ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Χρήση Υποδομής ως Κώδικας και Δυναμική Ανάθεση Εργασιών σε Συστήματα Κατανεμημένης Επεξεργασίας Δεδομένων Βασισμένα σε Υπολογιστικά Νέφη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΣΑΛΙΑΓΚΟΣ Ν. ΚΥΡΙΑΚΟΣ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Χρήση Υποδομής ως Κώδικας και Δυναμική Ανάθεση Εργασιών σε Συστήματα Κατανεμημένης Επεξεργασίας Δεδομένων Βασισμένα σε Υπολογιστικά Νέφη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΣΑΛΙΑΓΚΟΣ Ν. ΚΥΡΙΑΚΟΣ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Νοεμβρίου 2021.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.	Ιωάννης Κωνσταντίνου Επίκουρος Καθηγητής Πανεπιστήμιο Θεσσαλίας

Αθήνα, Νοέμβριος 2021

.....

Τσαλιαγκός Ν. Κυριάκος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Ευχαριστίες

Με την εκπόνηση της παρούσας διπλωματικής εργασίας ολοκληρώνεται ο κύκλος σπουδών μου στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών.

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον κ. Νεκτάριο Κοζύρη, Καθηγητή Ε.Μ.Π., για την επίβλεψή του και την ευκαρία που μου προσέφερε να ασχοληθώ με το σύγχρονο και εξαιρετικά ενδιαφέρον θέμα της εργασίας. Επίσης, θα ήθελα να ευχαριστήσω τον κ. Ιωάννη Κωνσταντίνου, Επίκουρο Καθηγητή Παν. Θεσσαλίας, για τη συνεργασία και τη συνεχή καθοδήγησή του.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου για την απεριόριστη στήριξη και συμπαράσταση που μου παρείχαν σε όλη τη διάρκεια των σπουδών μου.

Κυριάκος Τσαλιαγκός,
Αθήνα, 23η Νοεμβρίου 2021

Περίληψη

Στη δεύτερη δεκαετία του 21ου αιώνα, οι απαιτήσεις για λήψη αποφάσεων οδηγούμενη από δεδομένα είναι υψηλότερες από ποτέ. Είτε πρόκειται για ερμηνεία των κινήσεων της αγοράς και των προτιμήσεων των καταναλωτών, με σκοπό την ανάπτυξη μιας επιχείρησης και την μεγιστοποίηση των κερδών της, ή για την ανάλυση ιατρικών δεδομένων εν μέσω μιας παγκόσμιας πανδημίας, με σκοπό την καταπολέμηση ενός ιού και την ανάπτυξη μιας κατάλληλης θεραπείας, υπάρχει συλλογή ενός τεράστιου όγκου δεδομένων που προορίζεται για αποθήκευση και επεξεργασία.

Σε μία περίοδο που ο χρόνος-προς-την-αγορά διαρκώς μειώνεται, ειδικά στην ευρύτερη αγορά του Λογισμικού, παρατηρείται μία πρωτοφανής αλλαγή στον τρόπο επεξεργασίας αυτών των δεδομένων. Το μεγαλύτερο μέρος των προσπαθειών εστιάζει στην παράλληλη επεξεργασία δεδομένων και την ανάπτυξη αλγορίθμων και συστημάτων που υποστηρίζουν αυτή την προσέγγιση. Κρίνοντας ότι η δημιουργία και η συντήρηση ιδιωτικών συστάδων υπολογιστών είναι ασύμφορη οικονομικά, οι χρήστες στρέφονται σε περιβάλλοντα υπηρεσιών νέφους και χρησιμοποιούν την υποδομή που έχουν δημιουργήσει εταιρείες-κολοσσοί του χώρου.

Ταυτόχρονα, με την αύξηση στην πολυπλοκότητα των συστημάτων και ειδικότερα όταν αυτά είναι απομακρυσμένα και κατανεμημένα, γίνεται εξαιρετικά δύσκολο να επιχειρηματολογήσει κανείς για το τι είναι ανεπτυγμένο, καθώς και σε περίπτωση σφάλματος να μεταβεί σε μία προηγούμενη ορθή κατάσταση. Για αυτό το λόγο, αναπτύσσονται εργαλεία διαχείρισης που στηρίζονται στον κώδικα και επιτρέπουν τη εφαρμογή όλων των δοκιμασμένων τεχνικών του στη διαχείριση υποδομής.

Σε αυτή τη διπλωματική εργασία, γίνεται μία προσπάθεια συνδυασμού των διαφορετικών αυτών απαιτήσεων, ωστόσο με τρόπο που να συμπληρώνονται μεταξύ τους. Συγκεκριμένα, μετά από μία επαρκή θεωρητική ανάλυση των υπολογιστικών νεφών, της Υποδομής ως Κώδικα και της παράλληλης επεξεργασίας δεδομένων, θα χρησιμοποιηθεί το Terraform ώστε να αναπτυχθεί μία συστάδα υπολογιστών για το Spark στο Amazon EC2 νέφος. Επιπλέον, θα αξιολογηθούν οι δυνατότητες που προσφέρει το Spark για δυναμική ανάθεση εργασιών, με σκοπό την ελαχιστοποίηση σπατάλης στους χρησιμοποιούμενους πόρους.

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Υπολογιστικό Νέφος

1.1 Βασικές Αρχές και Πλεονεκτήματα

Ο όρος 'Υπολογιστικό Νέφος' σημειώνεται για πρώτη φορά σε εσωτερικό αρχείο της εταιρείας Compaq το 1996, ενώ το νέφος έχει χρησιμοποιηθεί ως αφαιρετικό σύμβολο για δίκτυα υπολογιστών ήδη από τα τέλη της δεκαετίας του ΄70 και το ARPANET, πρόγονο του σημερινού Internet. Η άνθηση του, όμως, ξεκινάει το 2006, με την κυκλοφορία του Elastic Compute Cloud (EC2) από την Amazon και την επανάσταση που αυτό έφερε στον τρόπο που αντιλαμβανόμαστε την Ανάπτυξη Λογισμικού.

Έχοντας κοινά χαρακτηριστικά με ένα αριθμό διαφορετικών υπολογιστικών μοντέλων, όπως το Μοντέλο Εξυπηρετητή-Πελάτη, τα Δίκτυα Ομότιμων Κόμβων, η Υπολογιστική Πλέγματος κ.ά., το Υπολογιστικό Νέφος προσφέρει σημαντικά πλεονεκτήματα στην ανάπτυξη, διαχείριση και επίβλεψη εφαρμογών. Αρχικά, το μοντέλο κόστους διαφοροποιείται και οι κεφαλαιουχικές δαπάνες αντικαθίστανται από τις λειτουργικές. Η ευθύνη εγκατάστασης και επιτήρησης της υπολογιστικής υποδομής αναλαμβάνεται εξ ολοκλήρου από τον εκάστοτε πάροχο υπηρεσιών νέφους, με αποτέλεσμα η κοστολόγηση να είναι ανάλογη της χρήσης υπολογιστικών πόρων. Συνεπώς, απαλείφεται η ανάγκη της εξαρχής κατάθεσης μεγάλων χρηματικών ποσών καθώς και η στελέχωση με προσωπικό με την κατάλληλη τεχνογνωσία διαχείρισης συστημάτων.

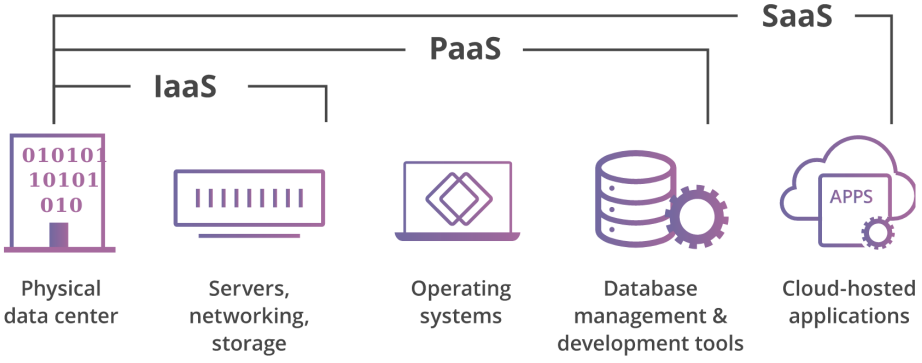
Παράγοντας που έχει συμβάλλει στην υιοθέτηση των υπολογιστικών νεφών είναι και η ανάπτυξη του κλάδου της Εικονικοποίησης και των Αρχιτεκτονικών Μικροϋπηρεσιών, οι οποίες τεχνικές στοχεύουν στο διαχωρισμό λογισμικού και υλικού και διευκολύνουν τη δημιουργία απομονωμένων, χαλαρά διασυνδεδεμένων 'εικονικών περιβάλλοντων'. Ο συνδυασμός τους δε με το υπολογιστικό νέφος δημιουργεί μια πλατφόρμα στην οποία η κατανομή πόρων είναι ταχεία, αποδοτική και η οποία επιτρέπει την πολυπληρωμή, της πλήρη συνύπαρξη δηλαδή διαφορετικών τύπων χρηστών, με διαφορετικές ανάγκες και απαιτήσεις.

Το μεγαλύτερο όμως πλεονέκτημα ίσως είναι η κλιμακωσιμότητα και η ελαστικότητα, δηλαδή η ικανότητα δυναμικά και σε πραγματικό χρόνο οι διαθέσιμοι πόροι

να αυξομειώνονται κατά βούληση. Σε ώρες αιχμής, ο αυξημένος φόρτος ικανοποιείται από αντίστοιχη αύξηση στους, φαινομενικά άπειρους, πόρους, ενώ σε διαστήματα μειωμένης κίνησης οι περιττοί πόροι αποδεσμεύονται. Η προηγμένη αυτή λειτουργία καθίσταται δυνατή από ικανά συστήματα επιτήρησης-προειδοποίησης που παρέχονται από τους παρόχους, ενώ για πιο λεπτόκοκκο έλεγχο μπορούν να εφαρμοστούν τεχνικές Μηχανικής Μάθησης, με απώτερο σκοπό την πιο αποδοτική και με λιγότερη αναμονή δέσμευση πόρων, την αύξηση της πολυπληρωμής, τη μείωση του κόστους ή και της ενεργειακής κατανάλωσης, ειδικά αν αναλογιστεί κανείς το περιβαλλοντικό αντίκτυπο ενός μεγάλου κέντρου δεδομένων.

Τελικά, παράλειψη θα ήταν να μην αναφερθούν μερικά από τα, λίγα αλλά σημαντικά, μειονεκτήματα της χρήσης υπολογιστικών νεφών. Η κατανεμημένη λογική πολλαπλασιάζει αδιαμφισβήτητα τη σχεδίαση και την ομαλή συνεργασία λειτουργιών, είτε αυτό αφορά την απουσία οικουμενικού ρολογιού και την προσπάθεια σειριοποίησης γεγονότων, είτε τη συνοχή και συνάφεια των δεδομένων όπως αυτά φαίνονται στους διάφορους χρήστες-λειτουργίες. Από πλευράς ασφάλειας, αν και τα συστήματα προστασίας είναι εξαιρετικά ανεπτυγμένα και εξειδικευμένα, σε μια κατανεμημένη αποθήκη δεδομένων πάντα ελοχεύει ο κίνδυνος μιας επίθεσης σε έναν αδύναμο κόμβο. Όσον αφορά το ζήτημα εμπιστοσύνης μεταξύ χρήστη και παρόχου, δεν είναι λίγες οι φορές που κολλοσοί έχουν κατηγορηθεί για πώληση ευαίσθητων δεδομένων σε τρίτους. Με την πρόοδο στο σχεδιασμό δικλείδων ασφαλείας και την αυστηροποίηση στις πολιτικές περί εκμετάλλευσης προσωπικών δεδομένων, είναι σίγουρο πως θα εξαλειφθούν τα όποια περιστατικά διαρροής.

1.2 Μοντέλα Υπηρεσιών



1.2.1 Υποδομή ως Υπηρεσία

Η Υποδομή ως Υπηρεσία είναι το πρώτο και χαμηλότερο στρώμα αφαίρεσης σε ένα υπολογιστικό νέφος και αναφέρεται στην παροχή ενός συνόλου υπηρεσιών που περιλαμβάνει μεταξύ άλλων φυσικές μηχανές, δομές για αποθήκευση δεδομένων, τείχη προστασίας, υποδομές δικτύου και αντίγραφα ασφαλείας. Τα παραπάνω καθίστανται

δυνατά μέσω ενός υπερεπόπτη, ο οποίος συγκεντρώνει πόρους για τη δημιουργία ανεξάρτητων Εικονικών Μηχανών και επιβλέπει τη χρήση τους. Εναλλακτικά, χρησιμοποιείται Linux Containerization για την ανάπτυξη απομονωμένων περιβάλλοντων χρήστη εντός ενός πυρήνα Linux, κατευθείαν πάνω από το φυσικό υλικό, απαλείφοντας έτσι την ανάγκη υπερεπόπτη, με τα αντίστοιχα κέρδη στην επίδοση. Το συγκεκριμένο μοντέλο συνεπώς αφήνει στον τελικό χρήστη την ευθύνη εγκατάστασης και διαχείρισης λειτουργικού συστήματος και εφαρμογών, ενώ το κόστος προκύπτει άμεσα από τη διάρκεια δέσμευσης υπολογιστικών πόρων.

1.2.2 Πλατφόρμα ως Υπηρεσία

Στο αμέσως ανώτερο επίπεδο παρεχόμενων υπηρεσιών νέφους βρίσκεται η Πλατφόρμα ως Υπηρεσία. Επιπρόσθετα με τις παρεχόμενες από την Υποδομή ως Υπηρεσία δυνατότητες, εδώ ο πάροχος έχει διαμορφώσει το λειτουργικό σύστημα και όλα τα απαραίτητα εργαλεία - βιβλιοθήκες, περιβάλλοντα χρόνου εκτέλεσης, υποστήριξη γλωσσών προγραμματισμού, διακομιστές ιστού - για την ανάπτυξη εφαρμογών. Ουσιαστικά, παρέχεται μια έτοιμη υπολογιστική πλατφόρμα και ο χρήστης μένει μόνο να την προσαρμόσει σύμφωνα με τις ανάγκες του και να αναπτύξει μια κατάλληλη εφαρμογή για την εκμετάλλευση της.

1.2.3 Λογισμικό ως Υπηρεσία

Τέλος, στην κορυφή της πυραμίδας βρίσκεται το Λογισμικό ως Υπηρεσία. Αποτελεί την πιο ολοκληρωμένη λύση εξωτερικής ανάθεσης συντήρησης υλικού και λογισμικού. Το παρεχόμενο λογισμικό, που αποτελείται από μία εφαρμογή ή/και βάση δεδομένων, καλύπτει απολύτως τις επιχειρηματικές ανάγκες του χρήστη. Ως μία κατά παραγγελία υπηρεσία, η κοστολόγηση πραγματοποιείται συνήθως με τη μορφή συνδρομής καθώς και με βάση τη χρήση που πραγματοποιείται (π.χ. μια βασική τιμή μέχρι να ξεπεραστεί ένα προκαθορισμένο πλαφόν και μετά αναλογικά με βάση κάποιο τιμοκατάλογο), ενώ άλλες φορές πιθανώς η χρήση βασικών αλλά περιορισμένων δυνατοτήτων του λογισμικού να είναι δωρεάν και η πλήρης εμπειρία να συγκρατείται για πακέτα επί πληρωμή. Η διαχείριση των πόρων, η αυξομείωση τους ανάλογα με το φόρτο, η επιτήρηση όπως επίσης και οποιαδήποτε άλλη διαχειριστική ανάγκη χρειάζεται να καλυφθεί από το χρήστη σε κάποιο από τα υπόλοιπα μοντέλα υπηρεσιών νέφους, εδώ αναλαμβάνεται με διαφανή τρόπο από τον πάροχο και ο χρήστης απλά χρειάζεται ένα λεπτό πελάτη, όπως ένα φυλλομετρητή ιστού, για την προσπέλαση της υπηρεσίας.

1.3 Μοντέλα Ανάπτυξης

Επιλέγοντας κανείς να χρησιμοποιήσει υπολογιστικό νέφος, ένα επιπλέον σημαντικό δίλημμα που συναντά είναι αυτό των Μοντέλων Ανάπτυξης. Συγκεράσοντας χαρακτηριστικά όπως η τοποθεσία των δεδομένων, ο τύπος του φόρτου εργασίας, οι ανάγκες επίδοσης καθώς και τα θέματα ασφαλείας, το καθένα από τα παρακάτω μοντέλα μοιάζει να ξεχωρίζει σε συγκεκριμένες περιπτώσεις.

1.3.1 Δημόσιο Νέφος

Η πιο συνηθισμένη και ίσως η πρώτη επιλογή που έρχεται στο μυαλό, όταν κανείς εξετάζει το ενδεχόμενο χρήσης υπηρεσιών νέφους, είναι το Δημόσιο Νέφος. Προσφέρει τη μεγαλύτερη δυνατή ελαστικότητα και κλιμακωσιμότητα, όπως επίσης και το μικρότερο χρόνο προσαρμογής. Ταυτόχρονα, όμως, η προσβασιμότητα του μέσω δικτύου και ο διαμοιρασμός των πόρων μεταξύ διαφορετικών χρηστών προκαλεί αυξημένες ανάγκες για ασφάλεια. Μοιάζει ιδανικό, μεταξύ άλλων, για υπολογισμούς παρτίδας καθώς και για περιπτώσεις που τα δεδομένα αποθηκεύονται επίσης στο δημόσιο νέφος, οπότε και για μείωση του κόστους μεταφοράς θα προτιμούσε κανείς να μεταφέρει τον 'υπολογισμό στα δεδομένα' και όχι το αντίθετο.

1.3.2 Ιδιωτικό/Εντός-του-Χώρου Νέφος

Αντίθετης φιλοσοφίας λύση είναι το Ιδιωτικό Νέφος. Πρόκειται για μια ιδιωτική υπολογιστική υποδομή που βρίσκεται εντός του χώρου μιας ευρύτερης επιχείρησης, οργανισμού ή ομάδας χρηστών και προσφέρει επιπρόσθετους πόρους για απαιτητικές εργασίες, ή αυξημένα πρωτόκολλα ασφάλειας. Σίγουρα αποτελεί μία καλή λύση για περιπτώσεις που δεδομένα τα οποία δεν μπορούν να αποθηκευτούν σε δημόσιο νέφος χρειάζεται να αποτελέσουν είσοδο σε μια, παραδείγματος χάριν, πολύπλοκη αναλυτική δουλειά. Επιπλέον, προσφέρει δυνατότητες προσαρμογής και αναβάθμισης του υλικού, ενώ δεν πρέπει να παραληφθεί το γεγονός ότι τον τελικό έλεγχο των δεδομένων τον έχει ο ίδιος ο ιδιοκτήτης τους και όχι κάποιος κεντρικός πάροχος. Αντικείμενο κριτικής, όμως, αποτελεί το γεγονός ότι μοιάζει να αναιρεί όλα εκείνα τα ελκυστικά

χαρακτηριστικά του δημόσιου νέφους, μιας και εισάγει επιπρόσθετα κόστη ιδιοκτησίας, λειτουργίας, διασύνδεσης και επιτήρησης ενός μεγάλου αριθμού υπολογιστικών πόρων και μέσων αποθήκευσης. Συγκριτικά με ένα δημόσιο νέφος, ο χρόνος-προς-την-αγορά καθώς και οι κεφαλαιακές ανάγκες αυξάνονται, προσφέροντας συνολικά μια αποδεκτή αλλά λιγότερο ευέλικτη και περισσότερο επικίνδυνη λύση.

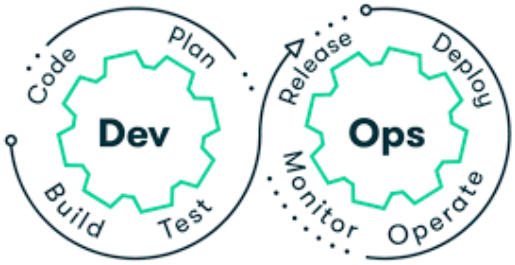
1.3.3 Υβριδικό Νέφος

Το Υβριδικό Νέφος, όπως μαρτυρά και το όνομα του, είναι ένα σύνολο λύσεων που σκοπό έχουν να συνδυάσουν τα θετικά των προηγούμενων δύο μοντέλων και να προσφέρουν μια, όσο το δυνατόν, βέλτιστη λύση. Παίρνει διάφορες μορφές, είτε ως παροχή μιας ολοκληρωμένης επιχειρησιακής υπηρεσίας, διασυνδέοντας λειτουργίες από ένα ιδιωτικό νέφος με εφαρμογές επιχειρησιακής ευφίας σε ένα δημόσιο νέφος, ή ως μία διεύρυνση της δεξαμενής διαθέσιμων πόρων. Συγκεκριμένα, η τελευταία περίπτωση είναι γνωστή ως 'έκρηξη νέφους'. Οι μηχανικοί, συνδυάζοντας τεχνικές ανάπτυξης λογισμικού και λειτουργιών πληροφορικής, σχεδιάζουν εφαρμογές που μπορούν εύκολα να επεκταθούν στο δημόσιο νέφος, για περαιτέρω δέσμευση πόρων, σε σενάρια που παρατηρείται μια ακίδα στη ζήτηση και το ιδιωτικό νέφος, το οποίο είναι σχεδιασμένο για μεσαίου μεγέθους φόρτη, αδυνατεί να παρέχει τις απαιτούμενες επιδόσεις, όπως αυτές ορίζονται από συμφωνίες σε επίπεδο υπηρεσίας ή από ανάγκες για ταχύτερη παραγωγή και λήψη αποφάσεων. Εναλλακτικά, ένα υβριδικό νέφος μπορεί να αποτελείται από ετερογενείς μεταξύ τους πόρους, για την εκμετάλλευση νέων τεχνολογιών από τις εφαρμογές. Ενδεικτικά παραδείγματα αποτελούν η συνύπαρξη επεξεργαστών αρχιτεκτονικών ARM και x86-64, ή η χρήση καρτών γραφικών για εκπαίδευση νευρωνικών δικτύων και για παράλληλη επεξεργασία. Τέλος, υβριδικό μπορεί να θεωρηθεί και το εικονικό ιδιωτικό νέφος, ένα ιδιωτικό νέφος δηλαδή το οποίο φιλοξενείται σε ένα δημόσιο νέφος. Με χρήση εικονικών τοπικών δικτύων, υποδικτύων, τειχών προστασίας και κρυπτογράφησης, ο πάροχος νέφους προσφέρει μια πλήρως απομονωμένη πλατφόρμα με ελεγχόμενη πρόσβαση, διατηρώντας όμως την κλιμακωσιμότητα, την πληθώρα διαθέσιμων πόρων και την γενικότερη άνεση και ελαστικότητα μιας υποδομής δημοσίου νέφους.

Κεφάλαιο 2

Υποδομή ως Κώδικας

2.1 Το Κίνημα ΑνάπτυξηΔιαχείριση - DevOps



Παραδοσιακά, η Ανάπτυξη Λογισμικού έχει διακριτά στάδια, στα οποία εμπλέκονται ανεξάρτητες ομάδες με συγκεκριμένες ευθύνες-στόχους. Ο στόχος, βέβαια, παραμένει η διανομή του εκάστοτε λογισμικού στον τελικό χρήστη-πελάτη, ο οποίος δεν ενδιαφέρεται για τη δομή μιας εταιρείας και το πως συνεργάζονται τα διάφορα κομμάτια της, αλλά θέλει μια, όσο το δυνατόν, άρτια εμπειρία χρήστη, η οποία προέρχεται από ποιοτικές μετρικές όπως η ευκολία στη χρήση ή η αξιοπιστία της εφαρμογής. Απαραίτητο είναι λοιπόν οι ομάδες αυτές να συνδυάσουν τις δυνάμεις τους, παρά τις όποιες διαφορές στη φιλοσοφία και τα εργαλεία τους.

Η πρώτη από αυτές τις ομάδες ασχολείται με το λογισμικό αυτό καθαυτό, ήτοι τον πηγαίο κώδικα. Στόχος η δημιουργία λογισμικού το οποίο δουλεύει τουλάχιστον στον προσωπικό τους σταθμό εργασίας και δεν δημιουργεί συγκρούσεις με τον αντίστοιχο κώδικα στους υπόλοιπους σταθμούς της ομάδας. Όταν η ομάδα φτάσει σε ένα σημείο που θεωρεί ότι είναι καλό οι νέες προσθήκες να γίνουν ορατές στην 'παραγωγή', τις συνδυάζει σε ένα εκτελέσιμο, το οποίο και παραδίδει στη δεύτερη ομάδα. Το έργο της πρώτης ομάδας έχει κάπου εδώ τελειώσει.

Η δεύτερη ομάδα είναι η υπεύθυνη για τη διαχείριση υλικού, ξεκινώντας από τα ράφια με εξυπηρετητές, τους δρομολογητές δικτύου, τις διασυνδέσεις τους και την

ψύξη και καταλήγοντας στην αρχικοποίησή τους, την εγκατάσταση του παραγόμενου λογισμικού και των εξαρτήσεων του. Με την αύξηση του μεγέθους της εταιρείας, του υλικού καθώς και των νέων εκδόσεων του λογισμικού, οι εξυπηρετητές, το στήσιμο των οποίων είναι σχεδόν απόλυτα χειρονακτικό, παρουσιάζουν ελαφρές αποκλίσεις μεταξύ τους, με συνέπεια οι εκδόσεις να είναι ασταθείς και επιδεκτικές σε σφάλματα διαμόρφωσης. Αυτή η μετατόπιση διαμόρφωσης είναι που καθιστά απαγορευτικές τις συχνές εκδόσεις, με τις μηνιαίες, μέχρι και εξαμηνιαίες, εκδόσεις να προκρίνονται ως λύση, για να περιοριστούν ο χρόνος διακοπής εφαρμογής αλλά και ο κόπος για το συνδυασμό των διαφορετικών και συχνά συγκρουόμενων κομματιών λογισμικού που λαμβάνουν από την ομάδα ανάπτυξης.

Η ΑνάπτυξηΔιαχείριση είναι μια νοοτροπία αλλαγής παραδείγματος που θεμελιώνεται στη χρήση του Υπολογιστικού Νέφους και της Εικονικοποίησης και στοχεύει στη γρηγορότερη, ορθότερη, αποδοτικότερη, πιο αυτοματοποιημένη και ελεγχόμενη παραγωγή και διανομή λογισμικού. Οι ομάδες προγραμματιστών και διαχειριστών συστήματος, ενώ διατηρούν κάποιες διακριτές αρμοδιότητες, έχουν πλέον στενότερη συνεργασία και αλληλεπίδραση. Συγκεκριμένα, οι διαχειριστές χρησιμοποιούν ήδη γνωστές και αποδεδειγμένες πρακτικές της Τεχνολογίας Λογισμικού και τις εφαρμόζουν στην παροχή και διαχείριση υποδομής μέσω κώδικα. Εναλλακτική ονομασία αυτής της πρακτικής είναι η Υποδομή ως Κώδικας.

2.2 Κατηγορίες εργαλείων Υποδομής ως Κώδικα

Η χρήση κώδικα για την ανάπτυξη, ενημέρωση ακόμα και καταστροφή της υποδομής απαιτεί νέες τεχνικές και εξειδικευμένα εργαλεία. Οι γενικού σκοπού γλώσσες προγραμματισμού και η ελευθερία που προσφέρουν αποτυγχάνουν να ορίσουν βέλτιστες πρακτικές, μιας και απαιτούν τη δημιουργία μιας νέας, εξατομικευμένης λύσης για κάθε πρόβλημα. Η χρήση επί τούτω σεναρίων είναι πολύ καλή επιλογή για τη γρήγορη ολοκλήρωση μικρών, μονότονων και εφάπαξ εργασιών. Κρίνεται ακατάλληλη, όμως, όταν πρόκειται για τη διαχείριση μέσω κώδικα μιας πολύπλοκης υποδομής, που μπορεί να αποτελείται από ένα εκτενές και ετερογενές σύνολο εξυπηρετητών, βάσεων δεδομένων, τειχών προστασίας, εξισορροπιστών φόρτου και διασυνδέσεων και απαιτεί συνεχή παρακολούθηση, συντήρηση και προσαρμογή. Ως αποτέλεσμα, νέα εργαλεία έχουν έρθει στο προσκήνιο, το καθένα από τα οποία προσφέρει κάτι διαφορετικό, αλλά όλα έχουν ως γνώμονα τη διεκόλυνση της αυτοματοποίησης και τον ενστερνισμό του 'DevOps' τρόπου σκέψης και ροής εργασιών.

2.2.1 Εργαλεία Διαχείρισης Διαμόρφωσης

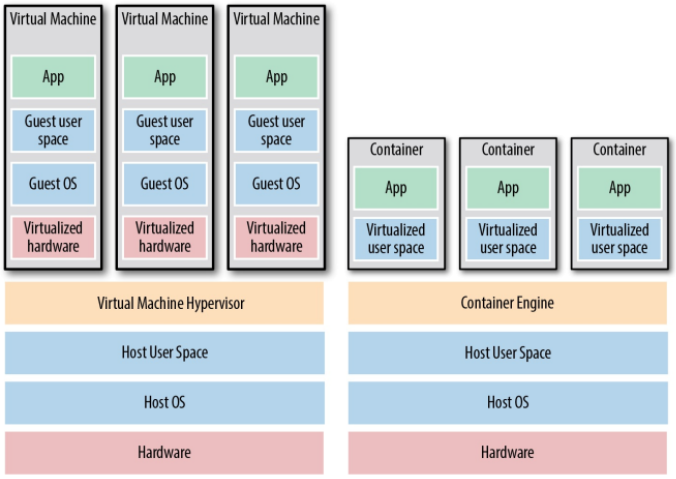
Τα εργαλεία διαχείρισης διαμόρφωσης είναι σχεδιασμένα για την πανομοιότυπη διαμόρφωση ενός μεγάλου αριθμού ήδη υπάρχοντων εξυπηρετητών, με τρόπο προβλεπόμενο, συστημικό και ταυτοτικό. Η επιβολή σταθερής δομής και συμβάσεων στα

αρχεία περιγραφής τους έχει ως αποτέλεσμα πιο ευανάγνωστο και διατηρήσιμο κώδικα, ενώ καθιστά τον εντοπισμό σφαλμάτων ευκολότερο και πιο στοχευμένο. Ταυτόχρονα, αναλαμβάνουν διάφορους ελέγχους ορθότητας στο παρασκήνιο, διευκολύνοντας και απλοποιώντας έτσι τη συγγραφή κώδικα.

Η διαμόρφωση των εξυπηρετητών γίνεται, κατά κύριο λόγο, μέσω δύο υποστηριζόμενων μεθόδων, της 'pull' και της 'push'. Όπως υποδηλώνουν και τα ονόματα τους, τα εργαλεία που χρησιμοποιούν την πρώτη μέθοδο στηρίζονται σε τακτικές-περιοδικές διερευνητικές ερωτήσεις ενός δαίμονα που τρέχει στους εξυπηρετητές-στόχους προς το master, δηλαδή προς το ίδιο το σύστημα, για πιθανές αλλαγές που χρειάζονται να γίνουν. Φυσικά, για να δουλέψει αυτό, χρειάζεται ο συγκεκριμένος δαίμονας να έχει εγκατασταθεί και να δουλεύει πάντα σωστά, ο εξυπηρετητής του εργαλείου, δηλαδή συνήθως ο σταθμός εργασίας του μηχανικού, θα πρέπει να δέχεται συνδέσεις από όλους τους πελάτες, ενώ για ανοχή στα σφάλματα θα πρέπει να υπάρχει και επιτήρηση. Αντιδιαμετρικά, τα εργαλεία της δεύτερης μεθόδου στέλνουν τις όποιες αλλαγές διαμόρφωσης χρειάζεται να γίνουν στους αντίστοιχους εξυπηρετητές, οι οποίοι δεν χρειάζονται κάποια περαιτέρω διαμόρφωση για να ξεκινήσουν να δουλεύουν. Αυτό γίνεται με τρόπο σύγχρονο και ξεκινάει από το σταθμό εργασίας του μηχανικού.

Τέλος, κατά παρόμοιο τρόπο με την παραδοσιακή Τεχνολογία Λογισμικού και τις Γλώσσες Προγραμματισμού, υπάρχει το δίλλημα μεταξύ δηλωτικής και προστακτικής προσέγγισης, ή αλλιώς 'τι' εναντίον 'πώς'. Τα δηλωτικά εργαλεία περιγράφουν μία επιθυμητή τελική κατάσταση, προσφέροντας έτσι μια αφαιρετικότητα στο πώς ακριβώς θα επιτευχθεί αυτή. Από την άλλη, τα προστακτικά εργαλεία απαιτούν μια σειρά από ακριβείς ενέργειες, δοσμένες με την σωστή σειρά και με τους κατάλληλους ελέγχους ενσωματωμένους στη λογική. Φανερή είναι η συσχέτιση και η επιρροή που έχει η γλώσσα προγραμματισμού με την οποία έχει αναπτυχθεί το κάθε εργαλείο στο ποιά προσέγγιση θα ακολουθήσει. Βέβαια, σημαντικό είναι το γεγονός ότι τα περισσότερα εργαλεία υποστηρίζουν και τις δύο προσεγγίσεις ή ένα συνδυασμό αυτών, σε μία προσπάθεια να προσελκύσουν ένα ευρύτερο, πιο ετερογενές σύνολο χρηστών, είτε αυτοί είναι προγραμματιστές, διαχειριστές συστημάτων ή μηχανικοί πλήρους-στοίβας.

2.2.2 Εργαλεία Προτυποποίησης Εξυπηρετητή



Μια εναλλακτική προσέγγιση στη διαχείρηση διαμόρφωσης αποτελούν τα εργαλεία προτυποποίησης. Εμπνευσμένα από μία από τις βασικές αρχές του συναρτησιακού προγραμματισμού, την αμεταβλητότητα, βρίσκονται στην άνοδο και αποτελούν βασικά εργαλεία στην προσπάθεια για αμεταβλητή υποδομή, η οποία διευκολύνει το να επιχειρηματολογήσει κανείς για το τί έχει αναπτυχθεί. Συγκεκριμένα, αντί να δίνουν οδηγίες στους εξυπηρετητές-στόχους για το τί χρειάζεται να εγκαταστήσουν και πώς, δημιουργούν μία ολοκληρωμένη εικόνα λειτουργικού συστήματος, συνδυασμένου με τα απαιτούμενα αρχεία και λογισμικό, η οποία και εγκαθίσταται στους εξυπηρετητές συνήθως από κάποιο άλλο εργαλείο Υποδομής ως Κώδικα. Η αμεταβλητότητα επιτυγχάνεται από το γεγονός ότι για να πραγματοποιηθεί κάποια αλλαγή στην ήδη υπάρχουσα διαμόρφωση, όπως για παράδειγμα μια αλλαγή στον κώδικα μιας εφαρμογής, θα πρέπει να δημιουργηθεί μία εικόνα με τη νέα έκδοση, η οποία στη συνέχεια θα πρέπει να αντικαταστήσει την παλιά σε όλους τους επηρεαζόμενους εξυπηρετητές.

Οι δύο μεγάλες κατηγορίες των εργαλείων προτυποποίσης προκύπτουν από τον τύπο Εικονικοποίησης που χρησιμοποιούν και συγκεκριμένα το πόση αναξαρτησία θέλουν να δώσουν στην παραγόμενη εικόνα. Συγκεκριμένα, η πρώτη κατηγορία εργαλείων δημιουργεί εικόνες για αυτούσια εικονικά μηχανήματα. Σε ένα περιβάλλον υπολογιστικού νέφους, χρησιμοποιείται ένας υπερόπτης, ο οποίος εικονικοποιεί του πόρους των χαμηλότερων επιπέδων, όπως επεξεργαστές, κύρια μνήμη, αποθηκευτικό χώρο και δικτυακές διεπαφές. Η εικόνα θα τρέξει πάνω στον υπερόπτη και η εικονική μηχανή που θα προκύψει θα μπορεί να δει και να εκμεταλλευτεί μόνο τους πόρους που της έχουν ανατεθεί, προσφέροντας έτσι απόλυτη ασφάλεια και απομόνωση. Η δεύτερη κατηγορία στηρίζεται στους containers. Οι containers εξομοιώνουν το χώρο χρήστη ενός λειτουργικού συστήματος, δημιουργώντας απομονωμένες διαδικασίες, μνήμη ακόμα και δίκτυα ξεχωριστά από τον υποκείμενο εξυπηρετητή, ο οποίος απλά χρειάζεται να έχει εγκατεστημένη μία μηχανή παραγωγής και διαχείρισης τους. Τρέχουν κατευθείαν πάνω στο λειτουργικό σύστημα και χρησιμοποιούν από κοινού τους πόρους του

εξυπηρετητή.

Και οι δύο κατηγορίες έχουν το πλεονέκτημα ότι οι παραγόμενες εικόνες είναι αυτοτελείς και θα έχουν την ίδια συμπεριφορά σε όποιο περιβάλλον και να τρέξουν, δεδομένου ότι ο κατάλληλος υπερόπτης ή η κατάλληλη μηχανή είναι παρόντες. Λόγω όμως των μοιραζόμενων πόρων και του πρακτικά μηδενικού επιπλέον κόστους επεξεργαστικών πόρων και μνήμης, οι containers είναι τάξεις μεγέθους πιο γρήγορο να φορτωθούν, με μειωμένες βέβαια δυνατότητες απομόνωσης, σε σχέση με τις εικονικές μηχανές.

2.2.3 Εργαλεία Παροχής Υποδομής

Ξεκινώντας από τα επί τούτου σενάρια και καταλήγοντας στα εργαλεία προτυποποίησης, όλα τα παραπάνω εργαλεία προσανατολίζονται στο πώς θα διαμορφώσουν έναν αριθμό ήδη υπάρχοντων εξυπηρετητών, μελών συνήθως μιας συστάδας, το καθένα φυσικά με τη δικιά του φιλοσοφία, όλα όμως με μία κοινή βάση και ένα κοινό αποτέλεσμα. Τα εργαλεία παροχής υποδομής έρχονται να γεμίσουν το κενό της παροχής αυτών καθαυτών των εξυπηρετηρών, υπό τη σκοπιά της Υποδομής ως Κώδικα. Τα εργαλεία αυτά χρησιμοποιούν τη διεπαφή προγραμματισμού που παρέχεται από τους παρόχους υπολογιστικών νεφών και δημιουργούν εξυπηρετητές, βάσεις δεδομένων, τείχη προστασίας, εξισορροπιστές φορτίου, κρυφές μνήμες, κανόνες δρομολόγησης, δομές δικτύων και υποδικτύων, δομές επιτήρησης καθώς και οποιοδήποτε άλλο κομμάτι μιας υποδομής ενός ευρύτερου συστήματος.

Κεφάλαιο 3

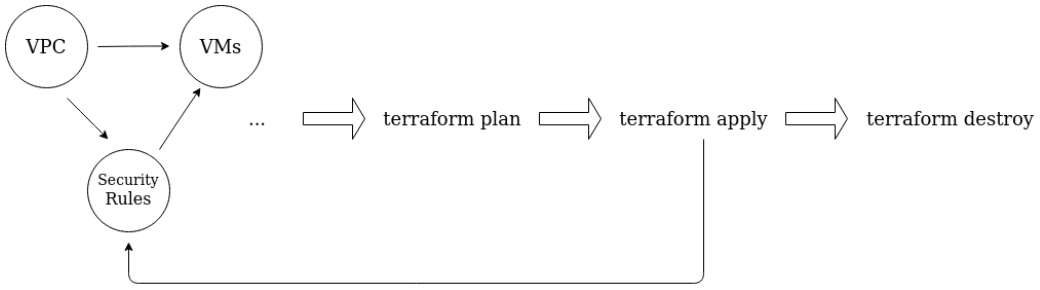
Terraform

3.1 Τί είναι το Terraform

Το Terraform είναι ένα εργαλείο Υποδομής ως Κώδικας ανοιχτού λογισμικού, το οποίο δημιουργήθηκε από την εταιρεία HashiCorp και κυκλοφόρησε για πρώτη φορά το 2014. Ανήκει στην κατηγορία των εργαλείων Παροχής Υποδομής, το οποίο σημαίνει ότι χρησιμοποιείται για τη δήλωση, δημιουργία, διαχείριση και καταστροφή εξυπηρετητών, βάσεων δεδομένων, δικτύων καθώς και πάσης φύσεως συστατικών μιας οσοδήποτε μεγάλης ή ετερογενούς υποδομής σε ένα ή περισσότερα περιβάλλοντα υπολογιστικού νέφους.

Ακολουθεί την προσέγγιση της αμεταβλητής υποδομής, με σκοπό την όσο δυνατόν ελαχιστοποίηση φαινομένων απόκλισης διαμόρφωσης στο σύνολο των εξυπηρετητών. Επιπλέον, για να διευκολύνει την επιχειρηματολογία για το τί είναι ανεπτυγμένο, χωρίς να είναι απαραίτητη η γνώση ολόκληρου του ιστορικού αλλαγών, χρησιμοποιεί μια δηλωτική γλώσσα περιγραφής, ορίζοντας εξαρχής μια επιθυμητή τελική κατάσταση και παρακολουθώντας κάθε φορά τις όποιες αλλαγές χρειάζονται να γίνουν, βρίσκοντας έτσι το ίδιο το βέλτιστο τρόπο για να επιτευχθεί η νέα κατάσταση. Για τη μείωση των κινούμενων μερών, την ελαχιστοποίηση των όποιων λαθών διαμόρφωσης αλλά και την απλοποίση της υποδομής, δεν χρησιμοποιεί κάποιον άμεσο αφέντη εξυπηρετητή για το διαμοιρασμό των ενημερώσεων, είναι δηλαδή 'masterless'. Αντιθέτως, χρησιμοποιεί εμμέσως την διεπαφή προγραμματισμού που προσφέρουν οι ίδιοι οι πάροχοι υπηρεσιών. Για τον ίδιο λόγο, η αρχιτεκτονική του Terraform δεν χρησιμοποιεί κάποιον εξωτερικό πράκτορα σε κάθε εξυπηρετητή που θέλει να διαμορφώσει, αλλά χρησιμοποιεί λογισμικό πράκτορα που ενσωματώνεται από τον εκάστοτε πάροχο στην εικόνα που θα χρησιμοποιηθεί για τη διαμόρφωση του πόρου (π.χ. AMI για Amazon EC2 Instance).

3.2 Ροή Εργασιών



Σχήμα 3.1: Ροή εργασιών του Terraform

Αποτελώντας στον πυρήνα του ένα εργαλείο Υποδομής ως Κώδικας, το Terraform στηρίζεται στον κώδικα και συγκεκριμένα σε αρχεία διαμόρφωσης γραμμένα σε μια υψηλού επιπέδου, ειδικού σκοπού γλώσσα συγκεκριμένου τομέα, παρόμοια σε σύνταξη με τις JSON και YAML μορφές, την HashiCorp Configuration Language (HCL). Τα αρχεία αυτά είναι η πηγή επιθυμητής κατάστασης είτε πρόκειται για την 1η μέρα και την δημιουργία των πρώτων μερών της υποδομής, είτε για τη 2η+ μέρα, όπου και χρειάζεται να τροποποιηθούν κάποια κομμάτια, ή για τη N-οστή μέρα, όταν και για κάποιο λόγο θέλουμε να καταστρέψουμε την υποδομή μας, π.χ. γιατί ήταν μέρος ενός δοκιμαστικού περιβάλλοντος.

Στα αρχεία διαμόρφωσης του Terraform (με κατάληξη .tf, π.χ. main.tf) περιγράφονται κομμάτια της υποδομής, καθώς και τα απαραίτητα για τη δημιουργία τους πεδία. Κάθε νέος πόρος που περιγράφεται μπορεί να έχει αναφορές σε κάποιον άλλο. Στα αρχεία αυτά, δηλαδή, αποτυπώνεται ο σύνθετος γράφος εξαρτήσεων μεταξύ των διαφόρων πόρων της υποδομής, σε μία πολύ υψηλού επιπέδου και κατανοητή από τον άνθρωπο περιγραφή.

Μόλις έχει ολοκληρωθεί η συγγραφή των αρχείων διαμόρφωσης, ακολουθεί το στάδιο σχεδιασμού. Το Terraform τελικά συμπυκνώνεται σε ένα μοναδικό εκτελέσιμο, το οποίο έχει τη μορφή Διεπαφής Γραμμής Εντολών (Command Line Interface - CLI). Η εντολή υπεύθυνη για το σχεδιασμό είναι η terraform plan. Σε πρώτο στάδιο, η εντολή αυτή ανανεώνει την οπτική που έχει το ίδιο το εργαλείο για το τί έχει αναπτυχθεί στην πραγματικότητα, επικοινωνώντας με τους διάφορους παρόχους υποδομής νέφους, αποκτώντας έτσι την πραγματική κατάσταση εκείνη τη στιγμή. Η ανανέωση αυτή συμβαίνει μόνο εντός μνήμης και δεν αποθηκεύεται κάπου. Στη συνέχεια, συνδυάζοντας την κατάσταση με τα αρχεία διαμόρφωσης, εκτυπώνει τους πόρους που χρειάζεται να αναπτυχθούν, να τροποποιηθούν ή να καταστραφούν, ώστε η επιθυμητή κατάσταση να γίνει πραγματικότητα. Είναι φανερό ότι η εντολή αυτή αποτελεί απλά ένα δοκιμαστικό βήμα, το οποίο δρα ως ένα δίχτυ προστασίας, αφού ο μηχανικός βλέπει μπροστά του ποιες αλλαγές θα γίνουν και μπορεί χωρίς κάποια επίπτωση να βρει πιθανά σφάλματα.

Έπειτα από το στάδιο σχεδιασμού, ακολουθεί το στάδιο εφαρμογής με την εντολή

terraform apply. Είναι σε αυτό το στάδιο που το Terraform χρησιμοποιεί τη Διεπαφή Προγραμματισμού των παρόχων υπηρεσιών νέφους και αναπτύσει τους διάφορους πόρους με τρόπο που να σέβεται τις διάφορες εξαρτήσεις μεταξύ τους, αλλά και όσο το δυνατόν πιο αποτελεσματικά, παραλληλοποιώντας τη δημιουργία ανεξάρτητων πόρων. Εφαρμόζοντας το πλάνο που προκύπτει από το προηγούμενο βήμα, δημιουργεί πόρους που υπάρχουν στη διαμόρφωση αλλά δεν σχετίζονται με πραγματικά κομμάτια της υποδομής όπως αυτή αποτυπώνεται στην υπάρχουσα κατάσταση, καταστρέφει πόρους που υπάρχουν στην υποδομή αλλά πλέον όχι και στη διαμόρφωση και ανανεώνει επί τόπου πόρους των οποίων οι παράμετροι έχουν αλλάξει, ή όπου αυτό δεν είναι δυνατό λόγω περιορισμών που θέτονται από τις απομακρυσμένες Διεπαφές Προγραμματισμού, τους καταστρέφει και τους επαναδημιουργεί (καθώς και όλες τις εξαρτήσεις αυτών).

Η ροή εργασιών αποτελείται, συνεπώς, από επαναλαμβανόμενες επαναλήψεις του βρόχου 'δημιουργία/τροποποίηση αρχείων διαμόρφωσης-σχεδιασμός-εφαρμογή' και παραμένει ακριβώς έτσι μέχρι και τη στιγμή που θα θελήσουμε να θέσουμε εκτός λειτουργίας την υποδομή μας. Το ρόλο αυτό αναλαμβάνει η εντολή terraform destroy. Η συγκεκριμένη εντολή δρα με αντίστροφο τρόπο από την εντολή εφαρμογής. Συμβουλεύεται την απεικόνιση της κατάστασης που έχει αποθηκευμένη εκείνη τη στιγμή το εργαλείο για το τί είναι ανεπτυγμένο, και στη συνέχεια επικοινωνεί με τους παρόχους για την καταστροφή, με τη σωστή σειρά, της υποδομής. Το βήμα αυτό αποτελεί και τη λήξη του κύκλου ζωής της ροής εργασιών.

3.3 Πάροχοι και Πόροι

3.3.1 Πάροχοι

Σύμφωνα με την έρευνα της Forrester Research για την IBM το 2020, το 62% όσων έχουν υιοθετήσει το δημόσιο νέφος χρησιμοποιούν 2+ ανεξάρτητες πλατφόρμες. Συνεπώς, απαραίτητο, ειδικά για ένα εργαλείο παροχής υποδομής, είναι να μην είναι προσκολλημένο σε ένα και μοναδικό πάροχο υπηρεσιών, ώστε να μην χρειάζεται ο χρήστης να πρέπει να μάθει μια νέα τεχνολογία από την αρχή, κάθε φορά που θέλει να μετακινηθεί ή να επεκταθεί σε μία διαφορετική πλατφόρμα νέφους. Αυτό επιτρέπει την εύκολη επέκταση και την εκμετάλλευση διαφόρων χαρακτηριστικών και προσφορών, αλλά και μειώνει την πιθανότητα σφάλματος ή παραβίασης ασφάλειας.

Ένα από τα πλεονεκτήματα του Terraform είναι ότι είναι αγνωστικιστικό ως προς τον πάροχο νέφους, μπορεί δηλαδή με την ίδια ευκολία να αναπτύξει υποδομή είτε πρόκειται για κάποιο μεγάλο δημόσιο νέφος, όπως τα AWS(Amazon), GCP(Google), Azure(Microsoft), ή για κάποιο ιδιωτικό κέντρο δεδομένων(OpenStack, VMware).

Κεντρική έννοια στο Terraform, η οποία επιτρέπει τη διασύνδεση με τις διάφορες πλατφόρμες, είναι ο πάροχος, ο οποίος ορίζεται με την λέξη-κλειδί provider. Οι providers είναι αρθρώματα λογισμικού (plugins) που είναι σχεδιασμένα για την αλληλεπίδραση με τις Διεπαφές Προγραμματισμού των παρόχων υπηρεσιών νέφους και διαχειρίζονται κλήσεις Διεπαφής Προγραμματισμού, επιβεβαιώσεις γνησιότητας καθώς και σφάλματα. Ο κάθε πάροχος διατηρεί το δικό του provider, οι οποίοι στην πραγμα-

τικότητα είναι μερικά εκτελέσιμα αρχεία γραμμένα στη γλώσσα Go(η οποία είναι και η γλώσσα συγγραφής του Terraform), και διαμοιρασμένα στο μητρώο καταγραφής providers που διατηρεί το Terraform. Υπάρχουν providers για μία ευρύτατη γκάμα υπηρεσιών, είτε πρόκειται για Υποδομή ως Υπηρεσία(AWS, Azure, VMware, ...), είτε για Πλατφόρμα ως Υπηρεσία(Heroku, Kubernetes, Lambdas, ...), ή για Λογισμικό ως Υπηρεσία(Domain Name Server, Content Delivery Network, Github, ...).

Οι providers είναι ανεξάρτητοι από το κεντρικό εργαλείο. Για να χρησιμοποιηθούν χρειάζεται ένα επιπλέον βήμα, το οποίο στην πραγματικότητα εκτελείται αμέσως μετά τη συγγραφή των αρχείων διαμόρφωσης και πριν το στάδιο του σχεδιασμού, αναλογιζόμενοι τον κύκλο ζωής της ροής εργασιών. Το βήμα αυτό είναι η εντολή terraform init. Με το terraform init, ουσιαστικά αρχικοποιείται ο τρέχων κατάλογος εργασιών ως κατάλογος εργασιών Terraform και δυναμικά εντοπίζονται και εγκαθιστώνται τα απαραίτητα plugins, μεταξύ των οποίων οι providers. Μάλιστα, αν εκτελέσουμε την εντολή terraform plan είτε πριν από την terraform init ή αφού έχουμε προσθέσει κάποιο νέο πόρο που χρησιμοποιεί κάποιο μέχρι πρότινος άγνωστο πάροχο, το Terraform θα ολοκληρώσει τη λειτουργία του εμφανίζοντας ένα μήνυμα σφάλματος και απαιτώντας επαναρχικοποίηση των plugins.

Για την παραμετροποίηση των παρόχων δεν χρειάζονται επιπλέον πεδία στα αρχεία περιγραφής, εκτός ίσως από την περιοχή, την έκδοση του plugin (π.χ. σε περίπτωση που κάποιο project χρειάζεται να είναι προσκολλημένο σε κάποια συγκεκριμένη έκδοση για λόγους συμβατότητας), κάποιο τελικό σημείο ή κάποια διαπιστευτήρια. Συγκεκριμένα, για τον provider της Amazon

```
provider "aws" {
    access_key = "MY_ACCESS_KEY"
    secret_key = "MY_SECRET_KEY"
    region = "eu-central-1"
}
```

το παραπάνω block κώδικα αρκεί για να χρησιμοποιηθεί ο λογαριασμός χρήστη AWS στον οποίον ανήκουν τα κλειδιά access_key και secret_key ώστε να αναπτυχθεί υποδομή εκ μέρους του στην περιοχή region, δηλαδή στο αντίστοιχο κέντρο δεδομένων της Amazon.

3.3.2 Πόροι

Οι πάροχοι είναι αυτοί που είναι απαραίτητοι να υπάρξουν, διότι χωρίς αυτούς το Terraform δεν μπορεί να αναπτύξει καμία υποδομή. Το μεγαλύτερο κομμάτι, όμως, των αρχείων καταγραφής, αυτό που ενδιαφέρει το μηχανικό-χρήστη και αυτό που εν τέλει θα αποτελέσει την υποδομή, είναι οι πόροι. Οι πόροι είναι αυτοί στους οποίους το Terraform οφείλει την θεμελιώδη περιγραφή του ως 'ένα εργαλείο διαχείρισης κατάστασης, το οποίο πραγματοποιεί ενέργειες δημιουργίας, ανάγνωσης, ανανέωσης και διαγραφής (Create, Read, Update, Delete - CRUD) σε διαχειριζόμενους πόρους'.

Στον κόσμο του Terraform, η λέξη-κλειδί για τον πόρο είναι η resource. Ως πόρος μπορεί να θεωρηθεί οτιδήποτε μπορεί να αναπαρασταθεί και να διαχειριστεί

πλήρως από ένα σύνολο CRUD λειτουργιών, τυπικά ό,τι εκτίθεται μέσω μιας Διεπαφής Προγραμματισμού από κάποιον πάροχο. Πόρος μπορεί να είναι ένα ιδιωτικό δίκτυο, μία βάση δεδομένων, μία εικονική μηχανή, ένα τείχος προστασίας.

Κάθε πόρος απαιτεί συγκεκριμένα, ξεχωριστά ανά περίπτωση συμπληρωμένα πεδία στο αντίστοιχο κομμάτι του αρχείου διαμόρφωσης του, τα οποία καθορίζονται από τους παρόχους. Η δήλωση τους πραγματοποιείται μέσα από block κώδικα, τα οποία ζουν μόνο μέσα στα αρχεία διαμόρφωσης, αφού οποιαδήποτε αλλαγή στον πραγματικό κόσμο θα γίνει με το στάδιο της εφαρμογής. Η σύνταξη ενός resource block ακολουθεί συγκεκριμένη δομή:

```
resource resource_type name {
    configuration_argument_1 = ...
    configuration_argument_2 = ...
    ...
    configuration_argument_n = ...
}
```

όπου resource_type είναι ένα κομμάτι υποδομής, όπως αυτό ορίζεται από τον αντίστοιχο πάροχο, name είναι ένα τοπικό όνομα που χρησιμοποιείται από το Terraform για να μπορούν να ανάφερονται άλλοι πόροι σε αυτόν και configuration_argument_{1,2,...,n είναι ένα σύνολο από πεδία διαμόρφωσης.

Κάθε πόρος διαθέτει συγκεκριμένα χαρακτηριστικά που μπορούν να προσπελαστούν μέσω εκφράσεων από κάποιο άλλο resource block, ως κομμάτι της διαμόρφωσης του. Χαρακτηριστικό παράδειγμα το μοναδικό αναγνωριστικό ενός ιδιωτικού δικτύου, όταν θέλουμε να δημιουργήσουμε εντός αυτού μια εικονική μηχανή,

```
...
vpc_id = aws_vpc.my_vpc.id
...
```

Η έκφραση aws_vpc.my_vpc.id ακολουθεί τη σύνταξη

```
<resource_type>.<name>.<attribute>
```

όπου vpc_id το μοναδικό αναγνωριστικό του δικτύου στο οποίο θα τοποθετηθεί η εικονική μηχανή που διαμορφώνεται στο συγκεκριμένο resource block, aws_vpc το resource που διαθέτει ο Amazon provider για να περιγράψει ένα εικονικό ιδιωτικό δίκτυο, my_vpc το όνομα του πόρου που έχουμε δημιουργήσει για το ιδιωτικό αυτό δίκτυο, και id το μοναδικό αναγνωριστικό αυτού του δικτύου.

Παράδειγμα δημιουργίας εικονικής μηχανής στο AWS EC2

Ένα ολοκληρωμένο παράδειγμα για της δημιουργία μιας εικονικής μηχανής Ubuntu στο AWS EC2 κέντρο δεδομένων της Φρανκφούρτης ("eu-central-1") μέσω Terraform, μαζί με τη δήλωση του παρόχου και του εικονικού δικτύου, είναι το παρακάτω:

```
provider "aws" {
    access_key = "MY_ACCESS_KEY"
    secret_key = "MY_SECRET_KEY"
    region = "eu-central-1"
}

resource "aws_vpc" "my_vpc" {
  cidr_block = "172.16.0.0/16"

  tags = {
    Name = "My Virtual Private Network"
  }
}

resource "aws_instance" "example" {
  ami = "ami-056114420b6ed624e"
  instance_type = "t2.micro"

  tags = {
    Name = "Example Instance"
  }
}
```

Tags είναι ένα σύνολο από ετικέτες στη μορφή ζευγών κλειδιού-τιμής για την ανάθεση μεταδεδομένων σε AWS πόρους. Χρησιμοποιούνται για την διαχείριση, κατηγοριοποίηση, αναζήτηση και αναγνώριση πόρων με βάση το σκοπό, τον ιδιοκτήτη ή το περιβάλλον και στη συγκεκριμένη περίπτωση, το tag Name χρησιμοποιείται για την ονομασία του πόρου, μια ονομασία που μεταξύ άλλων θα εμφανιστεί και στο AWS Console. Το instance_type δηλώνει τον τύπο της εικονικής μηχανής (εδώ επεξεργαστής 1 πυρήνα, κύρια μνήμη 1GB), ενώ το cidr_block προσδιορίζει το σχήμα διευθυνσιοδότησης του vpc δικτύου.

Δομικές μονάδες και χρήση μεταβλητών

Το Terraform επιτρέπει τη δημιουργία δομικών μονάδων (module), μιας ομαδοποίησης δηλαδή πόρων που χρησιμοποιούνται μαζί. Συμπληρωματικά με τη βασική μονάδα, τη ριζική μονάδα, που αποτελείται από τους πόρους που δηλώνονται στα .tf αρχεία στο βασικό κατάλογο εργασιών, μπορούν να δημιουργηθούν και θυγατρικές μονάδες, οι οποίες και θα χρησιμοποιηθούν σε ένα ή περισσότερα σημεία της διαμόρφωσης, ενώ μπορούν να έχουν οριστεί και εξωτερικά από άλλο χρήστη και να εγκατασταθούν μαζί με τους παρόχους στο στάδιο αρχικοποίησης (terraform init). Για την εξατομίκευση τους, εκθέτουν ένα σύνολο μεταβλητών, ένα σύνολο παραμέτρων, δηλαδή, στις οποίες πρέπει να δωθούν τιμές κατά την κλήση της μονάδας.

Μεταβλητές επίσης μπορούν να οριστούν για μεγαλύτερη ασφάλεια. Είναι φανε-

ρό ότι θα θέλαμε ιδιωτικές πληροφορίες, όπως κωδικοί πρόσβασης, κλειδιά, λεκτικές μονάδες (token) να μην φαίνονται στον πηγαίο κώδικα, όπως επίσης και στα στάδια σχεδιασμού και εφαρμογής, όπου εκτυπώνονται οι πόροι που πρόκειται να δημιουργηθούν/τροποποιηθούν/καταστραφούν. Με τη χρήση μεταβλητών, τα ευαίσθητα δεδομένα μπορούν να ορίζονται σε ξεχωριστά αρχεία μεταβλητών, τα οποία και δεν θα είναι μέρος του Συστήματος Ελέγχου Εκδόσεων (Version Control System, π.χ.Github), ή εναλλάκτικα να ορίζονται ως μεταβλητές περιβάλλοντος, σύμφωνα με το μοτίβο TF_VAR_<variable_name>, ή μπορούν να δίνονται ως επιλογές κατά την εκτέλεση της εντολής terraform apply.

Τέλος, μπορούμε να χρησιμοποιήσουμε μεταβλητές για μεγαλύτερη ευελιξία, π.χ. για επίλογη κατάλληλης εικόνας λειτουργικού συστήματος για μία εικονική μηχανή ανάλογα με τη γεωγραφική περιοχή που θέλουμε να την αναπτύξουμε.

Το παραπάνω παράδειγμα δημιουργίας εικονικής μηχανής στο AWS EC2, με χρήση μεταβλητών, τροποποιείται ως εξης:

Αρχικά, με τη δημιουργία του αρχείου vars.tf

```
variable "AWS_REGION" {
    default = "eu-central-1"
}
variable "AWS_ACCESS_KEY" {}
variable "AWS_SECRET_KEY" {}
variable "AWS_INSTANCE_TYPE" {
    default = "t2.micro"
}
variable "AMIS" {
    type = map(string)
    default = {
        eu-central-1 = "ami-056114420b6ed624e"
        eu-north-1 = "ami-08d18dd6fa8bd7fe6"
        eu-south-1 = "ami-04ae6b3b18f5fa3ea"
    }
}
```

που δηλώνει τις μεταβλητές, μερικές με κάποιες προεπιλεγμένες τιμές, ενώ φαίνεται και η πιο σύνθετη αντιστοίχιση μεταξύ κέντρου δεδομένων και εικόνας λειτουργικού συστήματος στη δήλωση της μεταβλητής AMIS.

Στη συνέχεια, το αρχείο που θα περιλαμβάνει τις τιμές για τα κλειδιά του λογαριασμού χρήστη AWS EC2 είναι το terraform.tfvars (προσοχή στην κατάληξη .tfvars!)

```
AWS_ACCESS_KEY = "MY_AWS_ACCESS_KEY"
AWS_SECRET_KEY = "MY_AWS_SECRET_KEY"
```

Τέλος, οι δηλώσεις εικονικής μηχανής και παρόχου

```
provider "aws" {
```

```
    access_key = var.AWS_ACCESS_KEY
    secret_key = var.AWS_SECRET_KEY
    region = var.AWS_REGION
}

resource "aws_instance" "example" {
  ami = lookup(var.AMIS, var.AWS_REGION)
  instance_type = var.AWS_INSTANCE_TYPE

  tags = {
    Name = "Example Instance"
  }
}
```

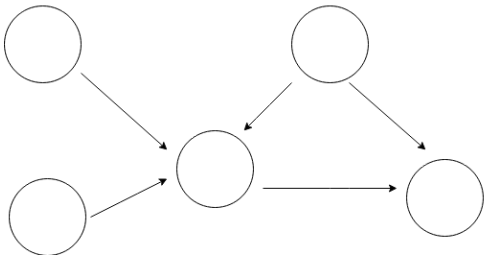
όπου και φαίνεται μία από τις διαθέσιμες συναρτήσεις του Terraform, η lookup, που χρησιμοποιείται για την προσπέλαση του κατάλληλου στοιχείου μιας μεταβλητής αντιστοίχισης όπως είναι η AWS_REGION

3.4 Πυρήνας και Γράφος Εξαρτήσεων

Οι πάροχοι είναι ο δίαυλος επικοινωνίας του Terraform με τις εξωτερικές διεπαφές προγραμματισμού των διαφόρων υπηρεσιών νέφους και οι πόροι παρέχουν τη λειτουργικότητα στην υποδομή, ο πυρήνας του εργαλείου όμως αναλαμβάνει την μετατροπή των αρχείων διαμόρφωσης σε ένα πλάνο το οποίο απαριθμεί τα βήματα που πρέπει να ακολουθηθούν για την υλοποίηση της. Τα βήματα αυτά πρέπει να σέβονται τις εξαρτήσεις μεταξύ των πόρων της υποδομής, να αποτυπώνουν λειτουργίες τόσο για τη δημιουργία όσο και καταστροφή πόρων και να επιτρέπουν την παράλληλη εκτέλεση όπου αυτό είναι δυνατόν, για καλύτερη επίδοση και αντιμετώπιση κωλυμμάτων, κάτι αρκετά πιθανό όταν μιλάμε για κατανεμημένα συστήματα σε υπολογιστικά νέφη. Η αυξημένη αυτή λειτουργικότητα που προσπαθεί να εισάγει η Υποδομή ως Κώδικας έρχεται με μία εγγενή αύξηση στην πολυπλοκότητα. Για ακόμη μία φορά, η λύση θα έρθει από την προσαρμογή της έρευνας ενός άλλου κομματιού της Επιστήμης Υπολογιστών στις ανάγκες της ανάπτυξης υποδομής μέσω κώδικα, της Θεωρίας Γράφων.

3.4.1 Βασικά στοιχεία της Θεωρίας Γράφων

Η Θεωρία Γράφων αποτελεί ένα γνωστικό πεδίο των Διακριτών Μαθηματικών που ασχολείται με την οπτική αναπαράσταση σχέσεων και εξαρτήσεων μεταξύ οντοτήτων. Τυπικά, ένας γράφος G αποτελείται από ένα σύνολο σημείων V και ένα σύνολο συνδέσεων E μεταξύ τους. Το σύνολο των σημείων ονομάζονται κόμβοι ή κορυφές (nodes/vertices) και το σύνολο των γραμμών που τα συνδέουν ονομάζονται ακμές (edges). Άν οι ακμές είναι κατευθυνόμενες, τότε μιλάμε για έναν κατευθυνόμενο γράφο.

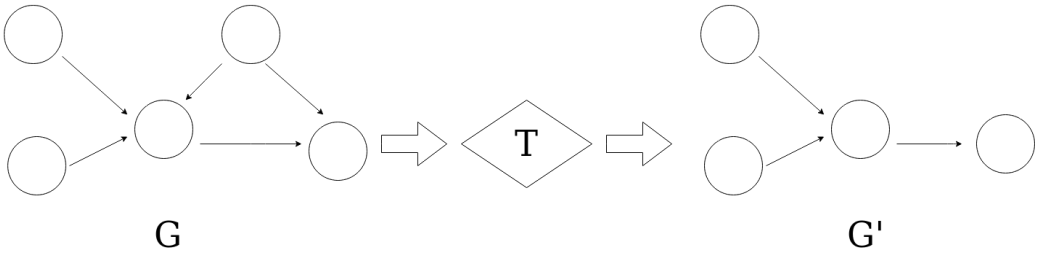


Σχήμα 3.2: Κατευθυνόμενος γράφος

Κύκλος ονομάζεται μια διαδρομή που ξεκινάει από ένα κόμβο, ακολουθεί τις κατευθυνόμενες ακμές και καταλήγει στον ίδιο κόμβο. Αν ένας κατευθυνόμενος γράφος δεν περιέχει κύκλους, δηλαδή για κάθε κόμβο δεν υπάρχει διαδρομή που να καταλήγει στον ίδιο κόμβο, τότε πρόκειται για έναν Κατευθυνόμενο Άκυκλο Γράφο (Directed Acyclic Graph - DAG). Η προσπέλαση των κόμβων ενός γράφου μπορεί να γίνει σειριακά ή και παράλληλα.

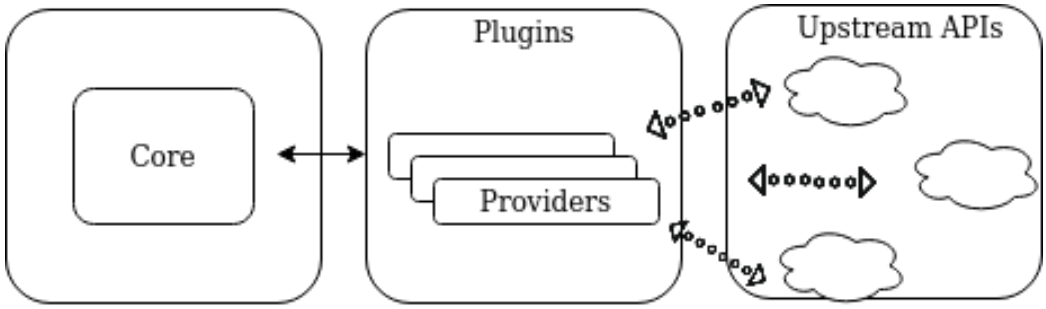
Διαδρομή είναι μια αφηρημένη ενέργεια επί ενός γράφου και ουσιαστικά πρόκειται για μία απαρίθμηση των κόμβων. Με την προσπέλαση κάθε κόμβου υπάρχει ελευθερία ως προς την ενέργεια που θα πραγματοποιηθεί (αν επιλέξουμε να πραγματοποιηθεί κάποια ενέργεια επί του κόμβου). Η διαδρομή μπορεί να ακολουθεί τις ακμές του γράφου ή μπορεί να έχει τυχαία σειρά, ενώ σημαντικότερο στα πλαίσια της Υποδομής ως Κώδικα είναι το γεγονός ότι, πέρα από σειριακά, μπορεί να εκτελείται και παράλληλα.

Μία από τις σημαντικότερες έννοιες στη Θεωρία Γράφων είναι ο μετασχηματισμός γράφων. Μετασχηματισμός T είναι μία συναρτησιακή διαδικασία που λαμβάνει ως είσοδο ένα γράφο και παράγει ως έξοδο μία τροποποιημένη έκδοση αυτού του γράφου. Η τροποποίηση αυτή μπορεί να αφορά την προσθήκη ή αφαίρεση κόμβων ή ακμών, ενώ μπορεί να μην πραγματοποιεί και καμία αλλαγή.



Σχήμα 3.3: Μετασχηματισμός γράφου

3.4.2 Πώς δουλεύει το Terraform και πώς εκμεταλλεύεται τη Θεωρία Γράφων

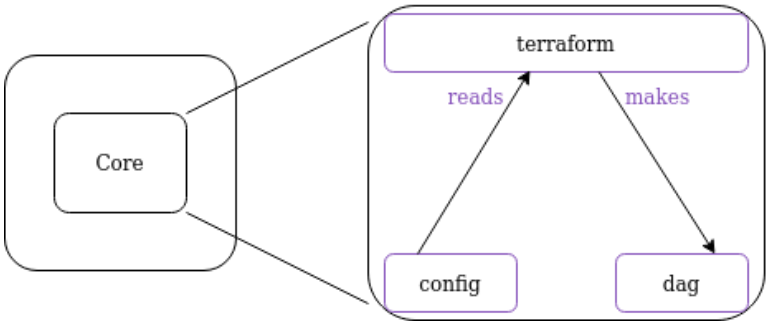


Σχήμα 3.4: Δομή του Terraform

Σε ένα περιβάλλον όπως αυτό της απομακρυσμένης, αυτοματοποιημένης παροχής υποδομής σε περιβάλλοντα νέφους, υπάρχουν ρητές εξαρτήσεις μεταξύ των απαραίτητων ενεργειών. Ταυτόχρονα, θα πρέπει να υπάρχει σαφώς ορισμένη αρχή και τέλος στο σύνολο των εξαρτήσεων. Απουσία αυτών θα σήμαινε μη προβλέψιμο και πιθανώς αένα-ο χρονοπρογραμματισμό εργασιών. Η βασική δομή της Θεωρίας Γράφων στην οποία στηρίζεται ο πυρήνας του Terraform για τη δημιουργία του πλάνου για την υλοποίηση μιας οποιαδήποτε υποδομής είναι ο Κατευθυνόμενος Άκυκλος Γράφος. Βασική ιδιότητα αυτού που τον προκρίνει ως ιδανική λύση είναι αυτή της τοπολογικής ταξινόμησης. Τοπολογική ταξινόμηση είναι μια γραμμική ταξινόμηση ενός κατευθυνόμενου γράφου, τέτοια ώστε για κάθε ακμή ο αρχικός κόμβος να εμφανίζεται νωρίτερα στην ακολουθία από τον τελικό κόμβο.

Σε πρώτο στάδιο, ο πυρήνας του Terraform χρησιμοποιεί στο εσωτερικό του ένα σύνολο από δηλωτικούς όρους (π.χ. Diff(), Apply(), Refresh()) οι οποίοι στη συνέχεια μέσω των provider plugins μετατρέπονται στα πιο γνώριμα και φιλικά προς τις Διεπαφές Προγραμματισμού των παρόχων CRUD ρήματα (Create(), Read(), Update(), Delete()). Συγκεκριμένα, Config είναι η παρεχόμενη από το χρήστη επιθυμητή κατάσταση, το σύνολο των πόρων και των μεταξύ τους σχέσεων που θέλει να δημιουργηθούν. Μετά από κάθε λειτουργία, το Terraform καταγράφει μία κατάσταση State, η οποία και είναι η πραγματικότητα για το εργαλείο όσον αφορά τους ανεπτυγμένους πόρους εκείνη τη στιγμή. Diff είναι η διαφορά μεταξύ επιθυμητής και πραγματικής κατάστασης και ο τρόπος για να επιτευχθεί η επιθυμητή κατάσταση παρουσιάζεται με το Plan και εφαρμόζεται με το Apply.

Τα βασικά πακέτα στον πυρήνα του Terraform είναι τρία. Αρχικά, το πακέτο config διαβάζει τη αρχεία διαμόρφωσης που παρέχονται από το χρήστη, τα εισάγει από το δίσκο στην κύρια μνήμη και παραδίδει την επιθυμητή κατάσταση στο πακέτο terraform. Στη συνέχεια, το πακέτο terraform κατασκευάζει τον κατευθυνόμενο άκυκλο γράφο εξαρτήσεων χρησιμοποιώντας το πακέτο dag, το οποίο υλοποιεί το θεωρητικό κομμάτι της Θεωρίας Γράφων.



Σχήμα 3.5: Πυρήνας του Terraform

Μία σχηματική απεικόνιση της παραπάνω διαδικασίας που λαμβάνει χώρα στον πυρήνα του Terraform βοηθάει να αντιληφθούμε έναν από τους λόγους που σχεδιάστηκε με αυτό τον τρόπο. Αυτός ο διαχωρισμός των ανησυχιών επιτρέπει στα ξεχωριστά πακέτα να αναπτύσσονται ανεξάρτητα το ένα από το άλλο, με αποτέλεσμα καλύτερη επιχειρηματολογία σε μία διαδικασία αποσφαλμάτωσης για το τί μπορεί να έχει πάει στραβά . Συγκεκριμένα, το πακέτο dag, το οποίο αναλύεται περισσότερο στη συνέχεια, υλοποιεί τις αφηρημένες έννοιες της Θεωρίας Γράφων με τρόπο απλό και συνθέσιμο, ώστε να είναι δυνατή η πραγματοποίηση δοκιμών μονάδος και η επαλήθευση της σωστής λειτουργίας του πακέτου, σε τέτοιο βαθμό ώστε τα τελευταία χρόνια να δέχεται σπάνια τροποποιήσεις στον πηγαίο του κώδικα.

Σε πιο τεχνικό επίπεδο, η διαδικασία κατασκευής του κατευθυνόμενου άκυκλου γράφου εξαρτήσεων, όπως αυτή υλοποιείται από το πακέτο dag, περιλαμβάνει μία σειρά από μετασχηματισμούς γράφων σε μορφή σωλήνωσης. Αρχικά, ο μετασχηματισμός ConfigTransformer δέχεται ως εισόδους τον κενό γράφο και τη διαμόρφωση Config, όπως αυτή έχει οριστεί από το πακέτο config. Η έξοδος του είναι ένας κατευθυνόμενος γράφος, του οποίου το σύνολο των κόμβων περιλαμβάνει κόμβους για κάθε πόρο που θα δημιουργηθεί καθώς και κόμβους που αναπαριστούν τους παρόχους, ενώ το σύνολο των ακμών αναπαριστά τις όποιες εξαρτήσεις υπάρχουν μεταξύ των κόμβων.

Έπειτα, ο μετασχηματισμός ProviderTransformer είναι υπεύθυνος για την ανακάλυψη των κόμβων-παρόχων και τη δημιουργία ακμών μεταξύ αυτών και των εξαρτόμενων από αυτούς κόμβους-πόρους. Σημειώνεται ότι μία επίσκεψη σε έναν οποιονδήποτε κόμβο είναι μία έννοια αφηρημένη όσον αφορά το πακέτο dag. Το πακέτο terraform είναι εκείνο που καθορίζει τί θα συμβεί σε κάθε επίσκεψη και ουσιαστικά διαφοροποιεί του κόμβους-παρόχους από τους κόμβους-πόρους. Συγκεκριμένα, οι κόμβοι-πάροχοι αποτελούν ένα βήμα αρχικοποίησης, είτε κάποιου πελάτη Διεπαφής Προγραμματισμού ή αυθεντικοποίησης, μιας λειτουργίας δηλαδή η οποία προκαλεί κάποια εξωτερική κλήση του πυρήνα σε κάποιον πάροχο και γίνεται με τρόπο διάφανο προς αυτόν, μιας και για τις λεπτομέρειες της κλήσης και την υλοποίηση της είναι υπεύθυνα τα αρθρώματα λογισμικού των παρόχων (Plugins).

Στη συνέχεια, ακολουθούν διάφοροι μετασχηματισμοί που σκοπό έχουν να ε-

μπλουτίσουν ή να τροποποιήσουν τον υπάρχοντα γράφο με νέους κόμβους ή νέες ακμές. Μερικά παραδείγματα είναι η προσθήκη κόμβων που αντιπροσωπεύουν κάποια λογική, όπως η καταστροφή ενός πόρου, και η ενσωμάτωση αγκιστρωμάτων κύκλου ζωής μέσω τροποποίησης ακμών, η οποία στα αρχεία διαμόρφωσης γίνεται με τη λέξη-κλειδί lifecycle και ορίζει ένα ενθετο block κώδικα εντός ενός resource block (π.χ. create_before_destroy).

Για τη δημιουργία του ελάχιστου δυνατού γράφου που σέβεται τις υπάρχουσες εξαρτήσεις αλλά ταυτόχρονα ξεφεύγει από το υπερσύνολο όλων των πιθανών εξαρτήσεων, ο προτελευταίος transformer κάνει χρήση αλγορίθμου Μεταβατικής Μείωσης, διευκολύνοντας έτσι σημαντικά σενάρια αποσφαλμάτωσης.

Στο τέλος της σωλήνωσης των μετασχηματισμών, στην περίπτωση που υπάρχουν παραπάνω από ένας κόμβοι δίχως εξαρτήσεις, προστίθεται ένας νέος, μοναδικός, ανεξάρτητος αρχικός κόμβος από τον οποίο θα εξαρτώνται οι παραπάνω μέχρι πρότινος ανεξάρτητοι κόμβοι. Μέσω του μετασχηματισμού RootTransformer, ο νέος κόμβος, ο root, θα είναι η αρχή όλων των διαδρομών επί του γράφου.

Αρκετές από τις λειτουργίες του Terraform απαιτούν μία διαδρομή επί του κατευθυνόμενου άκυκλου γράφου εξαρτήσεων που δημιουργούν τα πακέτα terraform και dag του πυρήνα, όπως για παράδειγμα η επαλήθευση του γράφου, η αναζήτηση μεταβλητών που δίδονται ως είσοδο από το χρήστη κατά το χρόνο εκτέλεσης και τα βήματα σχεδιασμού (plan) και εφαρμογής (apply). Με σαφή επιρροή από τα πεδία των Παράλληλων και των Κατανεμημένων Συστημάτων, οι διαδρομές αυτές γίνονται με τη σειρά που υποδεικνύουν οι ακμές, με παράλληλο τρόπο όπου είναι δυνατό, ενώ κάθε επίσκεψη κόμβου ξεκινάει αμέσως μόλις τελειώσουν οι λειτουργίες μόνο από τις οποίες έχει εξάρτηση, κάτι το οποίο επιτρέπει γρήγορη πρόοδο των λειτουργιών του γράφου σε περιπτώσεις καθυστερήσεων ή προσωρινής απώλειας λειτουργικότητας, φαινόμενα αρκετά συχνά σε περιβάλλοντα υπολογιστικών νεφών. Φανερός είναι ένας από τους λόγους επιλογής της γλώσσας προγραμματισμού Go για την υλοποίηση του Terraform, μιας γλώσσας με ενσωματωμένες δυνατότητες για ταυτοχρονισμό (goroutines, channels).

3.5 Κατάσταση

3.5.1 Γενικά

Από τη μέχρι τώρα παρουσίαση του Terraform, είναι φανερό ότι η κατάσταση έχει σημαντικό ρόλο και χρησιμοποιείται για την δημιουργία νέας υποδομής και την κατάλληλη ανανέωση της ήδη υπάρχουσας, κάθε φορά που εκτελείται μία εντολή terraform plan/apply/destroy. Δεδομένου ότι οι περισσότεροι πάροχοι προσφέρουν δυνατότητες για έλεγχο υποδομής είτε χειρονακτικά μέσω κάποιας ιστοσελίδας, ή μέσω διεπαφής γραμμής εντολών, αξίζει να αναφερθούμε στο πώς το Terraform γνωρίζει την υποδομή για την οποία είναι υπεύθυνο και πώς την καταγράφει στην εσωτερική του κατάσταση.

Η εσωτερική κατάσταση του Terraform, η οποία περιέχει πληροφορίες σχετικά με

την ανεπτυγμένη από αυτό υποδομή, καταγράφεται σε ένα αρχείο κατάστασης Terraform. Όταν σε έναν τρέχων κατάλογο, έστω /foo/bar, τρέξουμε το Terraform, δημιουργείται ένα αρχείο /foo/bar/terraform.tfstate. Το περιεχόμενο αυτού του αρχείου είναι μία απεικόνιση σε μορφή JSON από τους πόρους των αρχείων διαμόρφωσης στις αναπαραστάσεις αυτών των πόρων στον πραγματικό κόσμο. Αυτή η JSON απεικόνιση είναι που συμβουλεύεται το Terraform ώστε με κάθε εντολή να δημιουργήσει ένα πλάνο εκτέλεσης με βάση τη διαφορά των αρχείων διαμόρφωσης και της πραγματικά ανεπτυγμένης υποδομής.

Αν για έναν πόρο βρει ότι έχει ήδη μοναδικό αναγνωριστικό παρόχου για αυτό, σημαίνει ότι το έχει αναπτύξει ήδη στην πραγματικότητα και πλέον μένει να εφαρμόσει κάποια αλλαγή σε αυτό, αν υπάρχει. Επικοινωνώντας με τους παρόχους (μέσω των εξωτερικών plugins), λαμβάνει την πιο πρόσφατη κατάσταση του πόρου και καθορίζει τις αλλαγές που πρέπει να γίνουν, είτε πρόκειται για επί μέρους ανανέωση ή για καταστροφή και επαναδημιουργία.

Αν στο ήδη υπάρχον αρχείο κατάστασης Terraform υπάρχουν απεικονίσεις πόρων οι οποίοι δεν υπάρχουν στα ανανεωμένα αρχεία διαμόρφωσης, αυτό σημαίνει πως οι συγκεκριμένοι πόροι πρέπει να καταστραφούν. Συνεπώς, το Terraform χρησιμοποιεί τα μοναδικά αναγνωριστικά παρόχου που διατηρεί για αυτούς και κατά την επικοινωνία του με τους παρόχους δίνει οδηγία για καταστροφή αυτών. Στο ανανεωμένο αρχείο κατάστασης, οι συγκεκριμένες απεικονίσεις δεν θα υπάρχουν πλέον.

Τέλος, αν στα αρχεία διαμόρφωσης δηλώνονται πόροι οι οποίοι δεν υπάρχουν στο ισχύον αρχείο κατάστασης, οι πόροι αυτοί θα πρέπει να δημιουργηθούν. Το Terraform θα περιμένει να του επιστραφούν τα κατάλληκα μοναδικά αναγνωριστικά και θα αποθηκεύσει τις νέες απεικονίσεις στο ανανεωμένο αρχείο κατάστασης.

Πέρα από το αρχείο terraform.tfstate, για μεγαλύτερη ανοχή σε σφάλματα και δυνατότητα εύκολης επιστροφής στην προηγούμενη κατάσταση, με κάθε εντολή terraform plan/apply/destroy και αλλαγή κατάστασης, η προηγούμενη κατάσταση αποθηκεύεται εφεδρικά στο αρχείο terraform.tfstate.backup.

3.5.2 Εισαγωγή κατάστασης

Το αρχείο κατάστασης, λόγω της ιδιαίτερης μορφής που έχει, προορίζεται αποκλειστικά και μόνο για εσωτερική χρήση από το ίδιο το εργαλείο. Οποιαδήποτε χειρονακτική αλλαγή αποθαρρύνεται και η μόνη περίπτωση που ίσως χρειαστεί κανείς να στοχεύσει την κατάσταση είναι με την εντολή terraform import. Η συγκεκριμένη εντολή επιτρέπει να εισαχθεί στο αρχείο κατάστασης και κατ΄ επέκταση στον έλεγχο του εργαλείου υποδομή η οποία δημιουργήθηκε εκτός αυτού. Αν και είναι ένας αρκετά καλός τρόπος για να μεταβεί η υποδομή μας υπό τον πλήρη έλεγχο του Terraform, απαιτεί περισσότερα βήματα σε σχέση με την κλασική ροή εργασιών.

Αρχικά, χρειάζεται να αναγνωρίσουμε και να προσδιορίσουμε πλήρως τα κομμάτια υποδομής που θέλουμε να εισάγουμε, συνήθως μέσω των μοναδικών αναγνωριστικών που προσφέρει ο κάθε πάροχος.

Στη συνέχεια, δηλώνουμε πολύ βασικά τους αντίστοιχους πόρους στα αρχεία δια-

μόρφωσης και εισάγουμε την υποδομή, πραγματοποιώντας τις κατάλληλες αντιστοιχίσεις μεταξύ resource blocks και πραγματικών πόρων. Σε αυτό το σημείο είναι πολύ σημαντικό να τονιστεί ότι το Terraform δεν παράγει αυτόματα τη διαμόρφωση για τους εισαγόμενους πόρους, συνεπώς ακόμα δεν έχει συγχρονιστεί η κατάσταση με αυτούς.

Για να γίνει αυτό, εκτελούμε την εντολή terraform plan. Κατά τα γνωστά, η συγκεκριμένη εντολή προσπαθεί να παρουσιάσει ένα σχέδιο που θα δημιουργήσει τους πόρους των αρχείων διαμόρφωσης. Αυτό δεν είναι δυνατό, όσο παρουσιάζονται σφάλματα στη διαμόρφωση και συγκεκριμένα όσο δεν υπάρχουν απαραίτητες παράμετροι στις δηλώσεις πόρων. Για να λυθεί αυτό το πρόβλημα, υπάρχουν δύο προσεγγίσεις.

Σύμφωνα με την πρώτη προσέγγιση, μπορούμε να κάνουμε αντιγραφή του κατάλληλου κομματιού της κατάστασης και επικόλληση στη δήλωση του συγκεκριμένου πόρου που αυτό αφορά. Σίγουρα αποτελεί μία αρκετά εύκολη λύση, έχει όμως σαν αποτέλεσμα υπερβολικά ρητά resource blocks, με παραμέτρους που έχουν προκαθορισμένες τιμές και θα μπορούσαν να παραληφθούν από τα αρχεία διαμόρφωσης, δίνοντας έτσι ένα πιο ευανάγνωστο αποτέλεσμα.

Η δεύτερη προσέγγιση αποτελείται από επαναλαμβανόμενες εκτελέσεις της εντολής terraform plan. Με κάθε εκτέλεση, επιλέγουμε τις σωστές παραμέτρους που λείπουν από τη διαμόρφωση και τους δίνουμε τις κατάλληλες τιμές. Κατανοώντας τις διαφορετικές ανάγκες του κάθε πόρου και του κύκλου ζωής του, αυτή η διαδικασία επαναλαμβάνεται μέχρι το πλάνο εκτέλεσης να περιλαμβάνει μόνο επι μέρους ενημερώσεις των πόρων που στοχεύουμε, όποτε και εκτελούμε την εντολή terraform apply ώστε η κατάσταση να συγχρονιστεί με την πραγματικότητα και οι εξωτερικοί πόροι να περάσουν στον πλήρη έλεγχο του εργαλείου. Σαφώς πρόκειται για μία πιο δύσκολη διαδικασία και απαιτεί πιο προσεκτικές κινήσεις, οδηγεί όμως σε πιο διαχειρίσιμα αρχεία διαμόρφωσης.

3.5.3 Απομακρυσμένη αποθήκευση κατάστασης

Σε ένα μικρού μεγέθους ή προσωπικό έργο, η αποθήκευση ολόκληρης της κατάστασης του Terraform σε ένα και μοναδικό τοπικό αρχείο δουλεύει και είναι μία πλήρως ικανοποιητική λύση. Αναλογιζόμενοι όμως τη σημασία που έχει για το εργαλείο η κατάσταση, όντας η απεικόνιση που έχει για την πραγματικότητα και αποτελόντας την αρχή όλων των ενεργειών του, εύκολα προκύπτει ότι για μεγαλύτερα έργα ή έργα στα οποία εμπλέκεται μεγαλύτετος αριθμός μηχανικών, ακόμη και ομάδες, η λύση αυτή δεν μπορεί να είναι αποδεκτή.

Αρχικά, για χρήση του Terraform στο περιβάλλον μίας ομάδας, το αρχείο κατάστασης είναι απαραίτητο να βρίσκεται σε μία κοινόχρηστη τοποθεσία, ώστε όλα τα μέλη να έχουν πρόσβαση σε αυτό. Βέβαια, το πρόβλημα που εμφανίζεται σε όλα τις εφαρμογές που έχουν έναν κοινόχρηστο πόρο για διαφορετικούς πελάτες, αυτό της εμφάνισης καταστάσεων ανταγωνισμού, δημιουργεί την ανάγκη για κλείδωμα του αρχείου κατάστασης και σειριακής επεξεργασίας του. Ακόμα, ως βέλτιστη πρακτική είναι καλό τα διαφορετικά περιβάλλοντα δοκιμών και παραγωγής να είναι απομονωμένα

μεταξύ τους συνεπώς και χρειάζονται διαφορετικά αρχεία κατάστασης για το καθένα από αυτά.

Η λύση που προσφέρει το Terraform δεν είναι η χρήση κάποιου παραδοσιακού συστήματος ελέγχου έκδοσης (π.χ. git/Github), αλλά η ενσωματωμένη υποστήριξή του για δημοφιλή εξωτερικά απομακρυσμένα και κοινόχρηστα συστήματα αποθήκευσης, όπως τα Amazon S3, Azure Storage, Google Cloud Storage. Με αυτό τον τρόπο, πριν από κάθε εντολή terraform plan/apply/destroy, το εργαλείο αυτόματα θα φορτώσει την πιο πρόσφατη κατάσταση και με το τέλος της εκτέλεσης θα την αποθηκεύσει. Έτσι κανένας δεν θα δουλεύει με ξεπερασμένες εκδόσεις του αρχείου κατάστασης, ενώ η εγγενής υποστήριξη για κλείδωμα που προσφέρουν τα παραπάνω συστήματα εγγυάται ότι κάθε φορά μόνο ένα πελάτης θα τροποποιεί την κατάσταση. Επιπλέον πλεονεκτήματα στο κομμάτι της ασφάλειας και της αξιοπιστίας αποτελούν η δυνατότητα για κρυπτογράφηση τόσο κατά την αποθήκευση όσο και κατά τη μεταφορά του αρχείου, η διατήρηση των προηγούμενων εκδόσεων, για εύκολη επιστροφή σε προηγούμενη κατάσταση αλλά και αυτή καθαυτή η υποδομή, η οποία είναι πλήρως διαχειριζόμενη από τον αντίστοιχο πάροχο και έχει σχεδιαστεί για μέγιστη αντοχή και διαθεσιμότητα.

Παράδειγμα χρήσης του Amazon S3 ως remote backend

Για χρήση του Amazon S3 ως remote backend αρχικά χρειάζεται να δημιουργηθεί ένας κάδος αποθήκευσης (S3 bucket). Φυσικά και αυτό μπορεί να γίνει εξωτερικά του εργαλείου, στο πνεύμα όμως της φιλοσοφίας της υποδομής ως κώδικα αξίζει να το δημιουργήσουμε μέσω αυτού. Σε ένα ξεχωριστό φάκελο εργασιών, ώστε να έχουμε ανεξαρτησία από την υπόλοιπη υποδομή, ορίζουμε τον πάροχο aws και χρησιμοποιούμε τον πόρο aws_s3_bucket ως εξής:

```
provider "aws" {
    region = "eu-central-1"
}

resource "aws_s3_bucket" "my_s3_bucket" {
  bucket = "my_unique_bucket_name"
  versioning {
    enabled = true
  }
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
         sse_algorithm = "AES256"
       }
    }
  }
```

```
  tags = {
    Name = "My S3 bucket managed by Terraform"
  }
}
```

Η παράμετρος bucket ορίζει το όνομα του κάδου αποθήκευσης, για το οποίο πρέπει να είμαστε ιδιαίτερα προσεκτικοί ώστε να είναι καθολικά μοναδικό μεταξύ όλων των AWS πελατών, το versioning ενεργοποιεί τη διατήρηση προηγούμενων εκδόσεων και το server_side_encryption_configuration την κρυπτογράφηση.

Στη συνέχεια, στον κανονικό φάκελο εργασιών μας, κάνοντας χρήση ενός terraform block, δηλαδή ενός block κώδικα που αφορά διαμόρφωση για το ίδιο το Terraform, ορίζουμε ως backend τον κάδο που μόλις δημιουργήσαμε

```
terraform {
  backend "s3" {
    bucket = "my_unique_bucket_name"
    key = "terraform/terraform.tfstate"
  }
}
```

Εδώ, bucket το καθολικά μοναδικό όνομα του κάδου μας και key η πλήρης διαδρομή του αρχείου στο οποίο θα αποθηκευθεί η κατάσταση.

3.6 Προσαρμογή νέων εικονικών μηχανών στο νέφος μέσω Υποδομής ως Κώδικα - Χρήση του cloud-init

3.6.1 Εισαγωγή

Κατά την αρχικοποίηση μίας νέας εικονικής μηχανής σε ένα περιβάλλον υπολογιστικού νέφους χρησιμοποιείται ένα γενικό πρότυπο εικόνας λειτουργικού συστήματος που διαθέτει ο πάροχος νέφους ή ο πάροχος του λειουργικού συστήματος. Δίχως καμία παρέμβαση σε αυτή τη διαδικασία, κάθε νέα εικονική μηχανή είναι ίδια με την επόμενη και δεν επιτελεί καμία λειτουργία. Χρησιμοποιώντας τα δεδομένα χρήστη και ένα εργαλείο σαν το cloud-init, μπορούμε κατά το χρόνο εκτέλεσης στο πρώτο στάδιο της αρχικοποίησης, δηλαδή στο πρώτο ξεκίνημα της μηχανής, να την προσαρμόσουμε με βάση κάποια διαμόρφωση, ώστε να έχει τον επιθυμητό ρόλο στην υποδομή μας.

Ανεπτυγμένο αρχικά από την εταιρεία Canonical για διαμόρφωση Ubuntu Linux μηχανών όταν πρωτοεμφανίστηκε το Amazon EC2, το cloud-init είναι πλέον προεγκατεστημένο ως υπηρεσία σε μια πληθώρα από βασισμένα σε Unix λειτουργικά συστήματα (Ubuntu, Arch Linux, CentOS, Red Hat, FreeBSD, Fedora, κ.α.) και υποστηρίζεται από την πλειοψηφία των παρόχων υπηρεσιών νέφους (AWS, Azure, Google Cloud, Oracle Cloud, DigitalOcean, κ.α.). Χρησιμοποιείται για εγκατάσταση πακέτων, διαμόρφωση χρηστών και ασφάλειας, τροποποίηση αρχείων καθώς και

ορισμό εργασιών και σεναρίων που θέλουμε να εκτελεστούν στο πρώτο ξεκίνημα της μηχανής.

Ξεπερνώντας φυσικά την ιδέα της χειροκακτικής αρχικοποίησης μίας μηχανής, η χρήση του συγκεκριμένου εργαλείου προσφέρει πλεονεκτήματα που το κάνουν να υπερτερεί εναντίον της χρήσης εξειδικευμένων εικόνων μηχανής. Αρχικά, η δημιουργία ξεχωριστών εικόνων μηχανής για κάθε έναν από τους ρόλους μίας υποδομής είναι αρκετά κοπιαστικό και χρονοβόρο. Ομοίως, σε περιβάλλοντα ταχέως αναπτυσσόμενου λογισμικού, η δημιουργία και η ανανέωση των εικόνων μηχανής για κάθε νέα έκδοση ενός προγράμματος αυξάνει το χρόνο προς την αγορά. Ειδικά με την άνθηση της τεχνολογίας εικονοκοποίησης και τη χρήση containers, μία αρχική βασική διαμόρφωση με την κατάλληλη μηχανή διαχείρισης containers και χρήση αυτών από εκεί και πέρα μοιάζει ιδανικότερη προσέγγιση.

Ένα ακόμα πλεονέκτημα της χρήσης cloud-init είναι η φορητότητα που προσφέρει. Οι διαφορετικές διανομές Linux συχνά έχουν διαφορετικά κελύφη μεταξύ τους και ένα σενάριο γραμμένο για παράδειγμα σε bash μπορεί να μη δουλέψει σε μία έκδοση όπως η FreeBSD, όπου το προκαθορισμένο κέλυφος τόσο για το root χρήστη όσο και για τους υπόλοιπους δεν είναι το bash (tcsh και sh αντίστοιχα). Το cloud-init και η #cloud-config μορφή του παρέχει την απαραίτητη λειτουργικότητα με πολύ πιο κομψό, οικουμενικό και συνεπή τρόπο.

Όπως προαναφέρθηκε, το cloud-init είναι προεγκατεστημένο ως υπηρεσία στις εικόνες λειτουργικών συστημάτων των παρόχων υπηρεσιών νέφους. Συγκεκριμένα, στα βασισμένα στο systemd Linux, είναι το cloud-init.service που τρέχει κατά την εκκίνηση. Τότε, πραγματοποιεί την εξειδίκευση, όπως αυτή προκύπτει από τα παρεχόμενα από τον πάροχο μεταδεδομένα, που αφορούν μεταξύ άλλων τη δικτυακή διαμόρφωση και τους δίσκους της μηχανής, και τις παρεχόμενες από το χρήστη εντολές.

Παράδειγμα χρήσης του cloud-init

```
#cloud_config

package_update: true
package_upgrade: true

runcmd:
  - mkdir /run/foo
  - echo "Hello World $(date +%s)!" > /run/foo/boot_msg.txt
```

Από το συγκεκριμένο παράδειγμα φαίνεται πως στην πρώτη γραμμή είναι απαραίτητο να ορίσουμε το #cloud_config ως το διερμηνέα του αρχείου, το οποίο είναι σε YAML μορφή. Βλέπουμε ότι αντί για apt-get update, apt-get upgrade, χρησιμοποιούμε τα package_update, package_upgrade και έτσι ορίζουμε καθολικά την ενημέρωση της μηχανής και των πακέτων ανεξαρτήτως λειτουργικού συστήματος και διαχειριστή πακέτων. Στη συνέχεια, ακολουθώντας πάντα σωστή YAML σύνταξη, τυπώνουμε σε ένα τυχαίο αρχείο τη χρονοσφραγίδα της εκτέλεσης της εντολής. Σημαντικό είναι

να τονιστεί εκ νέου ότι οι runcmd εντολές θα εκτελεστούν μόνο κατά την πρώτη εκκίνηση της μηχανής, συνεπώς ακόμα και μετά από επανεκκινήσεις η χρονοσφραγίδα που θα δούμε στο αρχείο μας δεν θα έχει αλλάξει.

3.6.2 Ενσωμάτωση του cloud-init στο Terraform

Συνδυάζοντας τις πηγές δεδομένων template_file και template_cloudinit_config, το Terraform παρέχει δυνατότητα για δεδομένα χρήστη μέσω του cloud-init.

Συγκεκριμένα, το template_cloudinit_config χρησιμοποιεί μια πολυμερή και πολύμορφη δομή (multipart MIME format), ώστε να συμπεριλάβει στη διαμόρφωση της νέας εικονικής μηχανής αρχεία με οδηγίες σε διαφορετικές μορφές, ανάλογα με τις ανάγκες του χρήστη. Κάθε μέρος έχει το δικό του τύπο και το δικό του περιεχόμενο, το οποίο είναι μια αναφορά σε κάποια template_file πηγή δεδομένων. Εκεί, ορίζεται και το πραγματικό περιεχόμενο του μέρους, μαζί με πιθανές παραμέτρους που μπορεί να χρειάζεται να συμπεριληφθούν.

Κάθε template_cloudinit_config πηγή δεδομένων έχει μία και μοναδική ιδιότητα, τη rendered, η οποία παρέχει την τελική απόδοση της πολυμερούς διαμόρφωσης, και χρησιμοποιείται ως αναφορά σε αυτή από την παράμετρο για εισαγωγή δεδομένων χρήστη του εκάστοτε πόρου παροχής εικονικής μηχανής.

Παράδειγμα προσαρμογής νέας AWS EC2 μηχανής με χρήση Terraform και cloud-init

```
resource "aws_instance" "example" {
  ami = "ami-056114420b6ed624e"
  instance_type = "t2.micro"
  user_data = data.template_cloudinit_config.example.rendered
}

data "template_cloudinit_config" "example" {
  gzip= true
  base64_encode = true

  part {
    filename     = "init.cfg"
    content_type = "text/cloud-config"
    content      = "${data.template_file.example.rendered}"
  }
}

data "template_file" "example" {
  template = file("init.cfg")
}
```

Εδώ, βλέπουμε ότι με το user_data αναφερόμαστε στην παρεχόμενη από το χρήστη διαμόρφωση, η οποία θα δοθεί σε gzip και base64_encoded μορφή (προαιρετικά). Η διαμόρφωση είναι αυτή του παραδείγματος της προηγούμενης ενότητας και περιέχεται στο αρχείο init.cfg.

Κεφάλαιο 4

Spark

4.1 Μεγάλα Δεδομένα

4.1.1 Εισαγωγή

Ο όρος 'Μεγάλα Δεδομένα' χρησιμοποιείται για να περιγράψει το κομμάτι εκείνο της επιστήμης της Πληροφορικής που ασχολείται με την συλλογή, αποθήκευση, επεξεργασία, μεταφορά, αναζήτηση, οπτικοποίηση και ασφάλεια συνόλων δεδομένων τα οποία είναι υπερβολικά μεγάλα και σύνθετα για τα παραδοσιακά υπολογιστικά συστήματα και τις τεχνικές που αυτά προσφέρουν και μπορούν να υποστηρίξουν.

Έχοντας ήδη φτάσει τα 79 zettabytes το 2021, ο όγκος των συνολικών δεδομένων παγκοσμίως συνεχίζει να αυξάνεται ταχύτατα, με τις εκτιμήσεις να φτάνουν ακόμα και πάνω από τα 180 zettabytes έως το 2025. Η τάση αυτή είναι απολύτως δικαιολογημένη και οφείλεται σε μεγάλο βαθμό στην ευκολία με την οποία δημιουργούνται ψηφιακά δεδομένα στις μέρες μας. Κινητά τηλέφωνα, κάμερες, αισθητήρες, το 'Διαδίκτυο των Πραγμάτων', υπηρεσίες ροής πραγματικού χρόνου, αρχεία καταγραφής αποτελούν πηγές διαρκούς παραγωγής δεδομένων. Ταυτόχρονα, το κόστος των παραπάνω καθώς και των μέσων αποθήκευσης των παραγώμενων δεδομένων μειώνεται διαρκώς. Συνεπώς, θεωρείται έως και ασύμφορο να μην συλλέγει κανείς δεδομένα, συγκρίνοντας το κόστος παραγωγής-συντήρησής τους και το αναμενόμενο κέρδος που αυτά δύνανται

να προσφέρουν.

Την εξέλιξη των μέσων συλλογής και αποθήκευσης δεδομένων δυστυχώς δεν ακολουθεί η εξέλιξη στο υλικό των υπολογιστών και συγκεκριμένα στους επεξεργαστές. Η συνεχής ανάπτυξη μέχρι περίπου το 2005, η οποία σύμφωνα με το νόμο του Moore σήμαινε διπλασιασμό του αριθμού των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα κάθε περίπου 2 χρόνια, είχε ως αποτέλεσμα οι ταχύτητες των επεξεργαστών να αυξάνονται και μαζί με αυτές αυξάνονταν και η ταχύτητα εκτέλεσης των εφαρμογών, δίχως καμία αλλαγή στη λογική και τον πηγαίο κώδικά τους. Ο τεράστιος αριθμός τρανζίστορ που αρχικά οδήγησε σε αρχιτεκτονικές παραλληλισμού επιπέδου εντολών (Instruction Level Parallelism - ILP) και ισχυρότατους πυρήνες, στη συνέχεια παρουσίασε μεγάλα προβλήματα στη διάχυση θερμότητας, εμφάνισε φαινόμενα διαρροής ρεύματος και γενικά παρείχε επιδόσεις δυσανάλογες της κατανάλωσης ενέργειας που είχε.

Η σημαντική αλλαγή στον τρόπο σκέψης στο σχεδιασμό υλικού, αλλά στη συνέχεια και κατ΄ επέκταση στο σχεδιασμό λογισμικού, αφορά της στροφή στην κλιμακωσιμότητα. Αρχικά, με τη δημιουργία πολυπύρηνων επεξεργαστών και στη συνέχεια με τη δημιουργία συστημάτων που εκμεταλλεύονται ολόκληρες συστάδες υπολογιστών.

4.1.2 Αξιοποίηση Συστάδων Υπολογιστών

Με τη στροφή στις χρήση συστάδων υπολογιστών και τις λογικές 'διαίρει και βασίλευε' (divide and conquer) προέκυψαν θέματα παραλληλοποίησης και ανάθεσης εργασιών σε διεργασίες-εργάτες (workers), επικοινωνίας και συγχρονισμού μεταξύ αυτών για την προσπέλαση και επεξεργασία κοινόχρηστων πόρων, καθώς και προβλήματα διαμοιρασμού και συνδυασμού των ενδιάμεσων δεδομένων που προκύπτουν από τις επιμέρους εργασίες.

Τα παραπάνω προβλήματα δεν έχουν μία προφανή λύση, διότι πρόκειται για ασύγχρονα συστήματα και πρακτικά οτιδήποτε συμβαίνει είναι μη ντετερμινιστικό, συνεπώς αδύνατο να προβλεφθεί και εξαιρετικά δύσκολο να αναλυθεί σε σενάρια αποσφαλμάτωσης. Η χρήση τεχνικών ήδη γνωστών από το πεδίο των Λειτουργικών Συστημάτων, όπως οι σημαφόροι, τα κλειδώματα και τα φράγματα βελτιώνουν την κατάσταση, αλλά και πάλι προκύπτουν καταστάσεις ανταγωνισμού και αδιέξοδα (deadlock και livelock).

Τα προγραμματιστικά μοντέλα που εμφανίστηκαν, όπως το OpenMP για πολυπύρηνα συστήματα με μοιραζόμενη μνήμη και το MPI για ανταλλαγή μηνυμάτων μεταξύ κόμβων, αλλά και τα σχεδιαστικά μοντέλα, όπως αυτά των 'master' και 'slave' διεργασιών και οι ουρές 'παραγωγών-καταναλωτών', φυσικά και βοήθησαν και μάλιστα σε αυτά στηρίχθηκε η λειτουργία συστημάτων υπερυπολογιστών τεράστιας κλίμακας. Δεν μπορούν να παραβλεφθούν όμως τα σημαντικά τους μειονεκτήματα, όπως το γεγονός ότι με τις διαφορετικές υλοποιήσεις που έχουν, αποτελούν εξειδικευμένες βιβλιοθήκες, που πιθανώς παρουσιάζουν ελαφρώς αποκλίνουσες συμπεριφορές σε ορισμένες ακραίες περιπτώσεις, κάτι που κάνει την αποσφαλμάτωση, τη συνεργασία μεταξύ ομάδων και την οικουμενικότητα δυσκολότερη. Ταυτόχρονα, αποτελούν αρκετά χαμηλού επιπέδου λύσεις, απαιτούν ειδικές γνώσεις από τους χρήστες και έχουν

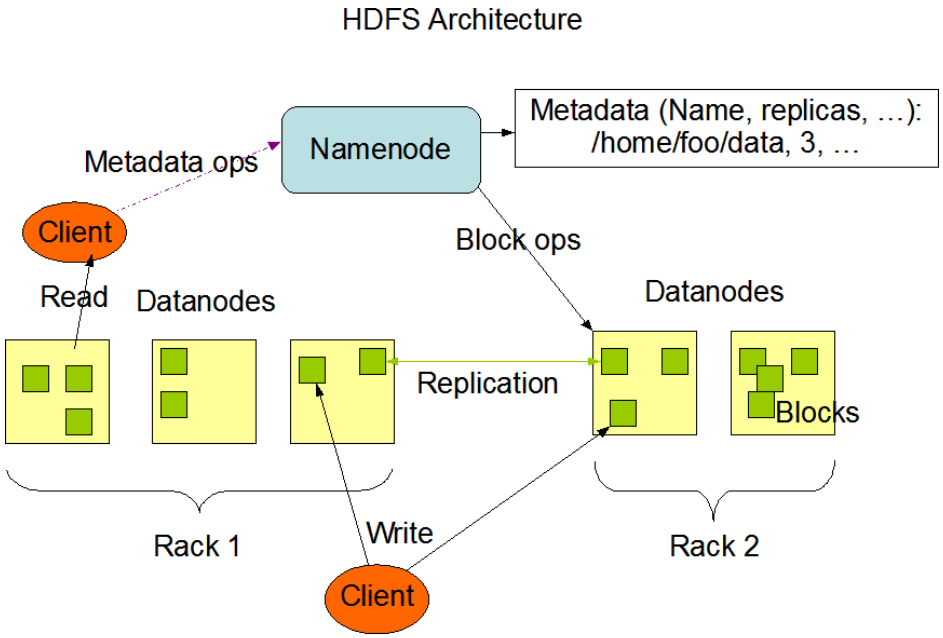
υψηλό προγραμματιστικό κόστος, κάτι που στο διεπιστημονικό πεδίο των Μεγάλων Δεδομένων δεν μπορεί να θεωρηθεί ιδανικό.

Σημαντική τομή και αφετηρία για τη άνθηση των Μεγάλων Δεδομένων αποτελεί η υλοποίηση του Hadoop MapReduce από την Apache το 2006, ως υλοποίηση ανοιχτού λογισμικού επιστημονικών εργασιών που προτάθηκαν από την Google το 2004, και αποτέλεσε το υπόβαθρο για τη δημιουργία του Apache Spark, το οποίο και θα αναλυθεί διεξοδικά στη συνέχεια.

4.2 MapReduce

4.2.1 HDFS και Λειτουργία του MapReduce

Το MapReduce είναι ένας συνδυασμός προγραμματιστικού μοντέλου και εργαλείου και χρησιμοποιείται για τη γρήγορη επεξεργασία τεράστιων συνόλων δεδομένων μέσω ενός παράλληλου και κατανεμημένου αλγορίθμου σε μία συστάδα υπολογιστών. Αποτελείται από 2 διακριτές φάσεις: τη 'Map' φάση, όπου η είσοδος διαμοιράζεται σε διαφορετικές διεργασίες με σκοπό την παραγωγή ενός ενδιάμεσου συνόλου δεδομένων της μορφής <κλειδί, τιμή>, και τη 'Reduce' φάση, όπου τα ενδιάμεσα δεδομένα τροφοδοτούνται σε νέες διεργασίες με σκοπό τη συνόψισή τους ανά κλειδί και τη δημιουργία ενός μικρότερου συνόλου της ίδιας μορφής, το οποίο πιθανώς θα είναι και το αποτέλεσμα του προγράμματος.



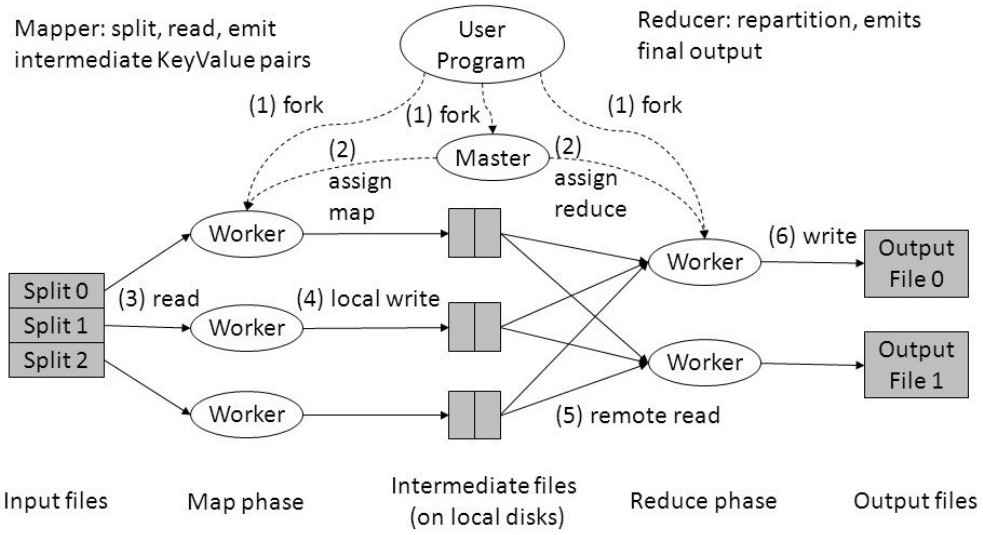
Σχήμα 4.1: Αρχιτεκτονική του HDFS

Η λειτουργία του στηρίζεται στην ύπαρξη ενός κατανεμημένου, κλιμακώσιμου,

με υψηλή ανοχή σε σφάλματα και σχεδιασμένου για λειτουργία σε χαμηλού κόστους υλικό συστήματος αρχείων, το οποίο χρησιμοποιείται σαν είσοδος και έξοδος των εφαρμογών. Το σύστημα αυτό είναι το HDFS (Hadoop Distributed File System). Το HDFS τρέχει επί μίας συστάδας υπολογιστών και ακολουθεί αρχιτεκτονική 'master-slave'. Ένας από τους κόμβους αναλαμβάνει το ρόλο του NameNode και οι υπόλοιποι κόμβοι είναι οι DataNodes. Μόλις ένα αρχείο ανέβει στο HDFS διασπάται σε blocks μεγέθους συνήθως 128 MB, τα οποία διαμοιράζονται στους DataNodes. Ταυτόχρονα, για να επιτευχθεί η ανοχή στα σφάλματα, τα blocks αντιγράφονται σε 3 κόμβους δεδομένων, μάλιστα με επίγνωση των διαφορετικών ραφιών υπολογιστών.

Ο NameNode είναι ο master εξυπηρετητής και διαχειρίζεται το namespace του συστήματος αρχείων, το οποίο εκτείνεται επί ολόκληρης της συστάδας, ρυθμίζει την προσπέλαση των αρχείων από τους πελάτες και διατηρεί πληροφορίες σχετικά με το ποιά αρχεία βρίσκονται σε κάθε DataNode.

Οι DataNodes ακολουθούν εντολές από το NameNode σχετικά με δημιουργία, διαγραφή και αντιγραφή blocks αρχείων και αναλαμβάνουν την υποστήριξη των αιτημάτων για γραφή και ανάγνωση από τους πελάτες του συστήματος αρχείων.



Σχήμα 4.2: Λειτουργία του MapReduce

Επιστρέφοντας στο επίπεδο του MapReduce, η επίσης τύπου 'master-slave' αρχιτεκτονική εκφράζεται με τις διεργασίες JobTracker και TaskTracker. Η διεργασία JobTracker εκτελείται στο master κόμβο (μαζί με τη NameNode) και είναι υπεύθυνη για την λήψη αιτημάτων για εκτέλεση εργασιών από τους πελάτες και επικοινωνία με το NameNode σχετικά με την τοποθεσία των απαραίτητων δεδομένων. Στη συνέχεια, αναθέτει εργασίες στους εργάτες και παρακολουθεί την εξέλιξή τους.

Η διεργασία TaskTracker εκτελείται σε όλους τους DataNode κόμβους και αποτελεί τον εργάτη. Λαμβάνει κώδικα, καθήκοντα και πληροφορίες σχετικά με τα δεδομένα

που θα επεξεργαστεί από τον JobTracker και στη συνέχεια εκτελεί τις εργασίες του επί των δεδομένων και αναφέρει την κατάσταση του στο master JobTracker.

Αναλυτικότερα, για μια map εργασία, ο TaskTracker αρχικά διαβάζει από το κατανεμημένο σύστημα αρχείων το κομμάτι των δεδομένων που του έχει ανατεθεί. Σημαντικό είναι να αναφερθεί ότι αυτό γίνεται με τρόπο που να σέβεται την τοπικότητα των δεδομένων, με σκοπό τη μείωση του φόρτου του δικτύου και της καθυστέρησης στην ολοκήρωση των εργασιών. Συνεπώς, στην πλειονότητά τους τα δεδομένα που αναλαμβάνει ο κάθε εργάτης βρίσκονται στον ίδιο κόμβο και εξυπηρετούνται από την τοπική DataNode διεργασία. Με την ολοκλήρωση της παραγωγής των ενδιάμεσων <κλειδί, τιμή> ζευγών, μια συνάρτηση διαίρεσης τα χωρίζει σε όσες είναι και οι reduce εργασίες που ακολουθούν, τα αποθηκεύει στο τοπικό σύστημα αρχείων και στη συνέχεια ενημερώνει το master για την τοποθεσία τους.

Όσον αφορά τις reduce διεργασίες, αφού ενημερωθούν από το master για την τοποθεσία όλων των δεδομένων που τους αναλογούν, τα συγκεντρώνουν από τους αντίστοιχους DataNodes, προχωρούν στην παραγωγή των νέων, συνοψισμένων <κλειδί, τιμή> ζευγών, τα οποία και είναι τα τελικά ζεύγη της συγκεκριμένης εργασίας MapReduce. Αν αυτή είναι και η τελική εργασία MapReduce, ο έλεγχος επιστρέφεται στο πρόγραμμα χρήστη, αλλιώς η διαδικασία επαναλαμβάνεται από την αρχή.

4.2.2 Περιορισμοί

Υπάρχουν κάποιοι βασικοί περιορισμοί που τίθενται λόγω της αρχιτεκτονικής του MapReduce και οδήγησαν σε αναζήτηση πιο ευέλικτων λύσεων για επεξεργασία τεράστιων συνόλων δεδομένων. Αρχικά, το Hadoop οικοσύστημα περιλαμβάνει τόσο το HDFS σύστημα αποθήκευσης όσο και τη MapReduce υπολογιστική μηχανή, η σύζευξη όμως των δύο είναι ισχυρή και το ένα δεν λειτουργεί χωρίς το άλλο. Σε περίπτωση κλιμάκωσης της υποδομής με περισσότερους εργάτες, είναι απαραίτητη και η προσθήκη αποθηκευτικής χωρητικότητας για το HDFS, αφού εκεί αποθηκεύονται τα δεδομένα των εργατών. Η εξάρτηση αυτή μόνο ιδανική δεν μπορεί να θεωρηθεί, ειδικά όταν η υποδομή βρίσκεται σε υπολογιστικό νέφος, όπου συνήθως η αγορά αποθηκευτικού χώρου είναι ανεξάρτητη από την αγορά υπολογιστικής ισχύος.

Ανελιξία προκαλεί επίσης το γεγονός ότι με κάθε εναλλαγή map και reduce διεργασιών τα ενδιάμεσα δεδομένα γράφονται στο δίσκο, ενώ ανεξαρτήτως εργασίας είναι απαραίτητο κάθε map διεργασία να ακολουθείται από μία reduce, ακόμα κι αν μία από αυτές δεν επιτελεί κάποια χρήσιμη λειτουργία. Αυτό μειώνει την εκφραστικότητα και την επίδοση που μπορούν να έχουν οι εφαρμογές μας, ενώ αυξάνει το κόστος διαβάσματος και γραψίματος στο δίσκο και το σύστημα αρχείων.

4.3 Εισαγωγή στο Spark

Το Spark είναι μια ενοποιημένη υπολογιστική μηχανή συνδυασμένη με ένα σύνολο από βιβλιοθήκες για παράλληλη επεξεργασία δεδομένων σε συστάδες υπολογιστών. Συγκεκριμένα, προσφέρει ένα σύνολο από βιβλιοθήκες για SQL, μηχανική μάθηση, επεξεργασία ροής και αναλύσεις γράφων, βελτιστοποιημένες για υψηλές επιδόσεις ακόμα και με συνδυασμένη χρήση τους. Ταυτόχρονα, εστιάζει τις προσπάθειες του μόνο στο κομμάτι της επεξεργασίας και δεν ασχολείται με την αποθήκευση των δεδομένων. Εμφανίστηκε το 2006, αρχικά ως ερευνητική εργασία των Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker και Ian Stoica του UC Berkeley, με σκοπό τη δημιουργία μιας μηχανής που θα αντιμετωπίζει την αναποδοτικότητα του MapReduce μέσω μιας σειράς βελτιστοποιήσεων, όπως συναρτησιακές λειτουργίες βασισμένες στη γλώσσα προγραμματισμού Scala, δυνατότητα για διαδραστική χρήση και αποδοτικός, εντός-της-μνήμης διαμοιρασμός δεδομένων μεταξύ των διαφόρων υπολογιστικών βημάτων.

Η λειτουργία του επί μιας συστάδας υπολογιστών στηρίζεται στην ύπαρξη ενός διαχειριστή συστάδας (cluster manager). Ο διαχειριστής αυτός μπορεί να είναι ο προκαθορισμένος του Spark, ή κάποιος εξωτερικός (Kubernetes, Mesos, YARN) και αναλαμβάνει την χορηγία πόρων στις εφαρμογές μας, οι οποίες, όπως ίσως είναι αναμενόμενο, ακολουθούν την αρχιτεκτονική 'master-slave'.

Σε μία εφαρμογή Spark, το ρόλο του master αναλαμβάνει η διαδικασία driver. Ο driver τρέχει τη main() συνάρτηση της εφαρμογής και διατηρεί όλες τις πληροφορίες για αυτή. Επιπλέον, κατανέμει και χρονοπρογραμματίζει εργασίες στους εργάτες, ενώ αποκρίνεται και σε είδοδο από το χρήστη, όταν το Spark χρησιμοποιείται διαδραστικά για επί τούτου ερωτήματα. Αυτή η διαδικασία, η οποία ονομάζεται SparkSession, είναι απολύτως απαραίτητη για μία Spark εφαρμογή, αφού μέσω αυτής εκετελούνται οι ορισμένες από το χρήστη λειτουργίες.

Οι εργάτες του Spark είναι οι executors. Ο ρόλος τους είναι απλός, να εκτελούν τις εργασίες που τους αναθέτει ο driver και να αναφέρουν την πρόοδο των εργασιών και την κατάστασή τους σε αυτόν.

Διεπαφές Προγραμματισμού

Η κύρια αφαίρεση που προσφέρει το Spark για την προσπέλαση δεδομένων είναι το RDD (Resilient Distributed Dataset), μια ανεκτική σε σφάλματα συλλογή δεδομένων, διαιρεμένη μεταξύ των κόμβων μιας συστάδας υπολογιστών για παράλληλη επεξεργασία. Είναι η πιο χαμηλού επιπέδου διεπαφή προγραμματισμού και μάλιστα χρησιμοποιείται και από τον πυρήνα του εργαλείου. Η χρήση της στις εφαρμογές απαιτεί προσεκτικούς χειρισμούς, έντονη δακτυλογράφηση και γνώση του πώς ακριβώς θέλουμε να επεξεργαστούμε τα δεδομένα μας.

Υπάρχουν ακόμα δύο νεότερες, υψηλότερου επιπέδου και πλέον κυρίαρχες διεπαφές προγραμματισμού για Δομημένες συλλογές: το DataFrame και το Dataset.

Αποτελούν κατανεμημένες συλλογές με δομή πίνακα, με σαφώς ορισμένες γραμμές και στήλες. Κάθε γραμμή έχει τον ίδιο αριθμό στηλών με τις υπόλοιπες και κάθε στήλη έχει πληροφορίες για τον τύπο δεδομένων της, ο οποίος είναι ο ίδιος για κάθε γραμμή. Ο ορισμός των ονομάτων των στηλών ενός DataFrame και των τύπων τους ονομάζεται σχήμα.

Η διαφορά των δύο διεπαφών εντοπίζεται στο ποιός διαχειρίζεται τους τύπους που χρησιμοποιούν. Στην περίπτωση του DataFrame, το Spark διαχειρίζεται πλήρως τους τύπους των στηλών του και τους ελέγχει μόνο κατά την εκτέλεση. Η διαχείριση αυτή πραγματοποιείται μέσω του εσωτερικού, βελτιστοποιημένου συστήματος τύπων που διαθέτει και επιτρέπει αποδοτικούς υπολογισμούς και υψηλή εξειδίκευση. Αντιθέτως, η διαχείριση των τύπων ενός Dataset αναλαμβάνεται από το Java Virtual Machine και ο έλεγχός του πραγματοποιείται κατά τη μεταγλώττιση ενός προγράμματος - υπενθυμίζεται ότι η γλώσσα συγγραφής του Spark είναι η Scala, δηλαδή γλώσσα που τρέχει επί του JVM. Ο ορισμός των τύπων γίνεται μέσω των case classes της Scala. Η διαχείριση των τύπων από το JVM μειώνει τη δυνατότητα για βελτιστοποιήσεις, αλλά ενισχύει την ασφάλεια τύπων και ξεκλειδώνει δυνατότητες για εφαρμογή πολύπλοκων κα εξειδικευμένων συναρτήσεων επί των δεδομένων. Αξίζει να σημειωθεί ότι στην πραγματικότητα ένα DataFrame είναι ένα Dataset τύπου Row, όπου Row η εσωτερική απεικόνιση της βελτιστοποιημένης εντός-της-μνήμης μορφής που χρησιμοποιεί το Spark για υπολογισμούς γραμμής.

Επιπλέον, σε αντίθεση με τα RDDs που είναι μία αδιαφανή συλλογή αντικειμένων, με καμία πληροφορία για τη μορφή των δεδομένων της, τα DataFrame/Dataset έχουν σχήμα και πληροφορίες τύπων συσχετισμένα με αυτά. Μειώνοντας το εύρος του τί μπορεί να εκφραστεί, ανοίγει το πεδίο για βελτιστοποιήσεις που προηγουμένως δεν ήταν δυνατές.

Ταυτόχρονα, συμπληρωματικά με τα DataFrame/Dataset, το Spark παρέχει τη Spark SQL, μια διεπαφή προγραμματισμού η οποία επιτρέπει στο χρήστη να χρησιμοποιεί τις γνώσεις και την εμπειρία του από τα κλασικά Συστήματα Διαχείρισης Βάσεων Δεδομένων (Database Management Systems) και την SQL πάνω στη μηχανή του Spark. Στην πραγματικότητα, οι χρήσεις DataFrame/Dataset και Spark SQL έχουν ταυτόσημη συμπεριφορά και επιδόσεις.

Ανεξαρτήτως διεπαφής προγραμματισμού, για να μπορούν οι πολλαπλοί executors να εκτελούν υπολογισμούς παράλληλα, τα δεδομένα διασπόνται σε κομμάτια που ονομάζονται διαιρέσεις (partitions). Πρόκειται για συλλογές από γραμμές δεδομένων που βρίσκονται σε ένα φυσικό κόμβο της συστάδας υπολογιστών. Αντιπροσωπεύουν το πώς τα δεδομένα έχουν κατανεμηθεί φυσικά στης συστάδα κατά την εκτέλεση και, σε συνδυασμό με τους executors, ορίζουν το μέγιστο επίπεδο παραλληλισμού που μπορεί να επιτευχθεί. Βέβαια, με μόνο 1 partition, ο παραλληλισμός είναι αυστηρά 1, ανεξαρτήτως του αριθμού των executors, όπως και με μόνο 1 executor, δηλαδή μόνο μία υπολογιστική μονάδα, ο παραλληλισμός θα είναι επίσης 1, ανεξαρτήτως των διαιρέσεων των δεδομένων.

4.4 Οκνηρή Αποτίμηση

Γνωστή ήδη από τη Θεωρία Γλωσσών Προγραμματισμού, το Συναρτησιακό Προγραμματισμό και το λ-λογισμό, η οκνηρή αποτίμηση είναι μία στρατηγική που καθυστερεί την αποτίμηση μίας έκφρασης μέχρι η τιμή της να χρειαστεί και αποφεύγει τις επαναλαμβανόμενες αποτιμήσεις.

Στο περιβάλλον του Spark, η οκνηρή αποτίμηση εκφράζεται μέσω του διαχωρισμού μεταξύ μετασχηματισμών και ενεργειών επί των δομών δεδομένων του. Επιπλέον, έχει ιδιαίτερη σημασία στη δημιουργία ενός κατευθυνόμενου ακυκλικού γράφου, μέσω του οποίου δηλώνονται οι εξαρτήσεις μεταξύ των διαφόρων μετασχηματισμών, ορίζεται η ορθή σειρά εκτέλεσης και παρέχεται ανοχή στα σφάλματα.

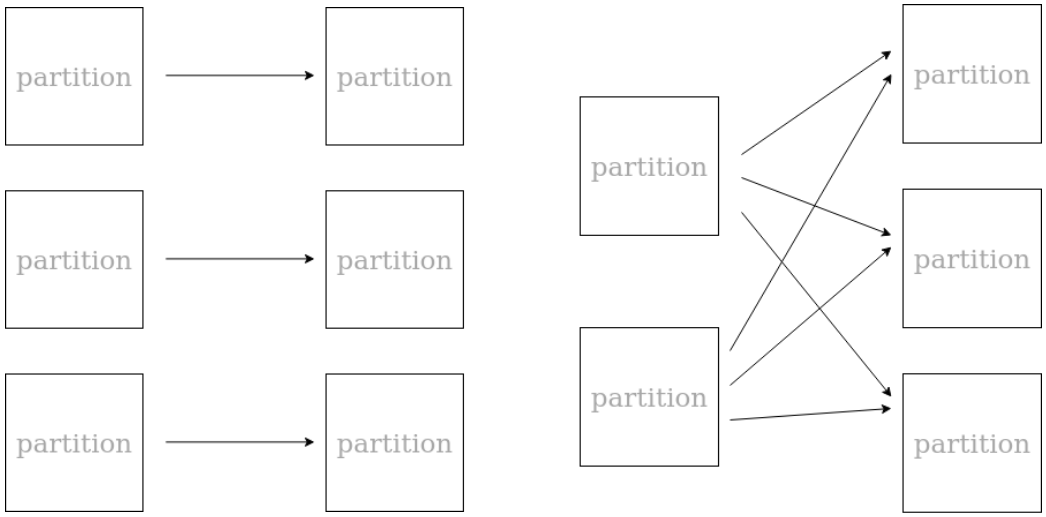
Μέσω της οκνηρής αποτίμησης και του γράφου μεταχηματισμών, το Spark μπορεί να προχωρήσει σε βελτιστοποιήσεις που δεν θα ήταν δυνατές αν τα δεδομένα τροποποιούνταν αμέσως. Το γεγονός αυτό το διαφοροποιεί από το MapReduce και το κάνει αποδοτικότερο.

4.4.1 Μετασχηματισμοί και Ενέργειες

Οι δομές δεδομένων του Spark, είτε για πρόκειται για DataFrame/Dataset ή για RDD, είναι αμεταβλητές συνεπώς δεν μπορούν να αλλάξουν μετά τη δημιουργία τους, παρά μόνο να αντικατασταθούν. Οι μετασχηματισμοί (Transformations) είναι αφηρημένες οδηγίες που ορίζουν το πως θα τροποποιηθούν τα δεδομένα και εκφράζουν την επιχειρησιακή λογική του χρήστη. Το Spark θα καθυστερήσει την εκτέλεση των μετασχηματισμών μέχρι να τους ζητηθούν δεδομένα. Τότε, όλοι οι αναγκαίοι μετασχηματισμοί θα έχουν σημειωθεί και το εργαλείο θα μπορέσει να προχωρήσει σε βελτιστοποιήσεις μεταξύ τους, ενώ ταυτόχρονα θα καταναλώσει λιγότερη υπολογιστική ισχύ και θα έχει μικρότερο αποτύπωμα μνήμης. Είναι ιδιαιτέρως σημαντικό να τονιστεί αυτό: το Spark δεν θα πραγματοποιήσει καμία ενέργεια μέχρι να του ζητηθεί κάποιο αποτέλεσμα.

Οι μετασχηματισμοί μπορούν να χωριστούν σε δύο κατηγορίες, με βάση τον τύπο των εξαρτήσεων που δημιουργούν. Από τη μία είναι οι μετασχηματισμοί που δημιουργούν στενές εξαρτήσεις. Με την εφαρμογή αυτών των μετασχηματισμών, κάθε διαίρεση της εισόδου θα συνεισφέρει δεδομένα σε το πολύ μία διαίρεση της εξόδου. Από την άλλη, οι μετασχηματισμοί που δημιουργούν ευρείες εξαρτήσεις. Με αυτούς, οι διαιρέσεις ειδόδου συνεισφέρουν σε πολλαπλές διαιρέσεις εξόδου.

Εδώ συναντάμε και ένα σημείο με έντονο ερευνητικό ενδιαφέρον, αυτό της ανάμιξης δεδομένων. Οι μετασχηματισμοί με ευρείες εξαρτήσεις δημιουργούν την ανάγκη για ανταλλαγή διαιρέσεων δεδομένων μεταξύ των εργατών-executors. Κάθε ανάμιξη διασπάει το γράφο υπολογισμών σε στάδια (stages). Ανάλογα το μέγεθος των διαιρέσεων, των αριθμό των εργατών και τη λοξότητα των δεδομένων (το πόσο ανισοκατανεμηνένα είναι μεταξύ των εργατών), η ανάμιξη μπορεί να αποτελέσει κώλυμα στην επίδοση μιας εργασίας. Εν αντιθέσει, οι μετασχηματισμοί που δημιουργούν στενές εξαρτήσεις επιτρέπουν τη δημιουργία σωλήνωσης εντός της μνήμης και το συνδυασμό



(α′) Στενή εξάρτηση (β′) Ευρεία εξάρτηση

Σχήμα 4.3: Είδη μετασχηματισμών

τους σε ένα βήμα. Σαφές είναι το πλεονέκτημα αυτής της βελτιστοποίησης έναντι του MapReduce και της διαφοράς στη φιλοσοφία των δύο εργαλείων. Με το MapReduce, η ιδιαιτέρως χρονοβόρα χρήση του δίσκου πραγματοποιείται μεταξύ όλων των βημάτων για αποθήκευση των ενδιάμεσων δεδομένων. Στο Spark, όσο δυνατόν περισσότερα βήματα συνδυάζονται εντός της μνήμης και το σύστημα αρχείων χρησιμοποιείται μόνο για την ανάμιξη δεδομένων. Όταν πρόκειται να πραγματοποιηθεί ανάμιξη δεδομένων, ο κάθε executor αποθηκεύει τα ενδιάμεσα δεδομένα του στο τοπικό σύστημα αρχείων, ενημερώνει τον driver για την τοποθεσία τους και λειτουργεί ως εξυπηρετητής προς τους άλλους executors που θα τα θελήσουν.

Μία ακόμα βελτιστοποιήση που είναι δυνατή λόγω της οκνηρής αποτίμησης και προσφέρει καλύτερη απόδοση είναι η προώθηση κατηγορήματος. Συγκεκριμένα, έστω μία δουλειά που ως τελευταίο μετασχηματισμό έχει ένα φιλτράρισμα (το φιλτράρισμα είναι ένα είδος μετασχηματισμού) που ορίζει τη λήψη μόνο ενός ή μερικών γραμμών του συνόλου δεδομένων. Με μία κλασική MapReduce υλοποίηση, θα είχαμε μία διαφορετική δουλειά για κάθε μετασχηματισμό που έχει οριστεί, συνεπώς υπολογισμό όλων των ενδιάμεσων αποτελεσμάτων και τέλος μία δουλειά για το φιλτράρισμα. Προφανώς, η επεξεργασία μόνο των δεδομένων που τελικά χρειαζόμαστε είναι ο πιο αποδοτικός τρόπος να πραγματοποιηθεί μία εργασία φιλτραρίσματος. Με το Spark, την οκνηρή αποτίμηση και την προώθηση κατηγορήματος, το φιλτράρισμα θα πραγματοποιηθεί στην αρχή και οι ενδιάμεσοι μετασχηματισμοί θα εφαρμοστούν μόνο στις γραμμές που μας ενδιαφέρουν.

Όσον αφορά τις ενέργειες (Actions), είναι το έναυσμα για εκκίνηση των υπολογισμών. Με μία ενέργεια, το Spark ξεκινάει την επεξεργασία των δεδομένων, όπως αυτή έχει οριστεί από μια σειρά μετασχηματισμών. Με το τέλος των υπολογισμών, ο

έλεγχος επιστρέφεται στο driver πρόγραμμα.

Παραδείγματα Μετασχηματισμών, Ενεργειών και χρήση DataFrame

Στο παρακάτω πολύ απλό παράδειγμα δημιουργούμε ένα DataFrame παραλληλοποιώντας ένα range αριθμών από το 1 ως το 1000 χρησιμοποιώντας τη μέθοδο toDF επί του Sequence που επιστρέφει η range μέθοδος. Για να έχουμε πρόσβαση στη μέθοδο range χρησιμοποιούμε τη μεταβλητή spark, που εδώ είναι ο driver της εφαρμογής μας.

```
val numbersDF = spark.range(1000).toDF("number")
val evenNumbersDF = numbersDF.where("number % 2 == 0")
val countOfEven = evenNumbersDF.count()
```

Το evenNumbersDF δημιουργείται με την εφαρμογή του μετασχηματισμού where (φιλτράρισμα) και του κατηγορήματος "number % 2 == 0". Φυσικά, αφού μέχρι και τη δεύτερη γραμμή δεν έχουμε ορίσει κάποια ενέργεια, το Spark δεν έχει προχωρήσει σε κανένα υπολογισμό. Στην τρίτη γραμμή, μέσω της μεθόδου-ενέργειας count(), θα υπολογιστούν όλα τα ενδιάμεσα DataFrames και θα επιστραφεί στο driver πρόγραμμα και τη μεταβλητή countOfEven το σύνολο των ζυγών αριθμών από το 1 ως το 1000.

Στο δεύτερο, πιο σύνθετο παράδειγμα, φαίνεται ο τρόπος που δημιουργούμε DataFrames από δεδομένα που βρίσκονται σε εξωτερικές πηγές καθώς και πώς αποθηκεύουμε δεδομένα από ένα DataFrame. Συγκεκριμένα, μέσω του DataFrame reader του driver, μπορούμε να ορίσουμε μια πληθώρα μορφών που μπορούν να έχουν τα αρχεία εισόδου (εδώ φαίνονται οι μορφές csv και json), την τοποθεσία των δεδομένων καθώς και μία σειρά δυνατών επιλογών. Για τη csv μορφή μπορούμε να ορίσουμε το διαχωριστικό (π.χ. comma ή tab), την κενή τιμή (null) και το αν η πρώτη γραμμή θεωρείται ως επικεφαλίδες των στηλών ή όχι. Με την επιλογή inferSchema επιτρέπουμε στο Spark να συμπεράνει τους τύπους των δεδομένων μας και να αναθέσει στις στήλες του DataFrame που θα προκύψει τους αντίστοιχους τύπους από το εσωτερικό του σύστημα τύπων.

```
val idAndSalaryDF = spark.read
                .format("csv")
                .option("inferSchema", "true")
                .option("header", "true")
                .option("sep", ",")
                .option("nullValue", "")
                .load("/resources/data/idAndSalary.csv")

val idAndNameDF = spark.read
                .format("json")
                .option("inferSchema", "true")
                .load("/resources/data/idAndName.json")
```

```
val cond = idAndSalaryDF.col("id") == idAndNameDF.col("id")

val nameAndSalaryDF = idAndSalaryDF
              .join(idAndNameDF, cond)
              .select("name", "salary")
              .orderBy("salary", "-1")

nameAndSalaryDF.write
              .format("csv")
              .mode(SaveMode.Overwrite)
              .option("header", "true")
              .option("sep", "\t")
              .save("/resources/data/nameAndSalary.csv")
```

Έστω ότι στο παράδειγμα μας θέλουμε να ταξινομήσουμε τους υπαλλήλους μιας εταιρείας με βάση το μισθό τους. Έχουμε δύο αρχεία, ένα csv με επικαφαλίδες "id" και "salary" και ένα json με field names "id" και "name". Μέσω του μετασχηματισμού join προχωράμε στο συνδυασμό των δύο DataFrames, την προβολή της πληροφορίας που μας ενδιαφέρει και στη συνέχεια στη ζητούμενη ταξινόμηση. Τονίζεται ότι η ένωση των δύο συνόλων δεδομένων μέσω του join ορίζει μία ευρεία εξάρτηση, οπότε θα χρειαστεί ανάμιξη των δεδομένων και ότι οι υπολογισμοί θα ξεκινήσουν μόλις οριστεί η ενέργεια save, μόλις δηλαδή ζητήσουμε ένα αποτέλεσμα από το Spark.

4.4.2 Βελτιστοποιητής Catalyst

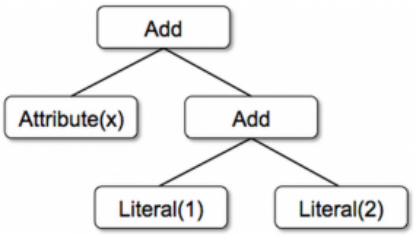
Όπως έχει σημειωθεί στη μέχρι τώρα ανάλυση του Spark, η οκνηρή αποτίμηση, η χρήση μετασχηματισμών και η δημιουργία ενός κατευθυνόμενου ακυκλικού γράφου υπολογισμών από το spark driver επιτρέπουν τη βελτιστοποίηση ενός ερωτήματος και της αποτίμησής του επί της συστάδας υπολογιστών. Αξίζει εδώ να αναφερθούμε στο βελτιστοποιητή του Spark, τον Catalyst, και να εμβαθύνουμε στη λειτουργία του.

Ο βελτιστοποιητής Catalyst εκ πρώτης όψεως μοιάζει με ένα μεταγλωττιστή γλώσσας προγραμματισμού. Ως είσοδο δέχεται μία υψηλού επιπέδου περιγραφή ενός παρεχόμενου από το χρήστη ερωτήματος, ορισμένου μέσω DataFrames ή Spark SQL, και ως έξοδο παρέχει μία σειρά από εργασίες πάνω σε RDDs, βελτιστοποιημένες για την εκτέλεσή τους επί της υποκείμενης φυσικής συστάδας υπολογιστών. Πρόκειται για ένα βελτιστοποιητή ερωτημάτων βασισμένο σε συναρτησιακές δομές της γλώσσας Scala, όπως οι Μερικώς Ορισμένες Συναρτήσεις (partial functions - PF) και η Αντιστοίχιση Προτύπων (pattern matching - PM). Συμπεριλαμβάνει μία βιβλιοθήκη για αναπαράσταση και τροποποίηση δέντρων και επί αυτής βιβλιοθήκες και κανόνες για ανάλυση ερωτημάτων, λογική βελτιστοποίηση, φυσικό σχεδιασμό και δημιουργία (Java bytecode) κώδικα.

Η βασική δομή δεδομένων που χρησιμοποιεί ο Catalyst είναι τα δέντρα. Ο βασικός τύπος των κόμβων του δέντρου είναι ο TreeNode και κάθε άλλους τύπος κόμβου

ορίζεται ως υποκλάση αυτού. Κάθε κόμβος του δέντρου έχει ένα δοσμένο τύπο και 0 ή περισσότερα παιδιά (μέσω του πεδίου children: Seq[TreeNode]). Φυσικά, οι δομές αυτές είναι αμεταβλητές και επεξεργάζονται μόνο μέσω συναρτησιακών μετασχηματισμών.

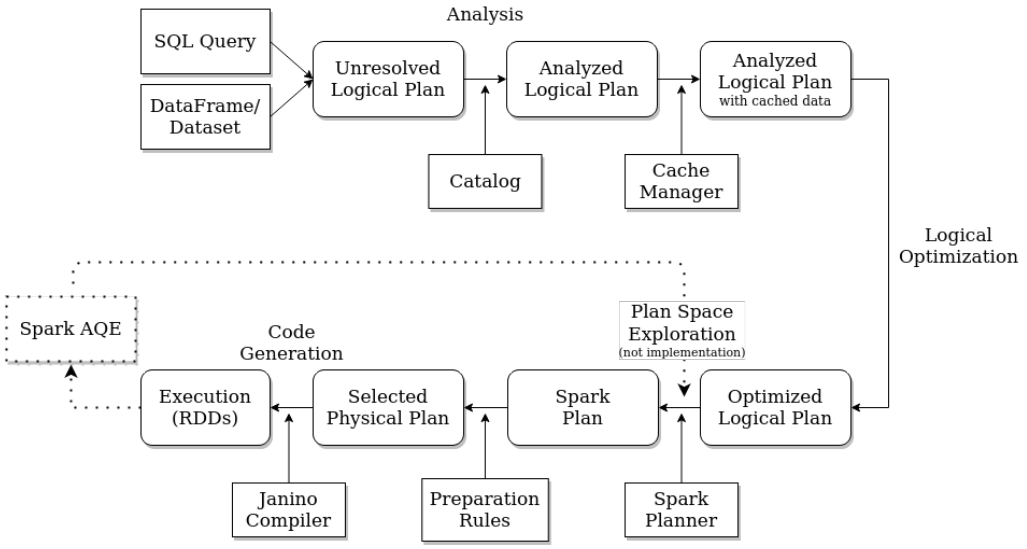
Έστω μία περίπτωση αντιστοίχισης γλώσσας μέσω εκφράσεων με τους ακόλουθους τύπους κόμβων: Literal(value: Int), Attribute(name: String), Add(left: TreeNode, right: TreeNode) για την αναπαράσταση αντίστοιχα μιας απλής σταθεράς, μιας τιμής από μία γραμμή εισόδου και ενός αθροίσματος δύο εκφράσεων. Τότε, μία έκφραση της μορφής x + (1 + 2) μπορεί να αναπαρασταθεί μέσω της γλώσσας Scala ως δέντρο ως εξής: Add(Attribute(x), Add(Literal(1) + Literal(2)))



Η τροποποίηση των δέντρων γίνεται μέσω εφαρμογής μετασχηματισμών (Transforms), δηλαδή κανόνων 'από δέντρο σε δέντρο' βασισμένων στο pattern matching. Ο Catalyst, ακολουθώντας μία αναδρομική, από κάτω προς τα πάνω προσέγγιση (bottom-up), εφαρμόζει κάθε μετασχηματισμό σε όλους τους κόμβους του δέντρου. Οι μετασχηματισμοί είναι ορισμένοι ως partial functions, οπότε οποιοσδήποτε κόμβος ή υποδέντρο δεν ταιριάζει με τα εξεταζόμενα πρότυπα, απλώς παραλείπεται.

Η διαδικασία βελτιστοποίησης και εκτέλεσης ενός ερωτήματος (QueryExecution) μέσω του Catalyst έχει αρκετά διακριτά στάδια. Αρχικά, ο κώδικας του χρήστη, εφόσον είναι συντακτικά ορθός, μετατρέπεται σε ένα Μη Επιλυμένο Λογικό Πλάνο (Unresolved/Parsed Logical Plan), το οποίο είναι ένα σύνολο από αφηρημένους μετασχηματισμούς που περιγράφουν την είσοδο του χρήστη. Στη συνέχεια, μετά το στάδιο της Ανάλυσης (Analysis) και της δημιουργίας του Επιλυμένου Λογικού Πλάνου (Analyzed Logical Plan), ακολουθεί η βελτιστοποίησή του και η δημιουργία τους Βελτιστοποιημένου Λογικού Πλάνου (Optimized Logical Plan). Βασισμένο σε αυτό, δημιουργείται μία σειρά από Φυσικά Πλάνα (Spark Plans) και με βάση ένα μοντέλο εκτίμησης κόστους (Cost-Based Model) επιλέγεται αυτό που θα εκτελεστεί. Τέλος, το Επιλεγμένο Φυσικό Πλάνο (Selected Physical Plan) θα χρησιμοποιηθεί για την παραγωγή Java bytecode κώδικα και τη δημιουργία των εργασιών που θα εκτελεστούν από τους executors, επιστρέφοντας έτσι τον κατευθυνόμενο ακυκλικό γράφο υπολογισμών επί RDDs στην τελική του μορφή.

Συγκεκριμένα, στο στάδιο της Ανάλυσης, ο Catalyst χρησιμοποιεί τον Catalog, ο οποίος είναι ένα αποθετήριο πληροφοριών σχετικών με όλα τα DataFrames που έχουν οριστεί στο πρόγραμμα του χρήστη, και επιλύει το αρχικό Μη Επιλυμένο Λογικό Πλάνο. Αυτό είναι απαραίτητο διότι ο κώδικας μπορεί να είναι συντακτικά ορθός, οπότε το αρχικό δέντρο να έχει δημιουργηθεί, αλλά να έχει αναφορές σε ιδιότητες ή



Σχήμα 4.4: Η σωλήνωση λογικού και φυσικού σχεδιασμού του Catalyst

σχέσεις που δεν υπάρχουν ή δεν έχουν αυστηρώς ορισμένους τύπους. Μέσω του Catalog, πραγματοποιείται η αναζήτηση σχέσεων ονομαστικά και η αντιστοίχιση σχέσεων και ιδιοτήτων στην πραγματικό τους όνομα, ώστε να μη δημιουργηθούν συγκρούσεις μεταξύ ιδιοτήτων διαφορετικών σχέσεων που έχουν το ίδιο όνομα, προσδιορίζονται ιδιότητες που αναφέρονται στην ίδια τιμή και τους δίνεται μοναδικό αναγνωριστικό και πραγματοποιούνται εξαναγκασμοί και μετατροπές τύπων σε εκφράσεις των οποίων τα συστατικά έχουν προηγουμένως προσδιοριστεί. Σε περίπτωση αποτυχίας των παραπάνω βημάτων, το Μη Επιλυμένο Λογικό Πλάνο θα απορριφθεί με μήνυμα λάθους.

Ένα επιπλέον βήμα βελτιστοποίησης είναι η withCachedData φάση, στην οποία ο διαχειριστής κρυφής μνήμης (CacheManager) ελέγχει το εντός-της-μνήμης αποθετήριό του (το πεδίο cachedData: LinkedList[CachedData]) για τμήματα του εξεταζόμενου δομημένου ερωτήματος που πιθανώς έχουν μονιμοποιηθεί εντός της κύριας μνήμης από προηγούμενο ερώτημα. Αν βρεθεί ένα τέτοιο τμήμα, τότε ο CacheManager το αντικαθιστά με το μονιμοποιημένο τμήμα, συμπεραλαμβανομένων όλων των υποερωτημάτων του. Σε περίπτωση που τίποτα δεν έχει μονιμοποιηθεί, το λογικό πλάνο δεν επηρεάζεται από την αναζήτηση στην κρυφή μνήμη και προχωράει αυτούσιο στην επόμενη φάση.

Ακολούθως, το Επιλυμένο Λογικό Πλάνο, το οποίο μέχρι στιγμής αποτελείται από τελεστές που αντιστοιχούν στη σειρά με την οποία έχουν οριστεί οι μετασχηματισμοί στον πηγαίο κώδικα (π.χ. read -> Relation, select -> Project, filter -> Filter), περνάει από το στάδιο της Λογικής Βελτιστοποίησης (Logical Optimization). Η βασισμένη σε κανόνες βελτιστοποίηση αναδιατάσσει και τροποποιεί το δέντρο μετασχηματισμών και περιλαμβάνει μεταξύ άλλων:

Αναδίπλωση Σταθερών Αντικατάσταση εκφράσεων που μπορούν να εκτιμηθούν στατικά με τις αντίστοιχες τιμές τους κατά τη μεταγλώττιση (π.χ. Add(Attribute(x), Add(Literal(1), Literal (2))) -> Add(Attribute(x), Literal(3))). Η εκτίμηση μίας στατικής έκφρασης θα γίνει πλέον μόνο 1 φορά και όχι όσες φορές είναι οι γραμμές του αντίστοιχου DataFrame.

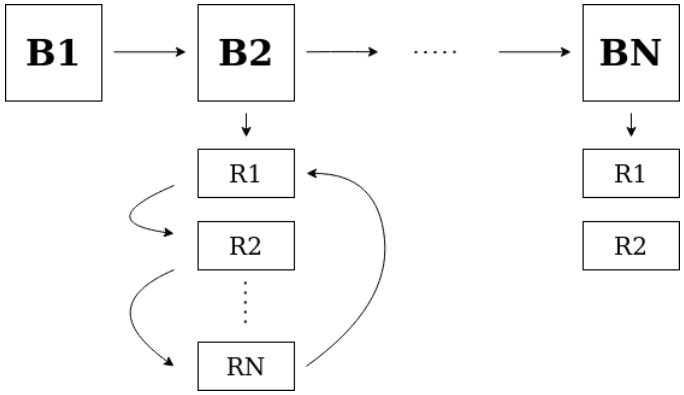
Προώθηση Κατηγορημάτων Εφαρμόζεται σε μετασχηματισμούς φιλτρατίσματος αμέσως μετά τη φόρτωση ενός συνόλου δεδομένων ή μετά από μία προβολή και στοχεύει στην αύξηση της επίδοσης ενός ερωτήματος μέσω της προώθησης του φιλτραρίσματος σε όσο το δυνατόν χαμηλότερο επίπεδο, ιδανικά στην πηγή των δεδομένων. Αντί να φορτωθεί ολόκληρο το σύνολο δεδομένων στη μνήμη του Spark, φορτώνονται μόνα όσα δεδομένα θα περάσουν τον έλεγχο, εξοικονομώντας έτσι κύρια μνήμη και μειώνοντας το φόρτο στο σύστημα αρχείων αλλά και στο δίκτυο. Μία ακόμα περίπτωση εφαρμογής του συγκεκριμένου μετασχηματισμού είναι στις ενώσεις (Joins). Όταν το φιλτράρισμα μπορεί να πραγματοποιηθεί έναντι ενός από τα δύο κομμάτια της ένωσης, τότε αυτό γίνεται πριν την ένωση.

Περικοπή μέσω Προβολής Επιτρέπει την ελαχιστοποίηση της μεταφοράς δεδομένων μεταξύ του συστήματος αποθήκευσής τους και του Spark (ή ακόμα και μεταξύ βημάτων βαθύτερα στο γράφο υπολογισμών) εξαλείφοντας τις στήλες που δεν χρειάζονται στο τελικό αποτέλεσμα. Αποτελεί δυϊκή τεχνική της Προώθησης Κατηγορημάτων (κέρδη στην επίδοση μειώνοντας τον αριθμό των γραμμών ή τον αριθμό των στηλών ανά γραμμή).

Διάδοση null Αντικατάσταση εκφράσεων με τιμές που μπορούν να εκτιμηθούν στατικά.

Απλοποίηση Boolean Εκφράσεων Απλοποίηση Boolean εκφράσεων μέσω συγχώνευσης, απαλειφής του τελεστή NOT, παραγοντοποίησης και αναδιαμόρφωσης ώστε να εκτιμηθούν όσο το δυνατόν λιγότερες σχέσεις.

Η εφαρμογή των συνολικά 109 κανόνων-μετασχηματισμών λογικής βελτιστοποίησης (69 μοναδικοί) πραγματοποιείται σειριακά σε 25 δεσμίδες (batches). Την εκτέλεσή τους αναλαμβάνει ο RuleExecutor, τυπικά διατηρώντας ένα γενικό σκοπό για κάθε δεσμίδα (π.χ. γενικές βελτιστοποιήσεις, αναθεωρήσεις εκφράσεων). Κάθε δεσμίδα ορίζεται από το όνομα (π.χ. Aggregate, Join Reorder, RewriteSubquery), τη στρατηγική εκτέλεσης και το σύνολο των κανόνων της. Η εκτέλεση των επιμέρους κανόνων των δεσμίδων γίνεται ακολουθώντας δύο στρατηγικές. Η μία είναι η επανάληψη σταθερού σημείου (FixedPoint(n)) και ορίζει τη συνεχόμενη εκτέλεση των κανόνων μίας δεσμίδας μέχρι το δέντρο να συγκλίνει, δηλαδή να μην υπάρχει κάποια αλλαγή, ή μέχρι ένα μέγιστο αριθμό n επαναλήψεων (όποιο συμβεί πρώτο). Η άλλη στατηγική είναι η εφάπαξ εφαρμογή (Once - μέγιστος αριθμός επαναλήψεων = 1) και χρησιμοποιείται σε δεσμίδες που αναλαβάνουν λειτουργίες όπως προσθαφαίρεση κόμβων και έλεγχοι εγκυρότητας.



Σχήμα 4.5: Ομαδοποίηση των κανόνων σε δεσμίδες από τον RuleExecutor και εκτέλεσή τους με τις τεχνικές σταθερού σημείου και εφάπαξ εφαρμογής

Σε αυτό το σημείο η σωλήνωση του Catalyst έχει ολοκληρώσει τα βήματα τροποποίησης των LogicalPlan και συνεχίζει με τη δημιουργία και βελτιστοποίηση των SparkPlan στο στάδιο του φυσικού σχεδιασμού.

Το Βελτιστοποιημένο Λογικό Πλάνο αναλαμβάνει ο SparkPlanner, ο οποίος εφαρμόζει μια σειρά από Στρατηγικές (Strategies) που μετατρέπουν τους λογικούς, αλγεβρικούς κόμβους σε φυσικούς κόμβους με τους αντίστοιχους χαμηλότερου επιπέδου τελεστές. Η εφαρμογή αυτή για ακόμα μία φορά είναι αναδρομική και με την τροποποίηση του κάθε κόμβου και μέσω των planLater κλήσεων όλα τα λογικά υποδέντρα μετατρέπονται στα αντίστοιχα φυσικά τους. Οι συνολικά 10 διακριτές Στρατηγικές (μεταξύ άλλων Aggregation, JoinSelection, InMemoryScans, BasicOperators) αναλαμβάνουν την προώθηση κατηγορημάτων και την περικοπή μέσω προβολής στις πηγές των δεδομένων, το σχεδιασμό της φόρτωσης των δεδομένων, τις διαφορετικές επιλογές συνάθροισης (SortAggregate, HashAggregate, ObjectHashAggregate) και τις διαφορετικές επιλογές ένωσης (broadcast hash, shuffle hash, sort merge, broadcast nested loop, cartesian product). Αυτές οι επιλογές χρησιμοποιούνται για την παραγωγή μιας σειράς από SparkPlans, από τα οποία επιλέγεται το βέλτιστο με βάση ένα Μοντέλο Εκτίμησης Κόστους (το οποίο στην πραγματικότητα, προς το παρόν, επιστρέφει απλώς το πρώτο -.next()- πλάνο).

Το SparkPlan περνάει από ένα σύνολο κανόνων προετοιμασίας (preparation rules - Rule[SparkPlan]) για τη μετατροπή του στο Επιλεγμένο Φυσικό Πλάνο. Οι κανόνες αυτοί χρησιμοποιούνται για επαληθεύσεις ορθότητας (όπως ότι όλα τα υποερωτήματα έχουν σχεδιαστεί και η διαμέριση και ταξινόμηση των δεδομένων είναι σωστή), συμπτίξεις φυσικών τελεστών σε μία Java συνάρτηση όπου αυτό είναι δυνατό καθώς και για βελτιστοποίηση του φυσικού πλάνου, μέσω της επαναχρησιμοποίησης υποερωτημάτων και των ανταλλαγών δεδομένων. Ενδεικτικά, σημειώνονται οι ακόλουθες βελτιστοποιήσεις φυσικού πλάνου:

CollapseCodegenStages Εισάγει WholeStageCodegenExec φυσικούς κόμβους, οι οποίοι συνδυάζουν πολλαπλούς φυσικούς τελεστές σε μία Java συνάρτηση. Η δημιουργία Java bytecode κώδικα για τη συγκεκριμένη συνάρτηση θα γίνει κατά τη χρόνο εκτέλεσης από το μεταγλωττιστή Janino. Η τεχνική αυτή συμβάλει στην αύξηση της επίδοσης στην εκτέλεση ενός ερωτήματος μέσω της χρήσης των καταχωρητών του επεξεργαστή για τα ενδιάμεσα δεδομένα και της εξάλειψης των κλήσεων εικονικών συναρτήσεων.

EnsureRequirements Επιβεβαιώνει ότι οι διαμερίσεις και η ταξινόμηση των δεδομένων είναι όπως την περιμένουν επόμενοι τελεστές και εισάγει ShuffleExchangeExec και SortExec φυσικούς κόμβους όπου χρειάζεται διανομή ή ταξινόμηση αντίστοιχα (π.χ. εισαγωγή SortExec κόμβου πριν από ένα sort-merge join).

PlanSubqueries Αφορά την προετοιμασία των Scalar Subqueries, τα οποία είναι ένθετα υποερωτημάτα που επιστρέφουν βαθμωτό αποτέλεσμα, δηλαδή μία ακριβώς στήλη από το πολύ μία γραμμή (αν το υποερώτημα επιστρέψει 0 γραμμές, τότε η τιμή της βαθμωτής έκφρασης του υποερωτήματος είναι null). Δημιουργεί το βελτιστοποιημένο φυσικό πλάνο για το λογικό πλάνο του υποερωτήματος και στη συνέχεια σχεδιάζει την εκτέλεσή του εισάγοντας έναν SubqueryExec φυσικό τελεστή.

Μία πολύ σημαντική (προαιρετική) βελτιστοποίηση που προσφέρει το Spark 3.0 και μπορεί να συμβάλει στην αύξηση της επίδοσης της εκτέλεσης ενός ερωτήματος είναι η Δυναμική Εκτέλεση Ερωτημάτων (Adaptive Query Execution - AQE). Πρόκειται για μια τεχνική που στηρίζεται στην κατάτμιση του φυσικού πλάνου εκτέλεσης σε στάδια (query stages) και την εκμετάλλευση παραγώμενων κατά το χρόνο εκτέλεσης στατιστικών. Το φυσικό σημείο που προκρίνεται για κατάτμιση του φυσικού πλάνου είναι τα σημεία όπου υπάρχει ήδη διακοπή των σωληνωμένων τελεστών για ανταλλαγή δεδομένων, είτε λόγω ανάμιξης ή λόγω μετάδοσης κάποιου συνόλου δεδομένου για broadcast join. Τα σημεία αυτά, τα οποία ονομάζονται και σημεία υλοποίησης (materialization points), ορίζουν ότι όλοι οι παράλληλοι υπολογισμοί πριν από αυτά έχουν ολοκληρωθεί και τα ενδιάμεσα δεδομένα έχουν παραχθεί, έχουν συλλεχθεί τα απαραίτητα στατιστικά και οι υπολογισμοί του επόμενου σταδίου μπορούν να ξεκινήσουν.

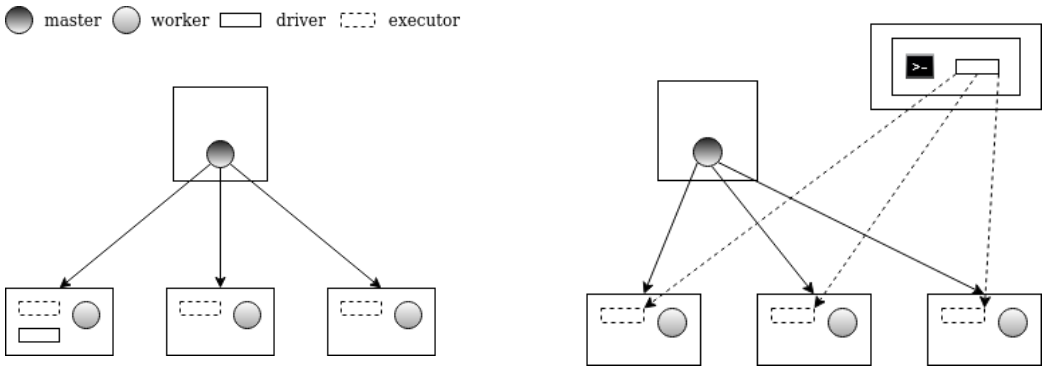
Η λειτουργία του Spark AQE στηρίζεται στη χρήση των στατιστικών που έχουν συλλεχθεί με την εκτέλεση του κάθε query stage και τον εμπλουτισμό με αυτά των κανόνων λογικής και φυσικής βελτιστοποίησης (καθώς και μερικών ειδικών κανόνων AQE φυσικής βελτιστοποίησης) που θα εφαρμοστούν εκ νέου στα στάδια που απομένουν. Το παλιό και το ανανεωμένο επιλεγμένο φυσικό σχέδιο συγκρίνονται και επιλέγεται αυτό που δύναται να προσφέρει την καλύτερη επίδοση. Μερικές από τις δυναμικές βελτιστοποιήσεις που προσφέρει το Spark AQE πλαίσιο εργασιών είναι οι CoalesceShufflePartitions (για συνένωση πολλών υπερβολικά μικρών διαιρέσεων μειώνοντας το φόρτο στο δίκτυο, το χρονοπρογραμματιστή εργασιών του Spark και το γενικό κόστος της ανάκτησης πολλών μικρών block δεδομένων και της αναποτελεσματικής I/O συμπεριφοράς αυτού), OptimizeSkewedJoin (για μετατροπή των ασύμμετρα

μεγάλων διαιρέσεων σε περισσότερες, μικρότερες υποδιαιρέσεις, βελτιώνοντας έτσι την επίδοση των ενώσεων) και OptimizeLocalShuffleReader (για αποφυγή ανάμιξης δεδομένων όταν το sort-merge join μετατρέπεται σε broadcast join). Προϋπόθεση για να εφαρμοστεί το AQE είναι αφενός να έχει ενεργοποιηθεί η συγκεκριμένη λειτουργία, αφετέρου το ερώτημα να μην είναι ερώτημα ροής και να έχει ένα τουλάχιστον υποερώτημα ή σημείο ανταλλαγής (δηλαδή κάποια ένωση ή συνάθροιση).

4.5 Λειτουργία σε Συστάδα Υπολογιστών

Το Spark σχεδιάστηκε με σκοπό να αξιοποιεί ολόκληρες συστάδες υπολογιστών για τις επεξεργαστικές του ανάγκες. Αξίζει να αναλύσουμε την κατανεμημένη 'master-slave' αρχιτεκτονική του και να εστιάσουμε στην υποδομή και τον κύκλο ζωής μιας εφαρμογής.

Όπως έχει προαναφερθεί συνοπτικά στην εισαγωγή, ο Spark driver είναι αυτός που ελέγχει την εκτέλεση μιας εφαρμογής και διατηρεί την κατάστασή της, όσον αφορά τους executors και τα tasks. Ταυτόχρονα, αλληλεπιδρά με τον cluster manager για να αποκτήσει φυσικούς πόρους για τους executors. Οι executors είναι οι εργάτες του συστήματος και η ευθύνη τους είναι η εκτέλεση των εργασιών που λαμβάνουν από τον driver και η αναφορά των αποτελεσμάτων και της κατάστασής τους. Ο cluster manager διατηρεί τη συστάδα των υπολογιστών που χρησιμοποιεί το Spark και παρέχει τους απαραίτητους φυσικούς πόρους.



(α′) cluster mode (β′) client mode

Σχήμα 4.6: Κατανεμημένη εκτέλεση του Spark

Σε αντίθεση με τον driver και τους executors οι οποίοι είναι διεργασίες, ο master και οι workers που χρησιμοποιεί ο cluster manager είναι φυσικοί κόμβοι. Συγκεκριμένα, ανεξαρτήτως cluster manager, συγκεκριμένοι δαίμονες τρέχουν σε αυτά τα μηχανήματα και τους δίνουν τις αντίστοιχες δυνατότητες και χαρακτηριστικά.

Για να διευκολύνει τα διαφορετικά σενάρια παραγωγής και τις διαφορετικές ανάγκες στη φυσική τοποθέτηση των πόρων, το Spark προσφέρει 3 διαφορετικούς

τρόπους εκτέλεσης. Στη λειτουργία συστάδας (cluster mode), μόλις ο cluster manager λάβει το JAR με την εφαρμογή του χρήστη, ξεκινάει τον spark driver σε έναν από τους worker κόμβους, μαζί με τους executors. Συνεπώς, είναι υπεύθυνος για τη διαχείριση όλων των spark διεργασιών. Η λειτουργία συστάδας είναι ιδανική για περιπτώσεις που η εφαρμογή υποβάλλεται στον cluster manager από ένα απομακρυσμένο από τη συστάδα μηχάνημα και επιτρέπει την αποφυγή καθυστέρησης λόγω του δικτύου στην επικοινωνία μεταξύ του driver και των executors.

Η λειτουργία πελάτη (client mode) προκρίνεται ιδανική για περιπτώσεις όπου για την υποβολή εφαρμογών χρησιμοποιείται ένα μηχάνημα 'πύλης' (gateway machine) το οποίο είναι τοποθετημένο στο ίδιο δίκτυο με τους workers. Με τη λειτουργία πελάτη, ο spark driver σηκώνεται μαζί με την υποβολή της εφαρμογής (μέσω της διεργασίας spark-submit) και δρα ως πελάτης στη συστάδα των υπολογιστών, συνεπώς διατηρείται από το gateway machine και όχι από τον cluster manager. Η είσοδος και η έξοδος της εφαρμογής γίνεται μέσω της κονσόλας της μηχανής και επιτρέπει τη χρήση του Spark shell (Spark REPL) για επί τούτου ερωτήματα.

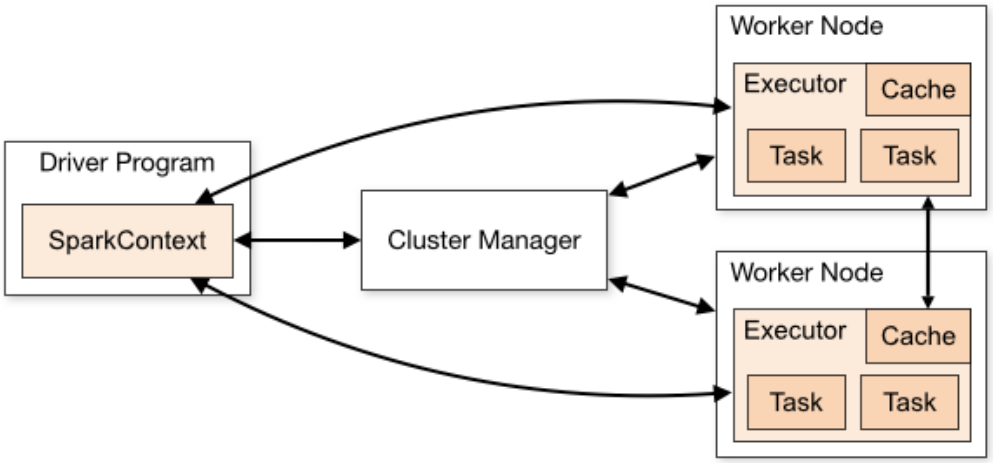
Ο τρίτος και τελευταίος τρόπος εκτέλεσης είναι η τοπική λειτουργία. Εκμεταλλεύεται το γεγονός ότι οι spark driver και spark executors είναι απλές διεργασίες και τις εκτελεί όλες στο ίδιο μηχάνημα, χρησιμοποιώντας τα τοπικά νήματα για να πετύχει τον απαραίτητο παραλληλισμό. Φυσικά, είναι μια λύση για γρήγορες δοκιμές και πειραματισμό και δεν συνίσταται για περιβάλλοντα παραγωγής.

4.5.1 Κύκλος ζωής μιας εφαρμογής Spark

Ο κύκλος ζωής μιας εφαρμογής στο Spark στο επίπεδο της υποδομής ξεκινάει με την υποβολή ενός προμεταγλωττισμένου JAR από το χρήστη στον cluster manager - έστω σε λειτουργία cluster mode. Με την υποβολή αυτή ζητάμε πόρους αυστηρά μόνο για το spark driver. Ο cluster manager κάνει δεκτή την αίτηση μας και τοποθετεί τον driver σε έναν από τους worker κόμβους.

Στη συνέχεια, ο κώδικας της εφαρμογής ξεκινάει να εκτελείται και ο driver, μέσω του SparkSession object, το οποίο πρέπει να έχει δηλωθεί, επικοινωνεί με τον cluster manager και αιτεί την εκκίνηση των executors σε όλη τη συστάδα, τους οποίους και θα διατηρήσει καθόλη τη διάρκεια της εφαρμογής. Ο αριθμός των executors που θα εκκινηθούν και οι πόροι που θα χρησιμοποιούν έχουν καθοριστεί από το χρήστη είτε μέσα από των κώδικα της εφαρμογής ή μέσω από ορίσματα γραμμής εντολών κατά την υποβολή της εφαρμογής. Ο cluster manager, δεδομένου ότι οι συγκεκριμένες αιτήσεις μπορούν να καλυφθούν από τους διαθέσιμους πόρους της συστάδας, εκκινεί τους executors και ενημερώνει το driver για την τοποθεσία τους. Πλέον η εκτέλεση της εφαρμογής μπορεί να ξεκινήσει.

Κατά τη διάρκεια της εκτέλεσης της εφαρμογής, ο driver και οι executors επικοινωνούν διαρκώς, είτε πρόκειται για μετακίνηση δεδομένων μεταξύ των executors ή για λήψη tasks από τον driver και αναφορά σε αυτόν της προόδου και τη καταστασής τους. Μόλις η εκτέλεση ολοκληρωθεί, ο spark driver πραγματοποιεί έξοδο με επιτυχία ή σφάλμα, επιστρέφει τον έλεγχο στο πρόγραμμα χρήστη και στη συνέχεια ο



Σχήμα 4.7: Εκτέλεση μιας εφαρμογής Spark σε συστάδα υπολογιστών

cluster manager απενεργοποιεί τον ίδιο και τους σχετιζόμενους με αυτόν executors και απελευθερώνει τους αντίστοιχους πόρους.

Εντός του Spark, η εφαρμογή ξεκινάει από το SparkSession και το SparkContext, το οποίο αναπαριστά τη σύνδεση με την υποκείμενη συστάδα, επιτρέπει την εκτέλεση κώδικα σε αυτόν και τη δημιουργία RDDs. Στη συνέχεια, ορίζονται οι απαραίτητοι λογικοί μετασχηματισμοί και μία ενέργεια πυροδοτεί την μετατροπή τους σε φυσικό πλάνο και την εκτέλεσή τους.

Κάθε μία ενέργεια αντιστοιχίζεται σε μία Spark δουλειά (job), ενώ πολλαπλές δουλειές εκτελούνται σειριακά. Κάθε δουλειά διασπάται σε στάδια (stages) ανάλογα το πόσοι μετασχηματισμοί που προκαλούν ευρείες εξαρτήσεις υπάρχουν. Η μικρότερη μονάδα υπολογισμού για το Spark είναι η εργασία (task), η οποία αντιστοιχεί σε ένα συνδυασμό από block δεδομένων και μια σειρά μετασχηματισμών επί αυτών. Ο αριθμός των tasks είναι αναλογος του αριθμου των partitions των δεδομένων. Κάθε stage αποτελείται από tasks που μπορούν να εκτελεστούν παράλληλα σε πολλαπλά μηχανήματα. Φυσικά, εντός των stages το Spark χρησιμοποιεί τη λογική σωλήνωση για συνδυασμό εντός-της-μνήμης όσο των δυνατών περισσότερων τύπου-map μετασχηματισμών.

Σε κάθε ανάμιξη δεδομένων, τα map tasks κατά την εκτέλεση τους γράφουν τα ενδιάμεσα δεδομένα προς ανάμιξη στο τοπικό σύστημα αρχείων. Έπειτα, στη φάση ομαδοποίησης/συνάθροισης/ένωσης, τα reduce tasks αναλαμβάνουν να φέρουν τα κατάλληλα δεδομένα από το σωστό αρχείο ανάμιξης και να εκτελέσουν τον υπολογισμό που τους έχει ανατεθεί.

4.6 Δυναμική Ανάθεση Εργασιών

Η προκαθορισμένη διαμόρφωση του Spark ακολουθεί μία στατική προσέγγιση στην ανάθεση των πόρων, σύμφωνα με την οποία μία εφαρμογή θα δεσμεύσει εξαρχής ένα σύνολο υπολογιστικών πυρήνων και μνήμης και θα τα διατηρήσει μέχρι την ολοκλήρωσή της, ανεξαρτήτως του πόσο τα χρησιμοποιεί στην πραγματικότητα κάθε στιγμή. Ωστόσο, με αυτό τον τρόπο οδηγούμαστε σε σπατάλη πόρων και μείωση της δυνατότητας για πολυχρησία μίας Spark συστάδας.

Η Δυναμική Ανάθεση Πόρων, ή αλλιώς Ελαστική Κλιμακωσιμότητα (Dynamic Resource Allocation - Elastic Scaling), είναι ένα χαρακτηριστικό του Spark που επιτρέπει την αυξομοίωση των διαθέσιμων πόρων μιας εφαρμογής ανάλογα το φόρτο εργασίας. Όταν οι executors είναι αδρανείς απομακρύνονται, οι πόροι τους επιστρέφονται στον cluster manager και γίνονται εκ νέου διαθέσιμοι για άλλες εφαρμογές/χρήστες της συστάδας. Συμπληρωματικά, όταν υπαρχουν εκκρεμή tasks, πραγματοποιείται αίτηση πόρων για δημιουργία νέων executors.

Η υπεύθυνη κλάση για τη δυναμική ανάθεση είναι η ExecutorAllocationManager, η οποία αρχικοποιείται ως μέρος της δημιουργίας του SparkContext (μαζί με, μεταξύ άλλων, τα DAGScheduler, TaskScheduler, BlockManager, LiveListenerBus). Με την αρχικοποίησή του, δηλώνει τον ExecutorAllocationListener στο LiveListenerBus, το οποίο είναι ένα κανάλι στο οποίο δηλώνονται γεγονότα σχετικά με μία Spark εφαρμογή. Χρησιμοποιώντας τον συγκεκριμένο Spark listener, ο ExecutorAllocationManager μπορεί να παρακολουθεί συγκεκριμένα γεγονότα που δημιουργούνται από τον χρονοπρογραμματιστή TaskScheduler (π.χ. onStageSubmitted, onStageCompleted, onTaskStart, onTaskEnd, onExecutorAdded, and onExecutorRemoved) και με βάση αυτά να παίρνει αποφάσεις για προσθήκη ή αφαίρεση executors. Ανάλογα την απόφαση του, για επικοινωνία με τον υποκείμενο cluster manager χρησιμοποιεί τη διεπαφή ExecutorAllocationClient.

Η απόλυτα ακριβής πρόβλεψη για το πότε ένας executor πρόκειται να αναλάβει ένα task και το βέλτιστο αριθμό executors δεν είναι δυνατή. Συνεπώς, ο ExecutorAllocationManager διατηρεί ένα κινητό στόχο για τον αριθμό, ο οποίος συγχρονίζεται τακτικά με τον cluster manager, σχετίζεται με τον αριθμό των εκτελούμενων και εκκρεμών tasks και στηρίζεται σε μια σειρά από ευρετικές.

Σχετικά με την Πολιτική Αίτησης νέων executors (Request Policy), βασική προϋπόθεση είναι να υπάρχουν εκκρεμή tasks στην ουρά του χρονοπρογραμματιστή. Συγκεκριμένα, η αίτηση πυροδοτείται όταν υπάρχουν tasks στην ουρά για πάνω από spark.dynamicAllocation.schedulerBacklogTimeout δευτερόλεπτα και στη συνέχεια πυροδοτείται ξανά, όσο συνεχίζουν να υπάρχουν tasks στην ουρά, κάθε spark.dynamicAllocation.sustainedSchedulerBacklogTimeout δευτερόλεπτα. Την πρώτη φορά, δημιουργείται αίτηση για έναν μόνο executor, ώστε σε περίπτωση που χρειαζόμαστε απλά μερικούς executors να μην σπαταλήσουμε χρόνο και ενέργεια στη δέσμευση και αποδέσμευσή τους. Έπειτα, όσο η ουρά των tasks παραμένει γεμάτη, ο αριθμός των αιτούμενων executors γίνεται γεωμετρικά μεγαλύτερος (2, 4, 8, ...), για περιπτώσεις όπου οι ανάγκες μιας εφαρμογής είναι πραγματικά μεγάλες και υπάρχουν άφθονοι

διαθέσιμοι πόροι.

Όσον αφορά την Πολιτική Αφαίρεσης (Remove Policy), ένας executor αφαιρείται όταν είναι αδρανής για περισσότερο από spark.dynamicAllocation.executorIdleTimeout δευτερόλεπτα. Η λειτουργία αυτή μπορεί να προκαλέσει προβλήματα αν ένας executor έχει cached δεδομένα στην κύρια μνήμη του. Για να μη χαθούν τα συγκεκριμένα δεδομένα, η επιλογή spark.dynamicAllocation.cachedExecutorIdleTimeout ορίζει ως μέγιστο χρονικό διάστημα που μπορεί ένας τέτοιος executor να είναι αδρανής το άπειρο, αλλά παράλληλα δίνει τη δυνατότητα να θέσουμε ένα άνω όριο στο διάστημα αυτό.

Ένα επιπλέον ζήτημα που προκύπτει με τη δυναμική ανάθεση πόρων είναι αυτό των ενδιάμεσων δεδομένων. Πλέον, όταν ένας executor αποσύρεται και μαζί του απελευθερώνονται η μνήμη και οι πυρήνες που χρησιμοποιούσε, η εφαρμογή συνεχίζει να εκτελείται. Όπως έχει ήδη αναφερθεί όμως, πριν από κάθε ανάμιξη, οι executors αποθηκεύουν τα ενδιάμεσα δεδομένα τους στο τοπικό σύστημα αρχείων και έπειτα δρουν ως εξυπηρετητές σε όποιον θέλει να τα προσπελάσει. Είναι πολύ πιθανό κάποιος executor να θέλει να προσπελάσει τα δεδομένα ενός executor που δεν υπάρχει πλέον και, εφόσον δεν θα τα βρει, θα αναγκαστεί να τα επανυπολογίσει.

Συνεπώς, είναι απαραίτητο να αποδεσμευθεί η αποθήκευση της κατάστασης ενός executor από τον ίδιο. Αυτό επιτυχάνεται μέσω της Εξωτερικής Υπηρεσίας Ανάμιξης (ExternalShuffleService). Πρόκειται για μια μακράς-διαρκείας διεργασία, η οποία τρέχει σε κάθε worker κόμβο της συστάδας, ανεξαρτήτως εφαρμογών και executors, και δρα ως ο εξυπηρετητής για την υποστήριξη των τοπικών δεδομένων, πέρα από τον κύκλο ζωής του executor που τα παρήγαγε.

Για την ενεργοποίηση του external shuffle service αλλά και γενικότερα της δυνατότητας για dynamic resource allocation είναι απαραίτητο οι επιλογές spark.dynamicAllocation.enabled και spark.shuffle.service.enabled (ή η spark.dynamicAllocation.shuffleTracking.enabled, για διατήρηση των executors που έχουν shuffle δεδομένα) να είναι true.

Κεφάλαιο 5

Υλοποίηση και Πειράματα

Στο κεφάλαιο αυτό θα αναλυθεί διεξοδικά το σύστημα που υλοποιήθηκε στο AWS Cloud και θα πραγματοποιηθούν μερικά πειράματα για την εξαγωγή γραφικών και αξιολόγηση της δυναμικής ανάθεσης εργασιών στο Spark.

5.1 Το προτεινόμενο σύστημα

Το προτεινόμενο σύστημα θα βασιστεί εξ' ολοκλήρου στην 'Υποδομή ως Κώδικας'. Ο ορισμός και η ανάπτυξη της υποδομής στο cloud θα γίνει μέσω του Terraform, ενώ η αρχικοποίηση και βασική διαμόρφωση των μηχανημάτων θα γίνει μέσω του cloud-init. Για τη διαφοροποίηση τους και τη λήψη των διαφορετικών ρόλων, θα χρησιμοποιηθούν docker containers και συγκεκριμένα για το spark cluster, θα υλοποιηθεί ένας single-container-per-host cluster, δηλαδή θα υπάρχει μόνο ο spark master/ένας spark worker ανά μηχάνημα.

Η αρχιτεκτονική του spark cluster θα βασιστεί στη λειτουργία του spark σε client mode. Συνεπώς, μαζί με τον κόμβο spark-master και το σύνολο των spark-worker κόμβων, θα αναπτυχθεί και ένας spark-gateway κόμβος από τον οποίο θα υποβάλλονται οι διάφορες εφαρμογές.

Για την απομακρυσμένη αποθήκευση του terraform state θα χρησιμοποιηθεί ένας Amazon S3 bucket, ο οποίος θα αναπτυχθεί επίσης από το Terraform, ανεξάρτητα από την προαναφερθείσα υποδομή. Ταυτόχρονα, ο συγκεκριμένος S3 bucket θα χρησιμοποιηθεί για την κατανεμημένη αποθήκευση αρχείων εισόδου/εξόδου των εφαρμογών που θα υποβάλει ο χρήστης στον spark cluster, ως εναλλακτική προσέγγιση της χρήσης ενός HDFS cluster.

Μία ακόμη υπηρεσία του AWS Cloud που θα χρησιμοποιηθεί είναι το Elastic Block Storage. Πρόκειται για μια κλιμακώσιμη, υψηλής διαθεσιμότητας και χαμηλής καθυστέρησης λύση για μόνιμη αποθήκευση επιπέδου block, βασισμένη σε ένα κατανεμημένο δίκτυο αποθηκευτικών μέσων στην περιοχή διαθεσιμότητας των κόμβων. Με την χρήση του AWS EBS για τις αποθηκευτικές ανάγκες των μηχανημάτων, αποσυνδέεται η αποθήκευση από το ίδιο το AWS EC2 instance, επιτρέποντας δυναμική

σύνδεση, αποσύνδεση και κλιμάκωση των όγκων δεδομένων, με κύκλο ζωής ανεξάρτητη από αυτό.

Για τον έλεγχο της υποδομής θα χρησιμοποιηθεί το σύστημα Prometheus. Το Prometheus, έργο υπό την αιγίδα του Cloud Native Computing Foundation (CNCF), είναι ένα σύστημα παρακολούθησης και ειδοποιήσεων βασισμένο σε μια βάση δεδομένων χρονοσειρών όπου αποθηκεύονται μετρικές πραγματικού χρόνου και επιτρέπει τη σύνθεση ευέλικτων ερωτημάτων επί αυτών. Οι μετρικές συλλέγονται από ένα σύνολο στόχων που ορίζονται κατά τη διαμόρφωσή του (μέσω ενός αρχείου YAML) και πραγματοποιούνται σύμφωνα με το pull μοντέλο, με τακτές HTTP κλήσεις προς αυτούς. Επιπλέον, για πιο φιλική προς το χρήστη εποπτία της υποδομής και δημιουργία δυναμικών και διαδραστικών γραφημάτων από τις παραγόμενες μετρήσεις του Prometheus, θα χρησιμοποιθεί το σύστημα οπτικοποίησης Grafana. Τα δύο αυτά συστήματα θα αναπτυχθούν επίσης με τη χρήση docker containers.

5.2 Υλοποίηση

Η υλοποίηση του προτεινόμενου συστήματος ακολουθεί συγκεκριμένα βήματα, τα οποία θα αναλυθούν στη συνέχεια.

5.2.1 Δημιουργία docker images

Αρχικά, στο φάκελο docker-images του πηγαίου κώδικα υπάρχουν οι υποφάκελοι με τα Dockerfile και τα απαραίτητα αρχεία για τους διαφορετικούς ρόλους στον cluster. Όσον αφορά το Spark δημιουργούνται 3 διαφορετικά images. Για βελτιστοποίηση των images και καλύτερη διαχείρηση και αναγνωσιμότητα δημιουργείται ένα βασικό spark-base-image που ουσιαστικά δίνει όλες τις spark δυνατότητες σε ένα container και στη συνέχεια, έχοντας ως βάση αυτό δημιουργούνται τα spark-master-image και spark-worker-image. Η διαφορά των δύο τελευταίων images εντοπίζεται στο σημείο διαμόρφωσης της λειτουργίας του προκύπτοντα container με την εντολή ENTRYPOINT. Εκεί, θα εκτελεστεί ένα ειδικό script που θα έχει αντιγραφεί προηγουμένως εντός του image και θα τρέχει το εκτελέσιμο spark-class με την class org.apache.spark.deploy.master.Master ή org.apache.spark.deploy.worker.Worker.

Αξίζει να σταθούμε στη δημιουργία του spark-base-image. Έπειτα από τα πρώτα βασικά βήματα (εγκατάσταση openjdk-8-jdk, scala-2.12.12, spark-3.0.1-bin-hadoop-3.2), είναι απαραίτητο να εγκαταστήσουμε δύο επιπλέον jars σε αυτά που συμπεριλαμβάνονται στο spark (φάκελος ${SPARK_HOME}/spark/jars/) για να μπορούμε να έχουμε πρόσβαση στο AWS S3. Αυτά είναι τα hadoop-aws-3.2.0.jar και aws-java-sdk-bundle-1.11.375.jar (οι εκδόσεις αντιστοιχούν σε αυτές των spark και hadoop και μπορούν να βρεθούν στο αποθετήριο www.mvnrepositoy.com).

Ακόμα, για να εξειδικεύσουμε τη λειτουργία του Spark, θα αντιγράψουμε στο φάκελο ${SPARK_HOME}/spark/conf/ εντός του image τα αρχεία spark-defaults-.conf και metrics.properties, τα οποία είναι βασισμένα στα αντίστοιχα .template του ίδιου φακέλου. Το πρώτο από αυτά περιλαμβάνει κάποιες επιλογές που επιτρέπουν

την τροποποιήση της προκαθορισμένης λειτουργίας του Spark και το χρησιμοποιούμε για να θέσουμε τις επιλογές spark.master.rest.enabled, spark.shuffle.service.enabled και spark.ui.prometheus.enabled σε true για την ενεργοποίηση του master REST endpoint (για δυνατότητα υποβολής εφαρμογών στο master μέσω REST API), του external shuffle service και της παραγωγής μετρικών σε μορφή που αναγνωρίζει το Prometheus. Το δεύτερο αρχείο χρησιμοποιείται επίσης για την εξαγωγή μετρικών για το Prometheus και μεταξύ άλλων ορίζει τα κατάλληλα endpoints για την προσπέλασή τους.

Το prometheus-image δημιουργείται για την αντιγραφή του αρχείου prometheus-aws.yml εντός του image ως prometheus.yml στο φάκελο /etc/prometheus/. Το αρχείο αυτό είναι η μόνη διαφοροποίηση του παραγόμενου image από το official (prom/prometheus στο dockerhub), αλλά είναι αυτό στο οποίο στηρίζεται η λειτουργία του όσο αναφορά το επίπεδο της εφαρμογής. Συγκεκριμένα, στο αρχείο αυτό δηλώνονται τα scrape configurations, τα οποία ορίζουν τους στόχους του εργαλείου και το πως θα τους ανακαλύψει, καθώς και διάφοροι παράμετροι όπως το διάστημα μεταξύ των ελέγχων των στόχων (scrape interval).

Τέλος, το grafana-image επίσης έχει απλή δημιουργία και χρησιμοποιείται για να ορίσουμε την πηγή δεδομένων και να συμπεριλάβουμε ένα εξειδικευμένο πίνακα (dashboard) για την οπτικοποίηση των μετρήσεων (με αντιγραφή των φακέλων provisioning και dashboards στα paths /etc/grafana/provisioning/ και /var/lib/grafana/dashboards/ αντίστοιχα εντός του image).

5.2.2 Δημιουργία VPC

Αρχικά, μέσω του Terraform και του resource 'aws_vpc', αναπτύσσεται στο EC2 και τη γεωγραφική περιοχή eu-central-1 το εικονικό δίκτυο VPC main, με τα δημόσια και ιδιωτικά υποδίκτυα main-public-{1-3} και main-private-{1-3} (resource 'aws_subnet') για την τοποθέτηση των μηχανημάτων με διαφορετικούς ρόλους. Για μεγαλύτερη ανοχή σε σφάλματα, ένα δημόσιο και ιδιωτικό υποδίκτυο τοποθετούνται σε κάθε ζώνη διαθεσιμότητας (availability zone). Οι ζώνες διαθεσιμότητας ανακαλύπτονται δυναμικά με τη χρήση ενός terraform data source.

Για ευελιξία, πειραματισμό και εξερεύνηση περισσότερων δυνατοτήτων του εργαλείου, για τη δημιουργία του συγκεκριμένου εικονικού δικτύου αναπτύχθηκε ένα Terraform module. Το module εξειδικεύεται από την είσοδο του χρήστη και διαθέτει παραμέτρους για τη γεωγραφική περιοχή AWS region, το σχήμα διευθυνσιοδότησης του δικτύου και όλων των υποδικτύων του και τις επιλογές για τον τρόπο μίσθωσης των εικονικών μηχανών, την υποστήριξη DNS και τη σύνδεση με το Amazon EC2-Classic κοινόχρηστο δίκτυο. Σε αρχικό στάδιο, το security group (εικονικό firewall για τον έλεγχο της εισερχόμενης και εξερχόμενης κίνσης) που χρησιμοποείται απλώς επιτρέπει την σύνδεση μέσω SSH και το βασικό έλεγχο διαθεσιμότητας ping. Για τη χρήση του, το module εκθέτει (output) τα μοναδικά αναγνωριστικά του δικτύου, των υποδικτύων και του security group.

5.2.3 Δημιουργία security groups

Το κάθε μηχάνημα, ανάλογα το ρόλο που έχει, έχει διαφορετικές απαιτήσεις ασφάλειας συνεπώς είναι αναγκαίο να δημιουργηθεί για κάθε τύπο μηχανήματος ένα διαφορετικό security group το οποίο, σε συνδυασμό με αυτό του υποκείμενου vpc, θα εκθέτει συγκεκριμένα ports και θα ελέγχει κατάλληλα την εισερχόμενη και εξερχόμενη κίνηση.

Η δημιουργία security group μέσω του Terraform γίνεται με χρήση του resource σαψaws_security_group. Στο resource block ορίζονται το όνομα του, μία σύντομη περιγραφή και δημιουργείται η απαραίτητη συσχέτιση με το main vpc μέσω του μοναδικού αναγνωριστικού αυτού. Η εισερχόμενη και εξερχόμενη κίνηση ελέγχονται μέσω των ingress και egress blocks, τα οποία διαμορφώνουν τους κανόνες ασφαλείας που θα ακολουθηθούν. Για κάθε κανόνα ορίζεται ένα block που περιλαμβάνει παραμέτρους για τα επιτρεπόμενα πρωτόκολλα, τις επιτρεπόμενες ip διευθύνσεις και τα ανοιχτά ports.

Στον Πίνακα 5.1, φαίνεται το σύνολο των ingress rules για όλα τα security groups, μαζί με τη χρησιμότητα του κάθε κανόνα. Η επικοινωνία μεταξύ του spark master και των workers πραγματοποιείται μέσω remote procedure call (RPC), το οποίο χρησιμοποιεί HTTP calls για την αποστολή JSON μηνυμάτων. Προφανώς και η προσπέλαση των web UIs γίνεται μέσω HTTP, συνεπώς το μοναδικό πρωτόκολλο που θα επιτραπεί είναι το tcp. Οι επιτρεπόμενες ip θα είναι οι ιδιωτικές ip που θα προκύψουν από το σχήμα διευθυνσιοδότησης του vpc, ενώ μόνο για τα web UIs των public instances θα επιτραπούν όλες οι ip. Όσον αφορά την εξερχόμενη κίνηση, θα είναι χωρίς περιορισμούς.

5.2.4 Δημιουργία και ανάθεση IAM roles

Το AWS Identity and Access Management (IAM) είναι μια υπηρεσία του AWS cloud που επιτρέπει λεπτόκοκκο έλεγχο στην πρόσβαση ανεπτυγμένων υπηρεσιών και πόρων από συστήματα και χρήστες. Δηλώνοντας ένα aws_iam_role_policy, ορίζεται ένα σύνολο υπηρεσιών, οι επιτρεπόμενες ενέργειες και οι πόροι που θα μπορούν αυτές να εφαρμοστούν. Στη συνέχεια, το policy ανατίθεται σε ένα aws_iam_role, ως ένα ολοκληρωμένο πακέτο δυνατοτήτων. Τέλος, δημιουργείται ένα aws_iam_instance_profile και του ανατίθεται ο συγκεκριμένος ρόλος. Τώρα, το profile είναι έτοιμο να ανατεθεί σε χρήστες και συστήματα και είναι αυτό στο οποίο θα γίνει αναφορά κατά τη δημιουργία των instances για την προσθήκη του κατάλληλου χαρακτήρα σε αυτά.

Θα χρειαστεί να δημιουργηθούν IAM roles για δύο δυνατότητες της υποδομής. Για την προσπέλαση του S3 bucket είναι απαραίτητο όλα τα spark instances να έχουν αναφορά στο 's3-bucket-access-role-instanceprofile', με το οποίο θα τους επιτρέπεται η πραγματοποίηση όλων των δυνατών ενεργειών επί του ίδιου του S3 bucket αλλά και των περιεχόμενων αυτού. Ο δεύτερος ρόλος αφορά τη δυνατότητα του prometheus να ανακαλύπτει δυναμικά τους στόχους του, στη συγκεκριμένη περίπτωση χρησιμοποιώντας την 'ec2_sd_configs' διαμόρφωση για service discovery στο AWS EC2 με βάση την '__meta_ec2_tag_Name' ετικέτα, δηλαδή το όνομα που θα δωθεί στα

ROLE	PORTS	PURPOSE
spark-master	7077	SPARK_MASTER_PORT
	8080	spark.master.ui.port
	6066	spark.master.rest.port
spark-worker	44444	SPARK_WORKER_PORT
	7337	spark.shuffle.service.port
	8081	spark.worker.ui.port
	33300-33315	spark.blockManager.port
spark-gateway	22200-22215	spark.driver.port
	55500-55515	spark.driver.blockmanager.port
	33300-33315	spark.blockManager.port
	4044-4055	spark.ui.port
prom-graf	9090	PROMETHEUS_UI_PORT
	3000	GRAFANA_UI_PORT

Πίνακας 5.1: Κανόνες εισερχόμενης κίνησης

instances. Μέσω των IAM roles μπορούμε να αναπτύσουμε την υποδομή μας αγνοώντας τις λεπτομέρειες σχετικά με την πρόσβαση των διαφόρων υπηρεσιών. Χωρίς αυτούς, κάθε μηχάνημα (ή χρήστης) που θα επιχειρούσε να χρησιμοποιήσει μια υπηρεσία του aws cloud θα έπρεπε να παρέχει τα κατάλληλα AWS_ACCESS_KEY_ID και AWS_SECRET_KEY, ενώ αυτά θα έπρεπε επίσης να είναι ορατά από τους διάφορους πιθανούς containers. Με τους IAM roles, η διαδικασία αυτή είναι πολύ απλούστερη, αφού, πλέον, όλες οι βιβλιοθήκες και οι εφαρμογές είναι αρκετά ανεπτυγμένες και ενημερωμένες ώστε είναι ικανές να 'υποθέσουν' τους ρόλους και να ανακαλύψουν τα συγκεκριμένα πιστοποιητικά, τα οποία για λόγους ασφαλείας ανανεώνονται αυτόματα ανά τακτά χρονικά διαστήματα.

5.2.5 Δημιουργία των EC2 instances

Σε αυτό το σημείο, με την ολοκλήρωση της δημιουργίας των docker images, του υποκείμενου vpc, των κατάλληλων security groups και των απαραίτητων IAM roles είναι δυνατή η ανάπτυξη όλων των μηχανημάτων με τη χρήση του resource 'aws_instance'.

Βασικές παράμετροι

Εντός του resource block, αρχικά θα καθοριστεί το κατάλληλο Amazon Machine Instance (AMI), ως η εικόνα του λειτουργικού συστήματος που θα χρησιμοποιηθεί. Για μεγαλύτερη ευελιξία και ευκολία στη χρήση, θα χρησιμοποιηθεί μία μεταβλητή terraform τύπου map(string), η οποία ως προκαθορισμένη τιμή θα έχει μία αντιστοίχιση μεταξύ των aws regions της Ευρώπης και των κατάλληλων AWS EC2 AMIs. Όλα AMIs θα είναι βασισμένα στα Ubuntu 20.04 (Focal Fossa), αρχιτεκτονικής amd64 και για instances τύπου hvm:ebs-ssd. Για την επιλογή της σωστής αντιστοίχισης θα χρησιμοποιηθεί η συνάρτηση lookup με κλειδί την aws region του χρήστη.

Έπειτα, με την παράμετρο instance_type επιλέγεται ο τύπος του μηχανήματος, με την παράμετρο iam_instance_profile γίνεται η ανάθεση του κατάλληλου IAM ρόλου και με την παράμετρο vpc_security_group_ids ορίζεται μία λίστα από μοναδικά αναγνωριστικά από security groups των οποίων οι κανόνες θα συνδυαστούν για την παραγωγή του τελικού. Στην συγκεκριμένη περίπτωση θα αποτελείται από το βασικό security group του υποκείμενου main vpc και από το κατάλληλο security group που δημιουργήθηκε για το ρόλο που θα αναλάβει το μηχάνημα.

Χρήση του count meta-argument

Στο σημείο αυτό είναι αναγκαία η διαφοροποίηση μεταξύ των μηχανημάτων τύπου spark-worker και των υπολοίπων. Όπως είναι γνωστό, η προκαθορισμένη χρήση του terraform ορίζει την ανάπτυξη ενός και μόνο ενός πραγματικού αντικειμένου της υποδομής ανά resource block. Δεδομένου ότι οι spark workers είναι σχεδόν πανομοιότυπα μηχανήματα, θα ήταν χρήσιμο να ομαδοποιηθούν εν μέρει, ώστε να μην χρειάζεται η συγγραφή ενός ξεχωριστού resource block για κάθε αντικείμενο. Το παραπάνω έχει ακόμη περισσότερη σημασία λόγω του γεγονότος ότι ο αριθμός των spark workers στην υποδομή που αναπτύσσεται παρέχεται από το χρήστη και άρα είναι μεταβλητός και μη προβλέψιμος.

Για το λόγο αυτό, θα χρησιμοποιηθεί το Terraform meta-argument count. Το count ορίζεται από την HCL και μπορεί να εφαρμοστεί σε block οποιουδήποτε τύπου πόρου. Δέχεται έναν ακέραιο αριθμό και δημιουργεί τόσα instances του συγκεκριμένου πόρου. Είναι σημαντικό να τονιστεί ότι κάθε τέτοιο instance είναι ανεξάρτητο από τα υπόλοιπα και αντιπροσωπεύει ένα διακριτό πραγματικό αντικείμενο της υποδομής, με ξεχωριστό κύκλο ζωής σε κάθε εφαρμογή της διαμόρφωσης. Με την προσθήκη του meta-argument count σε ένα resource block, δημιουργείται ένα επιπλέον count αντικείμενο το οποίο είναι διαθέσιμο στις εκφράσεις για τη διαμόρφωση των διαφορετικών instances. Μοναδική του ιδιότητα είναι η count.index για την προσπέλαση του κάθε instance (αρχίζοντας από το 0, <TYPE>.<NAME>[INDEX] για ένα instance, <TYPE>.<NAME> για ολόκληρο το block). Στη συνέχεια, οι πόροι που θα αναφέρονται σε μηχανήματα τύπου spark-worker θα έχουν το συγκεκριμένο meta-argument, το οποίο θα χρησιμοποιείται για την 1-προς-1 συσχέτιση πόρων και του αντίστοιχου μηχανήματος.

Επιλογή subnet

Η τοποθέτηση των μηχανημάτων στα υποδίκτυα του vpc θα πραγματοποιηθεί με την παράμετρο subnet_id και θα είναι σύμφωνη με τις ανάγκες για εξωτερική προσπέλαση, υψηλή διαθεσιμότητα και ανοχή σε πιθανά σφάλματα που επηρεάζουν ολόκληρες περιοχές διαθεσιμότητας. Στα δημόσια υποδίκτυα main-pubic-{1-3} θα τοποθετηθούν οι κόμβοι spark-master, spark-gateway και prom-graf-server, ένας στο καθένα. Θα τοποθετηθούν στα δημόσια υποδίκτυα γιατί είναι απαραίτητο να μπορούν να προσπελαστούν, είτε για την εποπτεία του cluster ή για την υποβολή εφαρμογών. Οι spark-worker κόμβοι θα τοποθετηθούν επίσης στα δημόσια υποδίκτυα με round-robin τρόπο (με χρήση της συνάρτησης element για επιλογή στοιχείου από λίστα - εδώ τη λίστα υποδικτύων που εκθέτει το vpc module).

Έλεγχος ταυτότητας

Η πρόσβαση στα instances και ο έλεγχος ταυτότητας θα είναι passwordless και θα πραγματοποιείται μέσω RSA ζευγών δημοσίων και ιδιωτικών κλειδιών. Πριν την ανάπτυξη της υποδομής, θα πρέπει να εκτελεστεί το script 'create_keypairs.sh' για τη δημιουργία του συνόλου των ζευγών κλειδιών. Στη συνέχεια, μέσω του resource 'aws_key_pair' θα δημιουργηθεί ένας πόρος για κάθε ζεύγος, όπου θα του δωθεί ένα όνομα key_name και θα δηλωθεί η τοποθεσία του δημόσιου κλειδιού του. Τέλος, στο resource block θα δημιουργηθεί η συσχέτιση με το κατάλληλο κλειδί μέσω της παραμέτρου key_name.

Διαμόρφωση μέσω του cloud-init

Ακολουθεί το σημαντικότατο βήμα της χρήσης των δεδομένων χρήστη και του cloud-init για τη διαμόρφωση των κόμβων της υποδομής σύμφωνα με τη λειτουργία τους. Όπως έχει προαναφερθεί, με την παράμετρο user_data εντός του resource_block δηλώνεται το κατάλληλο template_cloudinit_config που περιέχει όλα τα μέρη με τα rendered template files. Στο αρχείο init.cfg του φακέλου scripts βρίσκονται οι οδηγίες της βασικής διαμόρφωσης των μηχανημάτων. Στην πραγματικότητα, εκτός από τις βασικές οδηγίες package_update και package_upgrade για ενημέρωση και εγκατάσταση των νέων εκδόσεων των διάφορων πακέτων και την αλλαγή του hostname, οι υπόλοιπες οδηγίες είναι αυτές που χρειάζονται για την εγκατάσταση του docker. Αυτό είναι και ένα από τα πλεονεκτήματατα της προσέγγισης που ακολουθήθηκε, η διαμόρφωση των μηχανημάτων είναι εξαιρετικά απλή εφόσον η λειτουργικότητα της μηχανής δίνεται από τους containers που θα αναπτυθούν σε αυτή.

Με το τέλος της εγκατάστασης του docker, θα σηκωθεί ο κατάλληλος container. Αξίζει να τονιστεί ότι για τη διαμόρφωση όλων των μηχανημάτων χρησιμοποιείται το ίδιο αρχείο init.cfg. Αυτό είναι δυνατό λόγω της δυνατότητας που προσφέρει το cloud-init για παραμετροποίηση και το template_file data source για αντικατάσταση των συγκεκριμένων μεταβλητών. Για την εξειδίκευση της χρήσης του init.cfg θα χρη-

σιμοποιηθούν δύο μεταβλητές. Η πρώτη είναι η HOSTNAME η οποία προφανώς θα χρησιμοποιείται για τον ορισμό του ονόματος του μηχανήματος. Η δεύτερη μεταβλητή είναι η START_CONTAINER στην οποία θα περνιέται ως string η εντολή που θα ξεκινά τον κατάλληλο container για το μηχάνημα. Με το πέρασμα της εντολής ως string, θα χρησιμοποιηθεί κατευθείαν από τη sh shell διαδικασία που ορίζει το πεδίο runcmd (ως η τελευταία εντολή ${START_CONTAINER}). Η έξοδος όλων των υποδιεργασιών της διαμόρφωσης μέσω του cloud-init θα συγκεντρωθεί στο αρχείο /var/log/cloud-init-output.log και θα ολοκληρωθεί με το τελικό μήνυμα 'The system is up after $UPTIME seconds\nEND'. Για τους διαφορετικούς ρόλους θα δημιουργηθούν ξεχωριστά template_file και template_cloudinit_config για spark-master, spark-worker, spark-gateway και prom-graf-server.

Δέσμευση αποθηκευτικού χώρου EBS

Ο root EBS αποθηκευτικός χώρος των ec2 instances έχει μέγεθος 8GB και χρησιμοποιείται από το λειτουργικό σύστημα και τα αρχεία εφαρμογών. Για τη δέσμευση επιπλέον EBS αποθηκευτικού χώρου θα χρησιμοποιηθεί το resource 'aws_ebs_volume', όπου θα προσδιοριστεί η ζώνη διαθεσιμότητας (ίδια με αυτή του αντίστοιχου μηχανήματος) και το μέγεθος και ο τύπος αποθηκευτικού μέσου. Δεδομένου ότι ο όγκος δεδομένων που προκύπτει είναι αυτοτελής, η συσχέτιση του με το μηχάνημα που μας ενδιαφέρει γίνεται μέσω του resource 'aws_volume_attachment'. Η αρχικοποίηση του νέου αποθηκευτικού χώρου και η δημιουργία συστήματος αρχείων εντός αυτού θα γίνει μέσω του αρχείου volumes.sh του φακέλου scripts. Στο αρχείο αυτό, θα χρησιμοποιηθεί το λογισμικό Linux Volume Manager (LVM), το οποίο προσφέρει ευκολότερη και αποδοτικότερη διαχείριση του νέου όγκου δεδομένων, καθώς και όσων όγκων δεδομένων πιθανώς προκύψουν μελλοντικά. Ο νέος όγκος δεδομένων προσδεθεί στη διαδρομή /data, ενώ θα προστεθεί κατάλληλη οδηγία στο αρχείο /etc/fstab/ για αυτόματη πρόσδεση του όγκου δεδομένων κάθε φορά που επανεκκινεί το μηχάνημα. Το lvm θα είναι το πρώτο από τα πακέτα που θα εγκατασταθεί μέσω του πεδίου packages στο init.cfg, ενώ το volumes.sh θα συμπεριληφθεί στη διαμόρφωση ως το δεύτερο μέρος του πολυμερούς 'template_cloudinit_config'.

Προσοχή χρειάζεται στην ονοματοδοσία του νέου όγκου δεδομένων. Το όνομα που θα δωθεί αρχικά στη συσκευή κατά την πρόσδεσή της (έστω /dev/xvdh/) διαφέρει από αυτό που θα έχει τελικά. Με την πρόσδεση, το όνομα της αλλάζει αυτόματα και της ανατίθεται το πρώτο διαθέσιμο από την απαρίθμηση /dev/nvme[0-26]n1. Συνεπώς, στο volumes.sh και την αναζήτηση της συσκευής, θα πρέπει να χρησιμοποιηθεί το νέο όνομα /dev/nvme1n1, διαφορετικά η ανάπτυξη θα σταματήσει σε ένα ατέρμονο loop.

Για να μπορέσει να χρησιμοποιηθεί ο νέος αποθηκευτικός χώρος από τους διάφορους containers θα πρέπει να γίνει bind στα εσωτερικά directory που αποθηκεύουν δεδομένα. Στο spark, η προκαθορισμένη τοποθεσία αποθήκευσης δεδομένων είναι η /tmp και στο prometheus η /prometheus/data. Για αυτό, στην εντολή εκκίνησης των containers θα προστεθεί το '–mount type=bind,src=/data,dst={INSIDE_DIR}'.

Χρήση provisioners

Μετά και την αρχικοποίηση του επιπρόσθετου αποθηκευτικού χώρου, η υποδομή έχει σχεδόν ολοκληρωθεί. Μένουν δύο βήματα για τη βελτίωση της εμπειρίας χρήστη, τα οποία βασίζονται στη χρήση terraform provisioners. Οι provisioners είναι ένας μηχανισμός του terraform που επιτρέπει να οριστούν, στο τοπικό ή σε απομακρυσμένο μηχάνημα, ενέργειες που δεν μπορούν να περιγραφούν με δηλωτικό τρόπο. Στο πρώτο βήμα, στο resource block των instances, θα δηλωθεί ένας provisioner 'local-exec', ο οποίος θα εκτελείται τοπικά μόλις η δημιουργία του απομακρυσμένου μηχανήματος έχει ολοκληρωθεί, χωρίς απαραίτητα να είναι σε λειτουργική κατάσταση, και θα τυπώνει το όνομα και τις public και private ip αυτού στο αρχείο cluster_ips.txt.

Για το δεύτερο βήμα, θα χρησιμοποιηθεί ένα 'null_resource' resource block και εντός αυτού ένας provisioner τύπου 'remote-exec'. Ο πόρος 'νυλλ_ρεσουρσε' συμπεριφέρεται όπως ένας κανονικός πόρος, αλλά δεν κάνει τίποτα. Χρησιμοποιείται για την εκτέλεση provisioner οι οποίοι δεν σχετίζονται με κάποιο πραγματικό αντικείμενο της αναπτυσσόμενης υποδομής. Ο provisioner 'remote-exec' θα εκτελεστεί απομακρυσμένα και για αυτό χρειάζεται ένα nested 'connection' block για να πραγματοποιήσει μία σύνδεση SSH με το μηχάνημα-στόχο. Μέσω αυτού, θα ελέγχεται ανά 1 δευτερόλεπτο η τελευταία γραμμή του αρχείου /var/log/cloud-init-output.log για τη λέξη 'END', ως ένα σήμα ότι το cloud-init script ολοκληρώθηκε. Αυτό γίνεται ώστε η επιστροφή της εντολής terraform apply να μη γίνεται μόλις οι πόροι έχουν απλώς δημιουργηθεί, αλλα να καθυστερείται μέχρι η διαμόρφωση να έχει τελειώσει και η υποδομή να είναι πλήρως λειτουργική.

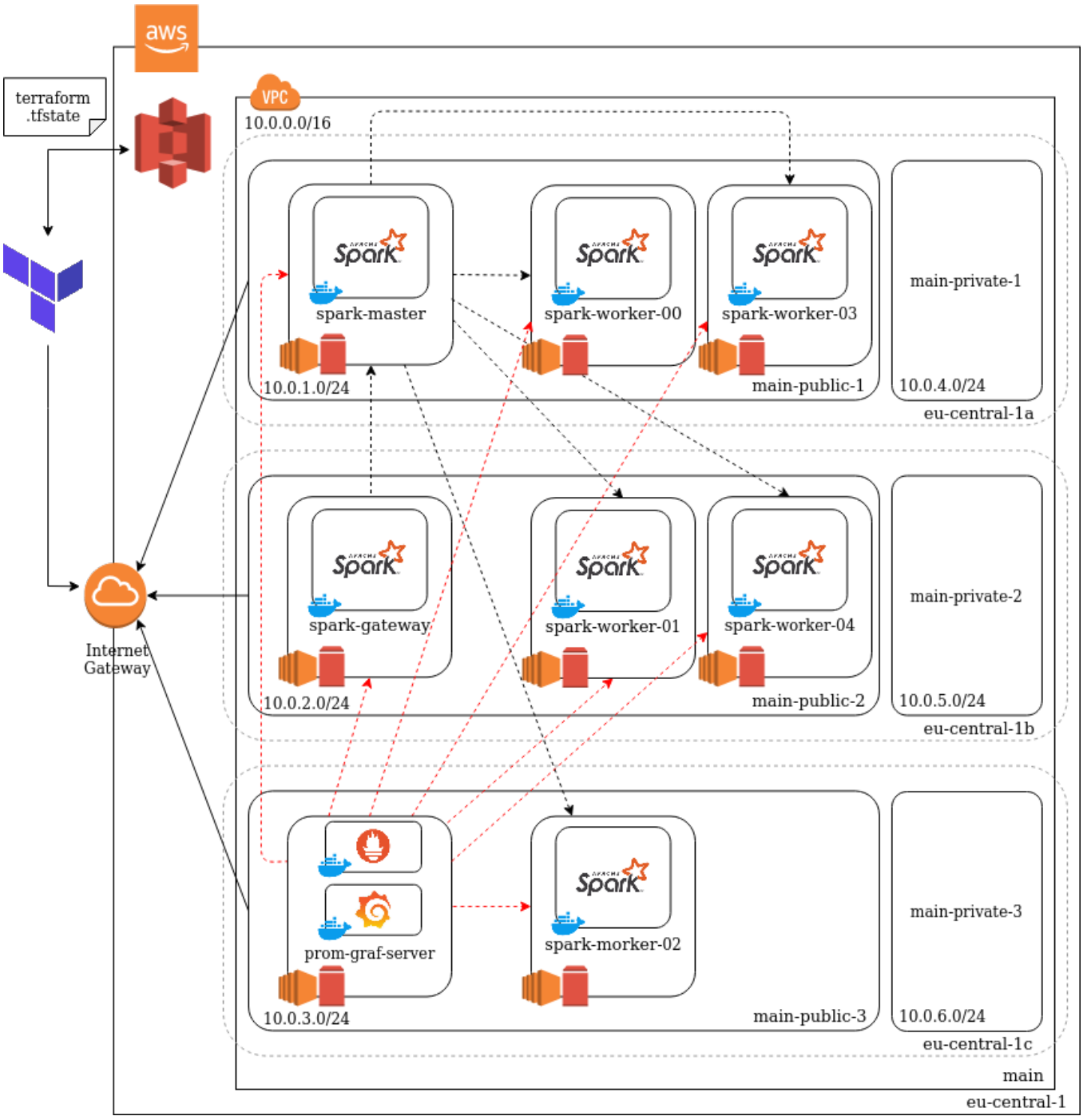
Σημειώνεται ότι και τα δύο αυτά βήματα είναι αποκλειστικά για βελτίωση της εμπειρίας χρήστη και μπορούν να παραληφθούν σε περίπτωση που ο όγκος των managed resources που προκύπτει είναι τόσο μεγάλος ώστε να ξεπερνάει το άνω όριο που θέτει το εργαλείο, ή προκύπτουν προβλήματα με την επίδοση του.

5.2.6 Δημιουργία S3 bucket και ορισμός remote backend

Το τελευταίο βήμα που απομένει είναι η δημιουργία του Amazon S3 bucket και ο ορισμός του ως το remote backend της υποδομής. Αρχικά, ξεχωριστά από τη μέχρι τώρα ανάπτυξη, στο φάκελο terraform-s3-bucket θα δημιουργηθεί μέσω του resource 'aws_s3_bucket' ένας S3 bucket, με προσοχή να έχει μοναδικό όνομα στο aws region. Επιπλέον, θα ενεργοποιηθεί μέσω της παραμέτρου versioning η διατήρηση προηγούμενων αρχείων καταστάσεων για δυνατότητα επαναφοράς κατάστασης και μέσω της παραμέτρου server_side_encryption_configuration η κρυπτογράφηση των δεδομένων του bucket.

Για τον ορισμό του bucket ως backend, στο φάκελο της κύριας υποδομής χρησιμοποιείται ένα terraform block, το οποίο είναι ένα block κώδικα για διαμόρφωση του ίδιου του εργαλείου. Εκεί, με ένα backend "s3" block, δηλώνεται το μοναδικό όνομα του bucket που θα χρησιμοποιηθεί και η πλήρης διαδρομή του αρχείου κατάστασης.

5.2.7 Σχηματική αναπαράσταση της υποδομής



Σχήμα 5.1: Η πλήρης υποδομή, έστω με 5 spark workers

5.3 Πειράματα

Στο σημείο αυτό θα πραγματοποιθεί μιας σειρά πειραμάτων που σκοπό έχουν να ανακαλύψουν το πραγματικό αντίκτυπο που έχουν στην εκτέλεση μιας εφαρμογής και την κατάσταση του spark cluster οι διαφορετικές επιλογές που προσφέρει το spark στο πλαίσιο της δυναμικής ανάθεσης πόρων μεταξύ εργασιών.

Για το σκοπό αυτό θα αναπτυχθεί μία υποδομή σύμφωνα με τις προδιαγραφές που φαίνονται στον Πίνακα 5.2. Από τους συγκεκριμένους πόρους, σε κάθε μηχάνημα θα δεσμευθεί για το spark το 75% και το υπόλοιπο θα είναι για το λειτουργικό σύστημα. Επιπλέον, κάθε spark executor θα αποτελείται από 1 επεξεργαστικό πυρήνα και 2GB κύριας μνήμης, συνεπώς ο μέγιστος αριθμός executors θα είναι 18.

ROLE	NUMBER	INSTANCE TYPE	vCPUs	MEMORY
spark-master	1	c5.large	2	4
spark-worker	6	c5.xlarge	4	8
spark-gateway	1	c5.xlarge	4	8
prom-graf-server	1	c5.large	2	4

Πίνακας 5.2: Προδιαγραφές μηχανημάτων

Το πρώτο σετ πειραμάτων αφορά τη δέσμευση πόρων. Αναλυτικότερα, θα εξεταστεί το πώς η 'επιθετικότητα' με την οποία μια εφαρμογή ζητάει πόρους από τον cluster manager επηρεάζει την συμπεριφορά της, την συμπεριφορά των πόρων της αλλά και την κατάσταση του spark cluster. Στη συνέχεια, το δεύτερο σετ πειραμάτων θα εξετάσει την αποδέσμευση πόρων και συγκεκριμένα πώς η 'προθυμία' μιας εφαρμογής να απελευθερώσει τους δεσμευμένους πόρους της επηρεάζει την εκτέλεσή της και τη σπατάλη πόρων στο spark cluster.

Για την προσομοίωση μιας εφαρμογής με έντονο επεξεργαστικό φορτίο και μεταβαλλόμενες ανάγκες για πόρους θα χρησιμοποιηθεί η εφαρμογή DynamicPi, η οποία είναι βασισμένη στην εφαρμογή SparkPi του Spark και υπολογίζει το π, αδρανοποιείται για 1 λεπτό και στη συνέχεια το επανυπολογίζει.

5.3.1 Σχετικά με τη δέσμευση πόρων

Οι παράμετροι που θα μεταβάλλονται στις εκτελέσεις της εφαρμογής DynamicPi φαίνονται στον Πίνακα 5.3. Η εκτέλεση 1 είναι η πιο επιθετική, τόσο στο μέγιστο και ελάχιστο αριθμό executors όσο και στην ταχύτητα με την οποία αιτεί νέους πόρους. Ακολουθούν οι εκτελέσεις 2 και 3, η καθεμία λιγότερο επιθετική από την προηγούμενη. Η εκτέλεση 4 αιτεί πόρους με τον ίδιο ρυθμό όπως η 2, αλλά μέσω της παραμέτρου executorAllocationRatio η μέγιστη παραλληλία μειώνεται στο μισό.

Στο Σχήμα 5.2 φαίνονται χρονοσειρές για τα μεγέθη 'Αριθμός executors', 'Αναλογία όλων των executors / Στόχος' και 'Συνολικά tasks ανά λεπτό ανά executor'

και προέρχονται από τις οπτικοποιήσεις του συστήματος Grafana. Συγκρίνοντας τα σχετικά με τον αριθμό των executors γραφήματα είναι φανερό ότι η πιο επιθετική προσέγγιση οδηγεί σε γρηγορότερη εκτέλεση. Ταυτόχρονα όμως, ο μεγαλύτερος ελάχιστος και μέγιστος αριθμός executors ενδεχομένως να προκαλεί σπατάλη πόρων και να στερεί τη δυνατότητα για αύξηση πόρων σε άλλες εφαρμογές που μοιράζονται το spark cluster.

PARAMETER	RUN 1	RUN 2	RUN 3	RUN 4
minExecutors	4	2	1	2
maxExecutors	40	16	16	16
schedulerBacklogTimeout	1	2	4	2
sustainedSchedulerBacklogTimeout	2	4	8	4
executorAllocationRatio	1	1	1	0.5

Πίνακας 5.3: Παράμετροι πρώτου σετ πειραμάτων

Με την πρώτη εκτέλεση, μαζεύονται γρήγορα πόροι, όμως ο υπερβολικά μεγάλος αριθμός maxExecutors οδηγεί σε κορεσμό, κάτι το οποίο φαίνεται και από το διάγραμμα σχετικά με το στόχο των executors, όπου ο λόγος είναι κάτω από το ένα και δείχνει ότι ο cluster δεν διαθέτει επιπλέον πόρους ώστε να ικανοποιήσει τις ανάγκες τις εφαρμογές. Αντίθετα, οι επόμενες εκτελέσεις έχουν ομαλότερη καμπύλη ζήτησης. Επιπλέον, με τον μικρότερο αριθμό minExecutors που διαθέτουν προκαλούν μικρότερη σπατάλη πόρων στο ενδιάμεσο διάστημα αδρανείας τους. Τα σχετικά με τον στόχο των executors γραφήματα αποδεικνύουν το ίδιο, μιας και ο λόγος ζήτησης είναι για μεγαλύτερο διάστημα στο 1. Οι ακίδες στα συγκεκριμένα γραφήματα προκύπτουν από τις στιγμές που το πρώτο στάδιο της εκτέλεσης έχει τελειώσει και οι επιπλέον πόροι δεν έχουν προλάβει να ελευθερωθούν.

Όσον αφορά τα διαγράμματα για το συνολικό αριθμό των tasks ανά λεπτό ανά executor, φαίνεται ότι η δυναμική ανάθεση δουλεύει ορθά και οι νέοι πόροι αναλαμβάνουν ανάλογο φόρτο, δεδομένου ότι οι καμπύλες είναι σχετικά 'παράλληλες'. Οι πόροι που προστίθενται αργότερα έχουν μικρότερο συνολικό αριθμό tasks, ενώ το μεγαλύτερο αριθμό τον έχουν οι executors που αντίστοιχουν στην ελάχιστη ποσότητα και διατηρούνται μεταξύ των διαστημάτων έντονο φόρτου, 4, 2 και 1 executors αντίστοιχα. Επιπλέον, φαίνεται ότι τα συνολικά tasks, και συνεπώς ο συνολικός φόρτος, ανά executor αυξάνονται όσο μειώνεται η επιθετικότητα με την οποία προστίθενται νέοι πόροι.

Η επίδραση της παραμέτρου executorAllocationRatio δεν είναι ιδιαιτέρως έντονη, δεδομένου ότι ο αριθμός των συνολικών executors είναι σχετικά χαμηλός, φαίνεται όμως ελαφρώς στο δεύτερο μισό των εκτέλεσων 2 και 4, όπου ο αριθμός των executors φτάνει γρηγορότερα στη μέγιστη τιμή του, παρόλο που η επιθετικότητα είναι η ίδια.

5.3.2 Σχετικά με την αποδέσμευση πόρων

Η παράμετρος που θα εξεταστεί στο δεύτερο σετ πειραμάτων είναι η executorIdleTimeout. Η εκτέλεση 1 δείχνει τη λιγότερη προθυμία να αποδεσμεύσει τους πόρους της και θέτει την συγκεκριμένη παράμετρο στα 60 δευτερόλεπτα, η οποία είναι και η προκαθορισμένη τιμή που θέτει το spark, όταν είναι ενεργοποιημένη η δυναμική ανάθεση πόρων. Η δεύτερη εκτέλεση μειώνει στο μισό την τιμή της παραμέτρου, ενώ η εκτέλεση 3 είναι η πλέον πρόθυμη και απελευθερώνει έναν executor μετά από μόλις 5 δευτερόλεπτα αδράνειας.

Στο Σχήμα 5.3 παρουσιάζονται τα αποτελέσματα των τριών εκτελέσεων σε ένα σύνολο γραφημάτων με χρονοσειρές αντίστοιχες με αυτές του πρώτου συνόλου πειραμάτων. Είναι φανερό ότι η δυναμική απροθυμία στην αποδέσμευση οδηγεί σε αύξηση της σπατάλης των πόρων spark cluster και πιθανώς να επηρεάζει και την εκτέλεση άλλων εφαρμογών. Συγκεκριμένα, από τα 'Number of Executors' διαγράμματα φαίνεται ότι παρά την πτώση του επεξεργαστικού φορτίου οι executors διατηρούνται, μέχρι σχεδόν το επόμενο κύμα εργασιών. Στην εκτέλεση 1, η μόνη στιγμή που υπάρχει μία ελαφριά μείωση είναι στη συμπλήρωση των 60 δευτερολέπτων, η μείωση αυτή όμως διατηρείται για λίγο αφού στη συνέχεια με τα νέα tasks υπάρχει εκ νέου αύξηση στη ζήτηση. Στη δεύτερη εκτέλεση, η αυξημένη προθυμία μειώνει τη σπατάλη, ενώ οι δέσμευση πόρων στην πλέον πρόθυμη εκτέλεση 3 σχεδόν ταυτίζεται με τις επεξεργαστικές ανάγκες.

Τα παραπάνω φαίνονται και στα διαγράμματα σχετικά με το στόχο των executors, όπου στα διαστήματα αδράνειας οι δεσμευμένοι πόροι είναι πολλαπλάσιοι ακόμα και των ελάχιστων απαιτήσεων. Εξαίρεση αποτελεί η εκτέλεση 3, όπου η ακίδα αντιπροσωπεύει το διάστημα μέχρι να αποδεσμευθούν η πόροι έπειτα από το τέλος του πρώτου φορτίου και στην υπόλοιπη εκτέλεση ο λόγος ζήτησης είναι στο 1. Τέλος, τα διαγράμματα για το ρυθμό εκτέλεσης tasks ανά executor δείχνουν ότι εκτός του ότι οι λιγότερο πρόθυμες εκτελέσεις συγκρατούν τους πόρους για μεγάλα διαστήματα εις βάρος άλλων εφαρμογών, ταυτόχρονα, με την αύξηση της απροθυμίας μειώνεται ο ρυθμός της συνολικής δουλειάς που εκτελούν.

Οι περιπτώσεις που φαίνεται να έχει νόημα η απροθυμία στη αποδέσμευση πόρων είναι σε περιβάλλοντα πολυμίσθωσης έντονου ανταγωνισμού, όπου η δέσμευση πόρων είναι αντίστοιχα δυσκολότερη και οι εφαρμογές ακόμα και στα διαστήματα αδρανείας τους θα θέλουν να διατηρήσουν τους πόρους τους, πιθανώς λόγων ισχυρών προθεσμιών. Επιπλέον, σε εφαρμογές που η μείωση στις ανάγκες είναι βραχυχρόνια σε σχέση με τα διαστήματα έντονου φόρτου.

(α′) Number of executors (β′) Executors/Target (γ′) Total tasks

Σχήμα 5.2: Αποτελέσματα πειραμάτων σχετικά με το request policy

(α′) Number of executors (β′) Executors/Target (γ′) Total tasks

Σχήμα 5.3: Αποτελέσματα πειραμάτων σχετικά με το remove policy

Κεφάλαιο 6

Επίλογος

Σε αυτή τη διπλωματική εργασία αρχικά παρουσιάστηκε η έννοια του υπολογιστικού νέφους ως απάντηση στις διαρκώς αυξανόμενες ανάγκες των χρηστών για πόρους. Αναλύθηκαν τα ιδιαίτερα χαρακτηριστικά των διαφορετικών μοντέλων υπηρεσιών και ανάπτυξης, καθώς και το τι προσφέρει το καθένα από αυτά, όσον αφορά την εξειδίκευση και το επίπεδο ετοιμότητας τους για χρήση υπηρεσιών. Στη συνέχεια, έγινε μια εισαγωγή στην Υποδομή ως Κώδικας και τη διαδρομή από τα πρώτα επί τούτου σενάρια ως και τα πλέον σύγχρονα εργαλεία προτυποίησης εξυπηρετητών και παροχής υποδομής.

Στο Κεφάλαιο 3 έγινε μια εκτενής αναφορά στο εργαλείο παροχής υποδομής Terraform και την προσέγγιση που ακολουθεί για την ανάπτυξη υποδομής. Παρουσιάστηκαν οι έννοιες των παρόχων και των πόρων, ως ο τρόπος που ενσωματώνει τα διάφορα συστατικά που προσφέρουν οι πάροχοι υπηρεσιών νέφους, τα διακριτά κομμάτια του πυρήνα του και ο ρόλος τους καθενός από αυτά, ενώ αναλύθηκε διεξοδικά η χρήση της Θεωρίας Γράφων για την αναπαράσταση και τροποποίηση της ανεπτυγμένης υποδομής και η χρήση της εσωτερικής κατάστασης, ως η μοναδική πηγή αλήθειας και συνέπειας. Τέλος, γίνεται αναφορά στο cloud-init και στο συνδυασμό του με το Terraform για την προσαρμογή νέων εικονικών μηχανών.

Το Κεφάλαιο 4 ξεκίνησε με το πρόβλημα των Μεγάλων Δεδομένων και των διαφορετικών προσεγγίσεων που έχουν προκύψει για την αντιμετώπιση του. Έπειτα, έγινε μια εισαγωγή στη βασική αρχιτεκτονική του, κυρίαρχου μέχρι προσφάτως, εργαλείου MapReduce, στον τρόπο λειτουργίας του αλλά και στους περιορισμούς που αυτό εισάγει. Στη συνέχεια, ακολούθησε το Spark ως η εναλλακτική, με τη χρήση της κύριας μνήμης και την οκνηρή αποτίμηση των μετασχηματισμών επί των δεδομένων, η οποία στηρίζεται στο βελτιστοποιητή Catalyst. Τέλος, εξετάστηκαν οι δυνατότητες που προσφέρει το Spark για δυναμική ανάθεση εργασιών και αναλύεται ο μηχανισμός που την υποστηρίζει.

Στο κομμάτι της υλοποίησης, το προτεινόμενο σύστημα αναπτύχθηκε στο AWS EC2 cloud με βάση το Terraform, χρησιμοποιώντας Docker containers, αρχιτεκτονική Spark client mode, παρακολούθηση από τα συστήματα Prometheus και Grafana και το AWS S3 για απομακρυσμένη αποθήκευση του Terraform state.

Συνολικά, αποκτήθηκε εξοικείωση με ένα ευρύ φάσμα εννοιών και υπηρεσιών του χώρου των Μεγάλων Δεδομένων, τόσο στο κομμάτι της υποδομής και των υπολογιστικών νεφών όσο και στο κομμάτι της ελαστικότητας στη διαχείριση πόρων. Τα διαφορετικά εργαλεία, το καθένα με το δικό του ρόλο και τις δικές του προκλήσεις, χρησιμοποιήθηκαν με τρόπο συνδυαστικό και δημιούργησαν ένα ολοκληρωμένο σύστημα.

Μελλοντική Δουλειά

Το σύστημα που αναπτύχθηκε έχει αρκετά σημεία που μπορούν να εξελιχθούν περαιτέρω και να εμπλουτιστούν με επιπλέον δυνατότητες για βελτιστοποίηση του αποτελέσματος.

Αρχικά, στο κομμάτι της προτυποποίησης θα μπορούσε να χρησιμοποιηθεί ένα εργαλείο σαν το Ansible. Το cloud-init περιορίζεται στην αρχική προτυποποίηση μιας εικονικής μηχανής και όπως φάνηκε αποτελεί κομμάτι των δεδομένων χρήστη που χρησιμοποιούνται στο πρώτο boot. Το Ansible θα έδινε τη δυνατότητα για πιο συνεχή και προηγμένο έλεγχο της διαμόρφωσης. Επιπλέον, θα μπορούσε να χρησιμοποιηθεί ένα εργαλείο Συνεχούς Ολοκλήρωσης/Συνεχούς Ανάπτυξης όπως το Jenkins, το οποίο σε συνδυασμό με τα Terraform, Docker και Ansible θα δημιουργούσε ένα ολοκληρωμένη σωλήνωση δημιουργίας και ανάπτυξης νέων εκδόσεων του λογισμικού.

Στο κομμάτι του Spark, ο cluster manager θα μπορούσε να είναι το de facto σύστημα ορχήστρωσης container, το Kubernetes. Η υποστήριξη που παρέχει το Spark για αυτό το σενάριο χρήσης είναι διαρκώς αυξανόμενη (παράδειγμα αποτελεί η προσθήκη ενός Kubernetes native χρονοπρογραμματιστή) και δημιουργεί μια υποδομή που ταιράζει καλύτερα στο σύγχρονο cloud περιβάλλον, προσφέρει πιο λεπτόκοκκο έλεγχο των εφαρμογών, βελτιώνει την ελαστικότητα και απρόσκοπτη ενσωμάτωση με συστήματα καταγραφής και παρακολούθησης. Ταυτόχρονα, είναι το επόμενο λογικό βήμα στη σωλήνωση συνεχούς ανάπτυξης που αναφέρθηκε προηγουμένως.

Τέλος, δεν μπορεί να παραβλεφθεί η τροποποίηση του πηγαία κώδικα του Spark με σκοπό τη βελτιστοποίηση της ελαστικότητας. Εστιάζοντας στη λειτουργία του ExecutionAllocationManager και χρησιμοποιώντας τις δυνατότητες που προσφέρει η χρήση του Kubernetes, θα μπορούσε να δημιουργηθεί ένα εξειδικευμένο πλαίσιο εργασιών για δυναμική ανάθεση πόρων, πέρα από τις προκαθορισμένες επιλογές που προσφέρει ήδη το εργαλείο.

Αναφορές

[1] Rishi Yadav. *Spark Cookbook*. Packt Publishing, 2015. ISBN: 9781783987061.

[2] Kirill Shirinkin. *Getting Started with Terraform*. Packt Publishing, 2017. ISBN: 9781786465108.

[3] Matei Zaharia Bill Chambers. *Spark: The Definitive Guide*. O'Reilly Media, Inc., 2018. ISBN: 9781491912218.

[4] Brian Brazil. *Prometheus: Up Running*. O'Reilly Media, Inc., 2018. ISBN: 9781492034148.

[5] Yevgeniy Brikman. *Terraform: Up Running, 2nd Edition*. O'Reilly Media, Inc., 2019. ISBN: 9781492046905.

[6] Paul Hinze (HashiCorp). *Applying Graph Theory to Infrastructure as Code*. URL: `https://www.youtube.com/watch?v=Ce3RNfRbdZ0`. (accessed: 23.11.2021).

[7] Arne Holst (Statista). *Amount of data created, consumed, and stored 2010-2025*. URL: `https://www.statista.com/statistics/871513/worldwide-data-created/`. (accessed: 23.11.2021).

[8] *Amazon Elastic Block Storage*. URL: `https://aws.amazon.com/ebs/`.

[9] *Amazon Elastic Compute Cloud*. URL: `https://aws.amazon.com/ec2/`.

[10] *Amazon Simple Storage Service*. URL: `https://aws.amazon.com/s3/`.

[11] *Ansible*. URL: `https://www.ansible.com/`.

[12] *Apache Hadoop*. URL: `https://hadoop.apache.org/`.

[13] *Apache Spark*. URL: `https://spark.apache.org/`.

[14] *Apache Spark Configuration*. URL: https://spark.apache.org/docs/latest/configuration.html.

[15] *Apache Spark Job Scheduling*. URL: https://spark.apache.org/docs/latest/job-scheduling.html.

[16] *Apache Spark Standalone Mode*. URL: https://spark.apache.org/docs/latest/spark-standalone.html.

[17] Jay Chapel. *Even if You're Not (Yet) Multi-Cloud, You Should Use Cloud-Agnostic Tools*. URL: https://www.dataversity.net/even-if-youre-not-yet-multi-cloud-you-should-use-cloud-agnostic-tools/. (accessed: 23.11.2021).

[18] Team Cleo. *On Premise vs. Cloud: Key Differences, Benefits and Risks*. URL: https://www.cleo.com/blog/knowledge-base-on-premise-vs-cloud. (accessed: 23.11.2021).

[19] *Cloud-init*. URL: https://cloud-init.io/.

[20] Cloudflare. *What is Platform-as-a-Service (PaaS)?* URL: https://www.cloudflare.com/learning/serverless/glossary/platform-as-a-service-paas/. (accessed: 23.11.2021).

[21] Doug Cutting. *Public Cloud vs. On-Premise: What to consider?* URL: https://www.comparethecloud.net/articles/public-cloud-vs-on-premise-what-to-consider/. (accessed: 23.11.2021).

[22] *Docker*. URL: https://www.docker.com/.

[23] *Dockerhub*. URL: https://hub.docker.com/.

[24] *Grafana*. URL: https://grafana.com/.

[25] Herman van Hövell (Databricks). *A Deep Dive into the Catalyst Optimizer*. URL: https://www.youtube.com/watch?v=GDeePbbCz2g. (accessed: 23.11.2021).

[26] IBM. *Configuring networking for Apache Spark*. URL: https://www.ibm.com/docs/en/izoda/1.1.0?topic=spark-configuring-networking-apache. (accessed: 23.11.2021).

[27] *Jenkins*. URL: https://www.jenkins.io/.

[28] *Kubernetes*. URL: https://kubernetes.io/.

[29] Jacek Laskowski. *The Internals of Spark SQL*. URL: https://jaceklaskowski.github.io/mastering-spark-sql-book/. (accessed: 23.11.2021).

[30] Mike Loukides. *What is DevOps?* URL: http://radar.oreilly.com/2012/06/what-is-devops.html. (accessed: 23.11.2021).

[31] Canonical Ltd. *Amazon EC2 AMI Locator*. URL: https://cloud-images.ubuntu.com/locator/ec2/. (accessed: 23.11.2021).

[32] Elvis Plesky. *IaaS vs PaaS vs SaaS – cloud service models compared*. URL: https://www.plesk.com/blog/various/iaas-vs-paas-vs-saas-various-cloud-service-models-compared/. (accessed: 23.11.2021).

[33] *Prometheus*. URL: https://prometheus.io/.

[34] Phil Schwab. *Catalyst Analyst: A Deep Dive into Spark's Optimizer*. URL: https://www.unraveldata.com/resources/catalyst-analyst-a-deep-dive-into-sparks-optimizer/.

[35] *Terraform*. URL: https://www.terraform.io/.

[36] *Terraform Registry*. URL: https://registry.terraform.io/. (accessed: 23.11.2021).

[37] MaryAnn Xue Wenchen Fan Herman van Hövell. *Adaptive Query Execution: Speeding Up Spark SQL at Runtime*. URL: https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html. (accessed: 23.11.2021).