέλεσμα αφού το σύστημα γνωρίζει ότι το λάθος ήταν μέρος της διαδικασίας. Όσον αφορά τους επικυρωτές, η ανίχνευση ενός ¨επίτηδες λάθους' δίνει μεγάλα χρηματικά βραβεία (Jackpots) τα οποία σε συνδυασμό με την συχνότητα εμφάνισης τους παρέχουν ικανά κίνητρα για να ελέγχονται εν τέλει τα πιο πολλά προβλήματα που φτάνουν για επίλυση. Από τον σχεδιασμό του πρωτοκόλλου είναι αδύνατο κάποιος που αναλαμβάνει τον έλεγχο λύσεων να γνωρίζει πότε είναι σε εφαρμογή ένα επίτηδες λάθος, οπότε αναγκάζεται να ελέγχει συστηματικά όλα τα προβλήματα. Τέλος, αξιζει να αναφερθεί ότι τα Jackpots δεν είναι στατικά καθορισμένα σε ένα ποσό αφού πρέπει πάντα να λαμβάνεται υπ΄ όψιν και η πολυπλοκότητα του κάθε προβλήματος. Το Jackpot αποτελείται από ένα ποσό που αντιστοιχεί σε αποζημίωση για τους πόρους που καταναλώθηκαν και από το κέρδος, το οποίο πρέπει να είναι ικανοποιητικό.

Φόροι

Οι φόροι (Taxes) είναι ο μηχανισμός μέσω του οποίου αποταμιεύονται κεφάλαια τα οποία θα δωθούν στην συνέχεια ως Jackpots. Μέσω του συστήματος αυτού το πρωτόκολλο μπορεί να συντηρείται μόνο του χωρίς να χρειάζεται κάποια εξωτερική παρέμβαση παρά μόνο μια αρχική κατάθεση από κάποιον για να τεθεί πρώτη φορά σε λειτουργία.

Ο κύριος τρόπος λειτουργίας του μηχανισμού αυτού μπορεί να περιγραφεί εν συντομία ως ένα συνολο κανόνων για τα ποσά τα οποία καταθέτουν και λαμβάνουν τα διάφορα εμπλεκόμενα μέρη. Αρχικά, αυτός που επιθυμεί να υποβάλλει ένα πρόβλημα προς επίλυση πρέπει να κάνει μια κατάθεση η οποία θα καλύπτει τους πόρους που απαιτούνται για την επίλυση, την επικύρωση αλλά και για την λειτουργία των κριτών. Επίσης, καταθέσεις απαιτούνται και από όσους επιθυμούν να συμμετάσχουν λύνοντας προβλήματα ή όσους επιθυμούν να συμμετάσχουν ελέγχοντας και επιβεβαιώνοντας τα. Τα ποσά των καταθέσεων είναι τέτοια ώστε να αποτρέπουν έναν λογικό παίκτη να συμμετάσχει ταυτόχρονα με δύο ρόλους αφού σε διαφορετική περίπτωση θα μπορούσε κάποιος επικυρωτής (που είναι ταυτόχρονα και λύτης) να γνωρίζει πότε έχει ¨επίτηδες λάθος' και να καρπώνεται έτσι το Jackpot χωρίς να κάνει πραγματικά δουλειά.

Ένα άλλο σύνολο κανόνων ρυθμίζει τα ποσά ανάληψης ούτως ώστε να μην αδειάσει ποτέ το ταμείο των Jackpot. Για παράδειγμα, υπάρχει ανώτατο όριο (κορεσμός) στο μέγιστο ποσό που μπορεί να δοθεί ως Jackpot και αυτό είναι το ένα τρίτο του ολικού διαθέσιμου κεφαλαίου εκείνη τη στιγμή και ένας άλλος κανόνας ο οποίος βρίσκεται σε εφαρμογή είναι η εκθετική μείωση του ποσού που λαμβάνει ο κάθε επικυρωτής που εντόπισε το

¨επίτηδες λάθος¨. Δηλαδή, όσο πιο πολλοί εντοπίζουν το λάθος τόσο μειώνεται το κέρδος του καθενός.

2.3 zk-Snarks

Το zk-SNARKS είναι ένα πρωτόκολλο το οποίο βασίζεται στην μαθηματική αυστηρότητα και όχι στα οικονομικά κίνητρα. Ο όρος zk-SNARKS είναι συντομογραφία του Zero Knowledge Succinct Not- Interactive Arguments of Knowledge ή στα ελληνικά Σύντομη Μη-Διαδραστική Απόδειξη Μηδενικής Γνώσης. Για να γίνει κατανοητή η διαδικασία του πρωτοκόλου αρχικά θα αναλυθεί η σημασία του κάθε όρου που εμφανίζεται στο όνομα:

- Succinct (Σύντομο): Με τον όρο αυτό δηλώνεται ότι τόσο το μέγεθος της απόδειξης πρέπει να είναι μικρό, της τάξεως των εκατοντάδων Bytes, όσο και ο χρόνος που απαιτείται για να διαβάσει και να επικυρώσει την ορθοτητα της απόδειξης να είναι επίσης μικρός, της τάξεως των milliseconds.

- Non-Interractive (Μη-Διαδραστική): Με τον όρο αυτό δηλώνεται ότι η απόδειξη δεν απαιτεί αλληλεπίδραση μεταξύ των δύο εμπλεκόμενων μερών, αυτού που θέλει να αποδείξει κάτι (prover) και αυτού που ζητάει την απόδειξη, την επικυρώνει ή την απορρίπτει (verifier). Με τον όρο μη-διαδραστική απόδειξη εννοείται ότι ο verifier λαμβάνει με μιας ό,τι του είναι απαραίτητο για να αξιολογήσει την απόδειξη, ενώ στην κατηγορία των διαδραστικών αποδείξεων κατατάσσονται τα πρωτόκολα αυτά τα οποία απαιτούν πάνω από μια ανταλλαγή δεδομένων μεταξύ των δύο μερών. Επίσης, αν υπάρχουν πολλοί verifiers και η αποδείξεις προκκύπτουν με διαδραστικό τρόπο τότε για κάθε έναν verifier απαιτείται και νέα απόδειξη, διότι η διαδικασία απόδειξης γίνεται σε πολλαπλούς γύρους στους οποίοιυς γίνεται ανταλλαγή πληροφοριών και παραμέτρων, οι οποίες δεν θα είναι ίδιες για όλους.

- Zero-Knowledge (Μηδενική Γνώση): Με αυτόν τον όρο δηλώνεται η ικανότητα του prover να μπορεί να πείσει τον verifier ότι μια υπόθεση είναι αλληθής χωρίς να του αποκαλύψει τίποτα παραπάνω. Ένα παράδειγμα μηδενικής γνώσης είναι η διαδικασία διαπίστευσης της ταυτότητας ενός χρήστη σε κάποια πλατφόρμα. Ο χρήστης (prover) αποδεικνύει στον διακομιστή (verifier) την ταυτότητα του, δίνοντας μια κωδικοποιημένη τιμή (hash) που αντιπροσωπεύει τον κωδικό του. Ο διακομιστής μπορεί να ταυτοποιήσει τον χρήστη εξετάζοντας την τιμή αυτή αλλά μη γνωρίζοντας τον πραγματικό του κωδικό.

- Απόδειξη (Argument of Knowledge): Ο όρος αυτός αφορά την ίδια την απόδειξη. Πρόκειται για μια διαδικασία με την οποία ενας τίμιος prover μπορεί να αποδείξει την αλήθεια μιας υπόθεσης σε κάποιον verifier σίγουρα, ενω από την άλλη ένας Verifier κατά μεγάλη πιθανότητα θα καταλάβει την απόπειρα ενός κακόβουλου prover να τον ξεγελάσει.

Η θεωρητική θεμελείωση του zk-SNARKS γίνεται με χρήση κάποιων εργαλείων και θεωρημάτων τα οποία εν συντομία θα αναλυθούν.

2.3.1 Ισότητα Πολυωνύμων

Ένα πολυώνυμο βαθμού $d \in \mathbb{Z}$ είναι μια παράσταση της μορφής

$$c_0x^0 + c_1x^1 + ... + c_dx^d, d \neq 0$$

.

Για συντομία, έναν πολυώνυμο της παραπάνω μορφής μπορεί να οριστεί μονοσήμαντα από το διάνυσμα των συντελεστών του. Επίσης, αν θεωρήσουμε δύο πολυώνυμα βαθμών κ,λ τότε είναι γνωστό από την άλγεβρα ότι τα κοινά τους σημεία δεν μπορούν να είναι περισσότερα από d, όπου $d = max(\kappa, \lambda)$.

Μια σημαντική ιδιότητα που προκύπτει από τα παραπάνω και είναι η βάση του zk-SNARKS είναι ότι αν κάποιος (prover) ισχυρίζεται ότι γνωρίζει ένα πολυώνυμο βαθμού α το οποίο ξέρει και ο verifier, τότε αρκεί ο verifier να διαλέξει μια αυθαίρετη τιμή $s \in \Delta$ (όπου Δ το πεδίο στο οποίο μπορεί να αποτιμηθεί το πολυώνυμο) και να ζητήσει από τον prover να αποτιμήσει το πολυώνυμο στο σημείο αυτό και να του δώσει το αποτέλεσμα. Στην συνέχεια, συγκρίνωντας το αποτέλεσμα με την αποτίμηση του δικού του πολυωνύμου, ο verifier μπορεί να καταλάβει αν ο prover ψεύδεται ή όχι. Η πιθανότητα τα δύο παραπάνω αναφερόμενα μέρη να μην γνωρίζουν τα δύο παραπάνω μέρη είναι $\frac{\alpha}{|\Delta|}$. Το παραπάνω κλάσμα συνήθως είναι αμελητέο, αφού το πεδίο ορισμού Δ μπορεί να είναι οι ακέραιοι, δηλαδή το σύνολο από 0 έως $2^{32} - 1$ και ο βαθμός του πολυωνύμου κάποιες χιλιάδες.

2.3.2 Παραγοντοποίηση Πολυωνύμων

Βάσει του θεμελειώδους θεωρήματος της άλγεβρας, ένα επιλύσιμο πολυώνυμο μπορεί να αναλυθεί σε γραμμικούς παράγοντες και να παρει τη μορφή:

$$(x - r_0)(x - r_1)...(x - r_d) = 0$$

, όπου d ο βαθμός του πολυωνύμου. Επίσης, βάσει των παραπάνω ισχύει ότι αν ένα πολυώνυμο p(x) έχει δύο (χωρίς βλάβη της γενικότητας) γνωστές ρίζες r_1, r_2 τότε θα ισχύει:

$$p(x) = (x - r_1)(x - r_2)h(x)$$

για κάποιο πολυώνυμο h. Το παραπάνω γενικεύεται και έτσι ισχύει ότι εαν ένα πολυώνυμο p(x) έχει τουλάχιστον τις ρίζες ενός $t(x) = (x - r_1)...(x - r_d)$, τότε το p γράφεται στην μορφή:

$$p(x) = t(x)h(x)$$

για κάποιο h. Αν ένας ένας prover θέλει να αποδείξη σε έναν verifier ότι γνωρίζει ένα πολυώνυμο p(x) το οποίο έχει όλες τις ρίζες ενός άλλου πολυωνύμου t(x), τότε αρκεί ο verifier να παράξει ένα τυχαίο $\in \Delta$, να αποτιμήσει το $t = t(s)$ τοπικά και να στείλει το s στον Prover. Εκείνος με την σειρά του, θα υπολογίσει το $h(x) = \frac{p(x)}{t(x)}$, θα αποτιμήσει το p(s) και το h(s) και θα τα επιστρέψει στον Verifier ο οποίος αρκεί να εξετάσει αν p(s) = h(s)t, με ίδια πιθανότητα να ξεγελαστεί όπως και παραπάνω.

2.3.3 Ομομορφική Κρυπτογράφιση

Στα δύο παραδείγματα συναλλαγών ανάμεσα σε Prover και Verifier που αναφέρθηκαν παραπάνω, καμιά τιμή δεν έχει κρυπτογραφηθεί. Το γεγονός αυτό μπορεί να οδηγήσει σε ευκολότερη πλαστογράφηση αποδείξεων και ο τρόπος για να αποφευχθεί αυτό είναι να μεταφέρονται όλες οι τιμές από το ένα μέρος στο άλλο με τέτοιον τρόπο ώστε να μην είναι μην μπορεί να προκύψει η πραγματική τιμή από την απεικόνιση της, αλλά ταυτόχρονα να είναι δυνατή η εφαρμογή πράξεων, όπως πρόσθεση και πολλαπλασιασμός. Αυτό είναι εφικτό μέσω της ομομορφικής κρυπτογράφισης. Έστω ένας αριθμός g και ένας αριθμός x, η ομομορφική κρυπτογράφιση του x θα είναι $E(x) = g^x$. Αν δοθούν οι κρυπτογραφίσεις E(x), E(y) δύο τιμών x, y τότε μπορεί να γίνει πρόσθεση των κρυπτογραφημένων τιμών ως εξής:

$$E(x)E(y) = g^x g^y = g^{x+y}$$

Επίσης, αν δοθεί ένας αριθμός α τότε μπορεί να υπολογιστεί το αx ώς:

$$(E(x))^\alpha = (g^x)^\alpha = g^{x\alpha}$$

Παρατηρούμε ότι με τα παραπάνω είναι δυνατό να γίνεται επεξεργασία των αριθμών x, y μέσω των κρυπτογραφημένων τιμών και χωρίς τις γνώση των αρχικών τιμών. Η ιδιότητα αυτή είναι πολύ χρήσιμη αφού επιτρέπει στον Verifier αντί να στέλνει τον αριθμό s (όπως παραπάνω) να στελνει όλα τα $E(s^i)$ $\forall i$ και ο Prover με τη σειρά του να υπολογίζει τις αποτιμήσεις των πολυωνύμων χρησιμοποιώντας τις κρυπτογραφημένες τιμές που έλαβε και τους δικούς του σταθερούς συντελεστές c_i ως εξής:

$$(E(s^0))^{c_0}(E(s^1))^{c_1}...(E(s^d))^{c_d} = g^{c_0 s^0 + ... + g^{c_d} s^d} = E(p(s))$$

Στην συνέχεια, προκειμένου να αποκρύψουμε στοιχεία των αρχικών τιμών που σχετίζονται με το μέγεθος τους (μεγάλοι αριθμοί θα έχουν μεγαλύτερες κρυπτογραφημένες τιμές) ενισχύεται περισσότερο η κρυπτογράφιση (ισχυρή ομομορφική κρυπτογράφιση) εισαγοντας μια παράμετρο n και κρατώντας όλες τις παραπάνω ιδιότητες. Πλέον, η κρυπτογράφιση της τιμής x είναι:

$$E(x) = g^x mod n$$

Η νέα αυτή κρυπτογράφιση αφενός καλύπτει πληροφορίες που θα μπορούσε κανείς να εξάγει από το μέγεθος του κρυπτογραφημένου μηνύματος αλλά και καθιστά την συνάρτηση κρυπτογράφισης όχι μονοσήμαντη, οπότε ακόμα και αν κάποιος γνωρίζει τα g, n δεν θα μπορούσε να βρει την πραγματική τιμή.

2.3.4 Knowledge-Of-Exponent Assumption

Το Knowledge-Of-Exponent Assumption (KEA) είναι μια μέθοδος που προτάθηκε από το paper [Dam91] και πρόκειται για μια μέθοδο η οποία επιβάλλει σε κάποιον να κάνει μόνο πράξεις ύψωσης σε δύναμη πάνω σε μια τιμή που το έδωσε ένας άλλος παίκτης. Πιο συγκεκριμένα, έστω ένας άριθμός a το οποίο ο παίκτης A θέλει να στείλει στον B για να τον υψώσει

σε έναν εκθέτη και θέλει να είναι σίγουρος ότι το μόνο που θα κάνει ο B είναι η επιθυμητή πράξη και τίποτα άλλο. Για να γίνει αυτό:

- Ο A επιλέγει έναν τυχαίο κρυφό αριθμό q και υπολογίζει την τιμέ $a' = a^q modn$ και στέλνει στον B το ζεύγος (a, a').
- Ο B επιλέγει έναν εκθέτη c και υψώνει τις δύο τιμές a, a'. Άρα, $b = a^c$ και $b' = a'^c$. Η τούπλα (β,β′) στέλνεται πίσω στον A.
- Ο A θα ελέγξει αν $b^q = b'$ και η ισότητα ισχύει τότε ο B έδρασε τίμια.

Η ιδιότητα αυτή είναι πολύ χρήσιμη και εφαρμόζεται από την μεριά του verifier για να είναι σίγουρος ότι ο prover χρησιμοποιεί μόνο πράξεις της μορφής $(E(s^i))^{c_i}$ για να χτίσει το πολυώνυμο του και δεν προσπαθεί με κάποιον άλλον τρόπο (π.χ. εισάγοντας νέες δυνάμεις στο πολυώνυμο) να φτιάξει μια ψεύτικη έγκυρη απόδειξη.

2.3.5 Cryptographic Pairings

Τα Cryptographic Pairings ή αλλιώς bilinear mappings είναι μια μαθηματική κατασκευή η οποία συμβολίζεται ως μια $e(g^*, g^*))$, για παράδειγμα $e(g^a, g^b))$. Η σχέση αυτή παίρνει στοιχεία από ένα σύνολο και τα προβάλλει ντετερμινιστικά στην συνδυασμένη απεικονηση τους στο σύνολο εξόδου. Για παράδειγμα:

$$e(g^a, g^b) = e(g, g)^{ab}$$

Τονίζεται ότι το σύνολο εξόδου δεν είναι ίδιο με το σύνολο εισόδου και συνεπώς δεν μπορεί η έξοδος μιας απεικόνισης να χρησιμοποιηθεί σαν είσοδος στην επόμενη. Τα cryptographic pairings βασίζονται στις ιδιότητες των ελλιπτικών καμπυλών για να πετυχαίνουν την συμπεριφορά τους. Πιο συγκεκριμένα, οι ιδιότητες τους είναι:

- $e(g^a, g^b) = e(g^b, g^a) = e(g^1, g^{ab}) = e(g^{ab}, g^1) = e(g^1, g^1)^{ab}$
- $e(g^a, g^b) * e(g^c, g^d) = g^{ab} * gcd = g^{ab+cd}$

Τα Cryptographic Pairings βοηθάν στο να υπολογιστούν ασφαλείς δημόσιοι και επαναχρησιμοποιούμενοι παράμετροι. Όλες οι παράμετροι που παράγονται τυχαία από τον Prover κρυπτογραφούνται ώς $g^a modn$ και στην συνέχεια διαγράφονται για να μην μπορεί κανείς με επίθεση ή καταλάθος να έχει πρόσβαση σε αυτές αφού αν αυτό γινόταν θα ήταν εύκολο να πλαστογραφηθούν αποδείξεις. Επόμενως, δεν υπάρχουν καθόλου απλές τιμές στο σύστημα παρά μόνο κρυπτογραφημένες και οι πράξεις μεταξύ των μπορούν να γίνουν μόνο με τα pairings.

2.3.6 Αναλυτική Περιγραφή Πρωτοκόλου για Πολυώνυμα

Έχοντας αναφέρει τα όλα τα παραπάνω εργαλεία, θα περιγραφεί αναλυτικά το πρωτόκολλο με το οποίο ένας Prover (P) αποδυκνύει σε έναν Verifier (V) ότι γνωρίζει ένα πολυώνυμο t του V, βαθμού d. Το πρωτόκολλο αυτό περιγράφεται παρακάτω:

- Ο V διαλέγει τυχαία αριθμούς s και a.
- Ο V υπολογίζει τα g^a και $(g^{s^i}, g^{as^i}), \forall i \in |d|$
- Ο V απόθηκεύει το κλειδί που θα χρειαστεί για την επιπεβαίωση του αποτελέσματος που θα έρθει από τον V, το οποίο είναι το $(g^a, g^{t(s)})$, όπου $g^{t(s)}$ είναι η αποτίμηση του τοπικού του πολυωνύμου στο σημείο s.
- Ο V διαγράφει τα s, a που έχει τοπικά και στην συνέχια στέλνει στον P τα $(g^{s^i}, g^{as^i}), \forall i \in |d|$
- Ο P έχει το δικό του πολυώνυμο το οποίο είναι $p(x) = c_d x^d + c_{d-1}x^{d-1} + ... + c_0 x^0$.
- Ο P υπολογίζει το $h(x) = \frac{p(x)}{t(x)}$, δηλαδή του συντελεστές του. Υπενθυμίζεται εδώ ότι στην απλοική προσέγγιση της ενότητας 2.3.2 ο P θα έστελνε τα h(s) και p(s) και ο V θα έλεγχε ότι $p(s) = h(s)t(s)$, αλλά τώρα η διαδικασία έγινε λίγο πολυπλοκότερη.
- Ο P εφαρμόζει τους συντελεστές των p, h στα g^{s_i} που έλαβε και έτσι υπολογίζει τα $g^{p(s)}, g^{h(s)}$ καθώς και τα $g^{p(s)}$ από τα g^{as^i} που επίσης έλαβε.
- Στην συνέχεια ο P διαλέγει ένα τυχαίο δ και στέλνει πίσω στον V την απόδειξη $(g^{p(s)\delta}, g^{h(s)\delta}, g^{ap(s)\delta})$
- Ο V ελέγχει αρχικά ότι έχουν γίνει μόνο έγκυρες πράξεις (ύψωση σε δύναμη) πάνω στα δεδομένα που είχε αρχικά στείλει, χρησιμοποιώντας το Knowledge-Of-Exponent Assumption και κάνοντας την πράξη στο πεδίο των κρυπτογραφημένων τιμών με χρήση των cryptographic pairings. Ο έλεγχος παίρνει την μορφή $e(g^{p'}, g) = e(g^p, g^a)$.
- Τέλος, ο V ελέγχει την ισότητα $p(s) = h(s)t(s)$ στο κρυπτογραφικό πεδίο και ο έλεγχος έχει τη μορφή $e(g^p, g) = e(g^{t(s)}, g^h)$

2.3.7 zk-Snarks σε πραγματικά προβλήματα

Η εφαρμογή για τον έλεγχο πολυωνύμων είναι απλή αλλά περιέχει όλα τα δομικά στοιχεία τα οποία εφαρμόζονται στο πρωτόκολλο. Από εκεί και πέρα, για πιο ρεαλιστικές εργασίες και εφαρμογές χρησιμοποιούνται πολυώνυμα πολλών μεταβλητών ή/και επαναληπτικές μέθοδοι.

Ωστόσο, σπάνιο είναι ένα πρόβλημα να δίνεται σε μορφή πολυονύμων ή το ζητούμενο να είναι να αποδειχθεί κάτι ευθέως για κάποιο πολυώνυμο. Ωστόσο, υπάρχει διαδικασία η οποία η οποία ανάγει έναν γενικό υπολογισμό σε ένα πρόβλημα με πολυώνυμα, όπως αυτά που αναφέρθηκαν παρά πάνω.

Αριθμητικά κυκλώματα

Τα αριθμητικά κυκλώματα είναι μια χαμηλού επιπέδου αναπαράσταση υπολογιστικών εργασιών. Αποτελείται από πύλες οι οποίες υλοποιούν τις βασικές πράξεις του πολλαπλασιασμού και της πρόσθεσης καθώς και καλώδια οποία ενώνουν τις εισόδους με τις εξόδους ούτως

ώστε να υπάρχει ροή πληροφορίας από την είσοδο, στα ενδιάμεσα αποτελέσματα και τέλος στο αποτέλεσμα.

Quadradic Arithmentic Problems (QAP)

Έχοντας το αριθμητικό κύκλωμα που αντιστοιχεί, ένας verifier θα μπορούσε να ελέγξει αν για κάθε πύλη τα στοιχεία εισόδου παράγουν την αναμενόμενη έξοδο. Για παράδειγμα, σε μια πύλη πολλαπλασιασμού με εισόδους c,d ο verifier εξετάζει αν το αποτέλεσμα είναι cd. Η προσέγγιση αυτή όνομάζεται Rank 1 constraint system. Όσο όμως το μέγεθος του κυκλώματος μεγαλώνει γίνεται δύσκολο να ελέγχεται κάθε πύλη με τον παραπάνω τρόπο. Τα Quadratic Arithmetic Problems είναι ένας τρόπος για να ομαδοποιούνται πολλές πύλες σε ένα πολυώνυμο και να είναι εύκολο να εξεταστεί η ορθότητα με αποτίμηση σε ένα μόνο σημείο όπως εξηγείται στην ενότητα Ïσότητα Πολυωνύμων¨.

Αναγωγή Γενικού τύπου υπολογισμού σε zk-SNARKS

Συνοψίζοντας, ένα γενικό υπολογιστικό πρόβλημα το οποίο δίνεται σε μια γλώσσα προγραμματισμού, μπορεί να μετατραπεί σε αριθμητικό κύκλωμα, στην συνέχεια σε QAP και τέλος να επαληθευτεί με χρήση zk-SNARKS.

Η διαδικασία δεν είναι στην πράξη τόσο εύκολη όσο περιγράφεται κυρίως λόγω τεχνικών περιορισμών. Προς το παρόν, δεν υπάρχουν για τις περισσότερες γνωστές υψηλού επιπέδου εργαλεία για παραγωγή αριθμητικών κυκλωμάτων. Ενδεικτικά, κάποια frameworks υπάρχουν για τις παραπάνω διαδικασίες είναι το ZoKrates και το Pinocchio. Το ZoKrates ένα framework το οποίο παρέχει εργαλεία για να παράγονται αποδείξεις Zk-Snarks για διάφορα προγράμματα στο Ethereum. Διαθέτει διεπαφές για να μπορούν να χρησιμοποιηθούν τα εργαλεία αυτά σε εφαρμογές Python και Js και κάνουν ευκολότερη τη διαδικασία παραγωγής αποδείξεων αποκρύπτοντας τις λεπτομέρειες της διαδικασίας. Ο μεγάλος περιορισμός της υλοποίησης αυτής είναι ότι τα προγράμματα πρέπει να είναι γραμμένα σε μια γλώσσα που αναπτύχθηκε αποκλειστικά για τη χρήση αυτή. Ένα άλλο εργαλείο που υπάρχει προς αυτή την κατεύθυνση είναι το Pinocchio, το οποίο δέχεται ως είσοδο ένα πρόγραμμα γραμμένο σε C και παράγει το ισοδύναμο QAP και όλες τις αρχικές παραμέτρους που χρειάζονται για την απόδειξη. Δεν είναι όλα τα προγράμματα σε C επιβεβαιώσιμα μέσω του Pinocchio αφού στην πραγματικότητα υποστηρίζεται ένα μόνο υποσύνολο της γλώσσας το οποίο όμως είναι αρκετά ευρύ.

Κεφάλαιο 3

Apache Spark

3.1 Εισαγωγή στο Spark

Το Spark είναι ένα εργαλείο για εφαρμογές μεγάλων δεδομένων το οποίο προσφέρει δυνατότητες επεξεργασίας τόσο σε ένα μηχάνημα όσο και δυνατότητα παράλληλης επεξεργασίας σε clusters. Αποτελείται από διάφορες βιβλιοθήκες, καθεμιά από τις οποίες αφορά στην επίλυση συγκεκριμένων προβλημάτων, όπως SQL Analytics, Machine Learning, Data Science και επίλυση προβλημάτων βελτιστοποίησης. Ξεκίνησε ώς ερευνητική δουλειά από το UC Berkeley AMPLab και πλέον χρησιμοποιείται από πολλές εταιρίες και πανεπιστήμια. Η αρχιτεκτονική του Spark επιτρέπει σε μια εργασία να έχει πολλά στάδια - και όχι μόνο 2 - όπως το map-reduce - με σκοπό να αναδεικνύονται οι εξαρτήσεις μεταξύ σταδίων και να τρέχουν όσο το δυνατον πιο πολλά στάδια παράλληλα.

Από πλευράς υλοποίησης, το Spark είναι γραμμένο στην γλώσσα Scala, μια γλώσσα που μοιάζει με τη Java, τρέχει πάνω σε JVM και στο ιδίωμα της έχει έντονο το στοιχείο του συναρτησιακού προγραμματισμού. Παρέχονται επίσης APIs για τις γνωστές γλώσσες Java, Python τα οποία καλύπτουν όλο το φάσμα των δυνατοτήτων του framework.

3.2 Αρχιτεκτονική του Spark

3.2.1 Δεδομένα στο Spark

RDD

Η βασική αφαιρετική δομή την οποία χρησιμοποιεί το Spark για την αναπαράσταση, αποθήκευση και επεξεργασία δεδομένων είναι το RDD, το οποίο είναι συντομογραφία για το Resilient Distributed Dataset. Όπως φαίνεται και από το όνομα, πρόκειται για κατανεμημένες συλλογές δεδομένων, οι οποίες αποθηκεύονται σε διαφορετικούς κόμβους του cluster και μπορούν να επεξεργαστούν παράλληλα. Το RDD πετυχαίνει την ανοχή σε σφάλματα (resilience) μέσω ενός γράφου RDD Linage Graph ο οποίος κρατά πώς προέκυψε το κάθε rdd για να μπορεί να επανακατασκευαστεί σε περίπτωση που προκύψει κάποιο σφάλμα. Τα δεδομένα μέσα στο RDD μπορούν να μην έχουν καποια συγκεκριμένη δομή (π.χ. οργανωμένα σε στήλες).

Dataframe - Dataset

Τα Dataframes και τα Datasets είναι επίσης κατανεμημένες συλλογές αντικειμένων που χρησιμοποιούνται στο Spark. Η διαφορά τους με το RDD είναι ότι τα δεδομένα έχουν σχήμα, δηλαδή πρόκειται πλέον για συλλογές οι οποίες έχουν δομή ορισμένη και τα δεδομένα είναι κολώνες, η καθεμιά με τον δικό της τύπο.

Η διαφορά μεταξύ των δύο διεπαφών εντοπίζεται στο πώς ορίζονται και επιβάλλονται οι τύποι στις επιμέρους στήλες των δεδομένων. Στην περίπτωση του Dataframe, οι τύποι διαχειρίζονται από το Spark, ενώ στην περίπτωση του Dataset η διαχείρηση γίνεται από κατά το στάδιο της μεταγλώτισης. Παρόλο που η ανάθεση τύπων επιβάλλει περιορισμούς και επιπλέον δυσκολία κατά των προγραμματισμό - σε σχέση με την ελεύθερη δομή των RDD - φανερώνει επιπλέον δυνατότητες βελτιστοποίησης.

3.2.2 Οκνυρή αποτίμηση

Η οκνυρή αποτίμηση είναι μια μέθοδος αποτίμησης εκφράσεων η οποία απαντάται συχνά στην μελέτη των γλωσσών προγραμματισμού και χρησιμοποιείται κυρίως στις γλώσσες συναρτησιακού προγραμματισμού. Η στρατηγική αυτή καθυστερεί την αποτίμηση της έκφρασης έως ότου να είναι αναγκαίος ο υπολογισμός του αποτελέσματος. Σκοπός της οκνυρής αποτίμησης είναι η αποφυγή επανυπολογισμού της ίδιας έκφρασης αλλά και η αποφυγή αποτίμησης εκφράσεων των οποίων το αποτέλσμα μπορεί να είναι περιττό.

Το Spark εφαρμοόζει την στρατηγική αυτή και υπάχει συγκεκριμένη κατηγορία πράξεων οι οποίες χρησιμοποιούνται προκειμένου να αποτιμηθεί μια έκφραση.

3.2.3 Τύποι πράξεων στο Spark

Οι πράξεις που γίνονται πάνω στα δεδομένα με τη χρήση του Spark χωρίζονται σε δύο κατηγορίες:

- Transformations (μετασχηματισμοί): Παραδείγματα μετασχηματισμών είναι οι μέθοδοι map(), filter(), reduceByKey() κλπ. Οι κλήσεις αυτές συνήθως δέχονται σαν ορίσματα ένα block δεδομένων και μια συνάρτηση και με χρήση των παράγουν νέα blocks δεδομένων τα οποία μπορούν να αποτελούν την είσοδο σε επόμενο μετασχηματισμό.

- actions (δράσεις): Με την χρήση μετασχηματισμών μπορούν να φτιαχτούν πολύ μεγάλες αλυσίδες επεξεργασίας δεδομένων (pipes). Όμως, επειδή το Spark χρησιμοποιεί οκνηρή αποτίμηση καμιά αλυσίδα μετασχηματισμών δεν θα παράξει αποτέλεσμα, παρά μόνο αν κληθεί κάποιο action, όπως οι μέθοδοι collect(), take(). Με την κλήση ενός action, ξεκινάει και η αποτίμηση των αλυσίδων.

3.2.4 Η έννοια της εργασίας στο Spark

Για την εκτέλεση μιας εφαρμογής, το spark ακολουθεί μια διαδικασία κατάτμησης της εργασίας σε μικρότερα μέρη πριν ανατεθεί κάτι σε κάποιον κόμβο του Cluster. Η κατάτμηση

αυτή δεν γίνεται μονομιάς αλλά με ενδιάμεσα στάδια καθένα από τα οποία δίνει κάποιες πληροφορίες. Για το Spark λοιπόν υπάρχουν 4 διαφορετικές αναπαραστάσεις για την εργασία, από πιο αφαιρετικές έως και την πιο ¨απτή¨, η οποία θα εκτελεστεί σε κάποιον κόμβο. Πιο συγκεκριμένα:

- App (εφαρμογή): Αυτή η έννοια είναι η πιο αφαιρετική και ευρεία έννοια. Αφορά αυτό το οποίο καταθέτει κάποιος στον spark cluster μέσω spark submit. Μια εφαρμογή χωρίζεται σε jobs.

- Job (δουλειά): Τα jobs προκύπτουν από την κλήση actions μέσα στο app. Δηλαδή ένα app μπορεί να έχει πολλά jobs. Κάθε Job στην συνέχεια κατατίθενται στον DAGScheduler ο οποίος παράγει έναν ακυκλικό γράφο αποτελούμενο από Stages (Στάδια).

- Stage (Στάδιο): Το στάδιο είναι η αμέσως μικρότερη μονάδα εργασίας από το job. Ένα σύνολο σταδίων παράγεται από τον DAGScheduler και είναι ένας γράφος ο οποίος εκφράζει εξαρτήσεις μεταξύ και αναδεικνύει ποιες πράξεις μπορούν να εκτελεστουν παράλληλα. Ένα stage μπορεί να χρησιμοποιεί τα αποτελέσματα όλων των προηγούμενων σταδίων αλλά όχι των επόμενων (ιδιότητα του DAG). Τα stageseίναι κυρίως λογικός σχεδιασμός για την εκτέλεση της εργασίας και ένας worker node δεν εκτελεί ένα ολόκληρο stage. Τα Stages χωρίζονται σε δύο υποκατηγορίες:

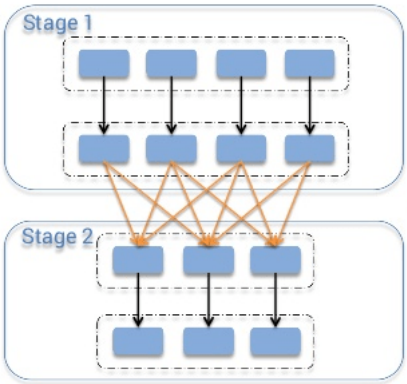
  - i. SuffleMapStage: Τέτοια stages είναι αυτά που δεν παράγουν το τελικό αποτέλεσμα που επιστρέφεται στον χρήστη (π.χ. με το collect()) αλλά παράγουν όλα τα ενδιάμεσα τα οποία είναι απαραίτητα.

  - ii. ResultStage: Το τελικό stage το οποίο μετά από ένα action και επιστρέφει το αποτέλεσμα του Job.

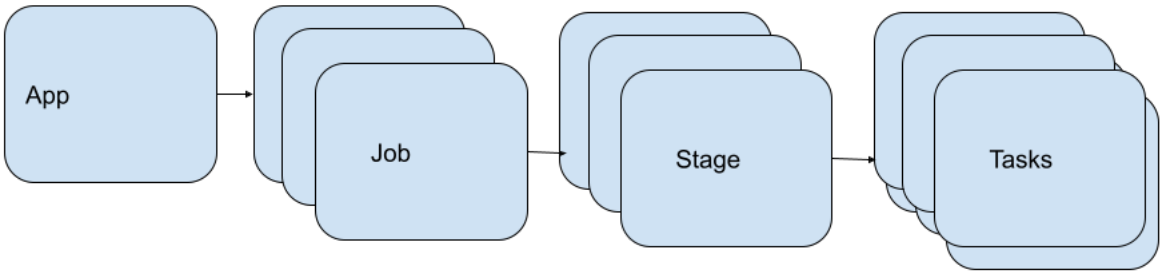
  Ο διαχωρισμός σε στάδια λαμβάνει υπόψιν κυρίως το shuffling, το οποίο γίνεται από τις κλήσεις reduceByKey() κλπ. Shuffling ονομάζεται η διαδικασία που λαμβάνει χώρα όταν ολοκληρώνεται ένα stage και τα αποτελέσματα πρέπει να μεταφερθούν στον κατάλληλο executor για να εκτελεστούν εργασίες των επόμενων σταδίων, όπως φαίνεται και στην εικόνα 2.1.

- Task (Εργασία): Η μικρότερη μονάδα εργασίας η οποία υπάρχει. Ουσιαστικά ένα stage χωρίζεται σε τασκς τα οποία εκτελούν τους ίδιους μετασχηματισμούς σε διαφορετικά παρτιτιονς των RDD. Η μονάδα αυτή εργασίας είναι αυτή που αποστέλλεται στους worker nodes και πιο συγκεκριμένα στους executors που ζουν στους workers. Πριν την αποστολή του, το task σειριοποιείται (serialization) με χρήση του αντίστοιχου artifact και αποστέλλεται σε κάποιον executor.

Κάθε Job, stage, task έχει ένα μοναδικό αναγνωριστικό JobId, stageId, taskId αντίστοιχα. Το taskId είναι μοναδικό για κάθε task και προκύπτει από την σειρά με την οποία το



Σχήμα 3.1: διαδικασία Shuffling



Σχήμα 3.2: Κατάτμιση εφαρμογής σε tasks

task ανατέθηκε σε έναν executor. Για παράδειγμα, αν π.χ. υπάρχει ένα stage με 4 tasks τα οποία εκφράζονται ως μια λίστα, το taskId=0 θα είναι το πρώτο task το οποίο θα βγει από την λίστα και θα ανατεθεί σε έναν executor και αυτό δεν σημαίνει απαραίτητα ότι θα είναι το πρώτο task της λίστας.

3.3 Spark Driver

Ο Spark Driver είναι το artifact το οποίο καλεί την main() της εφαρμογής και είναι υπέυθυνος για την αρχικοποίηση του συστήματος αλλά και για την μετάφραση του κώδικα σε ένα έγκυρο πλάνο εκτέλσης εργασιών που δύνανται να τρέξουν στον Cluster. Βασικά μέρη του υπάρχουν στον Driver και που θα εξηγηθούν στην συνέχεια είναι το Spark Context, ο DAG Scheduler και ο Task Scheduler. Ο Driver είναι ένας για κάθε εφαρμογή και ανάλογα με τις ρυθμίσεις του συστήματος μπορεί να ζει στον Master (Cluster Mode) ή να δημιουργείται στην μεριά του Client που υποβάλλει μια εφαρμογή (Client mode).

Spark Context

Το SparkContext artifact, το πιο σημαντικό απ' όλα τα artifacts τα οποία δημιουργούνται στα πλαίσια μιας εφαρμογής (spark application). Ένα SparkContext αντιστοιχεί σε ένα και μόνο App. Το sparkContext δημιουργεί τον DAGScheduler και τον Task Scheduler και διατηρεί όλες τις εσωτερικές μεταβλητές οι οποίες ορίζουν την κατάσταση στην οποία βρίσκεται η αντίστοιχη δουλειά. Με τον όρο "κατάσταση" εννοείται ένα σύνολο εσωτερικών μεταβλητών, μη ορατών από τον χρήστη οι οποίες περιέχουν:

- Πόσες προσπάθειες έχουν γίνει για να ολοκληρωθεί η δουλειά (Job)
- Το μοναδικό Ιδ της εφαρμογής που είναι ενεργή
- Τα αρχεία προγράμματος (*.jar) που έχουν φορτωθεί
- Τα αρχεία δεδομένων που έχουν φορτωθεί
- Όλο το configuration που έχει οριστεί για την εφαρμογή αυτή.
- Διατηρεί τους DAGScheduler και TaskScheduler.

DAG Scheduler

Ο DAGScheduler δημιουργείται και χρησιμοποιείται από το SparkContext. Διαθέτει μια μέθοδο submitJob() με την οποία το SparkContext εισάγει νέα jobs στον scheduler από τα οποία προκύπτει ο κατάλληλος ακυκλικός γράφος που αντιστοιχεί στο λογικό πλάνο εκτέλεσης της εφαρμογής. Ο ακυκλικός γράφος υπολογίζεται αναδρομικά ξεκινώντας από το το τέλος (finalStage). Υπολογίζονται τα στάδια- εξαρτήσεις για το final stage και στην συνέχεια τα στάδια- εξαρτήσεις των σταδίων αυτών. Όταν κάποιο στάδιο βρεθεί να μην έχει εξαρτήσεις , τότε έχει ολοκληρωθεί η διαδικασία και έχουν υπολογιστεί όλα τα στάδια που πρέπει να εκτελεστούν. Στην συνέχεια, για κάθε στάδιο υπολογίζονται τα tasks που το απαρτίζουν και τοποθετούνται σε μια δομή Taskset η οποία κατατίθεται στον Task Shceduler.

Taskset

Η δομή Taskset, όπως αναφέρεται και παραπάνω, περιέχει μια λίστα με όλα τα tasks τα οποία αφορούν κάποιο συγκεκριμένο στάδιο. Πέραν της λίστας με τα tasks, η δομή περιέχει και κάποια metadata όπως το στάδιο στο οποίο αφορούν τα tasks αυτά, πόσες έχει προσπαθήσει το σύστημα να εκτελέσει το στάδιο αυτό και κάποιες επιπλέον μη- υποχρεωτικές πληροφορίες όπως την προτεραιότητα που έχει το κάθε Taskset.

Task Scheduler

Για κάθε νέο Taskset που φτάνει στον Task Scheduler φτιάχνεται και ένας Taskset Manager ευθύνη του οποίου είναι να ελέγχει για τυχόν γεγονότα (events) που συμβαίνουν και αφορούν το συγκεκριμένο Taskset. Τέτοια γεγονότα είναι η περάτωση ή η αποτυχία ενός

task του Taskset ή η προσπάθεια επανυποβολής του συγκεκριμένου Taskset. Είναι εμφανές από τα παραπάνω ότι το Spark ομαδοποιεί τα tasks και αυτή η ομαδοποίηση είναι αρκετά αυστηρή. Ένα task που αποτυγχάνει σημαίνει την αποτυχία του Taskset και στην περίπτωση επανυποβολής πρέπει να υποβληθούν όλα τα tasks που περιέχονται σε αυτό και όχι κάποιο μεμωνομένο.

Βασική αρμοδιότητα του Task Scheduler είναι η ανάθεση των tasks σε executors προκειμένου να υπολογιστούν. Για να γίνει αυτό, αρχικά πρέπει να γίνουν γνωστοί στον Task Scheduler οι πόροι που κάθε κόμβος διαθέτει για αυτό το Taskset. Στην συνέχεια, διατρέχεται η λίστα των διαθέσιμων executors και των πόρων τους και για κάθε έναν βρίσκεται ένα κατάλληλο task το οποίο και θα του ανατεθεί. Κριτήρια για την καταλληλότητα ενός tasks αποτελούν οι πόροι (πυρήνες, μνήμη), η πληροφορία που υπάρχει στην cache του (π.χ. ήδη αποθηκευμένα rdd που είναι εξαρτήσεις του task) αλλά και η διαθεσιμότητα του (η οποία ελέγχεται μέσω heartbeat). Όταν κάποιο task ανατεθεί σε κάποιον κόμβο αποστέλλεται σχετικό μήνυμα στον TaskSet Manager αλλά και στον DagScheduler.

Όταν εν τέλει αποφασισθεί ποιο/ποια tasks θα ανατεθούν σε κάθε executor καλείται ο serializer, δουλειά του οποίου είναι να σειριοποιήσει ότι χρειάζεται ο executor (κώδικα, metadata) για να ξεκινήσει την εργασία. Τα σειριοποιημένα δεδομένα στέλνονται στον executor.

Task Result Getter

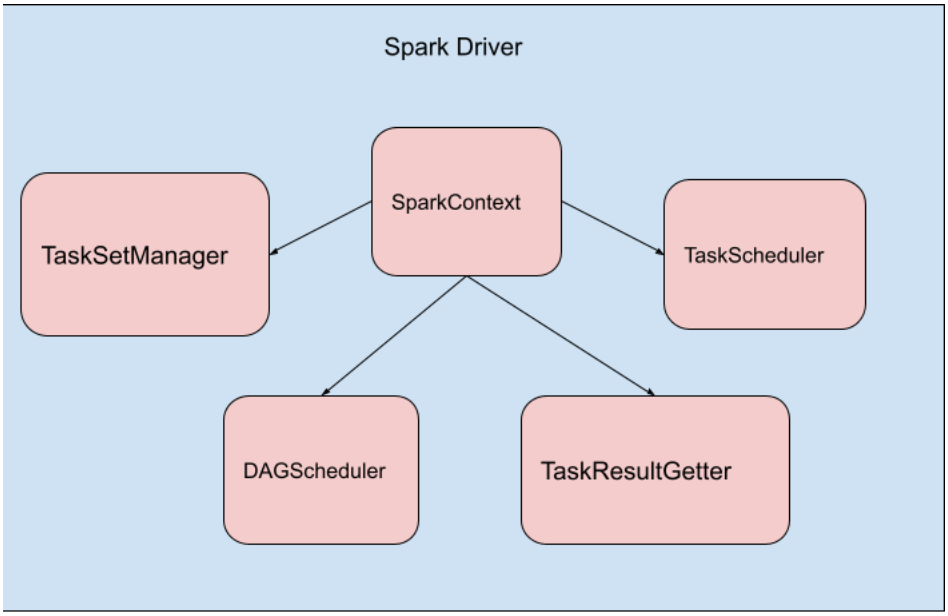
Όταν κάποιο task ολοκληρωθεί επιτυχώς σε κάποιο κόμβο εργάτη, πρέπει το αποτέλεσμα και κάποια metadata να αποσταλλούν πίσω στο Driver για να ενημερωθούν και τα υπόλοιπα ενδιαφερόμενα μέρη, όπως ο DAG Shceduler και ο Taskset Manager. Ο Task Result Getter είναι το component στο οποίο φτάνουν αυτές οι πληροφορίες, δέχονται μια πρώτη επεξεργασία και κυρίως αποσειριοποίηση και στην συνέχεια ειδοποιούνται τα υπόλοιπα services τα οποία μπορούν πλέον να διαβάσουν την επεξεργασμένη από τον Result Getter πληροφορία για να ολοκληρώσουν δικές τους λειτουργίες.

3.4 Η Αρχιτεκτονική Spark Master

Βασικός ρόλος του Spark Master είναι η διαχείρηση των κόμβων-εργατών και των πόρων τους. Όταν μια νέα εφαρμογή κατατείθεται στο σύστημα είναι ευθύνη του Master να διαθέσει πόρους στον driver της εφαρμογής προκειμένου να μπορέσει να γίνει σωστός προγραμματισμός και καταμερισμός της εργασίας στον cluster. Επίσης, ο Master χειρίζεται την εισαγωγή νέων κόμβων-εργατών και τις αποχωρήσεις.

Blacklist Manager

Ο Blacklist Manager είναι ένα Component το οποίο υπάρχει στον Master και παρόλο που δεν εμπλέκεται άμεσα στην διαδικασία υπολογισμού εφαρμογών, συμβάλλει στην ορθή και απρόσκοπτη λειτουργία του συστήματος. Πιο συγκεριμένα, ο Blacklist Manager ελέγχει διαρκώς για τυχόν αποτυχίες και αν προκύψουν τις κρατά μαζί με την χρονική στιγμή της



Σχήμα 3.3: Spark Driver

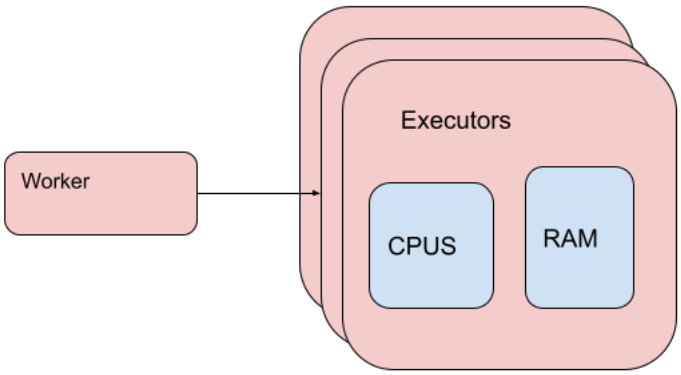
αποτυχίας και την συνδέει με τον Worker που την προκάλεσε. Ο λόγος για τον οποίο γίνεται αποθήκευση και του timestamp είναι για να μην αποκλείονται workers για σφάλματα πολύ πίσω στον χρόνο. Αν λοιπόν κάποιος Worker προκαλέσει πολλές αποτυχίες σε σχετικά μικρό διάστημα τότε ο Blacklist Manager τον αποκλείει για κάποιον χρόνο από το cluster.

3.5 Η Αρχιτεκτονική του Spark Worker

Όπως φαίνεται, σε κάθε μηχάνημα υπάρχει μοναδικός worker πάνω στον οποίο τρέχουν διαφορετικά threads, οι executors. Κάθε executor διαθέτει πυρήνες και μνήμη του μηχανήματος και μπορεί να δεσμευτεί προκειμένου να περατωθεί ένα task. Μετά την εκτέλεση, το αποτέλεσμα προωθείται στον Driver μέσω μιας διεπαφής που υπάρχει αποκλειστικά για να επιτελεί αυτή την επικοινωνία.

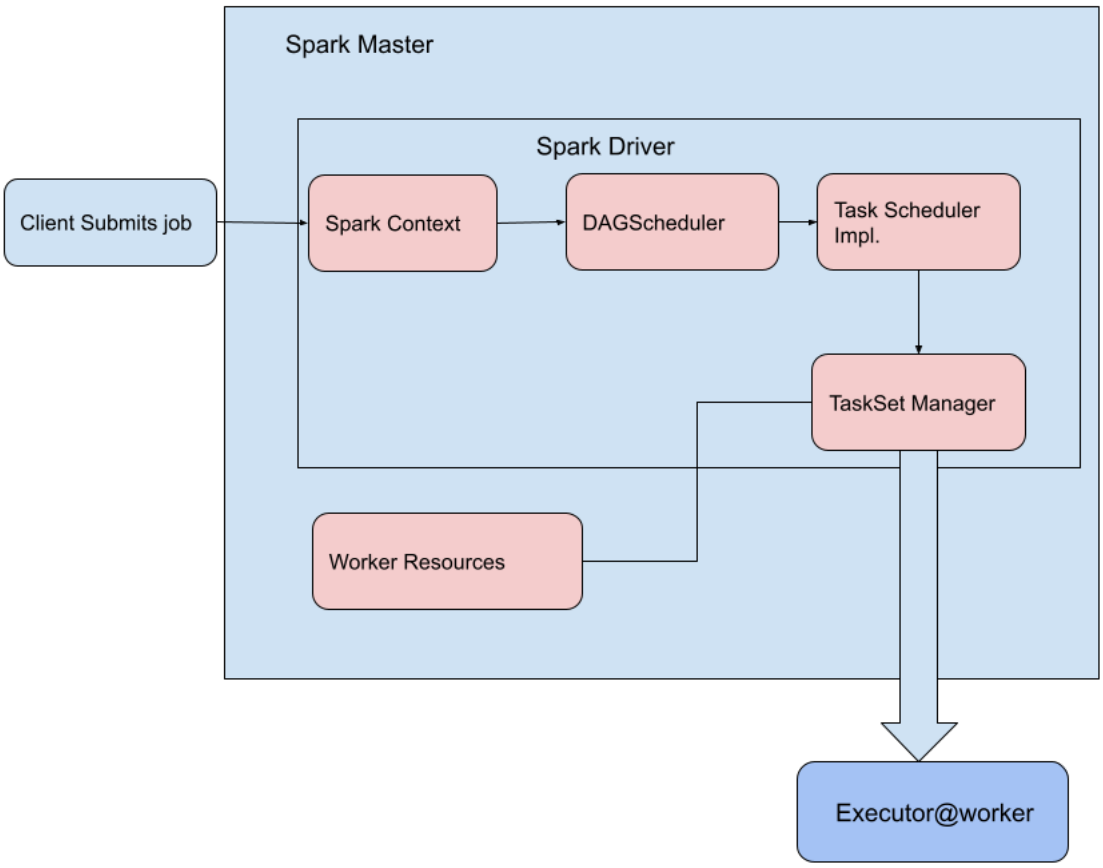
3.6 Υποβολή εργασίας στο Cluster

Υποβολή μιας νέας εφαρμογής στον Spark Cluster μπορεί να γίνει μέσω του script spark-submit. Η ροή από την στιγμή που ο πελάτης/ χρήστης εκτελεί το spark-submit ο πρωτο artifact το οποίο καλείται μετά το spark-submit είναι το org.apache.spark.deploy.SparkSubmit. Με αυτό, ετοιμάζεται περιβάλλον (JVM) στο οποίο θα τρέξει η εφαρμογή και μεταβιβάζονται τα ορίσματα του spark-submit στον Master. Στην συνέχεια, δημιουργειται ο Driver ο οποίος με τη σειρά του δημιουργεί τα υπόλοιπα αναγκαία μέρη. Τελικά, ο Task Scheduler δέχεται ως είσοδο από τον DAG Scheduler τα tasks τα οποία πρέπει να τρέξουν και από τον Master τους διαθέσιμους πόρους που υπάρχουν στο cluster. Με επεξεργασία αυτών, βρίσκει σε ποιον executor πρέπει να ανατεθεί το κάθε task και στην συνέχεια σειριοποιείται τόσο το



Σχήμα 3.4: Spark Worker

task όσο και κάποια metadata τα οποία περιέχουν πληροφορίες για τα partitions εισόδου και αποστέλλονται στον executor (σχήμα 2.5).



Σχήμα 3.5: Υποβολή εφαρμογής στον Cluster

3.7 Αποστολή αποτελέσματος ενός Task στον Driver

Μόλις κάποιος Executor ολοκληρώσει ένα Task και έχει το αποτέλεσμα διαθέσιμο πρέπει να το προωθήσει προς τον driver για να μπορέσει να χρησιμοποιηθεί ως είσοδος σε επόμενα tasks ή ως τελικό αποτέλεσμα. Για λόγους αποδοτικότητας, αν το μέγεθος του αποτελέσματος είναι μεγαλύτερο από ένα μέγιστο όριο τότε αυτό που στέλνεται δεν είναι το πραγματικό αποτέλεσμα αλλά metadata για το που είναι αποθηκευμένο. Με αυτό τον τρόπο μειώνεται η ανταλλαγή δεδομένων μεγάλου όγκου στο ελάχιστο, αφού όταν κάποιος άλλος executor χρειαστεί το αποτέλεσμα από κάποιο προηγούμενο task θα πάρει τα metadata από τον cluster manager και θα κάνει λήψη των δεδομένων. Στη διαδικασία αυτή έγινε μόνο μια μεταφορά δεδομένων μεγάλου όγκου, ενώ στην άλλη περίπτωση θα γινόταν δύο (μεταφορά αποτελέσματος από executor-1 σε cluster-manager και από cluster-manager σε executor-2).

Πέραν του αποτελέσματος, ο executor για κάθε task στέλνει και κάποια metadata τα οποία αφορούν σε χρόνους σειριοποίησης και αποσειριοποίησης του εκτελέσιμων, δεδομένων και αποτελέσμάτων, χρήση μνήμης, χρήση στοίβας και άλλα.

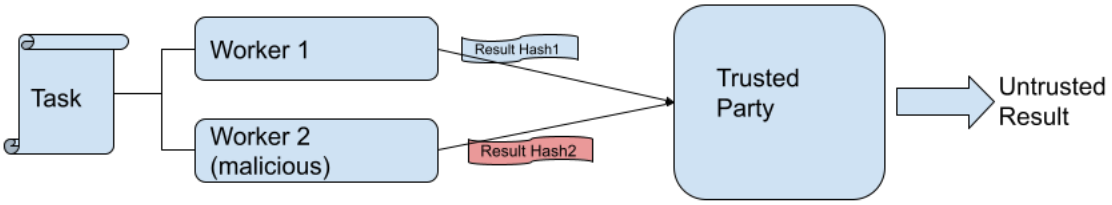
Κεφάλαιο 4

Επιβεβαιώσιμη επεξεργασία στο Spark

Προκειμένου να εξασφαλισθεί ότι το αποτέλεσμα μιας εφαρμογής που ανατέθηκε στο cluster είναι σωστό και για να μπορέσει να λάβει την σχετική πιστοποίηση ο client πρέπει να γίνουν κάποιες παραδοχές βάσει των οποίων θα γίνει η επέκταση του Spark σε επιβεβαιώσιμο Spark. Αξίζει εδώ να αναφερθεί ότι η υιοθέτηση του Spark ως βάση του προτεινόμενου συστήματος εισάγει περιορισμούς ως προς τον σχεδιαμό και την υλοποίηση της επέκτασης για την επιβεβαιώσιμη επεξεργασία. Οι περιορισμοί αυτοί αφορούν την αρχιτεκτονική του Spark αλλά και πιο τεχνικά ζητήματα, όπως για παράδειγμα την χρήση αποκλειστικά εργαλείων τα οποία παρέχουν διαπαφή σε Java ή Scala, την γλώσσα στην οποία είναι γραμμένο το Spark.

Η γενική ιδέα της υλοποίσης που παρουσιάζεται είναι το κάθε Task να ανατίθεται σε 2 τουλάχιστον κόμβους. Ο καθένας αφού υπολογίσει το αποτέλεσμα της εργασίας του, θα πρέπει να υπολογίσει και ένα hash για το αποτέλεσμα αυτό το οποίο και θα αποστέλλεται σε ένα έμπιστο μέρος το οποιο θα ελέγχει αν τα δύο hashes είναι ίδια. Αν για κάποιο Task υπάρχουν διαφορετικά hashes τότε θεωρείται ότι το αποτέλεσμα δεν είναι έμπιστο και ένα Exception από το έμπιστο μέρος τερματίζει την εφαρμογή.

Σε αντίθετη περίπτωση, αν για κάθε task υπάρχουν ίδια hashes τότε το τελικό αποτέλεσμα που προκύπτει θεωρείται έμπιστο.



Σχήμα 4.1: Επιβεβαίωση αποτελέσματος εργασίας

4.1 Συμπεριφορά Κακόβουλου

Ο σχεδισμός που έχει γίνει έχει βασιστεί σε κάποιες παραδοχές ως προς τις δυνατότητες και την συμπεριφορά των workers. Ένας τίμιος worker υπολογίζει και επιστρέφει πάντα σωστά αποτελέσματα, για όλα τα tasks που του αναθέτει ο master. Ένας κακόβουλος worker μπορεί να συμπεριφέρεται κατά το δοκούν, επιστρέφοντας κάποιες φορές σωστές και κάποιες φορές λάθος τιμές. Η απόφαση αυτή λαμβάνεται είτε τυχαία ,είτε με κάποια κριτήρια που στους υπόλοιπους συμμετέχοντες είναι άγνωστα. Επίσης, κανένας worker δεν μπορεί να πειράξει τα δεδομένα ή τα αποτελέσματα ενός άλλου και ο κάθε κόμβος δρα ανεξάρτητα οπότε δεν μπορούν δύο κακόβουλοι workers να συνεργάζονται, προκειμένου να εξαπατήσουν τον master και τον client.

4.2 Επιλογή έμπιστου μέρους

Σε προγούμενο κεφάλαιο αναφέρθηκε το πρωτόκολλο Truebit, το οποίο σε περίπτωση διαφωνίας εντοπίζει την πηγή της διαφωνίας την οποία εν τέλει λύνει το έμπιστο σώμα των κριτών. Έτσι και στην παρούσα υλοποίηση απαιτείται ένα έμπιστο μέρος το οποίο να ελέγχει για το αν έχει υπάρξει λάθος κατά την εκτέλεση μιας εφαρμογής ή αν το τελικό αποτέλεσμα είναι σωστό.

Ως έμπιστο μέρος της παρούσας υλοποίησης θεωρείται ο Master του cluster. Αυθαιρεσίες ή λάθη λοιπόν μπορούν να προοκύψουν μόνο από τους Workers. Ο λόγος για την υπόθεση αυτή είναι καθαρά συσχετισμένος με την αρχιτεκτονική του Spark. Ο Master έχει ευθύνες όπως ο καταμερισμός των πόρων του cluster οι οποίες είναι δύσκολο να ελεγχούν ή/και να επαληθευτούν. Επίσης, οι drivers των εφαρμογών τρέχουν στον Master με καθήκον τον προγραμματισμό των εργασιών και αυτό είναι άλλο ένα παράδειγμα εργασίας την οποία δεν μπορεί κάποιος να επαληθεύσει ως προς την ορθότητα και επιβάλλει το να θεωρηθεί ο Master έμπιστος.

4.3 Παρεμβάσεις στον Spark- Worker

Όπως περιγράφηκε παραπάνω είναι απαραίτητο πλέον για τα αποτελέσματα όλων των tasks να αποστέλλεται και ένα hash που να τα αντιπρωσοπεύει. Έτσι λοιπόν στον κώδικα του Executor εισάγεται ένα νέο component το οποίο δέχεται ως είσοδο το αποτέλεσμα ενός task και υπολογίζει ένα hash που αντιστοιχεί στο αποτέλεσμα. Από τεχνικής πλευράς, το αποτέλεσμα κάθε task περιγράφεται από έναν πίνακα με Bytes του οποίου το hash είναι αυτό χρησιμοποιείται. Στην συνέχεια, το hash αυτό προστίθεται στην δομή η οποία κρατά όλα τα metadata τα οποία ήδη στέλνει ο executor στον Master. Η προσθήκη αυτή επιφέρει γραμμική ως προς το μέγεθος του αποτελέεσματος επιβάρυνση executor.

4.4 Παρεμβάσεις στον Spark-Master

Στη μεριά του Master είναι πιο πολλές οι απαιτούμενες αλλαγές διότι οι ευθύνες του είναι περισσότερες και αφορούν τόσο τον καταμαμερισμό των εργασιών αλλά και την επιβεβαίωση τους.

DAG Scheduler

Η πρώτη προσθήκη στον Master αφορά τον Dag Scheduler και είναι η πολλαπλή εισαγωγή των ίδιων tasks στο σύστημα. Πιο συγκεκριμένα, όταν για ένα stage υπολογίζονται τα επιμέρους tasks, στο Taskset που δημιουργείται εισάγονται όλα 2 φορές, και το επεκταμένο αυτό Taskset αποστέλλεται στον Task Scheduler. Σκοπός αυτής της τροποποίησης είναι κάθε task να υπολογίζεται πολλές φορές , από διαφορετικούς Workers, δεδομένου ότι οι Workers δεν είναι έμπιστοι.

Task Scheduler

Στον κώδικα του Task Scheduler έγιναν τροποποιήσεις σε κάποιες ιδιωτικές συναρτήσεις προκειμένουν να μην επιτρέπεται τα δύο αντίτυπα που ίδιου Task να ανατεθούν σε executors του ίδιου Worker. Ο περιορισμός αυτός συμπεριλαμβάνεται πλέον στο σύνολο των παραμέτρων που εξετάζει ο Task Scheduler για να βρει που πρέπει να ανατεθεί το κάθε task.

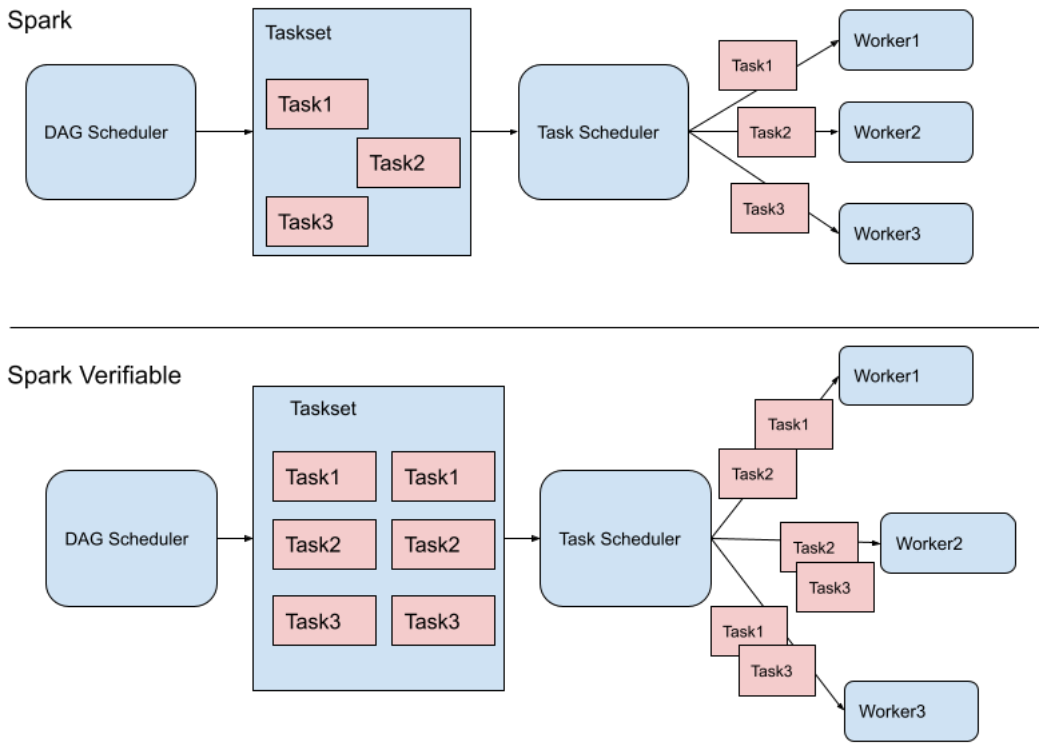
Στο σχήμα 3.2 φαίνεται πώς οι τρεις εργασίες ενός σταδίου θα κατανέμονταν σε έναν Cluster 3 κόμβων στο επιβεβαιώσιμο Spark και στο Spark.

Κατανάλωση Hashes

Στη συνέχεια, χρειάζεται μια δομή υπεύθυνη για την κατανάλωση των hashes που πλέον στέλνονται από τους Executors. Προκειμένου να γίνει αυτό, ο Master διατηρεί μια δομή στην οποία κρατά για κάθε (jobId, stageId, taskId) όλα τα hashes που λαμβάνει για κάθε τρίαδα. Αν για την ίδια τούπλα έρθουν διαφορετικά hashes τότε ο Master θεωρεί έχει γίνει λάθος στο task αυτό και έτσι η εκτέλεση της εφαρμογής σταματά με το exception NO CONSENSUS REACHED και δεν θα προκύψει αποτέλεσμα. Από την άλλη, αν για όλα τα tasks έρθουν ίδια hashes για όλες τις εκτελέσεις του ίδιου task, τότε το αποτέλεσμα θεωρείται σωστό και υπάρχει σιγουριά ως προς την ορθότητα. Η σιγουριά απορέει από τις ιδιότητες της συνάρτησης κατακερματισμού, ότι είναι πρακτικά απίθανο δύο διαφορτικά στοιχεία να έχουν ίδιο hash. Η δομή αυτή εισάγει γραμμική ως προς το πλήθος των tasks επιβάρυνση στον Master. Στον κώδικα του Task Result Getter προστίθεται η επιπλέον λειτουργικότητα να αντλεί το hash από τα metadata που φτάνουν από τους Executors και να το εισάγει στην δομή του master που αναφέρθηκε.

Επέκταση Blacklist Manager

Μια ¨επίθεση' την οποία μπορεί να πραγματοποιήσει ένας κακόβουλος κόμβος είναι να επιστρέφει συνεχώς λανθασμένα αποτελέσματα για όλες τις εργασίες που του ανατίθονται με



Σχήμα 4.2: Προγραμματισμός εργασιών στο Spark

σκοπό όλες οι εφαρμογές να τερματίζονται με μήνυμα λάθους και χωρίς να προκύψει τελικό αποτέλεσμα. Τέτοιες συμπεριφορές μπορούν να καταστήσουν τον cluster μη λειτουργικό, υπό την έννοια ότι κανείς πελάτης δεν θα μπορεί να πάρει αποτέλεσμα για κάποια εφαρμογή που καταθέτει. Προς αποτροπή τέτοιων επιθέσεων, επεκτείνεται ο Blacklist Manager έτσι ώστε να αποθηκεύει και τις αποτυχίες που οφείλονται σε έλλειψη Consensus σε μια ξεχωριστλη δομή. Επειδή όπως αναφέρθηκε παραπάνω, η παρούσα υλοποίηση δεν λύνει τις διαφορές μεταξύ workers που διαφωνούν πάνω σε μια υπολογιστική εργασία, αν για ένα task υπάρξει διαφωνία, τότε όλοι οι workers που εμπλέκονται στο συγκεριμένο task θα χρεωθούν μια αποτυχία παρόλο που είναι πολύ πιθανό ο ένας εξ΄ αυτών να είναι τίμιος και να έδωσε την σωστή απάντηση. Έτσι, η καταγραφή όλων των αποτυχιών και ο αποκλεισμός κάποιου όταν οι αποτυχίες γίνουν ίσες με ένα άνω όριο δεν είναι καλή πρακτική διότι όλοι κάποια στιγμή θα εμπλακούν σε αποτυχίες και σε βάθος χρόνου θα λάβουν άδικο αποκλεισμό. Αντ΄ αυτού, προσθέτουμε μια διεπαφή Reputation Algorithm η οποία έχει μια εσωτερική μεταβλητή workerReputation και δύο μεθόδους $reward()$ και penalize() που δρούν πάνω στο workerReputation αυξάνοντας ή μειώνοντας το αντίστοιχα. Ο κάθε ένας master μπορεί να έχει υλοποιήσει διαφορετικά το ReputationAlgorithm. Μετά από επιτυχή ολοκλήρωση ενός Job, ο master καλεί την reward() για όλους του workers, ενώ αν υπάρξει αποτυχία consensus, οι workers που εμπλέκονται στο task που προκάλεσε την αποτυχία λαμβάνουν penalize() και οι υπόλοιποι δεν δέχονται κάποια αλλαγή. Επίσης, υπάρχει ένα κατώφλι για τη φήμη και αν κάποιος κόμβος πέσει κάτω από το κατώφλι αυτό τότε αποκλείεται από το cluster για κάποιο χρόνο. Εκτιμάται ότι αν ο

αλγόριθμος είναι καλός, το σύστημα θα είναι εν τέλει ικανό σε βάθος χρόνου να αποκλείσει workers που συστηματικά επιστρέφουν λάθος αποτελέσματα για διάφορες εργασίες.

Τέλος, σε μια προσπάθεια να μειώθεί το συνολικό overhead το οποίο προκύπτει από το να τρέχουν όλα τα tasks 2 τουλάχιστον φορές, δίνεται η δυνατότητα να διαμορφωθεί ένα μικρότερο επίπεδο εμπιστοσύνης του αποτελέσματος με αντάλλαγμα μικρότερη αύξηση του χρόνου εκτέλεσης. Πιο συγκεκριμένα, ορίζεται ένα ελάχιστο reputation πάνω από το οποίο κάποιος θεωρείται σχετικά έμπιστος. Αν ένας σχετικά έμπιστος κόμβος αναλάβει ένα task, τότε το ίδιο task θα ανατεθεί και σε δεύτερο worker με κάποια πιθανότητα p ενώ με πιθανότητα 1-p μόνο αυτός ο ένας κόμβος θα τρέξει το task. Έτσι, το σύστημα επωφελείται από την ύπαρξη τίμιων κόμβων τους οποίους και ελέγχει περιοδικά για να βεβαιωθεί ότι εξακολουθούν να είναι τίμιοι. Έτσι, αν στο σύστημα μπεί ένα νέο Job από το οποίο προκύπτουν N tasks, ο μέσος αριθμός tasks που θα μπεί εν τέλει στο σύστημα είναι λιγότερα από 2N. Στην καλύτερη περίπτωση, όπου στο σύστημα απαρτίζεται μόνο από τίμιους κόμβους (μετά πολύ χρόνο και έχοντας αποκλείσει ενδεχομένως κάποιους κακόβουλους), τα tasks που θα εισαχθούν θα είναι $(1 + p)N$ (διωνυμική κατανομή) που αποτελεί βελτίωση σε σχέση με το 2N της βασικής υλοποίησης του επιβεβαιώσιμου Spark.

Κεφάλαιο 5

Πειραματική Αξιολόγηση

Οι προσθήκες και τροποποιήσεις του Spark που περιγράφησαν στο προηγούμενο κεφάλαιο είναι αναμενώμενο να εισάγουν κάποια καθυστέρηση στην διαδικασία εκτέλεσης των εφαρμογών. Σε αυτό το κεφάλαιο σκοπός είναι η αποτίμηση της καθυστέρησης αυτής. Η διαδικασία αυτή θα γίνει σε 3 στάδια:

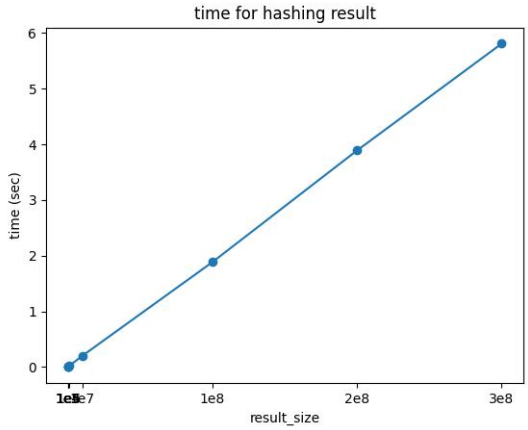
- Αποτίμιση καθυστέρησης των επιμέρους κλάσεων και μηχανισμών που εισήχθησαν, με προσομοίωση για μεταβλητό αριθμό κόμβων στον Cluster, και με μεταβλητό αριθμό tasks.
- Σύγκριση χρόνου εκτέλεσης για 3 εφαρμογές με χρήση Spark και της υπό ανάλυση παραλλαγής, χωρίς την εφαρμογή του reputation συστήματος.
- Σύγκριση χρόνου εκτέλεσης για εφαρμογές με χρήση Spark και της υπό ανάλυση παραλλαγής, με την εφαρμογή του reputation συστήματος.

Όλα τα πειράματα έγιναν στους υπολογιστές τις ουράς clones του cslab. Ο Master έχει 2 πυρήνες διαθέσιμους και 4G μνήμη και ο κάθε Worker ορίστηκε να έχει έναν Executor,με 2 πυρήνες και 2G μνήμη. Επίσης, για όλες οι μετρήσεις που εμφανίζονται στα παρακάτω διαγράμματα αντιπροσωπεύουν τον μέσο όρο 5 εκτελέσεων.

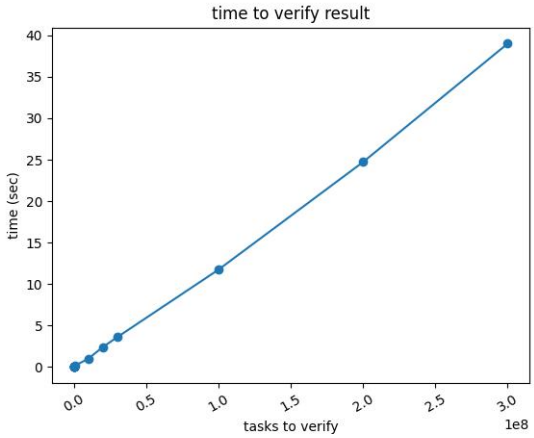
Αρχικά, μετρήθηκε ο χρόνος που απαιτείται από τον executor για να υπολογίσει το hash ενός αποτελέσματος μεγέθους μεταβλητού από της τάξης των KB έως και 300 MB. Τα αποτελέσματα φαίνονται στην επόμενη εικόνα 5.1.

Στην συνέχεια, μετρήθηκε ο χρόνος που απαιτείται από τον Master προκειμένου να διαβάσει τα hashes για μεταβλητό αριθμό tasks, έως και 100.000.000 tasks. Ίδιο πλήθος tasks πολύ να προκύψει για πολλούς διαφορετικούς συνδυασμούς αριθμού stages και tasks/ stage. Χρησιμοποιήθηκε μεταβλητός αριθμός σταδίων από 1 έως 10.000 και tasks/ stage από 2 έως 10.000 και στην συνέχεια ομαδοποιήθηκαν όλοι οι συνδυασμοί με τον ίδιο συνολικό αριθμό tasks και υπολογίστηκε ο μέσος χρόνος του απαιτείται για να πιστοποιήσει ο Master όλα τα αποτελέσματα (εικόνα 5.2).

Στην συνέχεια, εξετάζεται ο χρόνος εκτέλεσης end-to-end για 2 εφαρμογές, καθεμιά απο τις οποίες αποτελείται απο μια map(), ένα reduceByKey() και ένα sort(). Στην υποδομή που



Σχήμα 5.1: Executor hashing time



Σχήμα 5.2: Time for master to verify tasks

έχει στηθεί προσθέτουμε και ένα κατανεμημένο σύστημα αποθήκευσης αρχείων hdfs hadoop με replication 1, από το οποίο οι κόμβοι της συσταδας θα διαβάζουν τα απαραίτητα δεδομένα. Πιο συγκεκριμένα:

- Η εφαρμογή 1 δέχετεαι ως είσοδο αρχείο 1.3 GB το οποίο περιέχει στοιχεία του twitter με την δομή

$$< follower >< followee >$$

  και εκφράζει την σχέση ο $< follower >$ ακολουθεί τον $< followee >$. Σκοπός της εφαρμογή 1 είναι να βρει για κάθε χρήστη πόσους χρήστες ακολουθεί και να τους διατάξει σε φθίνουσα σειρά ως προς την τιμή αυτή.

- Η εφαρμογή 2 δέχεται ως είσοδο αρχείο 20 MB το οποίο περιέχει στοιχεία για ποδοσφαιριστές. Σκοπός της εφαρμογής είναι να βρει το πλήθος των ποδοσφαιριστών που

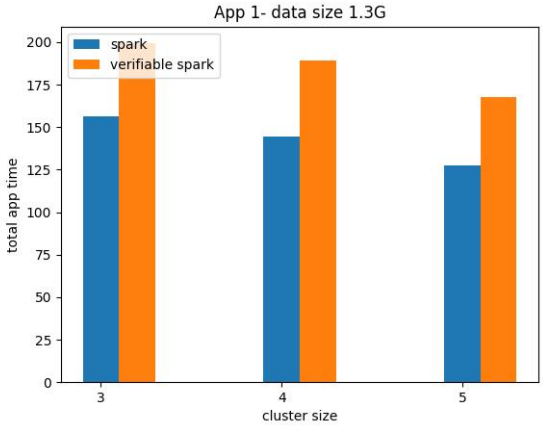
κατάγονται από κάθε χώρα και να κατατάξει τις χώρες σε φθίνουσα σειρά ως προς την τιμή αυτή.

- Η εφαρμογή 3 διαβάζει δύο αρχεία:

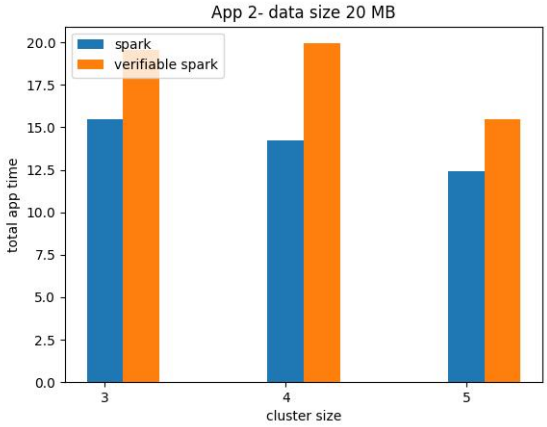
  1. Ένα αρχείο με το οποίο περιέχει στοιχεία για ταινίες, μεγέθους 1.7 MB και έχει το σχήμα:
  $$< movieid >< imdbid >< language >< title >$$

  2. Ένα αρχείο το οποίο περιέχει κριτικές χρηστών για διάφορες ταινίας και το σχήμα του είναι:
  $$< userid >< movieid >< rating >< timestamp >$$

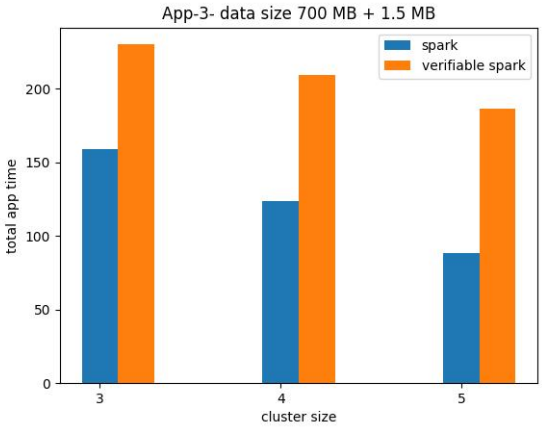
  Η εφαρμογή κάνει inner join τους δύο πίνακες πάνω στο πεδίο movie id και στην συνέχεια για κάθε τίτλο ταινίας υπολογίζεται το πλήθος των κριτικών που την αφορούν.



Σχήμα 5.3: Εφαρμογή 1

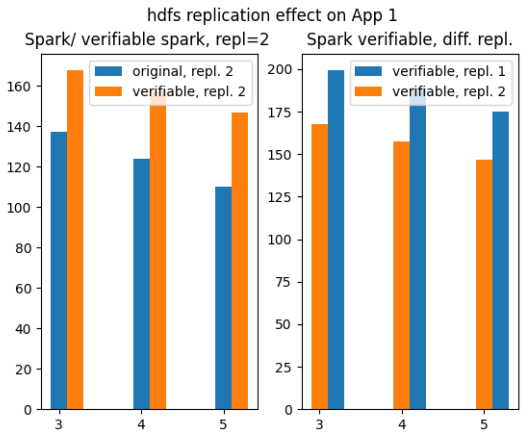


Σχήμα 5.4: Εφαρμογή 2



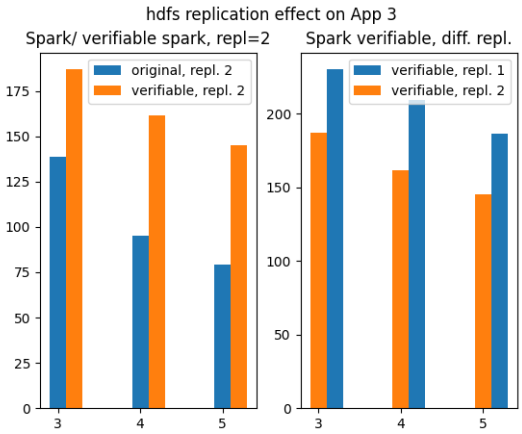
Σχήμα 5.5: Εφαρμογή 3

Από τα διαγράμματα παραπάνω είναι εμφανές ότι δεν δέχονται όλες οι εφαρμογές το ίδιο overhead όταν τρέχουν με επιβεβαιώση. Η διαφορά αυτή οφείλεται σίγουρα στο πλήθος των σταδίων και των εργασιών ανά στάδιο αλλά σημαντικό ρόλο παίζει και το κόστος της μεταφοράς δεδομένων στην μνήμη. Στην περίπτωση των εφαρμογών 1 και 3, ο όγκος των δεδομένων είναι σημαντικός και το replication factor του συστήματος κατανεμημένης αποθήκευσης που χρησιμοποιείται είναι 1. Ωστόσο, κάθε task χρειάζεται να τρέξει 2 φορές, μια σε κόμβο που έχει ήδη αποθηκευμένα τα απαραίτητα δεδομένα και μια σε κόμβο που δεν θα τα έχει και θα χρειάζεται μεταφορά. Οι μεταφορές αυτές είναι αρκετές σε πλήθος και η καθυστέρηση που εισάγουν είναι σημαντική. Για το λόγο αυτό, αυξάνεται το replication factor σε 2, και δοκιμάζονται οι εφαρμογές 1 και 3 εκ νέου. Στα επόμενα διαγράμματα φαίνεται η σύγκριση του Spark και του επιβεβαιώσιμου Spark για κάθε εφαρμογή αλλά και η σύγκριση των χρόνων εκτέλεσης του επιβεβαιώσιμου Spark για τις διαφορετικές τιμές του replication (σχήματα 5.6, 5.7).



Σχήμα 5.6: Εφαρμογή 1, replication 2

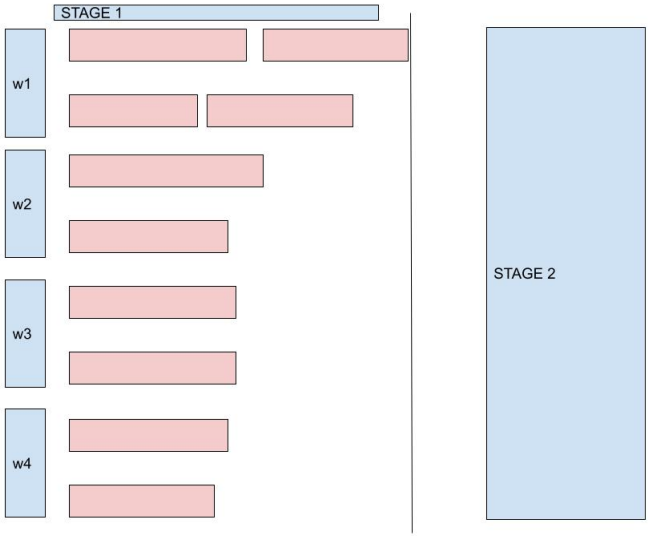
Αυτό που παρατηρείται στα 4.6, 4.7 είναι ότι η μεγαλύτερη τιμή του replication δεν βελ-



Σχήμα 5.7: Εφαρμογή 3, replication 2

τιώνει το επιβεβαιώσιμο Spark σε σχέση με το κανονικό Spark, διότι και το κανονικό Spark επωφελείται από την αύξηση του replication. Ωστόσο, η επιτάχυνση σε σχέση με το επιβεβαιώσιμο Spar με replication 1 είναι σημαντική, της τάξης του 15 τοις εκατό.

Στην συνέχεια, με χρήση του spark history server - ενός monitoring εργαλείου που ανήκει στην σουίτα του spark - παρατηρούμε ότι η εφαρμογή 2 έχει 2 στάδια και μέγεθος αρχείου εισόδου 20 MB, το οποίο χωρίζεται σε 10 partitions. Για κάθε ένα partition προκύπτει και ένα task κατά το πρώτο stage. Στην συνέχεια, είδαμε αρχικά ότι και κατά το δεύτερο stage δημιουργούνται 10 task αλλά κυρίως είδαμε πώς κατανεμήθηκαν τα tasks του πρώτου stage στους workers, το οποίο αποτυπώνεται και στο διάγραμμα 5.8.



Σχήμα 5.8: Προγραμματιμός εργασιών εφ. 2, cluster size: 4

Όπως φαίνεται, όλοι οι executors γεμίζουν με δύο tasks έκαστος εκτός από έναν ο οποίος παίρνει 4, δύο από τα οποία θα αρχίσουν να τρέχουν αμέσως και δύο από τα οποία

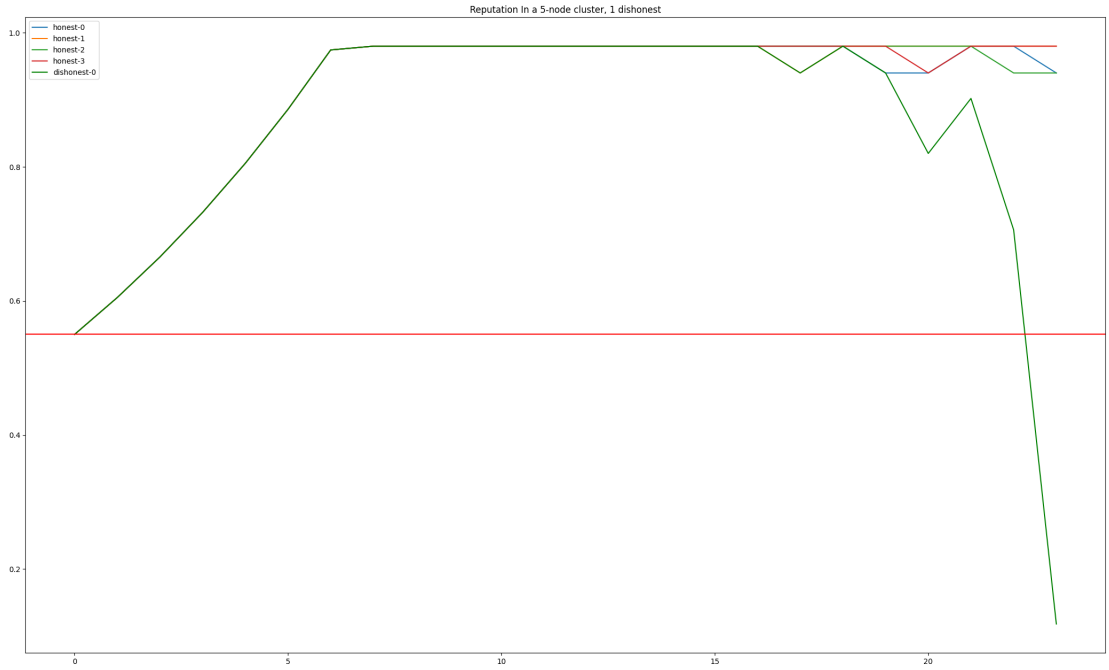
θα περιμένουν την ολοκλήρωση των προηγουμένων. Έως ότου ολοκληρωθούν και τα δύο τελευταία tasks, δεν ξεκινάει το επόμενο stage (λόγω εξαρτήσεων) και όπως φαίνεται από το history server τα resources 3 κόμβων μένουν για κάποιο διάστημα αχρησιμοποίητα. Αν λοιπόν χρησιμοπούσαμε το reputation system που προτείνεται στο προηγούμενο κεφάλαιο θα μπορούσαμε να αξιοποιήσουμε τα resources αυτά. Πιο συγκεκριμένα, υλοποιούμε ένα απλό αλγόριθμο φήμης:

$$reward() = \min(rep + 0.1rep, 1)$$

$$penalize() = \max(rep - 0.2(1.05 - rep), 0)$$

Στην συνέχεια, ορίζουμε κατώφλι αποκλεισμού 0.5, κάτω από το οποίο αποκλείεται ένα κόμβος από το σύστημα.

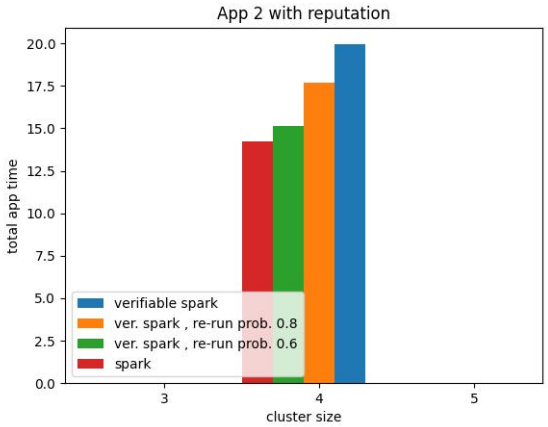
Στο διάγραμμα 5.9 φαίνεται μια προσομοίωση του παραπάνω αλγορίθμου για μια συστάδα 5 κόμβων εργατών, ο ένας εκ των οποίων κάνει επίτηδες λαθος ένα task με πιθανότητα 40 τοις εκατό. Το κάθε task αναλαμβάνεται από 2 workers και να είναι λάθος τότε και οι δύο λαμβάνουν ποινή, ενώ για τους άλλους τρεις δεν υπάρχει μεταβολή στη φήμη. Μετά από αρκετά Jobs, το σύστημα καταλαβαίνει και αποκλείει τον κακόβουλο.



Σχήμα 5.9: reputation algorithm simulation

Στην συνέχεια, θέτουμε κατώφλι έμπιστου 0.85. Αν ένας κόμβος έχει φήμη πάνω από 0.85, τότε τα tasks που αναλαμβάνει τα αναθέτουμε και σε άλλο κόμβο με πιθανότητα 0.6 και με πιθανότητα 0.4 ο έμπιστος αυτός κόμβος είναι ο μόνος που τρέχει το συγκεκριμένο task. Εφαρμόζοντας την λογική αυτή στην εφαρμογή App-2 για μέγεθος cluster 4 και θεωρώντας ότι το σύστημα ειναι σε μια κατάσταση όπου όλοι οι κόμβοι έχουν φήμη πάνω από 0.85, ο μέσος

αριθμός tasks που θα εισαχθεί και δεύτερη φορά είναι 6 από τα 10 (του stage 1) και 6 από τα 10 tasks του δεύτερου σταδίου (κατά μέσο όρο). Επαναλαμβάνουμε το ίδιο και με πιθανότητα δεύτερης ανάθεσης του ίδιου task 0.8, οπότε η πιθανότητα ένα task να τρέξει αποκλειστικά σε έναν σχετικά έμπιστο κόμβο είναι 0.2. Τα αποτελέσματα του χρόνου εκτέλεσης φαίνονται στο παρακάτων διάγραμμα (5.10).

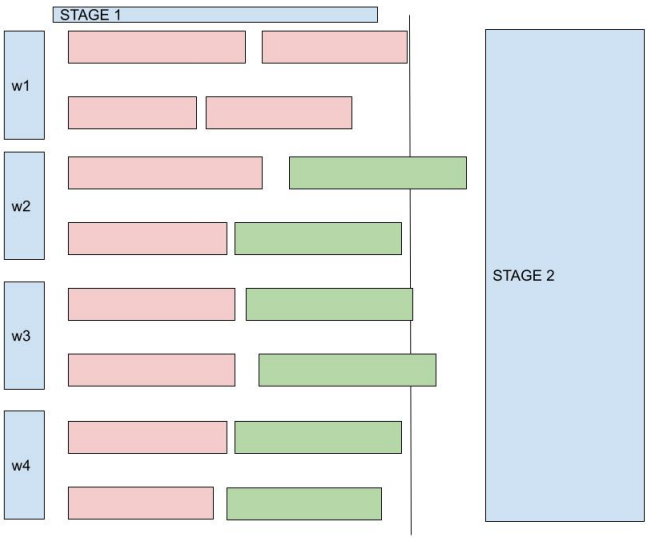


Σχήμα 5.10: App 2 reputation run

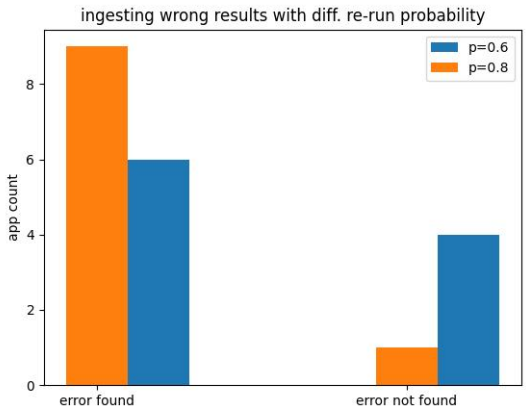
Αυτό που παρατηρείται στο 5.10 είναι ότι το επιβεβαιώσιμο spark τρέχει σε χρόνο πολύ μικρότερο σε σχέση με τη βασική υλοποίηση του επιβεβαιώσιμου Spark, όσο η πιθανότητα επανλυπολογισμού ενός task μειώνεται.

Η διαφορά ανάμεσα στον χρόνο εκτέλεσης του κανονικού Spark και του επιβεβαιώσιμου με σύστημα φήμης οφείλεται κυρίως στην διαφορά του χρόνου εκτέλεσης των διαφορετικών tasks. Προφανώς, όλα τα tasks κάνουν διαφορετικό χρόνο να ολοκληρωθούν και ανάλογα με το ποια tasks επιλέγονται για να επιβεβαιωθούν και από ποιους Workers, δημιουργείται overhead. Αυτό φαίνεται καλύτερα στην εικόνα 5.11 που δείχνει πώς περίπου είναι η νέα κατανομή των tasks του σταδίου 1 στους executors για πιθανότητα επανυπολογισμού 0.6 (και ανάλογη είναι και η εικόνα για το στάδιο 2).

Η βελτίωση αυτή στον χρόνο όμως συνοδεύεται με ένα τίμημα ως προς την εμπιστοσύνη που μπορεί να έχει ο πελάτης στο τελικό αποτέλεσμα. Είναι αναμενώμενο ότι σε κάποια στιγμή κάποιο task θα υπολογιστεί λάθος και επειδή θα έχει ανατεθεί σε έναν μόνο κόμβο το τελικό αποτέλεσμα θα είναι λάθος. Για να ποσοτικοποιηθεί αυτό το ενδεχόμενο, τρέχουμε 10 φορές την εφαρμογή 2 και μετράμε πόσες φορές οι εφαρμογή τερματίστηκε λόγω λάθους και πόσες φορές ολοκληρώθηκε επιτυχώς, για τις διαφορετικές πιθανότητες που αναφέρονται παραπάνω. Το πείραμα γίνεται στον cluster 5 κόμβων, ένας εκ των οποίων είναι κακόβουλος και κάνει λάθος επίτηδες ένα task από αυτά που ανατείθονται. Επίσης, το κατώφλι φήμης πάνω από το οποίο μπορεί ενας κόμβος να τρέξει ένα task μόνο αυτός είναι 0.85, όπως και στην παραπάνω ανάλυση. Επίσης, θεωρείται ότι όλοι οι κόμβοι έχουν αρχική φήμη 0.98, που σημαίνει ότι ο cluster για κάποιο διάστημα λειτουργούσε χωρίς προβλήματα μέχει που ένας άρχισε να δρα κακόβουλα. Το διάγραμμα 5.12 δείχνει τα αποτελέσματα του πειράματος αυτού.



Σχήμα 5.11: Task scheduling app-2, cluster size: 4



Σχήμα 5.12: Εκτέλεση εργασιών μια φορά από έμπιστο κόμβο

Από το διάγραμμα ότι όσο πιο μικρή είναι πιθανότητα να μην επαληθευτεί η εκτέλεση ενός task από άλλον κόμβο, τόσο λιγότερες είναι οι φορές που ένα λάθος περνάει χωρίς να γίνει αντιληπτό. Από την άλλη, όσο πιο πολλά είναι τα task που εκτελούνται με επιβεβαίωση, τόσο μεγαλύτερο είναι το προστιθέμενο overhead. Διαπιστώνεται λοιπόν ότι υπάρχει tradeoff το οποίο αφορά στον χρόνο εκτέλεσης εκτέλεσης των εφαρμογών και στο πόσα λάθη μπορεί να εντοπίσει το επιβεβαιώσιμο Spark. Στην συγκεκριμένη περίπτωση, με p=0.6 η εφαρμογή με το επιβεβαιώσιμο Spark έτρεξε σε χρόνο σχεδόν ίσο με το κανονικό Spark αλλά 4 στις 10 λάθος εκτελέσεις έγιναν δεκτές. Από την άλλη, η εκτέλεση με p=0.8 βρήκε 9 από τα 10 λάθη και έτρεξε σε χρόνο ενδιάμεσο του κανονικού και του επιβεβαιώσιμου. Ανάλογα με την εφαρμογή και με το αν χρειάζεται αυξημένο επίπεδο εμπιστοσύνης και αν υπάρχει ανοχή να τρέξουν πιο πολλές εργασίες (αλλά λιγότερες από 2N), τότε μπορεί να αυξηθεί το κατώφλι άνω του οποίου μπορεί ένας κόμβος να αναλάβει ένα task μόνος του ή/ και να αυξηθεί η συχνότητα με την

οποία ελέγχονται οι σχετικά έμπιστοι κόμβοι.

Κεφάλαιο 6

Μελλοντικές Επεκτάσεις

Το σύστημα που περιγράφεται αποτελεί μια πρώτη υλοποίηση, ένα proof of concept για ένα σύστημα επιβεβαιώσιμης κατανεμημένης επεξεργασίας βασισμένο στο Spark. Υπάρχουν πολλά σημεία που χρήζουν βελτίωσης αλλά και πολλές επεκτάσεις που μπορούν να γίνουν για να εξελιχθεί το σύστημα και ως προς την επίδοση και ως προς τα σενάρια τα οποία καλύπτει. Ενδεικτικά αναφέρονται κάποιες από τις επεκτάσεις που θα μπορούσαν να γίνουν:

Μη-έμπιστος Master

Η πρώτη παραδοχή η οποία έγινε στην παρούσα εργασία είναι αυτή του έμπιστου Master. Ο λόγος όπως εξηγήθηκε και παραπάνω έχει σχέση με την αρχιτεκτονική του Spark και συγκεκριμένα με τις ειδικές αρμοδιότητες τις οποίες αναλαμβάνει ο Master, όπως το scheduling των εργασιών. Αν στην παρούσα εργασία δεν εμπιστευόμασταν τον Master τότε δεν θα μπορούσαμε να εμπιστευτούμε καν ότι τα tasks τα οποία αναλαμβάνει ο κάθε Worker είναι σωστά και αφορούν στη σωστή εφαρμογή. Για να μπορέσει λοιπόν το σύστημα να δουλέψει σε σενάρια μη έμπιστου Master πρέπει όλες οι διαδικασίες που αναλαμβάνει είτε να αποκεντρωθούν και να τρέχουν απο πολλούς κόμβους οι οποίοι θα αλληλοελέγχονται, είτε να μεταφερθούν αυτές οι ευθυνες σε κάποιο έμπιστο τρίτο μέρος, όπως το Blockchain. Για τις ευθύνες του προγραμματισμού εργασιών, θα μπορούσε να χρησιμοποιηθεί ένας αλγόριθμος κατανεμημένου προγραμματισμού, όπως αυτός που περιγράφεται στο [2]. Το κομμάτι της σύγκρισης των αποτελεσμάτων των εκτελέσεων του ίδιου task από διαφορετικούς κόμβους θα μπορούσε να μεταφερθεί στο Blockchain. Το [1] περιγράφει ένα Ethereum Smart Contract που κάνει επιβεβαίωση του αποτελέσματος μιας εργασίας στο Blockchain και μαλιστά χρησιμοποιώντας τον μηχανισμό με τα hashes που χρησιμοποιήθηκε και στην παρούσα εργασία.

Εμπιστοσύνη vs Επίδοση

Η αρχική υλοποίηση του επιεβεβαιώσιμου Spark τρέχει όλα τα tasks μιας εφαρμογής τουλάχιστον 2 φορές ώστε να υπάρχει εμπιστοσύνη ως προς το αποτέλεσμα και προστασία πο αυθαιρεσία των άλλων κόμβων του Cluster. Διαπιστώθηκε ότι διπλάσιος αριθμός tasks δεν συνεπάγεται και διπλάσιος χρόνος εκτέλεσης, γιατί σημαντικό ρόλο παίζει και το σύνολο των

πόρων του συστήματος. Στην συνέχεια, είδαμε ότι αν υπάρχει σχετική εμπιστοσύνη - όπως αυτή ορίζεται απο τη συνάρτηση φήμης - μπορούν κάποια tasks να τρέξουν μόνο μια φορά με την προυπόθεση ότι θα τρέξουν σε έναν κόμβο ο οποίος έχει πολύ καλή φήμη, δηλαδή έχει συμπεριφερθεί σωστά για πολλές υποβληθείσες εργασίες στο παρελθόν. Αρχικά, θα μπορούσε να γίνει μια πιο ενδελεχής μελέτη σχετικά με τον αλγόριθμο διαχείρησης φήμης σε σχέση με τον μάλλον απλοικό που επιλέχθηκε στην παρούσα εργασία. Στην συνέχεια, θα μπορούσε να υπάρξει ένα νέο κομμάτι λογισμικού το οποίο με συστηματικό τρόπο θα προσπαθεί να επιτύχει το μέγιστο επίπεδο εμπιστοσύνης με το μικρότερο δυνατό overhead. Πιο συγκεκριμένα, λαμβάνοντας ως είσοδο από τον χρήστη το μέγιστο ανεκτό overhead, κάνοντας στατική ανάλυση στον κώδικα της εφαρμογής και λαμβάνοντας υπ΄ όψιν τους διαθέσιμους πόρους του συστήματος θα υπολογίζει έναν συνδυασμό του πόσα tasks να τρέξουν πολλαπλές φορές και ποιοί είναι αυτοί που μπορούν να αναλάβουν. Ο συνδυασμός αυτός θα πρέπει να έχει εκτιμώμενο overhead μικρότερο του μέγιστου ανεκτού και ικανοποιητικό επίπεδο εμπιστοσύνης.

Κίνητρο Συμμετοχής

Ένα σύστημα κατανεμημένης επεξεργασίας μπορεί να χρησιμοποιηθεί σε σενάρια που δίνεται η δυνατότητα σε αγνώστους να συνεισφέρουν στον cluster πόρους προκειμένου να είναι δυνατή η καλύτερη κλιμάκωση της συστάδας. Όσοι αποφασίζουν να συμμετάσχουν στον Cluster μπορούν επίσης να τον χρησιμοποιούν για τις δικιές τους εργασίες. Το πρόβλημα προκύπτει όταν όλοι μπαίνουν στην συστάδα μόνο για να τον χρησιμοποιήσουν και δεν είναι διατεθημένοι να συνεισφέρουν πόρους. Για να αντιμετωπισθεί το πρόβλημα αυτό πρέπει να υπάρχει ένα κίνητρο για τους συμμετεχοντες. Τέτοια κίνητρα μπορεί να είναι οικονομικά ή και πρακτικά, με εφαρμογή π.χ. ενός πρωτοκόλλου που εισάγει quotas για κάθε κόμβο. Ένα παράδειγμα quotas είναι το πηλίκο των πόρων που κατανάλωσε ο κάθε κόμβος προς τους πόρους που έχει συνεισφέρει να μην υπερβαίνει ένα όριο.

Βιβλιογραφία

[1] Sepideh Avizheh, Mahmudun Nabi, Reihaneh Safavi-Naini, Muni Venkateswarlu K.. 2019. Verifible Computation using Smart Contracts. In 2019 Cloud Computing Security Workshop (CCSW'19), November 11, 2019, London, United Kingdom. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/ 3338466.3358925.

[2] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). Association for Computing Machinery, New York, NY, USA, 69–84. DOI:https://doi.org/10.1145/2517349.2522716 .

[3] https://zokrates.github.io/introduction.html .

[4] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. Cryptology ePrint Archive, Report 2013/279. .

[5] https://books.japila.pl/apache-spark-internals/ .

[6] Jason Teutsch, Christian Reitwießner. 2017. A scalable verification solution for blockchains. https://people.cs.uchicago.edu/ teutsch/papers/truebit.pdf .

[7] Petkus, M. (2019). Why and How zk-SNARK Works. CoRR, abs/1906.07221. http://arxiv.org/abs/1906.07221 .

[8] https://hackage.haskell.org/package/arithmetic-circuits .

[9] https://z.cash/technology/zksnarks/ .

[10] https://www.investopedia.com/terms/z/zksnark.asp .