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

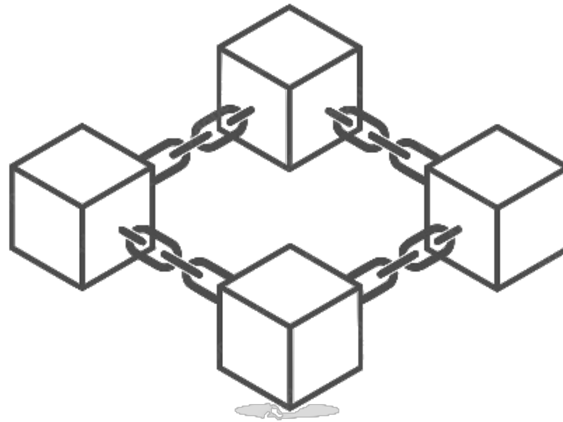
Μελέτη και βελτιστοποίηση του επιπέδου αποθήκευσης των Blockchain Clients

Μελέτη και υλοποίηση

Διπλωματικη Εργασια

του

ΙΩΑΝΝΟΥ Κ. ΣΠΥΡΙΔΩΝΟΣ



Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής

Αθήνα, Οκτώβριος 2022

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

Μελέτη και βελτιστοποίηση του επιπέδου αποθήκευσης των Blockchain Clients

Μελέτη και υλοποίηση

Διπλωματικη Εργασια

του

ΙΩΑΝΝΟΥ Κ. ΣΠΥΡΙΔΩΝΟΣ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24 Οκτωβρίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Νεκτάριος Κοζύρης Καθηγητής	Ιωάννης Κωνσταντίνου Επίκουρος Καθηγητής	Δημήτριος Τσουμάκος Αναπληρωτής Καθηγητής

Αθήνα, Οκτώβριος 2022

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

Copyright © – All rights reserved. Με την επιφύλαξη παντός δικαιώματος.
Σπυρίδων Ιωάννου, 2022.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.

Το περιεχόμενο αυτής της εργασίας δεν απηχεί απαραίτητα τις απόψεις του Τμήματος, του Επιβλέποντα, ή της επιτροπής που την ενέκρινε.

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

....
Σπυρίδων Ιωάννου

24 Οκτωβρίου 2022

Περίληψη

Το Blockchain είναι ένας τύπος κατανεμημένης δημόσιας βάσης δεδομένων, που αποτελείται από μία αυξανόμενη λίστα καταγραφών, που ονομάζοναι μπλοκ, τα οποία συνδέονται μεταξύ τους με την χρήση κρυπτογραφίας. Η τεχνολογία αυτή η οποία τα τελευταία χρόνια γίνεται ολονένα και πιο δημοφιλής, λογώ της δυνατότητας που παρέχει για ύπαρξη αμετάβλητων δεδομένων, η εγκυρότητα των οποίων είναι εξασφαλισμένη για όλους τους κόμβους του δικτύου, χωρίς την παρέμβαση ενδιάμεσων ρυθμιστικών αρχών. Μια από τις πιο δημοφιλής περιπτώσεις Blockchain είναι το Ethereum, λόγω της δυνατότητας που παρέχει για δημιουργία αποκεντρωμένων εφαρμογών στην κορυφή του blockchain, μέσω της λειτουργίας των έξυπνων συμβολαίων. Οι κόμβοι του Blockchain αποθηκεύουν μεγάλο όγκο δεδομένων που σχετίζονται με το state του, τα transactions που έχουν γίνει, καθώς και τα δεδομένα των έξυπνων συμβολαίων. Συνήθως τα δεδομένα αυτά φυλάσσονται σε δενδρικές δομές αποθήκευσης (tries) που προσφέρουν γρήγορη αναζήτηση και που υλοποιούνται με τη βοήθεια κάποιου key-value store (leveldb, rocksdb) . Ωστόσο, το storage layer μπορεί σε κάποιες λειτουργίες του πρωτοκόλου να αποτελεί performance bottleneck, όπως για παράδειγμα στο αρχικό sync ενός κόμβου που μόλις πρωτοεισέρχεται στο blockchain.

Στόχος της διπλωματικής εργασίας είναι (α) η μελέτη του workload που καλείται να εξυπηρετήσει το storage layer ενός Ethereum blockchain client (Geth) κατά τις διάφορες φάσεις της λειτουργίας του και (β) η βελτιστοποίηση της απόδοσής του με χρήση ενός διαφορετικού και κατάλληλα επιλεγμένου key-value store (BadgerDB).

Λέξεις Κλειδιά

Blockchain, Αλυσίδα, Μπλοκ, Κρυπτογραφικός Κατακερματισμός, Αποθήκη Ζευγαριών (κλειδί, τιμή), Ethereum, Patricia Merkle Trie, Geth, Δομή Επιτάχυνσης στιγμιότυπου, LevelDB, BadgerDB LSM-trees.

Abstract

A blockchain is a type of distributed public database, that consists of growing list of records, named blocks, that are securely linked together using cryptography. This technology, which in recent years has become more and more popular, due to the possibility it provides for the existence of immutable data, the validity of which is guaranteed for all network nodes, without the intervention of intermediate third-party regulatory authorities. One of the most popular cases of Blockchain is Ethereum, because it allows the development of decentralized applications on top of blockchain, through the functionality of smart contracts. Blockchain nodes store a large amount of data related to its state, transactions made, as well as smart contract data. Usually, these data are stored in tree storage structures (tries), that offer fast lookup, and which are implemented with the help of a key-value store (leveldb, rocksdb). However, the storage layer can be a performance bottleneck in some functions of the protocol, as for example in the initial sync of a node that is just entering the blockchain for the first time.

This diploma thesis aims to the (a) study of the workload called upon to serve the storage layer of an Ethereum blockchain client (Geth) during the various phases of its operation and (b) the optimization of its performance using a different and appropriately chosen key-value store (BadgerDB).

Keywords

Blockchain, Chain, Block, Cryptographic Hashing, Key-Value store, Ethereum, Patricia Merkle Trie, Geth, Snapshot Acceleration, LevelDB, BadgerDB, LSM-trees.

στους γονείς μου

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή κ. Νεκτάριο Κοζύρη για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να την εκπονήσω στο εργαστήριο Υπολογιστικών Συστημάτων. Επίσης ευχαριστώ ιδιαίτερα την Δρ. Κατερίνα Δόκα για την καθοδήγησή της και την εξαιρετική συνεργασία που είχαμε. Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Αθήνα, Οκτώβριος 2022

Σπυρίδων Ιωάννου

Περιεχόμενα

Περίληψη 1

Abstract 3

Ευχαριστίες 7

1 Εισαγωγή 17

1.1 Αντικείμενο της διπλωματικής . . . 18

1.2 Οργάνωση του τόμου . . . 19

I Θεωρητικό Μέρος 21

2 Θεωρητικό υπόβαθρο 23

2.1 Εισαγωγή στο Blockchain . . . 23

2.1.1 Το Πρόβλημα της Διπλής Δαπάνης . . . 23

2.1.2 Η Τεχνολογία Blockchain . . . 27

2.2 Blockchain Ethereum και Έξυπνα Συμβόλαια . . . 30

2.2.1 Εισαγωγή στο Ethereum . . . 30

2.2.2 Γενική Δομή και Έξυπνα Συμβόλαια . . . 30

2.3 Αποθήκευση Δεδομένων στο Ethereum Blockchain . . . 31

2.3.1 Merkle Patricia Tries . . . 31

2.3.2 Η Δομή του Επιπέδου Αποθήκευσης στο Ethereum . . . 34

2.4 Το Go Ethereum (Geth) . . . 35

2.4.1 Επιτάχυνση Snapshot . . . 38

2.4.2 Το Snap Sync . . . 41

2.4.3 Goerli Testnet . . . 42

2.4.4 LSM-trees και LevelDB . . . 43

2.4.5 BadgerDB . . . 44

II Πρακτικό Μέρος 47

3 Μελέτη του Geth Client 49

3.1 Ανάλυση του Workload . . . 49

3.1.1 Αναλογία Αναγνώσεων-Εγγραφών στον Δίσκο . . . 49

3.1.2 Χρονική καθυστέρηση της διαδικασίας του Compaction . . . 52

3.2 Χαρακτηριστικά του BadgerDB 52

4 Απόδοση Syncing με Χρήση BadgerDB 55

4.1 Στάδια Συγχρονισμού 55

4.2 Σύγκριση LevelDB και BadgerDB 55

4.2.1 Υλοποίηση ενσωμάτωσης BadgerDB 56

4.2.2 Εκτέλεση Πειραμάτων 56

4.3 Σχολιασμός Αποτελεσμάτων 57

III Επίλογος 59

5 Επίλογος 61

5.1 Συμπεράσματα 61

5.2 Μελλοντικές Επεκτάσεις 61

Παραρτήματα 63

Α΄ Ενσωμάτωση BadgerDB στο Geth Client 65

Α΄.1 Υλοποίηση Interface Geth 65

Β΄ Συλλογή δεδομένων κατά την εκτέλεση των πειραμάτων 69

Β΄.1 Επιλογές για το Τρέξιμο του Client 69

Β΄.2 Χαρακτηριστικά Μηχανημάτων 70

Βιβλιογραφία 72

Συντομογραφίες - Αρκτικόλεξα - Ακρωνύμια 73

Απόδοση ξενόγλωσσων όρων 75

Κατάλογος Σχημάτων

Κατάλογος Εικόνων

2.1 Το πρόβλημα της διπλής δαπάνης, τι ποσό θα έχει ο Β και ο Σ μετά τη διπλή αποστολή του Α. 23
2.2 Η ψηφοφορία των κόμβων σχετικά με το ποια συναλλαγή πραγματοποιείται πρώτη 26
2.3 Σχηματική απεικόνιση αλυσίδας blockchain (chain) . Η σύνδεση των blocks γίνεται μέσω των επιμέρους hashes αυτών. 28
2.4 Blockchain Ethereum και έξυπνα συμβόλαια 29
2.5 Patricia Trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn". 32
2.6 Merkle Tree όπου το hash των γονιών προκύπτει από πρόσθεση των αντίστοιχων hashes των παιδιών 32
2.7 Παράδειγμα Ethereum Modified Merkle Patricia Trie 33
2.8 Δομή Αποθήκευσης Δεδομένων στο Ethereum 35
2.9 Επισκόπηση State Trie - Storage Trie 36
2.10 Επισκόπηση Transactions Trie 36
2.11 Επισκόπηση Δομής Snapshot Acceleration 40
2.12 Snap Sync vs Fast Sync Downloads 42
2.13 LSM-Tree and LevelDB Architecture 44
2.14 LSM-Tree with Key-Value Seperation 45

Κατάλογος Πινάκων

3.1 Χρονική Καθυστέρηση Compaction ως Ποσοστό Ολικού Χρόνου Συντονισμού 52

4.1 Χρονική Διάρκεια Snap Sync LevelDB vs BadgerDB 57

Κεφάλαιο 1

Εισαγωγή

Η τεχνολογία Blockchain αποτελεί ένα θέμα το οποίο έχει συγκεντρώσει μεγάλη δημοσιότητα πρόσφατα. Η τεχνολογία αυτή ήρθε στην επιφάνεια μετά την εισαγωγή του Bitcoin το 2008 από το άτομο ή την ομάδα ανθρώπων με το ψευδώνυμο Satoshi Nakamoto. Πολλοί άνθρωποι μπερδεύονται και πιστεύουν ότι το Blockchain και το Bitcoin είναι έννοιες ταυτόσημες. Ωστόσο, το Bitcoin είναι απλά μια μόνο εφαρμογή της τεχνολογίας Blockchain. Υπάρχουν πολλές άλλες εφαρμογές και περιπτώσεις χρήσης που μπορούν να επιλυθούν χρησιμοποιώντας blockchain οι οποίες εμπεριέχουν αλλά δεν περιορίζονται σε συστήματα πληρωμών.

Ενώ το Bitcoin πρωτοστάτησε στην τεχνολογία blockchain όντας το πρώτο κρυπτονόμισμα, ένα άλλο κρυπτονόμισμα με το όνομα Ethereum είναι αυτό που τα τελευταία χρόνια έχει γίνει το κύριο θέμα συζήτησης στον χώρο του blockchain. Το Ethereum έχει επεκτείνει το αποκεντρωμένο ψηφιακό νόμισμα του Bitcoin δημιουργώντας ένα παγκόσμιο δίκτυο που στηρίζει μια διασυνδεδεμένη αγορά αποκεντρωμένων εφαρμογών (dApps), προσφέροντας πρωτοφανή αποτελεσματικότητα, ασφάλεια και έλεγχο στον χρήστη. Μέσω του πρωτοποριακού συνδυασμού χαρακτηριστικών του, όπως τα έξυπνα συμβόλαια(smart contracts), το Ethereum χρησιμοποιείται για μια ποικιλία καινοτόμων εφαρμογών στα οικονομικά, την περιήγηση στο διαδίκτυο, τα παιχνίδια, τη διαφήμιση, τη διαχείριση ταυτότητας και τη διαχείριση της αλυσίδας εφοδιασμού.

Υπάρχει μία πληθώρα από τομείς στους οποίους το Ethereum παρέχει χρησιμότητα και δημιουργεί αξία. Βιομηχανίες από την υγειονομική περίθαλψη μέχρι την ψυχαγωγία και την ακίνητη περιουσία δημιουργούν νέα εργαλεία στο πρωτόκολλο για να ενισχύσουν την αποτελεσματικότητα, την εμπιστοσύνη και να εκδημοκρατίσουν την πρόσβαση σε διάφορους τύπους υπηρεσιών. Για παράδειγμα, το Ethereum παρέχει μια ιδανική λύση για τη διαχείριση δικαιωμάτων εκμετάλλευσης στη μουσική βιομηχανία, διανέμοντας tokens που αντιπροσωπεύουν δικαιώματα ιδιοκτησίας που διευκολύνουν την αυτοματοποιημένη και απρόσκοπτη διανομή των πληρωμών δικαιωμάτων.

Το απαραβίαστο του Ethereum blockchain μπορεί να διαβεβαιώσει τους διαχειριστές εφοδιαστικής αλυσίδας και logistics σχετικά με την προέλευση των προϊόντων μέσω επαληθεύσιμης κρυπτογραφίας που βασίζεται στην τεχνολογία blockchain. Αυτές οι επιχειρήσεις μπορούν να παρακολουθούν τη διαδρομή ενός προϊόντος στο blockchain από τον κατασκευαστή μέχρι το ταμείο, γνωρίζοντας ότι τα δεδομένα δεν έχουν παραβιαστεί. Εν τω μεταξύ, οι τελικοί καταναλωτές μπορούν να είναι ήσυχοι γνωρίζοντας ότι τα προϊόντα που αγοράζουν

είναι στην πραγματικότητα αυθεντικά. Τα πάντα, από προϊόντα πολυτελείας έως βιολογικά τρόφιμα, παρακολουθούνται και ανιχνεύονται με το δίκτυο Ethereum.

Επιπλέον, μέσω της χρήσης κρυπτογραφικών μεθόδων, το Ethereum διασφαλίζει την ασφαλή ανταλλαγή πληροφοριών, η οποία είναι απαραίτητη για τη μεταφορά ευαίσθητων δεδομένων, όπως ιατρικά αρχεία και πληροφορίες ταυτότητας. Τέλος, τα tokens του Ethereum εκδημοκρατίζουν την πρόσβαση σε προϊόντα που προηγουμένως δεν ήταν προσιτά σε πολλούς. Υπάρχουν νεοφυείς επιχειρήσεις που βασίζονται στο Ethereum που προσφέρουν τμηματική ιδιοκτησία, δηλαδή υποστηρίζουν την κατοχή ενός κομματιού ενός αγαθού και όχι ολόκληρου. Η υπηρεσία αυτή αφορά κυρίως αγαθά πολυτελείας και ακίνητης περιουσίας, επιτρέποντας στους καταναλωτές να διαφοροποιήσουν τις επενδύσεις τους.

Το Ethereum είναι το δίκτυο που δίνει την δυνατότητα για καινοτομία στον χώρο του blockchain και των κρυπτονομισμάτων. Με την ευελιξία και τη στιβαρότητά του, νέες εφαρμογές συνεχίζουν να εμφανίζονται, κάτι που όπως είναι λογικό γεννά νέες απαιτήσεις, περιορισμούς χωρητικότητας, οπότε και αμφιβολίες για την δυνατότητα του δικτύου να ανταπεξέλθει. Η επεκτασιμότητα του δικτύου είναι ένα μεγάλο ζήτημα για το Ethereum και απασχολεί πολύ τους ερευνητές και τους Developpers του. Συγκεκριμένα, υπάρχουν πολλές λύσεις που ερευνώνται, δοκιμάζονται και εφαρμόζονται, που ακολουθούν διαφορετικές προσεγγίσεις για την επίτευξη παρόμοιων στόχων.

1.1 Αντικείμενο της διπλωματικής

Ένα βασικό ζήτημα που προκύπτει για την λειτουργία και την επεκτασιμότητα του Ethereum Blockchain είναι το γεγονός ότι οι κόμβοι του αποθηκεύουν μεγάλο όγκο δεδομένων που σχετίζονται με το state του, τα transactions που έχουν γίνει, καθώς και τα δεδομένα των έξυπνων συμβολαίων. Συνήθως τα δεδομένα αυτά φυλάσσονται σε δενδρικές δομές αποθήκευσης (merkle patricia tries) που προσφέρουν γρήγορη αναζήτηση και που υλοποιούνται με τη βοήθεια κάποιου key-value store (leveldb, rocksdb). Προφανώς, όσο το δίκτυο επεκτείνεται τόσο μεγαλώνουν και οι ανάγκες σε αποθηκευτικό χώρο, αλλά με ταυτόχρονη διασφάλιση της ταχείας απόκρισης των λειτουργιών. Έτσι, το storage layer του blockchain μπορεί σε κάποιες λειτουργίες του πρωτοκόλου να αποτελεί performance bottleneck, όπως για παράδειγμα στο αρχικό sync ενός κόμβου που μόλις πρωτοεισέρχεται στο blockchain, όπου πρέπει να κατεβάσει μεγάλο όγκο αρχείων και έχει μεγάλες απαιτήσεις σε ταχύτητα απόκρισης για να μπορεί η όλη διαδικασία να είναι εύκολη και γρήγορη.

Αντικείμενο της διπλωματικής εργασίας είναι (α) η μελέτη του workload που καλείται να εξυπηρετήσει το storage layer ενός Ethereum blockchain client (Geth) κατά τις διάφορες φάσεις της λειτουργίας του και (β) η βελτιστοποίηση της απόδοσής του με χρήση ενός διαφορετικού και κατάλληλα επιλεγμένου key-value store (BadgerDB). Επιχειρούμε, δηλαδή, να βελτιώσουμε τον χρόνο που απαιτείται ώστε οι νέοι κόμβοι που εισέρχονται στο δίκτυο να συγχρονιστούν με αυτό και να μπορούν να εκτελέσουν όλες τις επιθυμητές ενέργειες σε αυτό. Η μέλετη του workload έχει ως στόχο να μας υποδείξει την κατάλληλη επιλογή για την αντικατάσταση του υπάρχοντος key-value store (LevelDB), για την οποία επιλογή έχει αξία να μελετήσουμε σε τι βαθμό μπορεί να επηρεάσει την ταχύτητα του συγχρονισμού.

1.2 Οργάνωση του τόμου

Η εργασία αυτή είναι οργανωμένη σε επτά κεφάλαια: Στο Κεφάλαιο 2 δίνεται το θεωρητικό υπόβαθρο των βασικών τεχνολογιών που σχετίζονται με τη διπλωματική αυτή. Αρχικά περιγράφεται η λογική πίσω από την τεχνολογία του Blockchain και συγκεκριμένα του Ethereum, στην συνέχεια το πως το δεύτερο αποθηκεύει τα δεδομένα του και τέλος ασχολούμαστε με την δομή του Ethereum Client Geth καθώς και των key-value stores που θα χρησιμοποιήσουμε στα πειράματα μας. Στο Κεφάλαιο 3 παρουσίαζεται η ανάλυση του workload του Geth, καθώς και τα χαρακτηριστικά του BadgerDB μαζί με τους λόγους που μας οδήγησαν στην επιλογή του ως εναλλακτικό key-value store. Η παρουσίαση των αποτελεσμάτων της ενσωμάτωσης του BadgerDB στο Geth Client, στην θέση της LevelDB δίνεται στο Κεφάλαιο 4. Τέλος στο Κεφάλαιο 5 δίνονται τα συμπεράσματα, η συνεισφορά αυτής της διπλωματικής εργασίας, καθώς και μελλοντικές επεκτάσεις.

Μέρος I

Θεωρητικό Μέρος

Κεφάλαιο 2

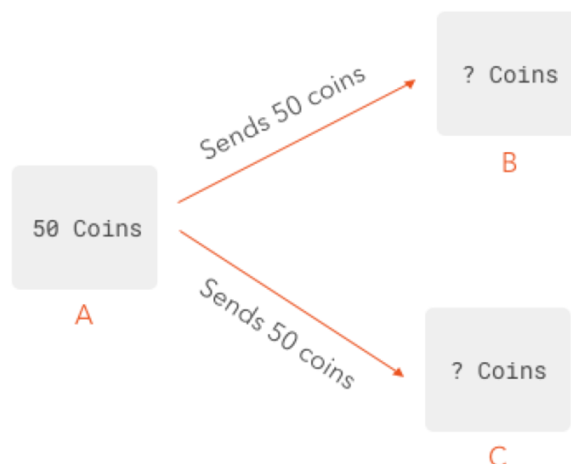
Θεωρητικό υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζονται αναλυτικά η τεχνολογία του Blockchain, εστιάζοντας στην συνέχεια στο Ethereum Blockchain και συγκεκριμένα στον τρόπο που αυτό αποθηκεύει τα δεδομένα.

2.1 Εισαγωγή στο Blockchain

2.1.1 Το Πρόβλημα της Διπλής Δαπάνης

Το 2009, ο Satoshi Nakamoto (ψευδώνυμο) συνέγραψε το εμβληματικό whitepaper του Bitcoin [1]. Το Bitcoin ήταν έτοιμο να λύσει ένα πολύ συγκεκριμένο πρόβλημα: πώς μπορεί να επιλυθεί το πρόβλημα της διπλής δαπάνης χωρίς μια κεντρική αρχή να ενεργεί ως διαιτητής σε κάθε συναλλαγή; Το πρόβλημα της διπλής δαπάνης είναι μια συγκεκριμένη περίπτωση επεξεργασίας συναλλαγών. Οι συναλλαγές, εξ ορισμού, πρέπει είτε να γίνονται είτε όχι. Επιπλέον, ορισμένες συναλλαγές πρέπει να παρέχουν την εγγύηση ότι θα πραγματοποιηθούν πριν ή μετά από άλλες συναλλαγές (με άλλα λόγια, πρέπει να είναι ατομικές). Η ατομικότητα (atomicity) γεννά την έννοια της σειράς: οι συναλλαγές είτε γίνονται είτε όχι πριν είτε μετά από άλλες συναλλαγές.



Εικόνα 2.1: *Το πρόβλημα της διπλής δαπάνης, τι ποσό θα έχει ο Β και ο Σ μετά τη διπλή αποστολή του Α.*

Η έλλειψη ατομικότητας είναι ακριβώς το ζήτημα του προβλήματος της διπλής δαπάνης: η «δαπάνη» ή η αποστολή χρημάτων από τον A στον B, πρέπει να συμβεί σε μια συγκεκριμένη χρονική στιγμή και πριν και μετά από οποιαδήποτε άλλη συναλλαγή. Εάν δεν συνέβαινε αυτό, θα ήταν δυνατό να αποσταλούν χρήματα περισσότερες από μία φορές σε ξεχωριστές αλλά ταυτόχρονες συναλλαγές.[2].

Όταν πρόκειται για καθημερινές νομισματικές πράξεις, οι συναλλαγές συνήθως επιβλέπονται από τις τράπεζες. Όταν ένας χρήστης συνδέεται στο τραπεζικό σύστημα και εκτελεί τραπεζικό έμβασμα, είναι η τράπεζα που διασφαλίζει ότι τυχόν προηγούμενες και μελλοντικές λειτουργίες είναι συνεπείς. Αν και η διαδικασία μπορεί να φαίνεται απλή σε τρίτους, είναι στην πραγματικότητα μια αρκετά εμπεριστατωμένη διαδικασία με εκκαθάριση και απαιτήσεις διακανονισμού. Στην πραγματικότητα, ορισμένες από αυτές τις διαδικασίες εξετάζουν την πιθανότητα μιας κατάστασης διπλής δαπάνης και τι πρέπει να συμβεί σε αυτές τις περιπτώσεις. Δεν πρέπει να προκαλεί έκπληξη το γεγονός ότι πρόκειται για αρκετά σύνθετες διαδικασίες, με αποτέλεσμα σημαντικές καθυστερήσεις [3].

Έτσι, το κύριο πρόβλημα που πρέπει να αντιμετωπίσει κάθε σύστημα συναλλαγών που εφαρμόζεται στη χρηματοδότηση είναι «πώς να πιστοποιούνται οι συναλλαγές όταν δεν υπάρχει κεντρική αρχή εποπτείας». Επιπλέον, δεν θα πρέπει να υπάρχει αμφιβολία για το εάν η σειρά των προηγούμενων συναλλαγών είναι έγκυρη. Για να πετύχει ένα νομισματικό σύστημα, δεν πρέπει να υπάρχει τρόπος να τροποποιήσουν τα μέρη προηγούμενες συναλλαγές. Αυτό ακριβώς σχεδιάστηκε για να αντιμετωπίσει το σύστημα blockchain στο Bitcoin [4].

Η επικύρωση συναλλαγών βασίζεται στο blockchain και χρησιμοποιεί κρυπτογραφία δημόσιου κλειδιού (Public-key cryptography). Η κρυπτογραφία δημόσιου κλειδιού υλοποιείται με την ύπαρξη δύο κλειδιών: ένα δημόσιο και ένα ιδιωτικό κλειδί. Αυτά τα κλειδιά χρησιμοποιούνται παράλληλα για συγκεκριμένους σκοπούς. Τα δεδομένα που είναι κρυπτογραφημένα με το δημόσιο κλειδί μπορούν να αποκρυπτογραφηθούν μόνο με τη χρήση του ιδιωτικού κλειδιού, ενώ τα δεδομένα που υπογράφονται με το ιδιωτικό κλειδί μπορούν να επαληθευτούν χρησιμοποιώντας το δημόσιο κλειδί. Το ιδιωτικό κλειδί δεν μπορεί να προέρχεται από το δημόσιο κλειδί, αλλά το δημόσιο κλειδί μπορεί να προέρχεται από το ιδιωτικό κλειδί. Το δημόσιο κλειδί προορίζεται για ασφαλή κοινή χρήση και συνήθως μπορεί να εκτεθεί ελεύθερα σε οποιονδήποτε. Αντίθετα το ιδιωτικό κλειδί θα πρέπει να είναι γνωστό μόνο σε αυτόν που το χρησιμοποιεί διασφαλίζοντας έτσι την ασφάλεια των συναλλαγών. [3].

Ας δούμε πώς μια πολύ απλή συναλλαγή μπορεί να επαληθευτεί. Ας υποθέσουμε ότι υπάρχει ο κάτοχος λογαριασμού A που κατέχει 50 νομίσματα. Αυτά τα νομίσματα του στάλθηκαν ως μέρος προηγούμενης συναλλαγής. Ο κάτοχος λογαριασμού A θέλει τώρα να στείλει αυτά τα νομίσματα στον κάτοχο του λογαριασμού B. Ο B, και οποιοσδήποτε άλλος θέλει να ελέγξει αυτή τη συναλλαγή, πρέπει να είναι σε θέση να επαληθεύσει ότι στην πραγματικότητα ο A ήταν αυτός που έστειλε τα νομίσματα στον B. Επιπλέον, πρέπει να μπορούν να δουν τον B και όχι κάποιον άλλον να τα εξαργυρώνει. Προφανώς, θα πρέπει επίσης να μπορούν να βρίσκουν το ακριβές χρονικό σημείο, σε σχέση με άλλες συναλλαγές, στις οποίες πραγματοποιήθηκε αυτή η συναλλαγή [2].

Για το απλό αυτό παράδειγμα, υποθέτουμε ότι τα δεδομένα στη συναλλαγή είναι ένα αναγνωριστικό για την προηγούμενη συναλλαγή (previous-transaction-id) που έδωσε στον A 50 νομίσματα, το δημόσιο κλειδί του τρέχοντος κατόχου A (owner-pubkey) και την υπογρα-

φή του προηγούμενου ιδιοκτήτη (prev-owner-signature) που επιβεβαιώνει ότι αυτός έστειλε αυτά τα νομίσματα στον A.

{
"previous-transaction-id": "FEDCBA987654321...",
"owner-pubkey": "123456789ABCDEF...",
"prev-owner-signature": "AABBCCDDEEFF112233..."
}

Η απόδειξη ότι ο A είναι ο κάτοχος αυτών των νομισμάτων υπάρχει ήδη: το δημόσιο κλειδί του είναι ενσωματωμένο στη συναλλαγή. Τώρα όποια ενέργεια γίνει από τον A πρέπει να επαληθευτεί με κάποιο τρόπο. Ένας τρόπος για να γίνει αυτό θα ήταν να προστεθούν πληροφορίες στη συναλλαγή και στη συνέχεια να δημιουργηθεί μια νέα υπογραφή. Εφόσον ο A θέλει να στείλει χρήματα στον B, οι προστιθέμενες πληροφορίες θα μπορούσαν απλώς να είναι το δημόσιο κλειδί του B. Μετά τη δημιουργία αυτής της νέας συναλλαγής, θα μπορούσε να υπογραφεί χρησιμοποιώντας το ιδιωτικό κλειδί του A. Αυτό αποδεικνύει ότι ο A, και μόνο ο A, συμμετείχε στη δημιουργία αυτής της συναλλαγής [3].

Το κρίσιμο σημείο είναι ότι τα δεδομένα που υπογράφονται με το ιδιωτικό κλειδί μπορούν να επαληθευτούν χρησιμοποιώντας το δημόσιο κλειδί. Επομένως η υπογραφή του A στη νέα συναλλαγή δημιουργεί έναν επαληθεύσιμο σύνδεσμο μεταξύ της νέας συναλλαγής και της παλιάς. Η νέα συναλλαγή παραπέμπει ρητά στην παλιά και η υπογραφή της νέας συναλλαγής μπορεί να δημιουργηθεί μόνο από τον κάτοχο του ιδιωτικού κλειδιού της παλιάς συναλλαγής (η παλιά συναλλαγή μας λέει ρητά ποιος είναι αυτός μέσω του πεδίου ιδιοκτήτης-pubkey). Έτσι, η παλιά συναλλαγή κατέχει το δημόσιο κλειδί εκείνου που μπορεί να το ξοδέψει, και η νέα συναλλαγή κρατά το δημόσιο κλειδί εκείνου που το έλαβε, μαζί με την υπογραφή που δημιουργήθηκε με το ιδιωτικό κλειδί του καταναλωτή [4].

Η μεγαλύτερη συνεισφορά που έφερε το Bitcoin στα υπάρχοντα συστήματα κρυπτονομισμάτων ήταν ένας αποκεντρωμένος τρόπος να γίνουν οι συναλλαγές ατομικές. Πριν από το Bitcoin, οι ερευνητές πρότειναν διαφορετικά σχήματα για να το πετύχουν αυτό. Ένα από αυτά τα σχέδια ήταν ένα απλό σύστημα ψηφοφορίας. Για να γίνει καλυτερα κατανοητή η απλότητα της προσέγγισης του Bitcoin, θα εξερευνήσουμε κάποιες από τις προηγούμενες προσπάθειες.

Σε ένα σύστημα ψηφοφορίας (voting system), κάθε συναλλαγή μεταδίδεται από τον κόμβο που την εκτελεί. Έτσι, για να συνεχίσουμε με το παράδειγμα του A που στέλνει 50 νομίσματα στον B, ο A προετοιμάζει μια νέα συναλλαγή δείχνοντας τη συναλλαγή με την οποία έλαβε τα 50 νομίσματα, μετά βάζει το δημόσιο κλειδί του B σε αυτό και χρησιμοποιεί το δικό του ιδιωτικό κλειδί για να το υπογράψει. Αυτή η συναλλαγή στη συνέχεια αποστέλλεται σε κάθε κόμβο που είναι γνωστός από τον A στο δίκτυο. Ας πούμε ότι εκτός από τον A και τον B, υπάρχουν άλλοι τρεις κόμβοι: C, D, E.

Υποθέτουμε τώρα ότι ο A είναι στην πραγματικότητα ένας κακόβουλος κόμβος. Αν και φαίνεται ότι ο A θέλει να στείλει 50 νομίσματα στον B και ο A εκπέμπει αυτήν τη συναλλαγή, αλλά ταυτόχρονα εκπέμπει και μια διαφορετική συναλλαγή: ο A στέλνει αυτά τα 50 νομίσματα στον C δείχνοντας την ίδια συναλλαγή από την οποία έλαβε τα 50 νομίσματα,

με σκοπό να στείλει δύο φορές 50 νομίσματα, εξού και το πρόβλημα ονομάζεται πρόβλημα διπλής δαπάνης. Σε αυτήν την περίπτωση έχουμε για τις δύο συνναλαγές με βάση τα παραπάνω :

```
const aToB = {
"previous-transaction-id": "FEDCBA987654321...",
"owner-pubkey": "123456789ABCDEF...", // B
"prev-owner-signature": "..."
}

const aToC = {
"previous-transaction-id": "FEDCBA987654321...",
"owner-pubkey": "00112233445566...", // C
"prev-owner-signature": "..."
}
```

Όπως προαναφέρθηκε το αναγνωριστικό προηγούμενης συναλλαγής δείχνει την ίδια συναλλαγή, δηλαδή ο κόμβος A στέλνει ταυτόχρονα αυτή τη συναλλαγή σε δυο διαφορετικούς κόμβους του δικτύου. Ποιος θα πάρει τα 50 νομίσματα;

Δεδομένου ότι πρόκειται για ένα κατανεμημένο δίκτυο, κάθε κόμβος έχει κάποιο βάρος στην απόφαση. Κάθε κόμβος ψηφίζει ποια συναλλαγή θα επιλέξει να πραγματοποιηθεί πρώτη. Υποθέτουμε ότι οι αποφάσεις των επιμέρους κόμβων είναι οι εξής :

Node	Vote
A	A to B
B	A to B
C	A to C
D	A to C
E	A to B

Εικόνα 2.2: *Η ψηφοφορία των κόμβων σχετικά με το ποια συναλλαγή πραγματοποιείται πρώτη*

Κάθε κόμβος δίνει μια ψήφο και κατά πλειοψηφία ο A στο B επιλέγεται ως η συναλλαγή που πρέπει να εκτελεστεί πρώτη. Προφανώς, αυτό ακυρώνει τη συναλλαγή A προς C που δείχνει τα ίδια νομίσματα με το A στο B. Φαίνεται ότι αυτή η λύση λειτουργεί, αλλά μόνο επιφανειακά. Ας δούμε γιατί.

Αρχικά, ας εξετάσουμε ότι η περίπτωση A έχει συνεννοηθεί με κάποιον άλλο κόμβο. Ο E ψήφισε τυχαία ή υποκινήθηκε κατά κάποιο τρόπο από τον A να επιλέξει τη μία συναλλαγή έναντι της άλλης· Δεν υπάρχει πραγματικός τρόπος να προσδιοριστεί αυτό. Δεύτερον, το μοντέλο μας δεν λαμβάνει υπόψη την ταχύτητα διάδοσης των συναλλαγών. Σε ένα αρκετά μεγάλο δίκτυο κόμβων, ορισμένοι κόμβοι μπορεί να δουν κάποιες συναλλαγές πριν από

άλλους. Αυτό προκαλεί μια ανισορροπία στις ψήφους. Δεν είναι δυνατό να προσδιοριστεί εάν μια μελλοντική συναλλαγή μπορεί να ακυρώσει αυτές που έχουν φτάσει. Επιπλέον δεν είναι δυνατό να προσδιοριστεί εάν η συναλλαγή που λήφθηκε έγινε πριν ή μετά από κάποια άλλη συναλλαγή σε αναμονή για ψηφοφορία. Εκτός εάν οι συναλλαγές είναι ορατές από όλους τους κόμβους, οι ψήφοι μπορεί να είναι άδικες. Ακόμη χειρότερα, κάποιος κόμβος θα μπορούσε να καθυστερήσει ενεργά τη διάδοση μιας συναλλαγής.

Τέλος, ένας κακόβουλος κόμβος θα μπορούσε να εισάγει μη έγκυρες συναλλαγές για να προκαλέσει στοχευμένη άρνηση υπηρεσίας. Αυτό θα μπορούσε να χρησιμοποιηθεί για να ευνοήσει ορισμένες συναλλαγές έναντι άλλων.

Ένα σύστημα ψηφοφορίας δεν μπορεί να διορθώσει αυτά τα προβλήματα επειδή είναι εγγενή στο σχεδιασμό του συστήματος. Οτιδήποτε χρησιμοποιείται για να ευνοήσει τη μία συναλλαγή έναντι της άλλης δεν μπορεί να αφεθεί στην επιλογή των κόμβων. Εφόσον ένας μεμονωμένος κόμβος ή μια ομάδα κόμβων μπορεί, κατά κάποιο τρόπο, να ευνοήσει ορισμένες συναλλαγές έναντι άλλων, το σύστημα δεν μπορεί να λειτουργήσει δίκαια. Αυτό ακριβώς το στοιχείο κατά τον σχεδιασμό των κρυπτονομισμάτων απαίτησε πολύ σκληρή προσπάθεια. Χρειαζόταν μια ιδιοφυής ιδέα για να ξεπεραστεί ένα τόσο βαθύ σχεδιαστικό ζήτημα [5].

2.1.2 Η Τεχνολογία Blockchain

Εδώ παρουσιάζεται μια εισαγωγή στην τεχνολογία blockchain. Για μια πληρέστερη παρουσίαση αυτού του υλικού, ο αναγνώστης θα πρέπει να συμβουλευτεί το Whitepaper του Bitcoin [1] και το Yellow Paper του Ethereum [6].

Το blockchain είναι μια κατανεμημένη και ασφαλής δομή δεδομένων που μοιράζεται σε ένα δίκτυο ισότιμων κόμβων (peer-to-peer) όπως φαίνεται στην Εικόνα 2.4(α,β) εξασφαλίζοντας εμπιστοσύνη μέσα σε αυτό. Τα κύρια χαρακτηριστικά του blockchain είναι :

- Είναι Αμετάβλητο: Τυχόν επικυρωμένες εγγραφές είναι μη αναστρέψιμες και δεν μπορούν να αλλάξουν. Αυτό σημαίνει ότι οποιοσδήποτε χρήστης στο δίκτυο δεν μπορεί να επεξεργαστεί, αλλάξει ή διαγράψει δεδομένα.

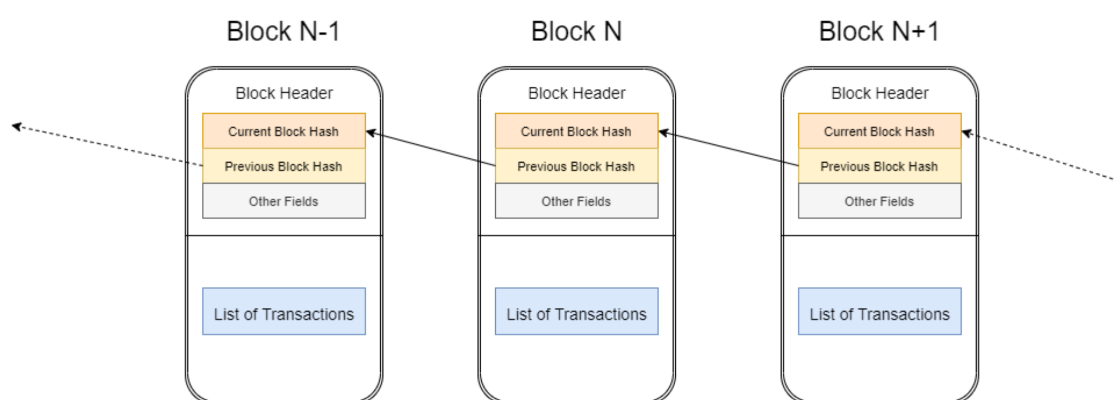
- Είναι Κατανεμημένο : Όλοι οι συμμετέχοντες στο δίκτυο έχουν ένα ίδιο αντίγραφο του blockchain εξασφαλίζοντας απόλυτη διαφάνεια σε αυτό.

- Είναι αποκεντρωμένο : Δεν υπάρχει κάποια κεντρική αρχή που παρεμβαίνει, ώστε να ¨ρυθμίζει¨ το δίκτυο, αλλά αυτό επιτυγχάνεται με ¨συνεργασία¨ από τους ίδιους του συμμετέχοντες.

- Είναι ασφαλές : Μέσω της διαδικασίας της κρυπτογράφησης αλλά και της κρυπτογραφικής σύνδεσης των blocks, καθιστά την επεξεργασία των δεδομένων πρακτικά απίθανη, καθώς μέσω αυτής θα άλλαζαν και όλα τα hash IDs σε όλο το μήκος της αλυσίδας.

- Υπάρχει συναίνεση : Το δίκτυο είναι σε θέση να καταλήγει σε γρήγορες και αντικειμενικές αποφάσεις, μέσω αλγορίθμων ομοφωνίας, τις οποίες όλοι οι συμμετέχοντες εμπιστεύονται, ακόμα και αν δεν υπάρχει εμπιστοσύνη μεταξύ τους.

Η αλυσίδα (chain) αποτελείται από μπλοκ (block) δεδομένων που συνδέονται προς τα εμπρός με κατακερματισμούς (hash), όπως φαίνεται και στην Εικόνα 2.3. Κάθε μπλοκ στην αλυσίδα περιέχει μια κεφαλίδα (header) και μια λίστα έγκυρων συναλλαγών. Η κεφαλίδα περιλαμβάνει πολλά πεδία που σχετίζονται τόσο με την ακεραιότητα της δομής δεδομένων (π.χ. τη χρονική σήμανση) όσο και με τις παραμέτρους του δικτύου (π.χ. παραμέτρους εξόρυξης). Αυτά τα πεδία περιλαμβάνουν τη χρονική σήμανση μπλοκ, τον αριθμό μπλοκ, τις παραμέτρους εξόρυξης και τον κατακερματισμό των περιεχομένων του προηγούμενου μπλοκ (που συνδέει το ένα μπλοκ με το επόμενο). Οι χρήστες ή οι κόμβοι μπορούν να υποβάλουν συναλλαγές που τροποποιούν την κατάσταση του δικτύου, για παράδειγμα στέλνοντας τιμή από μια διεύθυνση χρήστη σε άλλη ή αποθηκεύοντας μια τιμή σε ένα έξυπνο συμβόλαιο το οποίο αναλύεται περαιτέρω παρακάτω [7].



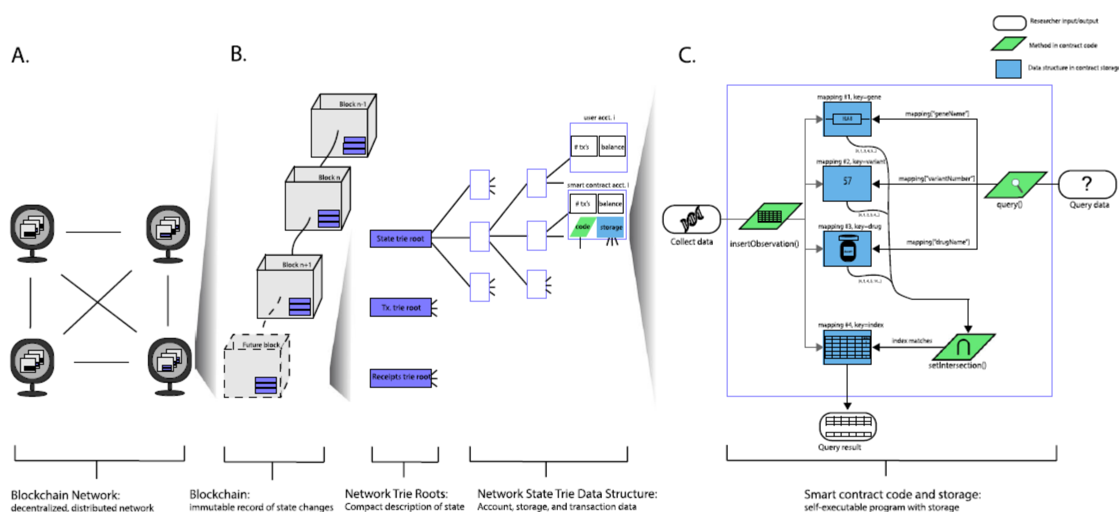
Εικόνα 2.3: *Σχηματική απεικόνιση αλυσίδας blockchain (chain) . Η σύνδεση των blocks γίνεται μέσω των επιμέρους hashes αυτών.*

Ο μηχανισμός συναίνεσης δικτύου (network consensus) καθορίζει ποιος χρήστης στο δίκτυο θα προσαρτήσει τις συναλλαγές στην αλυσίδα ως νέο μπλοκ. Αυτή είναι μια κρίσιμη διαδικασία. Εάν αποτύχει, η εγκυρότητα του προστιθέμενου μπλοκ θα διακυβευόταν. Οι πιο εξέχοντες μηχανισμοί συναίνεσης περιλαμβάνουν την απόδειξη της εργασίας(proof of work, PoW), την απόδειξη της συμμετοχής(proof of stake, PoS) και την απόδειξη της εξουσίας(proof of authority, PoA).

Σε ένα δίκτυο απόδειξης εργασίας (proof of work, PoW), οι κόμβοι εξορύσσουν μπλοκ. Δηλαδή, ασκούν υπολογιστική ενέργεια για να αναγνωρίσουν τιμές οι οποίες, όταν προστεθούν στα περιεχόμενα του εισερχόμενου μπλοκ, παράγουν ένα κατακερματισμό μπλοκ χαμηλότερο από αυτό που αναφέρεται ως δυσκολία, μια καθορισμένη παράμετρος του δικτύου. Η δυσκολία είναι μια παράμετρος σε όλο το δίκτυο, η οποία μπορεί να μεταβληθεί προκειμένου να ρυθμιστεί ο ρυθμός σχηματισμού μπλοκ. Για παράδειγμα στο δίκτυο του Bitcoin, η δυσκολία προσαρμόζεται κατά τέτοιο τρόπο, ώστε να σχηματίζεται νέο μπλοκ περίπου άνα 10 λεπτά. Ο μηχανισμός απόδειξης εργασίας είναι κρίσιμος για τα δημόσια δίκτυα blockchain κρυπτονομισμάτων όπως το Bitcoin, επειδή καθορίζει ένα κόστος για την τροποποίηση του blockchain, αποτρέποντας έτσι τους κακούς παράγοντες από το να διαφθείρουν την αλυσίδα [1]. Πολλοί έχουν επικρίνει τον Μηχανισμό PoW ότι είναι μη βιώσιμος μακροπρόθεσμα λόγω των μεγάλων ποσοτήτων ενέργειας που απαιτούνται για την εκτέλεση της υπολογιστικής εξόρυξης. Ειδικά για το Bitcoin, στην προσπάθεια η διάρκεια

σχηματισμού νέο μπλοκ να παραμείνει κοντά στα 10 λεπτά, η δυσκολία αυξάνεται συνεχώς, κάτι που σημαίνει ότι απαιτείται όλο και περισσότερη υπολογιστική ισχύς από τους miners, ώστε να ανταπεξέλθουν. Η ενεργειακή αυτή κατανάλωση εκτός από ιδιαίτερα κοστοβόρα έχει και περιβαλλοντολογικό αντίκτυπο, γεγονός που καθιστά την επεκτασιμότητα ενός δικτύου blockchain εξαιρετικά δύσκολη. [6].

Για να αντιμετωπιστεί αυτό το πρόβλημα, έχει προταθεί ο μηχανισμός απόδειξης συμμετοχής (proof of stake, PoS) [8]. Με την απόδειξη συμμετοχής, ένας αλγόριθμος δικτύου καθορίζει ποιος κόμβος θα προσθέσει το μπλοκ στην αλυσίδα με βάση τη συμμετοχή του κόμβου, η συμμετοχή είναι ένας συνδυασμός παραμέτρων συμπεριλαμβανομένου του υπολοίπου λογαριασμού. Ο μηχανισμός PoS δεν απαιτεί μεγάλους υπολογισμούς, μειώνοντας έτσι τη χρήση ενέργειας στο δίκτυο. Αυτοί οι μηχανισμοί συναίνεσης είναι απαραίτητοι για ένα δημόσιο δίκτυο blockchain όπου οποιοσδήποτε στον κόσμο μπορεί να εκτελέσει έναν κόμβο και ενδεχομένως να τροποποιήσει την αλυσίδα. Ωστόσο, σε ένα ιδιωτικό δίκτυο με άδεια, αυτοί οι μηχανισμοί μπορούν να αντικατασταθούν με έναν απλό μηχανισμό απόδειξης εξουσιοδότησης (proof of authority, PoA) [9]. Το PoA είναι μια τροποποιημένη έκδοση του PoS με την ταυτότητα ως τη μόνη συμμετοχή. Έχει γίνει δοκιμή των έξυπνων συμβολαίων σε ένα ιδιωτικό δίκτυο δοκιμών χρησιμοποιώντας απόδειξη εξουσιοδότησης.



Εικόνα 2.4: *Blockchain Ethereum και έξυπνα συμβόλαια. 'Ενα δίκτυο blockchain αποτελείται από μια αποκεντρωμένη, κατανεμημένη ψηφιακή δομή που μοιράζεται σε ομότιμους κόμβους. Το σχηματικό σχήμα δικτύου που εμφανίζεται εδώ περιέχει τέσσερις κόμβους (υπολογιστές), ο καθένας από τους οποίους συγχρονίζει ένα αντίγραφο της αλυσίδας. Το δίκτυο είναι αποκεντρωμένο (δεν υπάρχει κεντρικό σημείο ελέγχου) και κατανεμημένο (η αλυσίδα αποθηκεύεται σε πολλαπλές φυσικές τοποθεσίες). β) 'Ενα blockchain μπορεί να απεικονιστεί ως μια σειρά από μπλοκ που συνδέονται με κρυπτογραφικούς κατακερματισμούς. Δηλαδή, τα περιεχόμενα κάθε μπλοκ περιλαμβάνουν τον κατακερματισμό των περιεχομένων του προηγούμενου μπλοκ. Τα μπλοκ περιέχουν επίσης δεδομένα σχετικά με την κατάσταση του δικτύου που είναι αποθηκευμένα στη ρίζα μιας δενδρικής δομής. Η δομή δεδομένων δένδρου κατάστασης στο Ethereum αποθηκεύει πληροφορίες για λογαριασμούς χρηστών και έξυπνων συμβολαίων. γ)Ο κώδικας για ένα συγκεκριμένο έξυπνο συμβόλαιο στεγάζεται σε μια διεύθυνση στο χώρο αποθήκευσης δικτύου και διατηρεί επίσης τη δική του αποθήκευση (για την αποθήκευση μεταβλητών, για παράδειγμα). Εδώ φαίνεται ένα διάγραμμα ροής που απεικονίζει τους αλγόριθμους εισαγωγής και ερωτημάτων στο έξυπνο συμβόλαιο για τη λύση ενός προβλήματος.*

2.2 Blockchain Ethereum και Έξυπνα Συμβόλαια

2.2.1 Εισαγωγή στο Ethereum

Κατα το αρχικό στάδιο δημιουργίας των blockchain, ο βασικός στόχος της τεχνολογίας ήταν η υποστήριξη ψηφιακών συναλλαγών με τη χρήση κρυπτονομισμάτων. Καθώς ο κόσμος αναγνώριζε την δύναμη της τεχνολογία των blockchain, γεννήθηκε η ανάγκη για ανάπτυξη σύνθετων εφαρμογών, οι οποίες θα τρέχουν πάνω στο blockchain. Για τις εφαρμογές αυτές, η επιλογή του Bitcoin δημιουργούσε σοβαρούς περιορισμούς ως προς την ανάπτυξη τους, λόγω των περιορισμένων ειδών συνναλαγών και το μέγεθος των δεδομένων που μπορούσαν να περάσουν στο blockchain [10].

Στα τέλη του 2013, ένας νεαρός Ρώσο-Καναδός προγραμματιστής με το όνομα Vitalik Buterin ανέπτυξε την ιδέα του Ethereum Blockchain σε White Paper [11]. Σύμφωνα με τον Buterin, το Ethereum δημιουργήθηκε με σκοπό να επιτρέψει μεγαλύτερη ευελιξία και ελευθερία στην ανάπτυξη εφαρμογών μέσω της πλατφόρμας του για αποκεντρωμένες εφαρμογές. Επίσης, δημιούργησε μια built-in Turing-Complete γλώσσα προγραμματισμού, μέσω της οποίας θα μπορούσαν να δημιουργηθούν εύκολα εφαρμογές πάνω στο Ethereum, ανοίγοντας έτσι και τον δρόμο για την χρήση έξυπνων συμβολαίων (smart contracts).

2.2.2 Γενική Δομή και Έξυπνα Συμβόλαια

Το Ethereum μπορεί να χειριστεί ένα ευρύ φάσμα συναλλαγών μέσω έξυπνων συμβολαίων, αυτοεκτελούμενων προγραμμάτων που εκτελούνται στην εικονική μηχανή Ethereum Virtual Machine (EVM) και διατηρούν την κατάσταση στη δική τους αποθήκευση [6]. Το EVM έχει αρχιτεκτονική βασισμένη σε στοίβα και μπορεί είτε να αποθηκεύσει πράγματα στη στοίβα (π.χ. λειτουργίες bytecode), στη μνήμη (π.χ. προσωρινές μεταβλητές εντός συναρτήσεων) είτε π.χ. σε μόνιμες μεταβλητές που κρατούν καταχωρήσεις βάσης δεδομένων. Κάθε έξυπνο συμβόλαιο μπορεί να διαβάζει και να γράφει μόνο στο δικό του αποθηκευτικό χώρο. Προκειμένου να αποθαρρυνθούν οι προγραμματιστές από τη σύνταξη αναποτελεσματικών ή δυσκίνητων έξυπνων συμβολαίων, υπάρχει ένα κόστος καυσίμου (gas) που σχετίζεται με κάθε εντολή αποθήκευσης και ανάκτησης. Ακριβώς όπως οι χρήστες blockchain έχουν μια διεύθυνση, η κατάσταση ενός έξυπνου συμβολαίου βρίσκεται σε μια συγκεκριμένη μοναδική διεύθυνση στην παγκόσμια κατάσταση του δικτύου Ethereum, την οποία οι χρήστες μπορούν να καλέσουν. Εάν ένας χρήστης δεν έχει αρκετό καύσιμο, η κλήση συμβολαίου και η αντίστοιχη συναλλαγή δεν μπορούν να ολοκληρωθούν. Τα έξυπνα συμβόλαια παρέχουν την ευκαιρία ανάπτυξης εφαρμογών με πολύπλοκη λειτουργικότητα σε ένα δίκτυο blockchain [12].

Για να έχουμε μια γενική ιδέα για την δομή του Ethereum την χωρίζουμε σε τρία μέρη: Το blockchain, τις ρίζες δένδρων δικτύου και τις δενδρικές δομές δεδομένων, όπως φαίνεται στην Εικόνα 2.4(β). Το blockchain καταγράφει την κατάσταση του δικτύου σε συγκεκριμένες στιγμές, αφού οι συναλλαγές έχουν αλλάξει την κατάσταση. Η κατάσταση του δικτύου αποθηκεύεται σε δενδρικές δομές (Merkle Patricia Trie), καθεμιά από τις οποίες έχει μια ανώτατη τιμή κατακερματισμού (top hash). Το Merkle Patricia Trie παρέχει μια κρυπτογραφικά επικυρωμένη δομή δεδομένων που μπορεί να χρησιμοποιηθεί για την αποθήκευση

όλων των σχέσεων (κλειδί, τιμή). Τα μπλοκ στην αλυσίδα αποθηκεύουν αυτές τις κορυφαίες τιμές κατακερματισμού, αλλά δεν αποθηκεύουν όλα τα δεδομένα στο δίκτυο των blockchain καθώς αυτό θα απαιτούσε μεγάλο όγκο. Τα δεδομένα κατάστασης αποθηκεύονται σε ένα επίπεδο βάσης δεδομένων χρησιμοποιώντας τη leveldb [13]. Τα δεδομένα λογαριασμού χρήστη και τα δεδομένα έξυπνων συμβολαίων (συμπεριλαμβανομένου του κώδικα και των πραγματικών δεδομένων που εισάγονται μέσω του κώδικα) αποθηκεύονται στις δενδρικές δομές δεδομένων, οι οποίες συγχρονίζονται μόνο από ¨πλήρεις¨ και ¨αρχειακούς¨ κόμβους που απαιτούν σημαντικά μεγαλύτερη υπολογιστική ισχύ και αποθήκευση [11]. Αυτοί οι κόμβοι είναι απαραίτητοι για την ομαλή λειτουργία του δικτύου. Ωστόσο, το πρωτόκολλο Ethereum περιέχει επίσης μια επιλογή ¨ελαφρού¨ κόμβου, στην οποία συγχρονίζονται μόνο οι κεφαλίδες μπλοκ.

2.3 Αποθήκευση Δεδομένων στο Ethereum Blockchain

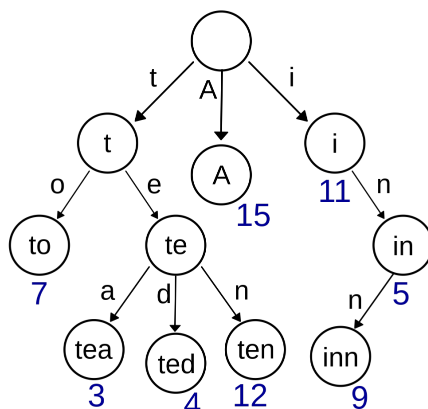
Μια σημαντική παράμετρος της τεχνολογίας του blockchain είναι η αποθήκευση δεδομένων. Το blockchain είναι μια κατανεμημένη δομή δεδομένων που επιτρέπει την ανταλλαγή πληροφοριών και αυτές οι πληροφορίες πρέπει να αποθηκεύονται στη δομή αυτή. Στη συγκεκριμένη ενότητα θα συζητηθεί η αποθήκευση δεδομένων στο Ethereum Blockchain. Για να είμαστε σε θέση να κατανοήσουμε τον τρόπο με τον οποίο το Ethereum αποθηκεύει δεδομένα, πρέπει πρώτα να μελετήσουμε το πως λειτουργούν οι επιμέρους δεντρικές δομές που χρησιμοποιεί, στις οποίες αναφερθήκαμε προηγουμένως.

2.3.1 Merkle Patricia Tries

Αναφερθήκαμε προηγουμένως στο ότι στο Ethereum Blockchain καταγράφεται η κατάσταση (state) του δικτύου κάθε φορά που ένα block συναλλαγών την έχει τροποποιήσει. Η κατάσταση αυτή, η οποία μπορεί να θεωρηθεί και σαν σχέση (κλειδί, τιμή) αποθηκεύεται μέσω μιας δομής την οποία το Ethereum προσδιορίζει ως Modified Merkle Patricia Trie (MPT). Η δομή αυτή αποτελεί μια μίξη της δομής των Merkle Trees και Patricia Tries. Για να καταλήξουμε στο πως λειτουργούν τα MPT πρέπει πρώτα να κατανοήσουμε τα χαρακτηριστικά των επιμέρους αυτών δομών [14].

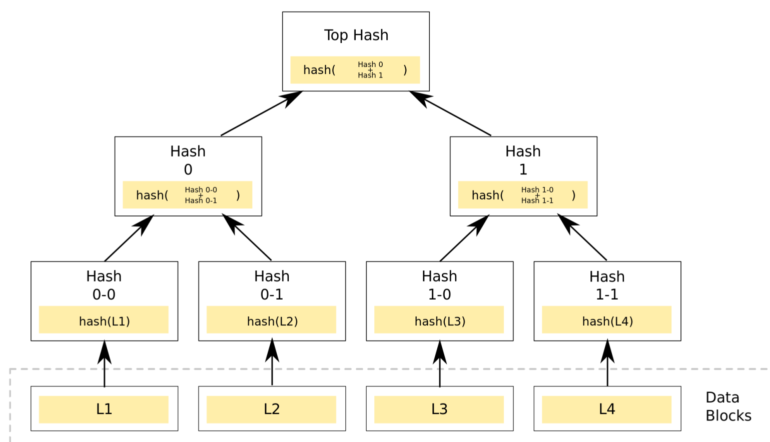
Patricia trie ή prefix tree ή radix tree ή απλά trie είναι μια δομή (κλειδί, τιμή), κατά την οποία τα κλειδιά που έχουν κοινό πρόθεμα (prefix), έχουν και το ίδιο μονοπάτι. Το γεγονός αυτό την καθιστά καλή επιλογή στην περίπτωση ύπαρξη κλειδιών με κοινά προθέματα προσφέροντας ταχύτητα στην αναζήτηση, δεσμεύοντας παράλληλα μικρά τμήματα μνήμης. Έτσι, στο παράδειγμα της εικόνας 2.5 για την εύρεση της τιμής των κλειδιών "tea" και "ted" ακολουθούμε το ίδιο αρχικό μονοπάτι ($\rightarrow t \rightarrow e$).

Merkle tree ή αλλιώς hash tree είναι ένα δέντρο, το οποίο έχει δύο τύπους κόμβων. Τα ¨φύλλα¨ (leaves), τα οποία παίρνουν την τιμή κατακερματισμού (hash) ενός μπλοκ δεδομένων, και όλους τους κόμβους που δεν είναι ¨φύλλα¨ των οποίων η τιμή προκύπτει από το hash ενός συνδυασμού των τιμών των παιδιών τους. Το μεγάλο πλεονέκτημα των Merkle trees, πηγάζει από το γεγονός ότι μπορούμε πολύ γρήγορα και εύκολα να ελέγξουμε αν πολλά μπλοκ δεδομένων είναι ίδια ένα προς ένα με άλλα μπλοκ δεδομένων απλά συγκρίνο-



Εικόνα 2.5: *Patricia Trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".*

ντας το Top Hash, δηλαδή τη ρίζα των δύο Merkle Trees, που σχηματίζουν. Ως αποτέλεσμα, η επικύρωση δεδομένων γίνεται πολύ γρηγορότερη, χωρίς να υπάρχει η ανάγκη να έχουμε όλη την πληροφορία. Επίσης εάν οι ρίζες δεν είναι ίδιες μπορούμε να βρούμε, με λογαριθμικά μικρότερη πολυπλοκότητα από το να ανατρέξουμε σε όλα τα δεδομένα ένα προς ένα, το μπλοκ δεδομένων που είναι διαφορετικό συγκρίνοντας προοδευτικά τα επιμέρους hash των εκάστοτε παιδιών .

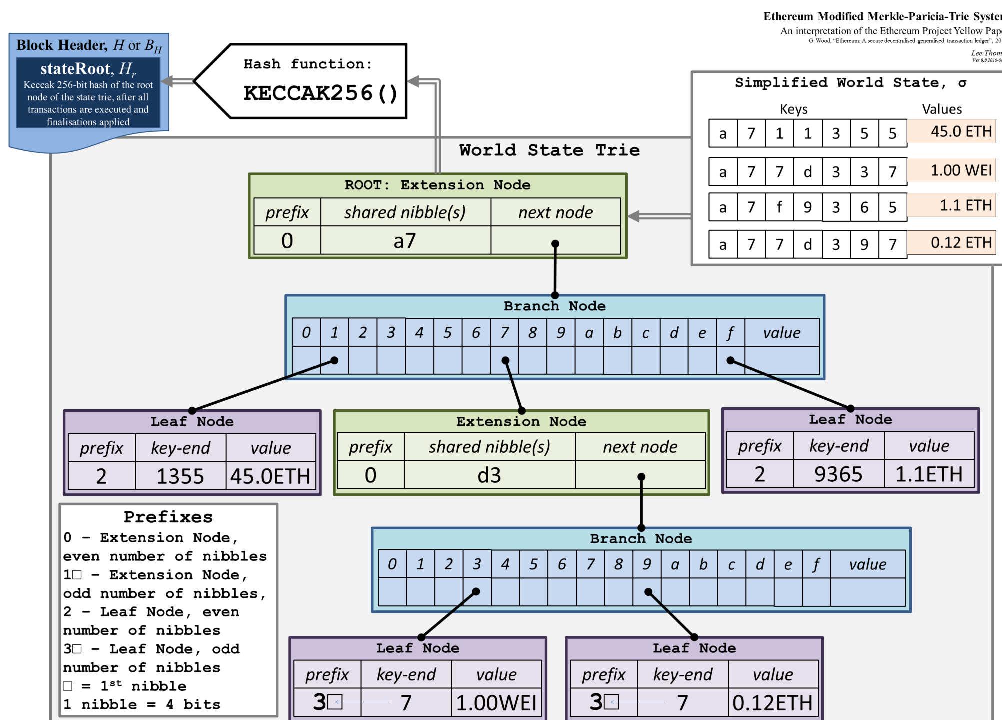


Εικόνα 2.6: *Merkle Tree όπου το hash των γονιών προκύπτει από πρόσθεση των αντίστοιχων hashes των παιδιών*

Όπως αναφέραμε προηγουμένως, το MPT που χρησιμοποιείται στο Ethereum, είναι μια μίξη των δύο δομών που προαναφέραμε. Συγκεκριμένα, όπως και στο Merkle tree, και εδώ οι κόμβοι έχουν κλειδί που προκύπτει από κρυπτογραφικό κατακερματισμό (hash) των περιεχομένων τους σε λογική αποθήκευσης δεδομένων(κλειδί, τιμή). Ο όρος Merkle υπονοεί ότι ο κόμβος ρίζα συνιστά κρυπτογραφικό ¨αποτύπωμα¨ όλης της δομής. Επίσης, όπως και στο Patricia trie και εδώ έχουμε λογική κοινού προθέματος στα κλειδιά με λίγο διαφορετική δομή. Αφού τα κλειδιά είναι hash values, αυτό σημαίνει ότι σε κάθε βήμα της δεντρικής δομής αυτής θα υπάρχουν 16 διαφορετικά πιθανά μονοπάτια, τα οποία συμβολίζονται με διαφορετικά 16-αδικό χαρακτήρα (nibble). Συγκεκριμένα, στο MPT, όπως φαίνεται και στην

Εικόνα 2.7 ένας κόμβος μπορεί να είναι ένα από τα ακόλουθα [15] :

- Empty Node : Ο κόμβος αναπαριστάται με κενή συμβολοσειρά
- Branch Node : Είναι κόμβος με 17 αντικείμενα. 16 από αυτά αντικατοπτρίζουν ένα δείκτη για το επόμενο βήμα στο πρόθεμα κλειδιού και το 1 σχετίζεται με την τιμή του ίδιου του κόμβου.
- Extension Node : Είναι κόμβος ο οποίος χρησιμοποιείται για την ομαδοποίηση κοινών μερικών μονοπατιών. Δηλαδή για κλειδιά με περισσότερα από ένα κοινά nibble, δεν χρειάζεται να υπάρχουν απαραίτητα αντίστοιχα τόσα παιδιά στο δέντρο όταν κάνουμε την αναζήτηση. Ουσιαστικά, χωρίς αυτού του τύπου τους κόμβους στη θέση τους θα υπήρχαν αντίστοιχα πολλοί Branch Nodes, όσος και ο αριθμός των κοινών nibbles στο κλειδί. Ως αποτέλεσμα, η ύπαρξη των κόμβων αυτών αντιμετωπίζει το μειονέκτημα των Patricia Tries για αργή αναζήτηση όταν υπάρχει μεγάλη ομοιότητα προθεμάτων μεταξύ των κλειδιών.
- Leaf Node : Οι καταληκτικοί αυτόι κόμβοι συνιστούν το τέλος των υπάρχοντων μονοπατιών και περιέχουν τις τιμές των ζευγών (κλειδί, τιμή).



Εικόνα 2.7: *Παράδειγμα Ethereum Modified Merkle Patricia Trie*

Η δομή του MPT αλλάζει δυναμικά με κάθε νέα εισαγωγή στοιχείων και προσαρμόζεται έτσι ώστε η αναζήτηση να απαιτεί το μικρότερο δυνατό βάθος αναζήτησης. Συγκεκριμένα, , μετά από νέα εισαγωγή (κλειδιού,τιμής) ακολουθείται από την ρίζα το μονοπάτι του κλειδιού και ανάλογα που θα σταματήσει η αναζήτηση, οι γενικοί κανόνες για την ανανέωση των MPTs είναι οι ακόλουθοι:

- Αν η αναζήτηση σταματήσει σε Empty Node, τότε αυτός αντικαθίσταται από Leaf Node με το εναπομείναν μονοπάτι.
- Αν σταματήσει σε Leaf Node, τότε αυτός μετατρέπεται σε Extension Node και προστίθενται αντίστοιχα ένας νέος Branch Node και ένας Leaf Node με την κατάληξη του εισαγόμενου κλειδιού.
- Αν σταματήσει σε Extension Node, τότε αυτός μετατρέπεται σε διαφορετικό Extension Node με μικρότερο μονοπάτι και συνδέεται στην συνέχεια με ένα νέο Branch Node.
- Αν σταματήσει σε Branch Node, τότε προφανώς προστίθεται νέος Leaf Node, με το εναπομείναν μονοπάτι.

Συνεπώς, με τα παραπάνω χαρακτηριστικά, παρατηρούμε ότι το MPT είναι μια δομή που αποθηκεύει ζευγάρια (κλειδί, τιμή) με τέτοιο τρόπο, ώστε να είναι εύκολη η πιστοποίηση της ακεραιότητας των δεδομένων, κρατώντας παράλληλα σε χαμηλά επίπεδα την πολυπλοκότητα της αναζήτησης. Συγκεκριμένα, έχουμε εγγυημένη λογαριθμική πολυπλοκότητα για εισαγωγές, διαγραφές, επικυρώσεις και τροποποιήσεις. Μια τελευταία λεπτομέρεια των MPTs είναι η κρυπτογράφηση των κλειδιών πριν την εισαγωγή, ώστε αυτά να είναι όσο το δυνατόν καλύτερα 'ισορροπημένα' στο δέντρο. Για τους λόγους αυτούς το Ethereum επιλέγει να τα χρησιμοποιεί στο επίπεδο αποθήκευσης του, το οποίο θα αναλύσουμε στην συνέχεια.

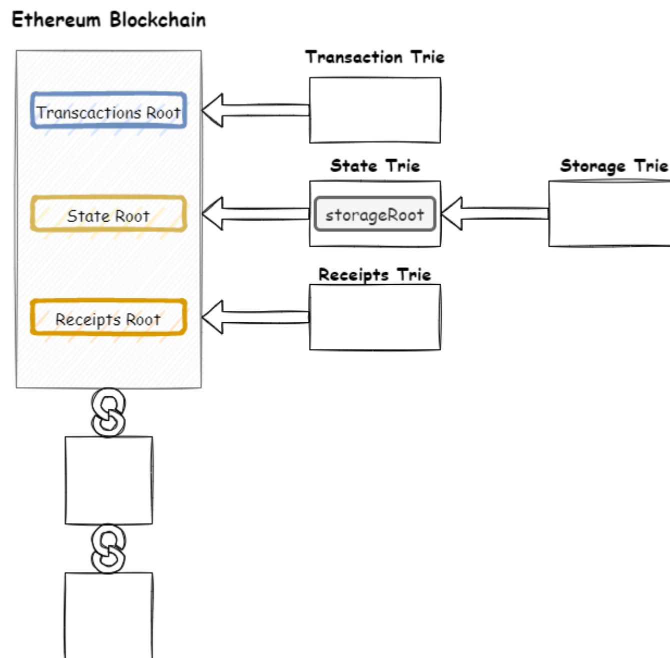
2.3.2 Η Δομή του Επιπέδου Αποθήκευσης στο Ethereum

Το Ethereum είναι μια πλατφόρμα blockchain ανοιχτού κώδικα που επιτρέπει τη δημιουργία αποκεντρωμένων εφαρμογών και είναι ένα δημόσιο κατανεμημένο δίκτυο blockchain. Ο κεντρικός του σκοπός είναι η παροχή μιας πλατφόρμας για απρόσκοπτη εκτέλεση κώδικα για οποιαδήποτε αποκεντρωμένη εφαρμογή και για το λόγο αυτό έχει ιδιαίτερο ενδιάφερον να ασχοληθούμε με το τρόπο με τον οποίο αυτό αποθηκεύει τα δεδομένα.

Τα δεδομένα στο Ethereum blockchain αποθηκεύονται χρησιμοποιώντας δενδρικές δομές και συγκεκριμένα Modified Merkle Patricia Tries, για τα οποία μιλήσαμε προηγουμένως. Είναι απόλυτα θεμιτό να αποθηκεύονται χωριστά μόνιμα δεδομένα όπως η εξορυσσόμενη συναλλαγή και προσωρινά δεδομένα όπως το υπόλοιπο λογαριασμού. Το Ethereum χρησιμοποιεί αυτή τη δενδρική δομή δεδομένων για τη διαχείριση προσωρινών και μόνιμων δεδομένων. Όπως έχουμε προαναφέρει, το Ethereum αποτελείται από καταγραφές καταστάσεων, οι οποίες πυροδοτούνται από την εξόρυξη νέων συναλλαγών. Για να είναι εφικτή η ξεχωριστή αποθήκευση προσωρινών και μόνιμων δεδομένων, το Ethereum, χρησιμοποιεί τέσσερα διαφορετικά MPTs, όπως φαίνεται και στην εικόνα 2.8, και είναι τα ακόλουθα [4] :

- Δένδρο κατάστασης (State Trie)
- Δένδρο αποθήκευσης (Storage Trie)
- Δένδρο συναλλαγών (Transactions Trie)
- Δέντρο αποδείξεων (Receipts Trie)

Ένα αξιοσημείωτο σημείο εδώ είναι ότι δεδομένα όπως το υπόλοιπο λογαριασμού δεν αποθηκεύονται απευθείας στα μπλοκ. Μόνο οι κατακερματισμοί του ριζικού κόμβου του δένδρου συναλλαγής, του δένδρου αποδείξεων και των δένδρων κατάστασης αποθηκεύονται στο blockchain. Η λειτουργία αποθήκευσης του Ethereum blockchain είναι αρκετά απλή. Η διαίρεση μόνιμων και προσωρινών δεδομένων σε διαφορετικά δένδρα διασφαλίζει ότι τα δεδομένα είναι ασφαλή και εύκολα διαχειρίσιμα, όπως και τα δεδομένα που αποθηκεύονται στο blockchain ως έξυπνα συμβόλαια είναι επίσης ασφαλή και εύκολα προσβάσιμα [11].



Εικόνα 2.8: *Δομή Αποθήκευσης Δεδομένων στο Ethereum*

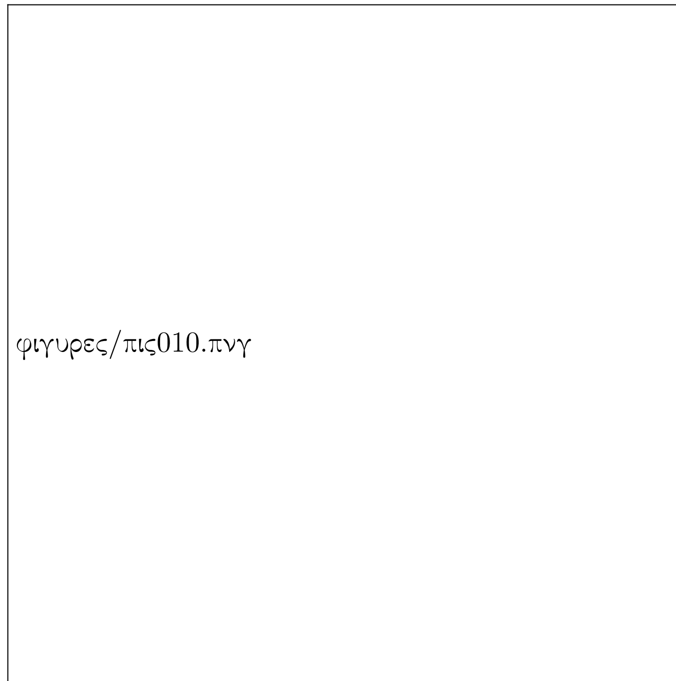
Σε ότι αφορά το Δένδρο κατάστασης στο Ethereum, υπάρχει το Global State Trie, ςαντο οποίο είναι μοναδικό και ενημερώνεται συνεχώς. Το State Trie αποτελείται από ένα κλειδί και μια τιμή για κάθε λογαριασμό που υπάρχει στο Ethereum. Το κλειδί είναι ένα αναγνωριστικό 160bit. Η τιμή προκύπτει από την κρυπτογράφηση των ακόλουθων πεδίων: nonce, balance, codeHash και storageRoot.

Στο Δένδρο αποθήκευσης αποθηκεύονται τα δεδομένα του συμβολαίου. Κάθε λογαριασμός Ethereum έχει το δικό του δένδρο αποθήκευσης. Ο κατακερματισμός 256bit του ριζικού κόμβου αποθηκεύεται στο καθολικό δένδρο κατάστασης ως storageRoot.

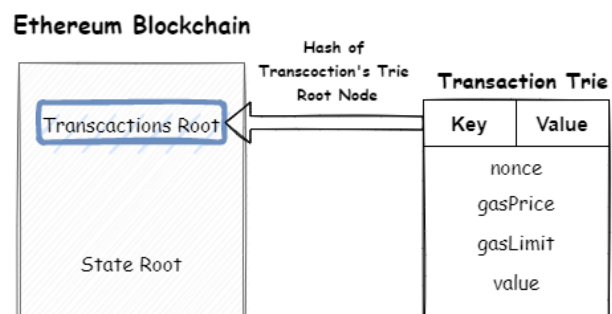
Υπάρχει ένα δένδρο συναλλαγών σε κάθε μπλοκ Ethereum. Η σειρά της συναλλαγής που υπάρχει σε κάθε μπλοκ βασίζεται στην πορεία προσθήκης της συναλλαγής στο μπλοκ και στον εκάστοτε εξορύκτη(miner). Η θέση ενός μπλοκ δεν αλλάζει ποτέ, οπότε και η θέση της εκάστοτε συναλλαγής παραμένει σταθερή, καθιστώντας έτσι δυνατό τον εντοπισμό της, εφόσον είναι προσβάσιμη η διαδρομή που οδηγεί σε αυτήν.

2.4 To Go Ethereum (Geth)

Το Go Ethereum είναι η μια από τις τρεις πρωτότυπες υλοποιήσεις του πρωτοκόλλου Ethereum. Έχει γραφεί σε γλώσσα Go, πλήρως ανοικτού κώδικα και με άδεια GNU LGPL



Εικόνα 2.9: *Επισκόπηση State Trie - Storage Trie*



Εικόνα 2.10: *Επισκόπηση Transactions Trie*

v3. Οι άλλες δυο υλοποιήσεις είναι σε C++ και Python. Το Go Ethereum είναι διαθέσιμο για μεταφόρτωση είτε ως ανεξάρτητος client που ονομάζεται Geth και μπορεί να εγκατασταθεί πρακτικά σε κάθε λειτουργικό σύστημα, είτε ως βιβλιοθήκη που μπορεί να ενσωματωθεί σε Go, Android ή iOS projects [16].

Όταν κυκλοφόρησε το Ethereum, υπήρχε δυνατότητα επιλογής ανάμεσα σε δύο διαφορετικούς τρόπους συγχρονισμού του δικτύου: πλήρης συγχρονισμός (full sync) και γρήγορος συγχρονισμός (fast sync) στον οποίο παραλείπονται οι ελαφρείς πελάτες. Ο πλήρης συγχρονισμός λειτουργεί με λήψη ολόκληρης της αλυσίδας και εκτέλεση όλων των συναλλαγών, αντίθετα ο γρήγορος συγχρονισμός τοποθετεί μια αρχική εμπιστοσύνη σε ένα πρόσφατο μπλοκ και κατεβάζει απευθείας την κατάσταση που σχετίζεται με αυτό (μετά από την οποία αλλάζει στην εκτέλεση μπλοκ όπως ο πλήρης συγχρονισμός). Παρόλο που και οι δύο τρόποι λειτουργίας κατέληξαν στο ίδιο τελικό σύνολο δεδομένων, επιλέγουν διαφορετικό εύρος συμβιβασμών [17]:

Ο πλήρης συγχρονισμός ελαχιστοποιεί την εμπιστοσύνη, επιλέγοντας να εκτελέσει όλες τις συναλλαγές από τη γένεση μέχρι την κεφαλή. Αν και μπορεί να είναι η πιο ασφαλής επιλογή, το Ethereum mainnet περιέχει επί του παρόντος πάνω από 1,03 δισεκατομμύρια συναλλαγές, με ρυθμό αύξησης 1,25 εκατομμυρίων / ημέρα. Η επιλογή να εκτελεσθούν τα πάντα από τη γένεση σημαίνει ότι ο πλήρης συγχρονισμός έχει ένα διαρκώς αυξανόμενο κόστος. Προς το παρόν χρειάζονται 8-10 ημέρες για την επεξεργασία όλων αυτών των συναλλαγών σε ένα αρκετά ισχυρό μηχάνημα .

Ο γρήγορος συγχρονισμός επέλεξε να βασίζεται στην ασφάλεια των PoW. Αντί να εκτελούνται όλες οι συναλλαγές από την αρχή, υπέθεσε ότι ένα μπλοκ με 64 έγκυρα PoW στην κορυφή θα ήταν απαγορευτικά ακριβό για να το κατασκευάσει κάποιος, ως εκ τούτου, αρκεί να κατέβει η κατάσταση που σχετίζεται με το HEAD-64. Ως αποτέλεσμα, στον γρήγορο συγχρονισμό εμπιστευόμενοι τη ρίζα κατάστασης από ένα πρόσφατο μπλοκ, γίνεται να κατεβάζουμε κατευθείαν το State Trie παραλείποντας όλες τις προηγούμενες καταστάσεις. Αυτό αντικατέστησε την ανάγκη για CPU IO δίσκου με την ανάγκη για εύρος ζώνης δικτύου και καθυστέρηση. Συγκεκριμένα, το mainnet του Ethereum περιέχει επί του παρόντος περίπου 675 εκατομμύρια κόμβους δένδρων κατάστασης, που χρειάζονται περίπου 8-10 ώρες για τη λήψη σε ένα αρκετά γρήγορο συνδεδεμένο μηχάνημα.

Ο πλήρης συγχρονισμός παρέμεινε διαθέσιμος για οποιονδήποτε ήθελε να δαπανήσει τους πόρους για να επαληθεύσει ολόκληρο το ιστορικό του Ethereum, αλλά για τους περισσότερους ανθρώπους, ο γρήγορος συγχρονισμός ήταν κάτι παραπάνω από επαρκής. Υπάρχει ένα παράδοξο της επιστήμης των υπολογιστών, ότι όταν ένα σύστημα φτάσει να υλοποιεί 50-πλάσια χρήση από αυτή για την οποία σχεδιάστηκε, θα χαλάσει. Η λογική είναι ότι άσχετο πώς δουλεύει κάτι, πιέστε το αρκετά και θα εμφανιστεί ένα απρόβλεπτο σημείο συμφόρησης.

Στην περίπτωση του γρήγορου συγχρονισμού, το απρόβλεπτο σημείο συμφόρησης ήταν η καθυστέρηση, που προκλήθηκε από το μοντέλο δεδομένων του Ethereum. Το δέντρο κατάστασης του Ethereum είναι ένα MPT, όπου τα φύλλα περιέχουν τα χρήσιμα δεδομένα και κάθε κόμβος παραπάνω είναι ο κατακερματισμός 16 πιθανών παιδιών. Συγχρονίζοντας από τη ρίζα του δέντρου (το hash που είναι ενσωματωμένο σε μια κεφαλίδα μπλοκ), ο μόνος τρόπος για να κατέβουν τα πάντα είναι να ζητηθεί κάθε κόμβος ένας προς ένας. Με 675 εκατομμύρια κόμβους για λήψη, ακόμη και με τη συγκέντρωση 384 αιτημάτων μαζί, καταλήγει

να χρειάζεται 1,75 εκατομμύρια χτύπηματα με τις επιστροφές(round trips). Υποθέτοντας ένα υπερβολικά γενναιόδωρο 50μs RTT σε 10 ομότιμους κόμβους, ο γρήγορος συγχρονισμός ουσιαστικά περιμένει πάνω από 150 λεπτά για την άφιξη των δεδομένων. Αυτή η καθυστέρηση δικτύου(network latency) είναι μόνο ένα από τα τρία μέρη του προβλήματος.

Όταν ένας ομότιμος εξυπηρετητής λαμβάνει ένα αίτημα για κόμβους δένδρων, πρέπει να τους ανακτήσει από το δίσκο. Το Merkle Patricia Trie του Ethereum δεν βοηθά ούτε σε αυτήν την περίπτωση. Εφόσον οι κόμβοι δένδρων δεικτοδοτούνται με κατακερματισμό (keyed by hash), δεν υπάρχει ουσιαστικός τρόπος αποθήκευσης και ανάκτησής τους ομαδικά, καθένας από τους οποίους απαιτεί την ανάγνωση της δικής του βάσης δεδομένων. Για να γίνουν τα πράγματα χειρότερα, το LevelDB (χρησιμοποιείται από το Geth) αποθηκεύει δεδομένα σε 7 επίπεδα, επομένως μια τυχαία ανάγνωση θα αγγίζει γενικά πολλά αρχεία. Πολλαπλασιάζοντάς τα όλα προς τα πάνω, ένα μόνο αίτημα δικτύου 384 κόμβων - σε 7 αναδυόμενες αναγνώσεις - ανέρχεται σε 2,7 χιλιάδες αναγνώσεις δίσκου. Με την ταχύτερη ταχύτητα SATA SSD των 100.000 IOPS, αυτό είναι 37 ms επιπλέον καθυστέρηση. Με την ίδια υπόθεση 10 ομότιμων κόμβων όπως παραπάνω, ο γρήγορος συγχρονισμός με τον τρόπο αυτό πρόσθεσε επιπλέον 108 λεπτά χρόνο αναμονής. Αλλά και η καθυστέρηση εξυπηρέτησης (serving latency) είναι μόνο ένα από τα τρία μέρη του προβλήματος.

Η αποστολή αιτημάτων για λήψη τόσων πολλών κόμβων δένδρων μεμονωμένα συνεπάγεται στην πραγματικότητα την ανάγκη να ανέβουν (upload) αντίστοιχα πολλά hash. Με 675 εκατομμύρια κόμβους για λήψη, αυτό είναι 675 εκατομμύρια hashes για μεταφόρτωση ή 675 M * 32 byte = 21 GB. Με παγκόσμιο μέσο όρο ταχύτητας μεταφόρτωσης 51 Mbps ο γρήγορος συγχρονισμός προσθέτει έτσι περίπου 56 λεπτά επιπλέον χρόνο αναμονής. Οι λήψεις είναι λίγο περισσότερες από δύο φορές μεγαλύτερες, επομένως με παγκόσμιους μέσους όρους 97 Mbps, ο γρήγορος συγχρονισμός καθυστερεί για άλλα 63 λεπτά. Οι καθυστερήσεις εύρους ζώνης (bandwidth latency) είναι το τελευταίο 1/3 του προβλήματος.

Συνοψίζοντας, στην καλύτερη περίπτωση ο γρήγορος συγχρονισμός δαπανά 6,3 ώρες χωρίς να κάνει τίποτα, απλώς περιμένοντας δεδομένα εφόσον: Υπάρχει σύνδεση δικτύου άνω του μέσου όρου, υπάρχει επαρκής αριθμός ομότιμων κόμβων που επικοινωνούν (peers) και επίσης όταν οι ομότιμοι δεν εξυπηρετούν κανέναν άλλο εκείνη τη στιγμή.

2.4.1 Επιτάχυνση Snapshot

Εξηγήσαμε παραπάνω γιατί το Ethereum αποθηκεύει την κατάστασή του σε Merkle Patricia Trie. Το κόστος που πληρώνουμε για να έχουμε λογαριθμική ενημέρωση και επαλήθευση είναι λογαριθμικές αναγνώσεις και λογαριθμική αποθήκευση για κάθε μεμονωμένο κλειδί. Αυτό συμβαίνει επειδή κάθε εσωτερικός κόμβος δένδρου πρέπει να αποθηκευτεί στο δίσκο ξεχωριστά [18].

Το 2019 που υπολογίστηκε, το βάθος του δένδρου λογαριασμών που παρατηρήθηκε περισσότερο ήταν το 7. Αυτό σημαίνει ότι κάθε λειτουργία δένδρου (π.χ. ανάγνωση υπολοίπου, εγγραφή) αγγίζει τουλάχιστον 7-8 εσωτερικούς κόμβους, έτσι θα πραγματοποιηθούν τουλάχιστον 7-8 προσβάσεις στη βάση δεδομένων. Επίσης, η LevelDB οργανώνει τα δεδομένα της σε μια δομή με μέγιστο 7 επιπέδων, οπότε υπάρχει ένας επιπλέον πολλαπλασιαστής από εκεί. Το καθαρό αποτέλεσμα είναι ότι μια πρόσβαση σε μία μεμονωμένη κατάσταση αναμένεται να

οδηγεί σε 25-50 τυχαίες προσβάσεις στο δίσκο. Πολλαπλασιάζοντας με όλες τις αναγνώσεις και εγγραφές κατάστασης που προκαλούν όλες οι συναλλαγές σε ένα μπλοκ, το αποτέλεσμα είναι ένας πολύ μεγάλος αριθμός. Αυτό είναι το κόστος λειτουργίας ενός κόμβου Ethereum ώστε να διατηρείται η δυνατότητα κρυπτογραφικής επαλήθευσης των καταστάσεων ανά πάσα στιγμή.

Το Ethereum βασίζεται σε κρυπτογραφικές αποδείξεις για την κατάστασή του. Δεν υπάρχει τρόπος παράκαμψης των προσβάσεων στο δίσκο εάν είναι επιθυμητή η επαλήθευση όλων των δεδομένων. Βέβαια, τα ήδη επαληθευμένα δεδομένα μπορούν να θεωρηθούν και θεωρούνται αξιόπιστα.

Δεν υπάρχει λόγος συνεχόμενων επαληθεύσεων σε κάθε στοιχείο κατάστασης, κάθε φορά που διαβάζεται από το δίσκο. Το Merkle Patricia Trie είναι απαραίτητο για εγγραφές, αλλά δημιουργεί επιβάρυνση (overhead) στην ανάγνωση. Δεν μπορούμε να απαλλαγούμε από αυτό, και δεν μπορούμε να το ¨αδυνατίσουμε¨, αλλά αυτό δεν σημαίνει ότι πρέπει απαραίτητα να το χρησιμοποιούμε παντού. Αξίζει να δούμε σε ποιες περιπτώσεις ένας κόμβος Ethereum ζητά πρόσβαση σε κατάσταση. Αυτές είναι :

Κατά την εισαγωγή ενός νέου μπλοκ, η εκτέλεση κώδικα στο EVM πραγματοποιεί έναν ισορροπημένο αριθμό αναγνώσεων και εγγραφών της κατάστασης. Ωστόσο, ένα μπλοκ άρνησης υπηρεσίας (denial-of-service) μπορεί να κάνει πολύ περισσότερες αναγνώσεις παρά εγγραφές.

Όταν ένας χειριστής κόμβου ανακτά την κατάσταση (π.χ. με eth_call και συναφείς συναρτήσεις), η εκτέλεση κώδικα EVM κάνει μόνο αναγνώσεις (μπορεί επίσης να κάνει εγγραφές, αλλά αυτές απορρίπτονται στο τέλος και δεν διατηρούνται).

Όταν ένας κόμβος συγχρονίζεται, ζητά κατάσταση από απομακρυσμένους κόμβους που πρέπει να τον ανακαλύψουν και να τον εξυπηρετήσουν μέσω του δικτύου.

Με βάση τα παραπάνω μοτίβα πρόσβασης, εάν μπορούν να αποφευχθούν οι προσπελάσεις των αναγνώσεων στο δένδρο κατάστασης, μια σειρά λειτουργιών κόμβου θα γίνει σημαντικά ταχύτερη. Θα μπορούσε ακόμη και να επιτρέψει ορισμένα νέα μοτίβα πρόσβασης (όπως το state iteration) που ήταν απαγορευτικά ακριβή προηγουμένως. Φυσικά, υπάρχει πάντα ένα trade-off που πρέπει πάντοτε να λαμβάνεται υπόψη. Χωρίς την απαλλαγή από το trie, οποιαδήποτε νέα δομή επιτάχυνσης δημιουργεί επιπλέον επιβάρυνση και πρέπει να αναρωτηθούμε εαν αξίζει να επομιστούμε αυτό το επιπλέον βάρος.

Το Merkle Patricia Trie λειτουργεί αποδοτικά για τις εγγραφές, αλλά υπάρχει πρόβλημα στις αναγνώσεις. Όπως αναφέρθηκε, η θεωρητική ιδανική αποθήκευση δεδομένων για την κατάσταση του Ethereum θα ήταν είναι ένας απλός flat χώρος αποθήκευσης κλειδιού-τιμής (ξεχωριστός για λογαριασμούς και κάθε συμβόλαιο). Ωστόσο, χωρίς τους περιορισμούς του Merkle Patricia Trie, αυτή η ιδανική λύση μπορεί να εφαρμοσθεί.

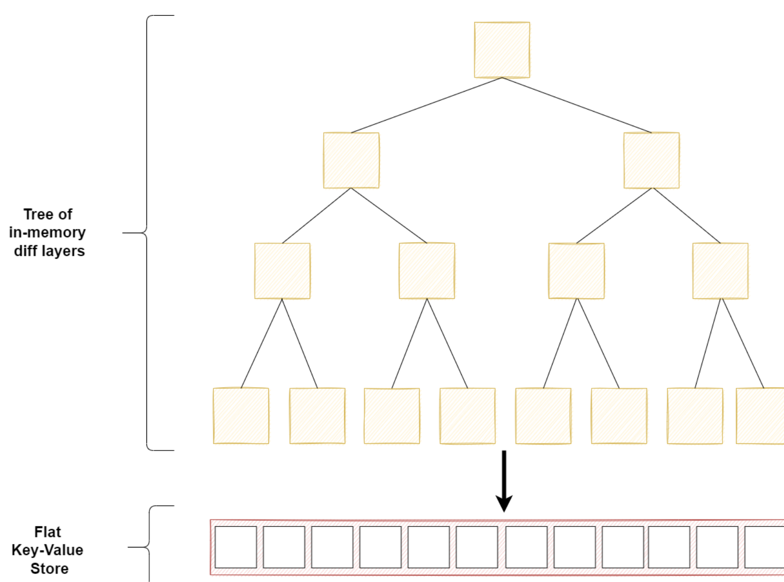
Σε αυτό το πλαίσιο, το Geth εισήγαγε και παρουσίασε τη δομή της επιτάχυνσης στιγμιότυπου (Snapshot Acceleration) [18]. Ένα στιγμιότυπο είναι μια πλήρης προβολή της κατάστασης Ethereum σε ένα δεδομένο μπλοκ. Μπορούμε να θεωρήσουμε το snapshot ως μια συλλογή όλων των λογαριασμών και των υποδοχών αποθήκευσης, που αντιπροσωπεύονται από ένα επίπεδο χώρο αποθήκευσης τύπου 'κλειδί, τιμή' (flat key-value store). Έτσι, όποτε επιθυμούμε πρόσβαση σε έναν λογαριασμό ή σε μια υποδοχή αποθήκευσης, το κόστος είναι μόνο 1 αναζήτηση LevelDB αντί για 7-8 που απαιτούνταν στο δένδρο. Η ενημέρωση

του στιγμιότυπου είναι επίσης απλή θεωρητικά, αφού επεξεργαστούμε ένα μπλοκ κάνουμε 1 επιπλέον εγγραφή LevelDB ανά ενημερωμένη υποδοχή.

Το snapshot ουσιαστικά μειώνει την πολυπλοκότητα των αναγνώσεων από O(log n) σε O(1), ενώ αυξάνει την πολυπλοκότητα εγγραφών από O(log n) σε O(1 + log n). Προφανώς και στις δύο περιπτώσεις έχουμε και ένα πολλαπλασιαστικό παράγοντα(περίπου 7) που προκύπτει από το overhead που εισάγει η LevelDB. Επίσης προκαλείται αύξηση του χώρου που απαιτείται στο δίσκο από O(n log n) σε O(n + n log n).

Η διατήρηση ενός χρησιμοποιήσιμου στιγμιότυπου της κατάστασης Ethereum έχει ωστόσο την πολυπλοκότητά της. Όσο τα μπλοκ έρχονται το ένα μετά το άλλο, χτίζοντας πάντα πάνω από το τελευταίο, η αφελής προσέγγιση της συγχώνευσης των αλλαγών στο snapshot λειτουργεί. Ωστόσο, εάν υπάρχει μια ελάχιστη αναδιοργάνωση έστω και ενός μπλοκ, δημιουργείται πρόβλημα γιατί δεν υπάρχει αναίρεση σε επίπεδη αναπαράσταση δεδομένων. Για να γίνουν τα πράγματα χειρότερα, είναι αδύνατη η πρόσβαση σε παλαιότερη κατάσταση (π.χ. 3 μπλοκ παλιά για κάποιο DApp ή 64+ για fast sync). Για να ξεπεραστεί αυτός ο περιορισμός, το στιγμιότυπο του Geth αποτελείται από δύο οντότητες: ένα persistent στρώμα δίσκου που είναι ένα πλήρες στιγμιότυπο ενός παλαιότερου μπλοκ (π.χ. HEAD-128) και ένα δέντρο από επίπεδα διαφορών στη μνήμη που συγκεντρώνει τις εγγραφές στην κορυφή.

Κάθε φορά που υποβάλλεται σε επεξεργασία ένα νέο μπλοκ, δεν συγχωνεύονται οι εγγραφές απευθείας στο επίπεδο του δίσκου, αλλά δημιουργείται ένα νέο επίπεδο διαφοράς στη μνήμη το οποίο περιέχει τις αλλαγές, όπως φαίνεται αφηρημένα στην εικόνα 2.11. Όταν έχουν συγκεντρωθεί αρκετά επίπεδα διαφοράς στη μνήμη στην κορυφή, τα κάτω αρχίζουν να συγχωνεύονται μεταξύ τους και τελικά ωθούνται στο δίσκο. Κάθε φορά που πρόκειται να αναγνωσθεί ένα στοιχείο κατάστασης, η ανάγνωση ξεκινά από το ανώτερο επίπεδο διαφοράς και συνεχίζει προς τα πίσω μέχρι να βρεθεί ή να φτάσει στο δίσκο.



Εικόνα 2.11: *Επισκόπηση Δομής Snapshot Acceleration*

Αυτή η αναπαράσταση δεδομένων είναι πολύ ισχυρή καθώς επιλύει πολλά ζητήματα. Δεδομένου ότι τα επίπεδα διαφοράς στη μνήμη συναρμολογούνται σε ένα δέντρο, οι αναδιοργανώσεις που είναι μικρότερες από 128 μπλοκ μπορούν απλώς να επιλέξουν το επίπεδο

διαφοράς που ανήκει στο γονικό μπλοκ και να δημιουργήσουν από εκεί προς τα εμπρός. Οι εφαρμογές DA (DApps) και οι απομακρυσμένοι συγχρονιστές που χρειάζονται παλαιότερη κατάσταση έχουν πρόσβαση σε 128 πρόσφατες. Το κόστος όντως αυξάνεται κατά 128 αναζητήσεις αντιστοιχήσεων, αλλά οι 128 αναζητήσεις in-memory είναι ταχύτερες κατά τάξεις μεγέθους από 8 αναγνώσεις δίσκου που πολλαπλασιάζονται 4x-5x από τη LevelDB. Σημειώνουμε βέβαια, ότι σε περίπτωση που πιθανή αναδιοργάνωση είναι βαθύτερη από το peristent στρώμα δικτύου, τότε το snapshot πρέπει να δημιουργηθεί από την αρχή, κάτι που είναι ιδιαίτερα ¨ακριβό¨.

Η δομή επιτάχυνσης στιγμιότυπου Geth μειώνει την πολυπλοκότητα ανάγνωσης κατάστασης κατά περίπου μια τάξη μεγέθους. Αυτό σημαίνει ότι η άρνηση παροχής υπηρεσίας (DoS) που βασίζεται σε ανάγνωση γίνεται πιο δύσκολο να επιτευχθεί κατά μια τάξη μεγέθους και οι κλήσεις της συνάρτησης eth_call επιταχύνονται κατά μια τάξη μεγέθους.

Το στιγμιότυπο επιτρέπει επίσης το γρήγορο state iteration των πιο πρόσφατων μπλοκ. Αυτός ήταν στην πραγματικότητα ο κύριος λόγος για τη δημιουργία στιγμιότυπων, καθώς επέτρεψε τη δημιουργία του νέου αλγόριθμου snap sync που επιταχύνει τον συγχρονισμό νέων κόμβων και θα αναλυθεί στην συνέχεια. [11].

Παρά την σημαντική επιτάχυνση που προσφέρει η δομή αυτή, δεν γίνεται να αποφύγουμε την ύπαρξη trade-offs. Μετά την ολοκλήρωση του αρχικού συγχρονισμού, χρειάζονται περίπου 9-10 ώρες στο mainnet για τη δημιουργία του αρχικού στιγμιότυπου (διατηρείται ζωντανά στη συνέχεια) και χρειάζονται περίπου 15+ GB επιπλέον χώρου στο δίσκο.

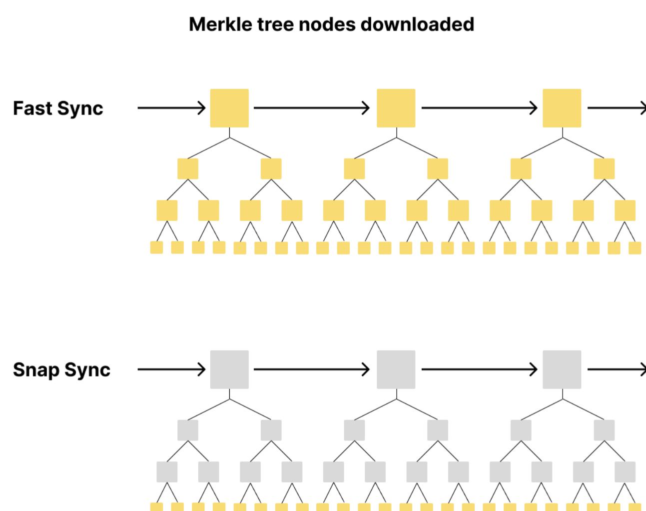
Σημειώνουμε ότι η συγκεκριμένη βελτίωση απαίτησε μεγάλο όγκο εργασίας, ενδεικτικά η πρώτη έκδοση του snap sync γράφτηκε πριν από 2,5 χρόνια. Επίσης απαιτήθηκαν πάνω από 6 μήνες για τις δοκιμές και την αποδοχή του τελικού κώδικα στιγμιότυπου.

2.4.2 Το Snap Sync

Το Snap sync σχεδιάστηκε για να λύσει και τα τρία προβλήματα του γρήγορου συγχρονισμού (fast sync) που απαριθμήθηκαν στην αρχή. Η βασική ιδέα είναι αρκετά απλή: αντί να κατεβάζει το δένδρο κόμβο προς κόμβο, το snap sync κατεβάζει τα συνεχόμενα κομμάτια χρήσιμων δεδομένων κατάστασης σε flat μορφή (snapshot), όπως εξηγήσαμε και παραπάνω, τα οποία εποπτικά βρίσκονται στα φύλλα των MPTs, όπως φαίνεται και στην Εικόνα 2.12, και έπειτα ανακατασκευάζει το δένδρο τοπικά. Οι βελτιώσεις στα προβλήματα που παρουσιάζονται στο fast sync είναι οι ακόλουθες [17]:

- Χωρίς λήψη ενδιάμεσων κόμβων δένδρων Merkle, τα δεδομένα κατάστασης μπορούν να ανακτηθούν σε μεγάλες παρτίδες, καταργώντας την καθυστέρηση που προκαλείται από την καθυστέρηση δικτύου (latency).

- Χωρίς λήψη κόμβων Merkle, τα δεδομένα που λαμβάνονται (download) μειώνονται στο μισό. Και χωρίς να αντιμετωπίζεται κάθε τμήμα δεδομένων ξεχωριστά, τα δεδομένα που αποστέλλονται (upload) είναι ασήμαντου όγκου, αφαιρώντας την καθυστέρηση που προκαλείται από το εύρος ζώνης.



Εικόνα 2.12: *Snap Sync vs Fast Sync Downloads*
https://docs.nethermind.io/nethermind/ethereum-client/sync-modes

- Χωρίς να ζητούν δεδομένα με τυχαία κλειδιά, οι ομότιμοι κόμβοι κάνουν μόνο δύο συνεχόμενες αναγνώσεις δίσκου για να εξυπηρετήσουν τις απαντήσεις, αφαιρώντας την καθυστέρηση IO του δίσκου ,με την προϋπόθεση βέβαια οι ομότιμοι κόμβοι να έχουν ήδη αποθηκευμένα τα δεδομένα σε κατάλληλη επίπεδη μορφή (snapshot).

Ενώ το snap sync είναι αρκετά παρόμοιο με το warp sync του parity - και πράγματι πήρε πολλές ιδέες σχεδίασης από αυτό - υπάρχουν σημαντικές βελτιώσεις σε σχέση με το τελευταίο [17]:

Το warp sync βασίζεται σε στατικά στιγμιότυπα που δημιουργούνται κάθε 30000 μπλοκ. Αυτό σημαίνει ότι οι κόμβοι εξυπηρέτησης πρέπει να αναπαράγουν τα στιγμιότυπα κάθε 5 ημέρες περίπου, αλλά η επανάληψη ολόκληρης της προσπάθειας κατάστασης μπορεί στην πραγματικότητα να πάρει περισσότερο χρόνο από αυτόν. Αυτό σημαίνει ότι το warp sync δεν είναι βιώσιμο μακροπρόθεσμα. Σε αντίθεση με αυτό, το snap sync βασίζεται σε δυναμικά στιγμιότυπα, τα οποία δημιουργούνται μόνο μία φορά, αδιάφορο πόσο αργά αυτό γίνεται, και στη συνέχεια διατηρούνται ενημερωμένα καθώς προχωρά η αλυσίδα.

Η μορφή στιγμιότυπου του warp sync δεν ακολουθεί τη διάταξη δένδρου Merkle και, ως εκ τούτου, τα κομμάτια των δεδομένων warp δεν μπορούν να αποδειχθούν μεμονωμένα. Οι κόμβοι συγχρονισμού πρέπει να κατεβάσουν ολόκληρο το σύνολο δεδομένων των 20+ GB για να μπορέσουν να το επαληθεύσουν. Αυτό σημαίνει ότι οι κόμβοι warp sync θα μπορούσαν θεωρητικά να αδυνατούν να ανταποκριθούν. Σε αντίθεση με αυτό, η μορφή στιγμιότυπου του snap sync είναι απλώς τα διαδοχικά φύλλα του δένδρου Merkle, τα οποία επιτρέπουν την απόδειξη οποιουδήποτε εύρους, επομένως τα κακά δεδομένα εντοπίζονται αμέσως.

2.4.3 Goerli Testnet

Το Goerli είναι ένα δίκτυο δοκιμών για την απόδειξη εξουσιοδότησης μεταξύ πελατών για το Ethereum. Το Goerli είναι το πρώτο δοκιμαστικό δίκτυο μεταξύ πελατών με απόδειξη

εξουσιοδότησης, που συγχρονίζει τα Parity Ethereum, Geth, Nethermind, Hyperledger Besu (πρώην Pantheon) και EthereumJS. Αυτό το δοκιμαστικό δίκτυο είναι ένα έργο που βασίζεται στην κοινότητα χρηστών, εντελώς ανοιχτού κώδικα. Γεννήθηκε τον Σεπτέμβριο του 2018 κατά τη διάρκεια του ETHBerlin και έκτοτε αυξάνεται σε συντελεστές [19].

Το δίκτυο δοκιμών Goerli είναι το πιο σταθερό testnet και για το λόγο αυτό έχει επιλεγεί για πλήθος πειραμάτων που σχετίζονται με την ανάπτυξη και εξέλιξη του Ethereum. Αυτός είναι και ο λόγος που έχει επιλεχθεί στα πλαίσια αυτής της εργασίας για την διεξαγωγή των πειραμάτων.

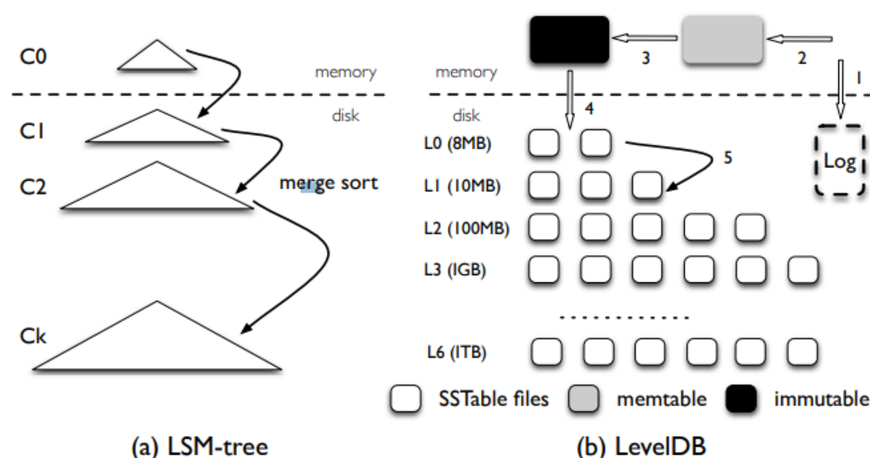
2.4.4 LSM-trees και LevelDB

Η LevelDB είναι μια βιβλιοθήκη ανοιχτού κώδικα της Google, για την αποθήκευση κλειδιού-τιμής που παρέχει μεταξύ άλλων, επαναλήψεις προς τα εμπρός και προς τα πίσω σε δεδομένα, ταξινομημένη αντιστοίχιση από κλειδιά συμβολοσειρών σε τιμές συμβολοσειρών, προσαρμοσμένες λειτουργίες σύγκρισης και αυτόματη συμπίεση. Τα δεδομένα συμπιέζονται αυτόματα χρησιμοποιώντας το "Snappy" μια βιβλιοθήκη συμπίεσης/αποσυμπίεσης ανοιχτού κώδικα Google. Ενώ το Snappy δεν στοχεύει στη μέγιστη συμπίεση, στοχεύει σε πολύ υψηλές ταχύτητες. Η Leveldb είναι ένας σημαντικός μηχανισμός αποθήκευσης και ανάκτησης που διαχειρίζεται την κατάσταση του δικτύου Ethereum. Ως εκ τούτου, η leveldb είναι μια εξάρτηση για τους πιο δημοφιλείς πελάτες Ethereum (κόμβους) όπως το go-ethereum, το cpp-ethereum και το pyethereum [11].

Η LevelDB είναι ένα ευρέως χρησιμοποιούμενο key-value store, το οποίο βασίζεται σε LSM-trees. Τα LSM-trees είναι μια persistent δομή, η οποία είναι αποτελεσματική στις περιπτώσεις των key-value stores, με μεγάλο αριθμό εισαγωγών και διαγραφών. Συγκεντρώνει τα δεδομένα προς εγγραφή σε μεγάλα κομμάτια, ώστε να εκμεταλλευτεί το υψηλό διαδοχικό εύρος ζώνης των σκληρών δίσκων. Επειδή, λοιπόν, οι διαδοχικές εγγράφες (sequential writes) είναι ταχύτερες από τις τυχαίες εγγραφές (random writes) περίπου κατα 2 τάξεις μεγέθους, τα LSM-trees προσφέρουν καλύτερη απόδοση εγγραφών σε σχέση με τα παραδοσιακά B-δέντρα, που περιλαμβάνουν random accesses.

Ένα LSM-tree αποτελείται από τμήματα με εκθετικά μεγαλύτερα μεγέθη, C_0 έως C_k, όπως φαίνεται και στην εικόνα 2.13(α). Το C_0 είναι ταξινομημένο δέντρο που διατηρείται in-memory, ενώ τα παρακάτω τμήματα είναι B-δέντρα αποθηκευμένα στον δίσκο. Κατά την εισαγωγή ενός ζευγαριού (key,value), αυτό αποθηκεύεται σε ένα on-disk sequential log file, ώστε να είναι εφικτή η ανάκτηση σε περίπτωση που έχουμε crash. Στην συνέχεια, το ζευγάρι προστίθεται στην μνήμη C_0, η οποία επιτρέπει αποτελεσματικές αναζητήσεις σε ζευγάρια (key, value) που έχουν εισαχθεί πρόσφατα. Όταν το C_0 φτάσει το όριο μνήμης του, τότε συγχωνεύεται στο on-disk C_1 με μια λογική αντίστοιχη του αλγορίθμου Merge Sort. Η διαδικασία αυτή είναι γνωστή ως συμπύκνωση (compaction) και εφαρμόζεται αντίστοιχα και για όλα τα τμήματα που είναι διαδοχικά μεταξύ τους (C_i και C_{i+1}). Για μια αναζήτηση τα LSM-trees, μπορεί να χρειαστεί να διασχίσουν όλα τα τμήματα ένα προς ένα μέχρι να βρούν το επιθυμητό στοιχείο, γεγονός που τα καθιστά καλύτερη λύση όταν οι εισαγωγές είναι πιο συχνές από τις αναζητήσεις [20].

Η αρχιτεκτονική της LevelDB φαίνεται στην εικόνα 2.13(β) και περιλαμβάνει ένα on-disk



Εικόνα 2.13: *LSM-Tree and LevelDB Architecture*
https://learn-sys.github.io/cn/slides/r0/week13-1.pdf

log file, δύο ταξινομημένες in-memory skip λίστες (memetable και immutable memtable) και 7 επίπεδα από ταξινομημένους πίνακες συμβολοσειρών (Sorted String Table), οι οποίοι αποθηκεύονται στον δίσκο. Αρχικά τα ζευγάρια (key, value) που εισάγονται αποθηκεύονται στο log file και το memtable. Όταν το δεύτερο γεμίσει, τότε τα νέα δεδομένα αποθηκέυονται σε νέο memetable και log file, ενώ το παλιό memetable τροποποείται σε immutable memtable και στην συνέχεια μέσω συμπύκνωσης(compaction) τα δεδομένα περνούν στο δίσκο και το επίπεδο L_0, δημιουργώντας νέο αρχείο SSTable. Όπως έχουμε αναφέρει και προηγουμένως, κατά την αναζήτηση στην LevelDB στην χειρότερη περίπτωση μπορούμε να έχουμε 7 ¨χτυπήματα¨ στον δίσκο σε περίπτωση που για την εύρεση του κλειδιού χρειαστεί να φτάσουμε μέχρι το επίπεδο L_6.

Ένα μεγάλο πρόβλημα των LSM-trees, οπότε και της LevelDB είναι το λεγόμενο Write and Read amplification και έχει να κάνει με την αναλογία των δεδομένων που γράφτηκαν (ή διαβάστηκαν) από την μνήμη, σε σχέση με το πλήθος των δεδομένων που ο χρήστης ζήτησε. Για να πετύχει διαδοχική οργάνωση των δεδομένων και κατά συνέπεια διαδοχικά accesses, η LevelDB γράφει περισσότερα δεδομένα, περισσότερες φορές από το απαραίτητο, δημιουργώντας έτσι μεγάλο Write amplification, το οποίο σχετίζεται με την διαδικασία του compaction. Προφανώς, όσο μεγαλώνει το μέγεθος της βάσης, τόσο μεγαλώνει και το Write amplification, καθώς τα ζευγάρια (key, value) πιθανότητα θα χρειαστεί να διασχίσουν περισσότερα επίπεδα μνήμης κατά την διαδικασία της συμπύκνωσης. Δηλαδή, στην LevelDB, τα δεδομένα γράφονται πολλές φορές καθώς γίνεται compaction από 'χαμηλά' σε 'υψηλά' επίπεδα.

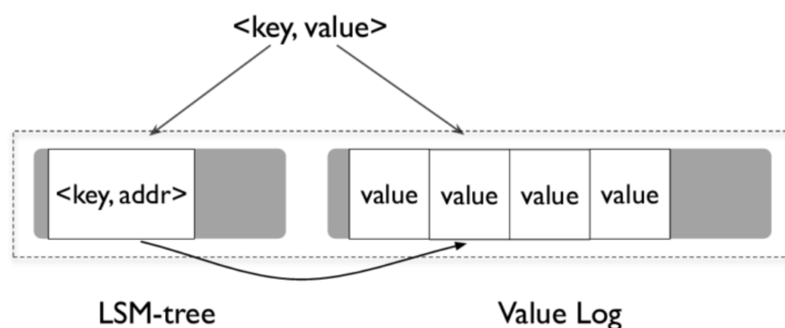
2.4.5 BadgerDB

Το BadgerDB είναι μία ενσωματώσιμη, γρήγορη και persistent key-value βάση, γραμμένη αποκλειστικά στην γλώσσα Go (Golang). Το BadgerDB δημιουργήθηκε κυρίως για να αποτελέσει μια αποτελεσματική εναλλακτική σε key-value stores, που δεν είναι βασισμένα στην γλώσσα Go, όπως η RocksDB. Το Badger είναι σταθερό και χρησιμοποείται για να

σερβίρει δεδομένα μεγέθους εκατοντάδων terabytes. Επίσης, το Badger υποστηρίζει ταυτόχρονές ACID συναλλαγές με serializable snapshot isolation (SSI). Το Badger έχει ακόμα δοκιμαστεί ότι δουλεύει καλά όταν αντιμετωπίζει ανωμαλίες επιπέδου συστήματος αρχείων, εξασφαλίζοντας ακρίβεια και συνέπεια[21].

Το σχέδιο του Badger βασίζεται σε ένα paper με όνομα WiscKey: Separating Keys from Values in SSD-conscious Storage[20], η κεντρική ιδεά του οποίου είναι ο διαχωρισμός των κλειδιών και των τιμών κατά την διαδικασία του compaction των LSM-trees. Αναφέραμε και προηγουμένως, ότι το μεγαλύτερο πρόβλημα απόδοσης των LSM-trees είναι ακριβώς αυτή η διαδικασία της συμπύκνωσης. Πολλαπλά αρχεία διαβάζονται στην μνήμη, ταξινομούνται και στην συνέχεια γράφονται ξανά επηρεάζοντας σημαντικά την απόδοση. Πληρώνοντας το παραπάνω κόστος τα LSM-trees επιτυγχάνουν αποτελεσματική αναζήτηση, λόγω διαδοχικών (sequential) προσβάσεων για πολλαπλά αρχεία.

Η σηματική αλλαγή που συναντάμε εδώ έχει να κάνει με την ανακάλυψη, ότι για την διαδικασία του compaction, χρειάζεται να ταξινομηθούν μόνο τα κλειδιά, ενώ οι τιμές μπορούν να αντιμετωπιστούν ξεχωριστά. Από την στιγμή που τα κλειδιά είναι συνήθως μικρότερα από τις τιμές, η συμπύκνωση μόνο των κλειδιών μπορεί να μειώσει δραστικά το πλήθος των δεδομένων που χρειάζονται κατά τη διαδικασία της ταξινόμησης. Με αυτόν τον τρόπο, το μήκος του LSM-tree σε αυτήν την περίπτωση είναι πολύ μικρότερο σε σχέση με την περίπτωση της LevelDB, γεγονός που μπορεί να μειώσει σημαντικά το Write και Read Amplification για workloads με σχετικά μεγάλη ποσότητα δεδομένων. Μια σχηματική αναπαράσταση της δομής αυτής φαίνεται στην εικόνα 2.14.



Εικόνα 2.14: *LSM-Tree with Key-Value Seperation*
https://distributed-computing-musings.com/wp-content/uploads/2022/07/Screen-Shot-2022-07-11-at-10.42.46-AM-768x318.png

Κατά τη διάρκεια της αναζήτησης μιας τιμής, πρώτα θα ελεχθεί το LSM-tree για το κλειδί και στην συνέχεια όταν βρεθεί η τιμή, γίνεται νέα κλήση read για να βρεθεί η τιμή. Παρότι, η δομή αυτή στην αναζήτηση μοιάζει πιο αργή από την αντίστοιχη της LevelDB, λόγω του επιπλέον I/O που εισάγει, στην πραγματικότητα επειδή το LSM-tree είναι σημαντικά μικρότερο, η αναζήτηση θα ελέγξει λιγότερα επίπεδα στο δέντρο, γλιτώνοντας έτσι αρκετά accesses στο δίσκο. Αυτό μοιάζει να είναι ένα ικανοποιητικό trade-off, τέτοιο ώστε να μας οδηγήσει στην υπόθεση ότι η δομή αυτή μπορεί να αποδώσει καλύτερα σε σχέση με τα συμβατικά LSM-Trees.

Πρακτικό Μέρος

Κεφάλαιο 3

Μελέτη του Geth Client

Στο κεφάλαιο αυτό παρουσιάζεται αρχικά η μελέτη που έγινε για την καταγραφή του workload του Geth Client, ενώ στη συνέχεια περιγράφονται οι λόγοι που επιλέγουμε να ενσωματώσουμε το επιλεγμένο key-value store (badgerdb) με σκοπό την βελτιστοποίηση της απόδοσης του client κατά την διαδικασία του syncing.

3.1 Ανάλυση του Workload

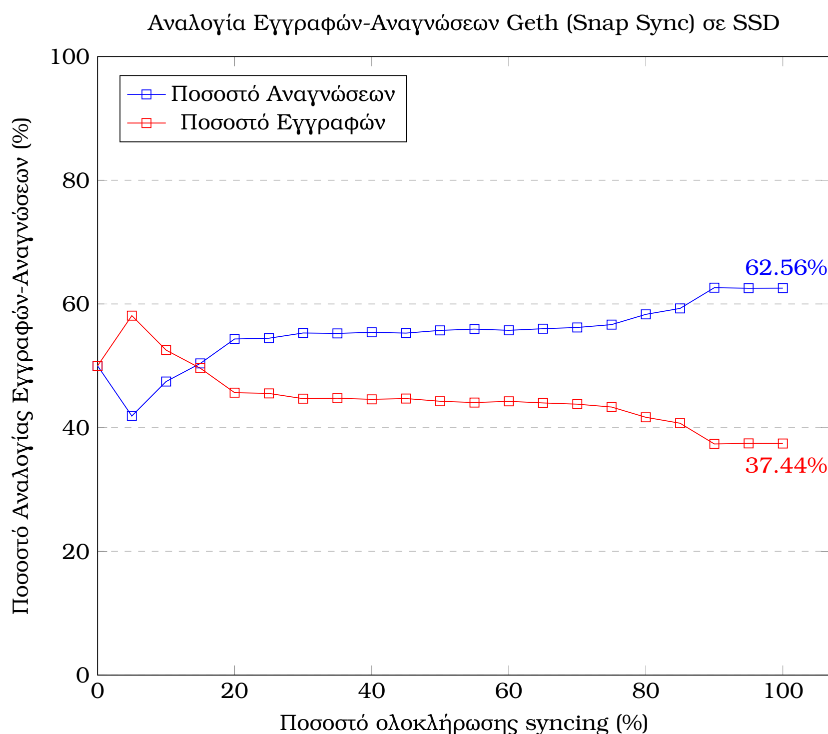
Στην ενότητα αυτή παρουσιάζεται η μελέτη και ανάλυση του workload του Geth Client στην υπάρχουσα μορφή του με την χρήση της LevelDB ως επιλεγμένου key-value store για αποθήκευση των δεδομένων του blockchain.

Συγκεκριμένα, εξετάζουμε μέσω πειραμάτων, τα οποία χωρίζονται ανάλογα την συσκευή αποθήκευσης που χρησιμοποιούμε (HDD ή SSD), την αναλογία αναγνώσεων εγγραφών στον δίσκο, προοδευτικά, κατά την διάρκεια του Snap Sync, το αποτέλεσμα των οποίων παρουσιάζεται σχηματικά σε διαγράμματα δεδομένων στην συνέχεια. Έπειτα, ασχολούμαστε με την χρονική καθυστέρηση που εισάγεται από την διαδικάσία του compaction στον συνολικό χρόνο συγχρονισμου και για τις δύο περιπτώσεις αποθήκευσης δεδομένων.

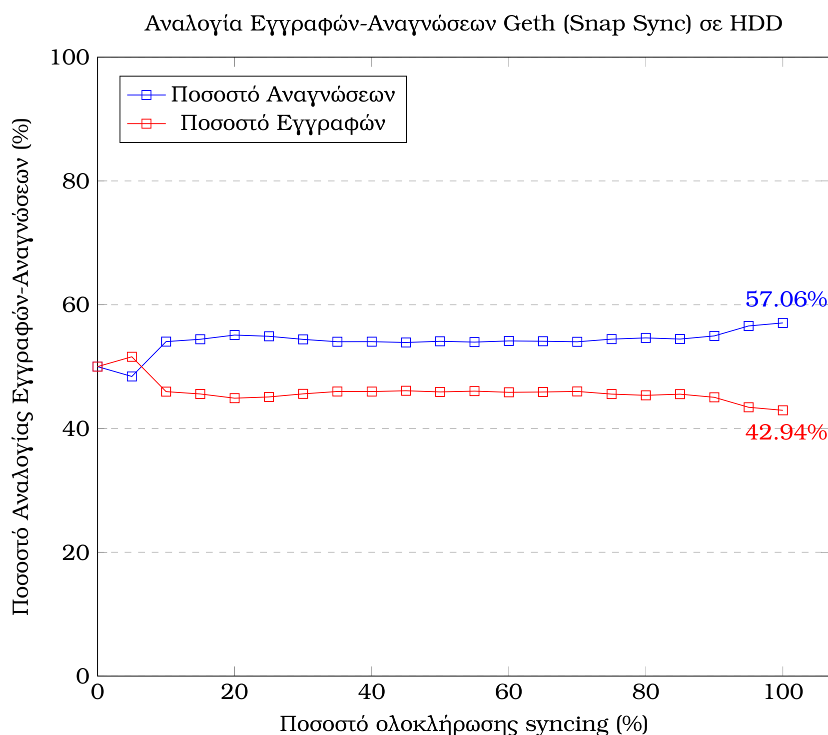
3.1.1 Αναλογία Αναγνώσεων-Εγγραφών στον Δίσκο

Μία σημαντική παράμετρος στην επιλογή κατάλληλου key-value store είναι η αναλογία των αναγνώσεων και των εγγραφών που γίνονται στο δίσκο κατά την διάρκεια του syncing. Ανάλογα της δομή τους διαφορετικά key-value stores είναι πιο αποτελεσματικά στις εγγραφές ή στις αναγνώσεις, οπότε η αναλογία αυτή μας επιτρέπει να έχουμε μια καλύτερη εικόνα, ώστε να έχουμε καλύτερες σχεδιαστικές επιλογές σχετικά με το ποιο key-value store μπορεί να δώσει τα καλύτερα δυνατά αποτελέσματα.

Έτσι, εκτελούμε πειράματα για τις περιπτώσεις όπου χρησιμοποιούμε HDD και SSD, αναζητώντας την εν λόγω αναλογία καθ'όλη την διάρκεια της διαδικασίας συντονισμού. Οι μετρήσεις αυτές παρουσιάζονται στα παρακάτω διαγράμματα. Αρχικά, η μελέτη κατά την διεξαγωγή του πειράματος με SSD είχε τα ακόλουθα αποτελέσματα.

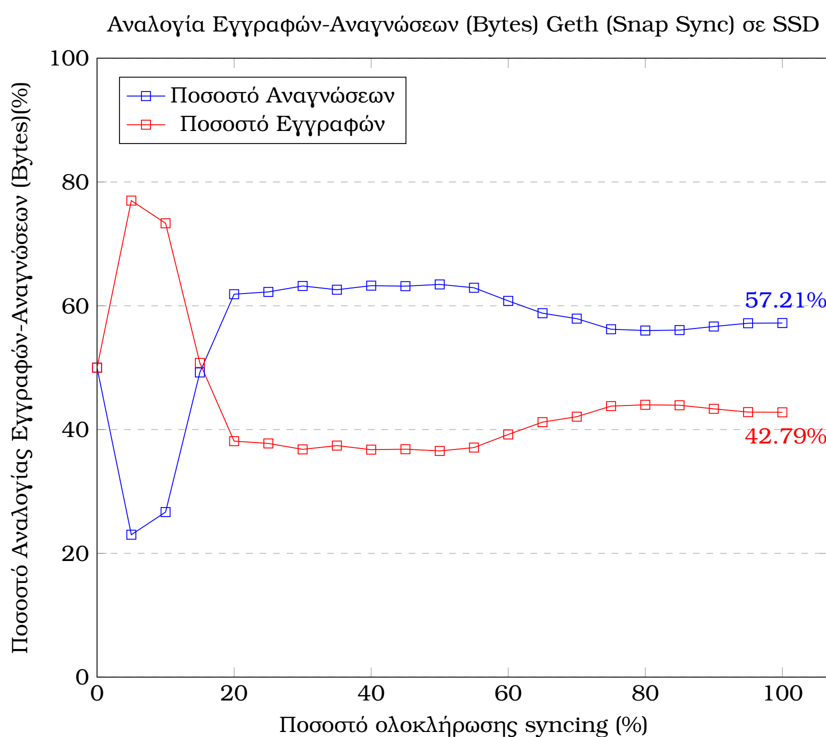


Η ίδια μελέτη κατά την διεξαγωγή του πειράματος με HDD είχε τα παρακάτω αντίστοιχα αποτελέσματα.

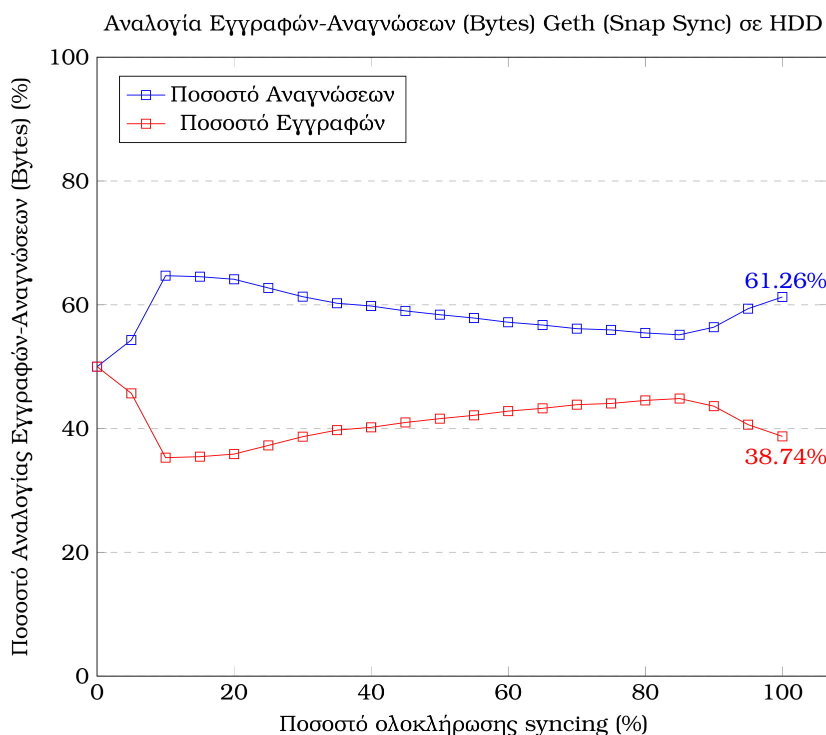


Παρατηρούμε ότι και στις δύο περιπτώσεις οι αναγνώσεις υπερτερούν των εγγραφών σε απόλυτο αριθμό 'χτυπημάτων' στο δίσκο, περισσότερο στην περίπτωση χρήσης SSD. Παρ'όλα αυτα για να έχουμε μια πιο καθαρή εικόνα για το μέγεθος των δεδομένων που διαβάζονται-/γράφονται, στην συνέχεια και στα ακόλουθα διαγράμματα παρουσιάζουμε την αντίστοιχη

ποσοστιαία μελέτη που σχετίζεται με τα ποσοστά των αναγνώσεων και εγγραφών στο δίσκο, όχι όμως σαν 'καθαρό' αριθμό, αλλά με βάση το μέγεθος των αρχείων που διαβάζονται και γράφονται αντίστοιχα.



Η ίδια μελέτη κατά την διεξαγωγή του πειράματος με HDD είχε τα παρακάτω αντίστοιχα αποτελέσματα.



Παρατηρούμε ότι και στην περίπτωση που μετράμε το μέγεθος των δεδομένων, οι αναγνώσεις κατά την διάρκεια του συγχρονισμού υπερτερούν ελαφρώς των εγγραφών. Συνεπώς,

θεωρητικά ψάχνουμε ένα key-value store, το οποίο να είναι εξίσου καλό και στην περίπτωση των αναγνώσεων και των εγγραφών, εστιάζοντας ίσως λίγο περισσότερο στο πρώτο κομμάτι.

3.1.2 Χρονική καθυστέρηση της διαδικασίας του Compaction

Η διαδικασία της συμπύκνωσης (compaction) είναι νευραλγική διαδικασία για όλα τα key-value stores που βασίζονται σε LSM-Trees, μεταξύ των οποίων και η LevelDB, που το Geth χρησιμοποιεί.

Συγκεκριμένα στο Geth, κατά τη διαδικασία του compaction, η αλυσίδα δεν προχωρά, οπότε ο χρόνος που απαιτείται για την συμπύκνωση εισάγει καθυστέρηση σε όλη την διαδικασία του sync.Για το λόγο αυτό έχει αξία να μελετήσουμε το χρόνο που απαιτείται για την διαδικασία αυτή διαχωρίζοντας πάλι τις περιπτώσεις όπου χρησιμοποιούμε HDD και SSD.

Από την μελέτη αυτή προέκυψαν τα ακόλουθα αποτελέσματα, σχετικά με το ποσοστό του ολικού χρόνου συγχρονισμού που καταλαμβάνει το compaction κατά την είσοδο ενός νέου κόμβου στο δίκτυο μέσω της επιλογής του Snap Sync.

Compaction Delay (Snap Sync)		
/////////////	*HDD*	*SSD*
Percentage of Total Syncing	71.59%	1.63%

Πίνακας 3.1: *Χρονική Καθυστέρηση Compaction ως Ποσοστό Ολικού Χρόνου Συντονισμού*

Παρατηρούμε, λοιπόν, ότι παρότι στην περίπτωση που χρησιμοποιούμε SSD, το compaction είναι μια σύντομη διαδικασία που δεν καθυστερεί τον συγχρονισμό, αντίθετα στην περίπτωση που χρησιμοποιούμε HDD, το compaction αποτελεί την πιο 'βαριά' διαδικασία της όλης προσπάθειας, καταλαμβάνοντας περίπου τα 2/3 του συνολικού χρόνου. Μοιάζει, λοιπόν, θεωρητικά η περίπτωση του HDD, να είναι αυτή που επιδέχεται μεγαλύτερης βελτίωσης στην συγκεκριμένη περίπτωση.

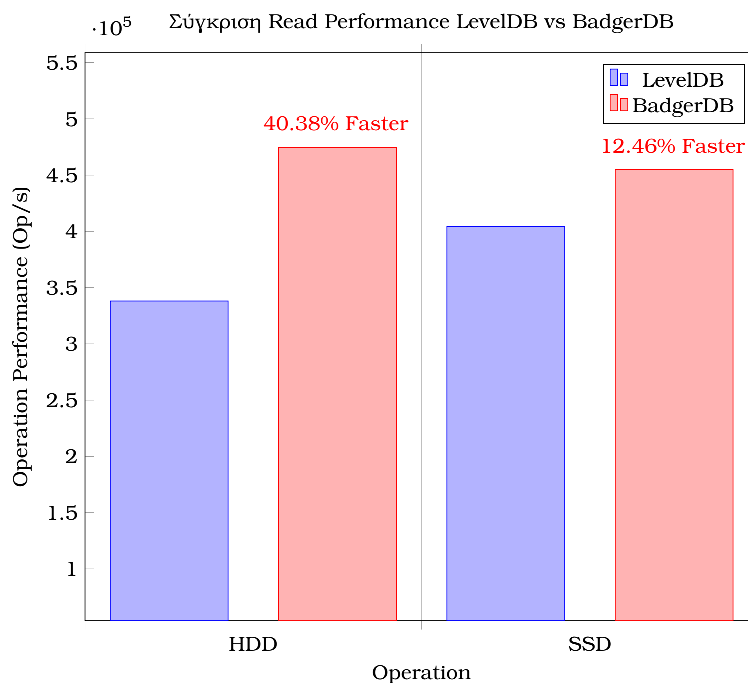
3.2 Χαρακτηριστικά του BadgerDB

Αφού μελετήσαμε τη λειτουργία του Geth Client, στην υπάρχουσα δομή του(LevelDB), έχει τώρα νόημα να δούμε τα χαρακτηριστικά του key-value store που στα πλαίσια αυτής της εργασίας θέλουμε να αντικαταστήσει την LevelDB, δηλαδή το BadgerDB.

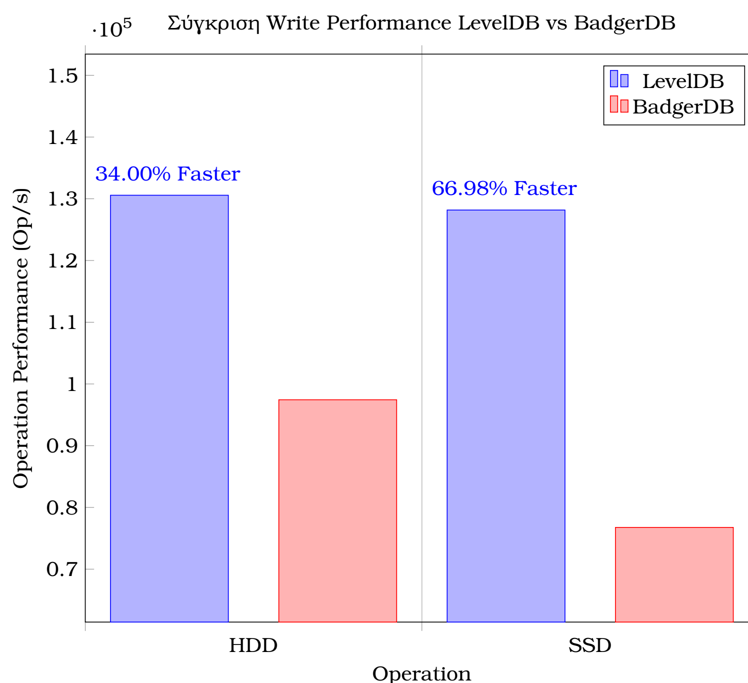
Εξηγήσαμε και στο θεωρητικό μέρος, ότι το BadgerDB βασίζεται στη λογική διαχωρισμού των κλειδιών και των τιμών κατά την διαδικασία του compaction των LSM-trees. Προφάνως, λόγω της δομής του αναμένουμε σημαντική επιτάχυνση της διαδικασίας του compaction με την ενσωμάτωση του BadgerDB στο Geth Client. Είδαμε και προηγουμένως ότι η διαδικασία της συμπύκνωσης μπορεί να αποτελέσει bottleneck κατά την διάρκεια του συγχρονισμού, ειδικά στην περίπτωση του HDD, συνεπώς οποιαδήποτε ενέργεια μειώνει τον χρόνο της διαδικασίας αυτής αξίζει να ερευνηθεί περαιτέρω.

Έχει, επίσης, ενδιαφέρον να δούμε το πως επηρεάζεται και η διαδικασία των Read και Write στο δίσκο από την νέα αυτή δομή. Είδαμε προηγούμενως, από την δομή του Geth,

ότι ψάχνουμε ένα key-value store, που να αποδίδει καλύτερα από την LevelDB σε αναγνώσεις, χωρίς να είναι πολύ πιο αργό σε εγγραφές. Κίνητρο για την επιλογή του BadgerDB για την αντικατάσταση της LevelDB, αποτέλεσαν τα πειράματα του χρήστη smallnest[21], όπου συγκρίνονται διαφορετικά key-value stores, ως προς την απόδοση τους σε random reads/writes. Συγκεκριμένα, ενδιαφέροντα είναι τα αποτελέσματα που σχετίζονται με τη LevelDB και το BadgerDB, των οποίων το throughput παρουσιάζεται στα ακόλουθα διαγράμματα. Αρχικά για την περίπτωση των αναγνώσεων έχουμε:



Αντίστοιχα, για την περίπτωση των εγγραφών έχουμε τα ακόλουθα αποτελέσματα:



Παρατηρούμε, λοιπόν, ότι σε επίπεδο αναγνώσεων το BadgerDB φαίνεται να αποδίδει καλύτερα τόσο στην περίπτωση του HDD, όσο και του SSD. Εξηγήσαμε και προηγουμένως, ότι οι αναγνώσεις αποτελούν το κύριο ποσοστό των συνολικών 'χτυπημάτων' στο δίσκο στο Geth, οπότε η υπεροχή του BadgerDB εδώ είναι πολύ σημαντικό κίνητρο, ώστε να επιχειρηθεί αλλαγή σε επίπεδο key-value store. Εντοπίζουμε, βέβαια, ότι σε επίπεδο HDD η υπεροχή αυτή είναι μεγαλύτερη, με το BadgerDB να δίνει 40.38% μεγαλύτερο Throughput, ενώ σε επίπεδο SSD το BadgerDB δίνει 12.46% μεγαλύτερο Throughput.

Αντίθετα, σε επίπεδο εγγραφών , η LevelDB φαίνεται να αποδίδει καλύτερα τόσο στην περίπτωση του HDD, όσο και του SSD. Συγκεκριμένα, στην περίπτωση του SSD η LevelDB έχει αισθητά καλύτερη απόδοση, καθώς έχει 66.98% μεγαλύτερο Throughput. Και πάλι εδώ, στην περίπτωση του HDD, η εικόνα είναι πιο εναθαρρυντική για το BadgerDB με την LevelDB να έχει 34.00% μεγαλύτερο Throughput.

Συνεπώς, η υπεροχή του BadgerDB σε επίπεδο Read Throughput (ειδικά στον HDD), καθώς και η επιτάχυνση της διαδικασίας του compaction καθιστά υποσχόμενη την επιλογή αντικατάστασης του key-value store του Geth Client. To trade-off που υπάρχει είναι η υστέρηση της απόδοσης του BadgerDB σε επίπεδο εγγραφών, ειδικά στην περίπτωση του SSD, το οποίο θα δούμε στην πράξη πόσο επηρεάζει την ταχύτητα του συγχρονισμού στην συνέχεια.

Κεφάλαιο 4

Απόδοση Syncing με Χρήση BadgerDB

Στο κεφάλαιο αυτό παρουσιάζονται τα αποτελέσματα της ενσωμάτωσης του BadgerDB στο Geth Client, στην θέση της LevelDB, εστιάζοντας στην διαδικασία συγχρονισμού μέσω Snap Sync και κυρίως στον συνολικό χρόνο που απαιτείται για αυτόν.

4.1 Στάδια Συγχρονισμού

Για να μελετήσουμε την διαδικασία του συγχρονισμού πιο αναλυτικά, επιλέγουμε να την αποδομήσουμε. Συγκεκριμένα, επειδή μιλάμε για συγχρονισμό μέσω Snap Sync, χωρίζουμε την όλη διαδικασία σε δύο στάδια τα οποία είναι τα ακόλουθα :

- Στάδιο 1: Το Στάδιο αυτό περιλαμβάνει την λήψη των block bodies, headers, receipts, που απαιτούνται για τον συγχρονισμό, μέχρι και 64 μπλοκ πριν το τελευταίο μπλοκ της αλυσίδας. Η διαδικασία αυτή είναι ίδια με αυτή που γινόταν στο Fast sync, που ήταν η προηγούμενη μέθοδος γρήγορου συγχρονισμού, όταν στόχος ήταν η αποφυγή του πλήρη συγχρονισμού (Full Sync), απλά εδώ το state trie κατασκευάζεται τοπικά από τα δεδομένα που έρχονται, όπως αναλύσαμε και στο θεωρητικό μέρος.

- Στάδιο 2: Το Στάδιο αυτό περιλαμβάνει την δημιουργία της δομής του Snapshot με βάση τα δεδομένα που έχουν ληφθεί από τους ομότιμους κόμβους. Προφανώς, το στάδιο αυτό εντοπίζεται μόνο στην περίπτωση του Snap Sync και η χρονική καθυστέρηση που εισάγει είναι ένα trade-off που πληρώνουμε, ώστε να έχουμε τα πλεονεκτήματα της δομής του Snapshot.

Θα δούμε στην συνέχεια τους επιμέρους χρόνους για το εκάστοτε στάδιο με χρήση BadgerDB και LevelDB, συγκρίνοντας την ταχύτητα συγχρονισμού στις δύο αυτές περιπτώσεις.

4.2 Σύγκριση LevelDB και BadgerDB

Το κύριο ζητούμενο της εργασίας είναι εάν η ενσωμάτωση του BadgerDB βελτιώνει την απόδοση του συγχρονισμού, κάτι που ερευνάμε σε αυτήν την ενότητα. Συγκεκριμένα, μετράμε πειραματικά την απόδοση της LevelDB στην διαδικασία του συγχρονισμού και στα επιμέρους στάδια αυτού και στην συνέχεια κάνουμε το ίδιο για την περίπτωση του BadgerDB, διατηρώντας τις ίδιες προδιαγραφές κατά την εκτέλεση των δύο διαφορετικών πειραμάτων.

4.2.1 Υλοποίηση ενσωμάτωσης BadgerDB

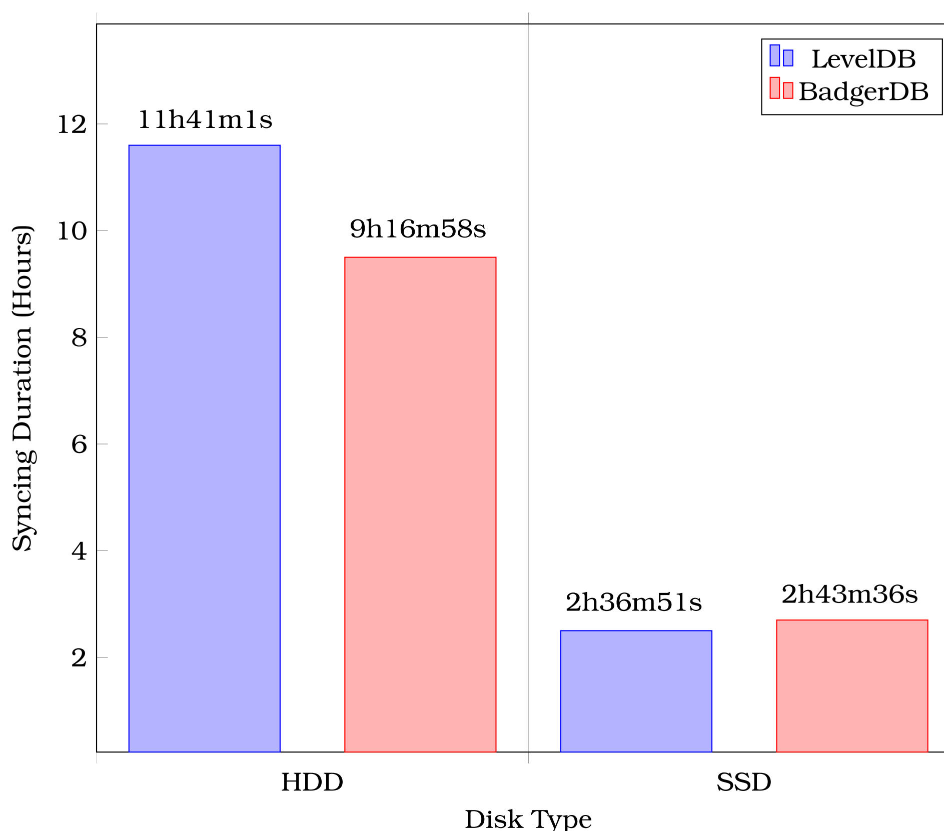
Η διαδικασία ενσωμάτωσης του BadgerDB περιγράφεται αναλυτικότερα από πλευράς κώδικα στο **Παράρτημα Α΄** και βασίζεται στην υλοποίηση του interface που ορίζει το Geth για το key-value store του.

Η διαδικασία αυτή ήταν ιδιαίτερη απαιτητική καθώς, χρειάστηκαν αρκετές προσαρμογές προκειμένου να καταστεί εφικτή η αντικατάσταση της LevelDB από το BadgerDB. Συγκεκριμένα :

- Όλες οι συναρτήσεις του interface που έπρεπε να υλοποιηθούν ήταν προσαρμοσμένες στα χαρακτηριστικά της LevelDB, οπότε η υλοποίηση τους αρχικά περιλάμβανε συνήθως μερικές γραμμές κώδικα με χρήση έτοιμων built-in συναρτήσεων της LevelDB. Αντίθετα, στην περίπτωση του BadgerDB, πολλές συναρτήσεις χρειάστηκε να υλοποιηθούν από την αρχή αναλυτικά, με βάση τα χαρακτηριστικά του BadgerDB. Συγκεκριμένα, η μεγαλύτερη δυσκολία εμφανίστηκε στην περίπτωση των συναρτήσεων που εισάγουμε τα badger.WriteBatch και badger.Iterator, η λειτουργικότητα των οποίων εξηγείται στο **Παράρτημα Α΄**.
- Έπρεπε να προσαρμοστούν τα προκαθορισμένα integrity tests του Geth, ώστε να μπορούν να δημιουργούν και να έχουν πρόσβαση σε badgerdb database.

4.2.2 Εκτέλεση Πειραμάτων

Τα πειράματα διεξήχθησαν ξεχωριστά για τις περιπτώσεις SSD και HDD, με τα συνολικά χρονικά αποτελέσματα τους να παρουσιάζονται στο παρακάτω διάγραμμα.



Στην συνέχεια, παρουσιάζουμε πιο αναλυτικά τα αποτέλεσματα του παραπάνω διάγραμματος, συμπεριλαμβάνοντας την απόδοση των δύο περιπτώσεων και στα επιμέρους στάδια συγχρονισμού, παραθέτοντας και την αντίστοιχη ποσοστιαία χρονική μεταβολή, ώστε να υπάρχει μια πιο συνολική και ολοκληρωμένη εικόνα των αποτελεσμάτων.

Total Syncing Time (Snap Sync)						
/////////////	*HDD*			*SSD*		
/////////////	*LevelDB*	*BadgerDB*	*Comparison*	*LevelDB*	*BadgerDB*	*Comparison*
Stage 1 Duration	9h24m1s	7h57m56s	18.01% faster	2h7m59s	1h59m44s	6.89% faster
Stage 2 Duration	2h17m0s	1h19m2s	73.34% faster	28m52s	43m50s	34.14% slower
Total Syncing Duration	**11h41m1s**	**9h16m58s**	**25.86% faster**	**2h36m51s**	**2h43m36s**	**4.30% slower**

Πίνακας 4.1: *Χρονική Διάρκεια Snap Sync LevelDB vs BadgerDB*

Σημειώνουμε ότι στα αποτελέσματα μας δεν ασχολούμαστε με τον όγκο του δίσκου που απαιτούταν για την εκτέλεση των πειραμάτων, διότι σε όλες τις περιπτώσεις η τιμή μεταξύ BadgerDB και LevelDB σχεδόν ταυτιζόταν. Ως αποτέλεσμα, δεν υφίσταται συγκριτικό πλεονέκτημα για καμία από τις δύο περιπτώσεις key-value store, όσον αφορά το ποσοστό του δίσκου που χρησιμοποιήθηκε.

Η διαδικασία για την συλλογή των δεδομένων, οι εντολές που χρησιμοποιήθηκαν, καθώς και τα χαρακτηριστικά των μηχανημάτων που χρησιμοποιήθηκαν κατά την διαδικασία των πειραμάτων περιγράφονται αναλυτικότερα στο Παράρτημα Β.

4.3 Σχολιασμός Αποτελεσμάτων

Μετά την εκτέλεση των πειραμάτων μας παρατηρούμε ότι η ενσωμάτωση του BadgerDB βελτιώνει την ταχύτητα του συγχρονισμού στην περίπτωση χρήσης HDD, ενώ φαίνεται να υστερεί σε απόδοση στην περίπτωση χρήσης SSD. Συγκεκριμένα, στην περίπτωση χρήσης HDD το BadgerDB δίνει κατά 25.86% ταχύτερα αποτελέσματα, ενώ στον SSD είναι πιο αργός ο συντονισμός, σε σχέση με την LevelDB, κατά 4.30%.

Με βάση την μελέτη του workload και των benchmarks για τα key-value stores, θα χαρακτηρίζαμε τα παραπάνω αποτελέσματα εν μέρει αναμενόμενα. Όπως αναμενόταν, το BadgerDB βελτιώνει την ταχύτητα στον HDD αρχικά λόγω της μείωσης του compaction time, που βασίζεται στην δομή του ίδιου. Ταυτόχρονα όμως, η σαφής υπεροχή του στα disk reads, τα οποία αποτελούν και το μεγαλύτερο μέρος των χτυπημάτων στο δίσκο, υπερκερνά την αντίστοιχη αδυναμία στα disk writes με το παραπάνω δίνοντας καλύτερη συνολική απόδοση και στα δύο στάδια του συντονισμού.

Αντίθετα, για την περίπτωση του SSD, η οριακή υπεροχή του BadgerDB στα disk reads, δεν φαίνεται ικανή να αντισταθμίσει την σχετική αδυναμία του στα disk writes, καθιστώντας τον συνδυασμό αυτό συνολικά κατώτερο του αντίστοιχου συνδυασμού με χρήση LevelDB. Επίσης, όπως είδαμε νωρίτερα, το compaction delay στον SSD στο Snap Sync είναι πολύ

μικρό, καθιστώντας την μείωση του πρακτικά ανούσια στην συνολική προσπάθεια ενίσχυσης της ταχύτητας συντονισμού.

Επίσης, η καλύτερη εικόνα στον HDD μπορεί να αποδοθεί και στο ότι στην αναλογία Εγγραφών-Αναγνώσεων, με βάση το μέγεθος των αρχείων, οι αναγνώσεις κατείχαν μεγαλύτερο ποσοστό (61.26% έναντι 57.21%. Επίσης, ειδικά στην περίπτωση του Stage 2(Snapshot Generation), στον HDD βλέπουμε μεγάλη βελτίωση από την αλλαγή key-value store. Το γεγονός αυτό μοιάζει να βασίζεται στην παρατήρηση του workload, όπου εντοπίζουμε ένα spike στο ποσοστό των αναγνώσεων (bytes) στο τελευταίο στάδιο του συγχρονισμού, κάτι που δεν έχουμε στην περίπτωση του SSD.

Συνεπώς, με βάση τα παραπάνω αποτελέσματα, συμπεραίνουμε ότι το BadgerDB, μπορεί να βελτιώσει την απόδοση του Snap Sync του Geth όταν αυτό γίνεται σε μηχάνημα με HDD. Αντίθετα, στην περίπτωση ύπαρξη μηχανήματος SSD, όχι μόνο δεν βελτιώνει την ταχύτητα του συγχρονισμού, αλλά κάνει την όλη διαδικασία ακόμα πιο αργή μεγενθύνοντας ουσιαστικά το ήδη σημαντικό πρόβλημα της καθυστέρησης εισόδου ενός νέο κόμβου στο δίκτυο του Ethereum.

Μέρος III

Επίλογος

Κεφάλαιο 5

Επίλογος

5.1 Συμπεράσματα

Το Blockchain είναι μια αναδυόμενη τεχνολογία, η οποία έχει τραβήξει την προσοχή του επιστημονικού, και όχι μόνο, κόσμου τα τελευταία χρόνια και όχι άδικα. Στα πλαίσια της συνεχόμενης προσπάθειας εξέλιξης των συστημάτων που βασίζονται σε αυτό, η εργασία αυτή προσέγγισε μια πτυχή του Ethereum Blockchain που σχετίζεται με την ταχύτητα του αρχικού συγχρονισμού ενός νέου κόμβου στο δίκτυο.

Η συνεισφορά της παρούσας διπλωματικής εργασίας έχει δύο σκέλη. Το πρώτο αφορά τη μελέτη του workload του Geth Client σχετικά με την αναλογία αναγνώσεων/εγγραφών στο δίσκο και το δεύτερο σκέλος σχετίζεται με την ενσωμάτωση του BadgerDB ως key-value store και τα αποτελέσματα που προκύπτουν από την αλλαγή αυτή.

Από την σύγκριση των αποτελεσμάτων μεταξύ των δύο υλοποιήσεων (πριν και μετά την ενσωμάτωση) στις διάφορες περιπτώσεις επιλογής δίσκου παρατηρήσαμε ότι για την περίπτωση HDD η επιλογή του BadgerDB δίνει ικανοποιητικά αποτελέσματα, βελτιώνοντας την ταχύτητα συγχρονισμού του Client. Αντίθετα, στην περίπτωση δίσκων SSD, η εφαρμοσμένη επιλογή της LevelDB υπερέχει οριακά πραγματοποιώντας ταχύτερα την συνολική διαδικασία του syncing.

Συμπερασματικά, μπορούμε να πούμε ότι η ενσωμάτωση του BadgerDB, μπορεί να έχει νόημα για το Geth από την στιγμή που βελτιώνει την απόδοση σε δίσκους HDD, ωστόσο όσο περνούν τα χρόνια η τεχνολογία απομακρύνεται όλο και περισσότερο από τους HDD, με τους SSD να αποτελούν μια καλύτερη λύση αποθήκευσης σε σχεδόν όλα τα επίπεδα. Συνεπώς, πιθανή βελτίωση της απόδοσης για την περίπτωση των SSD θα ήταν περισσότερο θεμιτή στην μελέτη μας. Παρ'όλα αυτά μέσω της εργασίας αυτής, έχουμε μια πιο καθαρή εικόνα στο τι μπορεί να προσφέρει μια πιθανή αλλαγή key-value store στο Geth Client και το τι χαρακτηριστικά θα ήταν καλό να έχει αυτό για να προσφέρει πιθανότητες ενίσχυσης της απόδοσης.

5.2 Μελλοντικές Επεκτάσεις

Το ζήτημα που μελετήθηκε στα πλαίσια αυτής της διπλωματικής εργασίας θα μπορούσε να εξεταστεί περαιτέρω τουλάχιστον ως προς τρεις κατευθύνσεις. Συγκεκριμένα, αναφέρονται τα ακόλουθα:

- Ενσωμάτωση διαφορετικού, κατάλληλα επιλεγμένου, key-value store στο Geth Client στην θέση της LevelDB. Ένα ενδιαφέρον παράδειγμα είναι το Pebble, το οποία είναι ένα key-value store, εμπνευσμένο από τις LevelDB/RocksDB, γραμμένο σε Go που αναπτύχθηκε στα Cockroach Labs[22]. Πέρα από πιθανή βελτίωση στην απόδοση η ενσωμάττωση του Pebble έχει νοήμα, μόνο και μόνο επειδή η LevelDB δεν διατηρείται πλέον (η έκδοση Go) και δεν θα ήταν επιθυμητό να κολλήσει το Geth σε κάτι που δεν λαμβάνει πια ενημερώσεις κώδικα.

- Δοκιμή των αντίστοιχων μετρήσεων μελέτης του workload για την περίπτωση πλήρους συγχρονισμού (Full Sync) και επιλογή κατάλληλου key-value store για αντικατάσταση της LevelDB. Ενδεχομένως, το Badger να συνιστά μια ενδιαφέρουσα επιλογή και σε αυτήν την περίπτωση και έχει νόημα να εξεταστεί.

- Μελέτη των τεχνικών caching που χρησιμοποιούνται από το Geth και το αν υπάρχει περιθώριο περαιτέρω βελτίωσης τους, εστιάζοντας πιθανώς σε τεχνικές πρόβλεψης δεδομένων.

Παραρτήματα

Παράρτημα Α΄

Ενσωμάτωση BadgerDB στο Geth Client

Α΄.1 Υλοποίηση Interface Geth

Στην προσπάθεια ενσωμάτωσης του BadgerDB στο Geth Client, έπρεπε να υλοποιηθούν τα παρακάτω interfaces, με βάση τον κώδικα του Geth [23]. Συγκεκριμένα οι συναρτήσεις που υλοποιήθηκαν ήταν οι ακόλουθες:

Database

- `func New(file string, cache int, handles int, namespace string, readonly bool) (*Database, error)`
  Η συνάρτηση New αρχικοποιεί την βάση με βάση τις προδιαγραφές που παίρνει σαν όρισμα.

- `func (db *Database) Path() string`
  Η συνάρτηση Path επιστρέφει το path του directory της βάσης.

- `func (db *Database) NewBatch() ethdb.Batch` και `func (db *Database) NewBatchWithSize(size int) ethdb.Batch`
  Η συνάρτηση NewBatch δημιουργεί μια βάση δεδομένων μόνο για εγγραφή(Batch), που αποθηκεύει ως buffer τις αλλαγές μέχρι να κληθεί η τελική εγγραφή. Η συνάρτηση NewBatchWithSize κάνει το ίδιο για προκαθορισμένο μήκος Batch.

- `func (db *Database) Put(key []byte, value []byte) error`
  Η συνάρτηση Put τοποθετεί το δοσμένο ζευγάρι (key, value) στην ουρά.

- `func (db *Database) Has(key []byte) (exists bool, err error)`
  Η συνάρτηση Has ελέγχει εάν το δοσμένο κλειδί υπάρχει. Αν ναι γυρίζει true, αλλιώς false.

- `func (db *Database) Get(key []byte) (data []byte, err error)`
  Η συνάρτηση Get επιστρέφει την τιμή που αντιστοιχεί στο δοσμένο κλειδί.

- `func (db *Database) Delete(key []byte) error`
  Η συνάρτηση Delete αφαιρεί την εγγραφή που αντιστοιχεί στο δοσμένο κλειδί από την ουρά και την βάση.

- `func (db *Database) Flush() error`
  Η συνάρτηση Flush κάνει commit τις εκκρεμείς εγγραφές στον δίσκο.

- `func (db *Database) Close() error`
  Η συνάρτηση Close κλείνει την βάση.

- `func (db *Database) NewIterator(prefix []byte, start []byte) ethdb.Iterator`
  Η συνάρτηση NewIterator δημιουργεί έναν δυαδικό-αλφαβητικό iterator σε ένα υποσύνολο του περιεχομένου της βάσης δεδομένων με ένα συγκεκριμένο πρόθεμα κλειδιού (prefix), ξεκινώντας από ένα συγκεκριμένο αρχικό κλειδί (start) ή μετά, εάν αυτό δεν υπάρχει.

- `func (db *Database) NewSnapshot() (ethdb.Snapshot, error)`
  Η συνάρτηση NewSnapshot δημιουργεί ένα snapshot της βάσης βασιζόμενο στο τωρινό της state.

Batch

- `func (b *BadgerBatch) Put(key []byte, value []byte) error`
  Η συνάρτηση Put τοποθετεί το δοσμένο ζευγάρι (key, value) στο Batch.

- `func (b *BadgerBatch) Delete(key []byte) error`
  Η συνάρτηση Delete αφαιρεί την εγγραφή που αντιστοιχεί στο δοσμένο κλειδί από το key-value store.

- `func (b *BadgerBatch) Write() error`
  Η συνάρτηση Write στέλνει (flush) τα συσσωρευμένα δεδομένα στον δίσκο.

- `func (b *BadgerBatch) ValueSize() int`
  Η συνάρτηση ValueSize ανακτά τον όγκο των δεδομένων που βρίσκονται στην ουρά για εγγραφή.

- `func (b *BadgerBatch) Reset()`
  Η συνάρτηση Reset αρχικοποιεί εκ νέου το Batch, ώστε να μπορεί να ξαναχρησιμοποιηθεί.

- `func (b *BadgerBatch) Replay(w ethdb.KeyValueWriter) error`
  Η συνάρτηση Replay αναπαράγει το περιεχόμενο του Batch.

Iterator

- `func (iter *BadgerIterator) Key() []byte`
  Η συνάρτηση Key επιστρέφει το κλειδί του τρέχοντος ζευγαριού (key, value) ή nil εάν έχει ολοκληρωθεί η διάσχιση.

- `func (iter *BadgerIterator) Value() []byte`
  Η συνάρτηση Value επιστρέφει την τιμή του τρέχοντος ζευγαριού (key, value) ή nil εάν έχει ολοκληρωθεί η διάσχιση.

- `func (iter *BadgerIterator) Next() bool`
  Η συνάρτηση Next μετακινεί τον iterator στο επόμενο ζευγάρι (key, value), ενώ επιστρέφει εάν ο iterator έχει εξαντληθεί.

- `func (iter *BadgerIterator) Seek(key []byte)`
  Η συνάρτηση Seek αναζητά το ζητούμενο κλειδί και εάν υπάρχει μετακινεί τον iterator σε αυτήν την θέση. Εάν δεν το βρει, ψάχνει το αμέσως μικρότερο κλειδί.

- `func (iter *BadgerIterator) Error() error`
  Η συνάρτηση Error επιστρέφει όλα τα errors που πιθανόν έχουν συσσωρευτεί.

- `func (iter *BadgerIterator) Release()`
  Η συνάρτηση Release απελευθερώνει οποιουσδήποτε σχετικούς πόρους.

Ο υλοποιημένος κώδικας της εργασίας μπορεί να εντοπιστεί στο ακόλουθο GitHub repository: https://github.com/SprIoan/GethBadger .

Παράρτημα Β΄

Συλλογή δεδομένων κατά την εκτέλεση των πειραμάτων

Στο παράρτημα αυτό παρουσιάζεται η διαδικασία που ακολουθήθηκε για την συλλογή των μετρήσεων απόδοσης του συγχρονισμού του Geth Client. Συγκεκριμένα, παρουσιάζονται οι εντολές που χρησιμοποιήθηκαν για την έναρξη του Client, καθώς και τα χαρακτηριστικά των μηχανημάτων που αξιοποιήθηκαν για την εκτέλεση των πειραμάτων.

Β΄.1 Επιλογές για το Τρέξιμο του Client

Η λύση που επιλέχθηκε για την εκτέλεση των πειραμάτων ήταν το clone και η εκτέλεση να γίνει από το source code του Go Ethereum [23], το οποίο και επεξεργαστήκαμε για την περίπτωση του BadgerDB. Συγκεκριμένα, μέσα στο directory go-ethereum :

- Εκτελούμε το build χρησιμοποιώντας την έκδοση 1.18.2 της γλώσσας Go.
- Στην συνέχεια εκκινούμε τον client μέσω την εντολής `./build/bin/geth --goerli --verbosity 5 --metrics --pprof &> log.txt` . Τα flags που επιλέγουμε έχουν τις εξής ιδιότητες :
  - Με το flag --goerli επιλέγουμε ο συγχρονισμός του Geth να γίνει στο Goerli Testnet
  - Με το flag --verbosity 5 επιλέγουμε το πόσο αναλυτικά θα είναι τα δεδομένα στο log file και δίνοντάς του τιμή 5, ορίζουμε λεπτομερή περιγραφή (detail), η οποία αποθηκεύεται στο αρχείο log.txt.
  - Με το flag --metrics ενεργοποιούμε την επιλογή για την built-in συλλογή δεδομένων και μετρικών του Geth.
  - Με το flag --pprof ενεργοποιούμε HTTP Server μέσω του οποίου το Geth θα σερβίρει τις μετρικές στο `127.0.0.1:6060/debug/metrics`.
  - Για το μέγεθος cache που θα χρησιμοποιηθεί επιλέγουμε την default τιμή που ορίζει το Geth (1024 MB).
- Ταυτόχρονα εκτελούμε την συλλογή μετρικών μέσω ενός script που εκτελεί διαδοχικά την εντολή `wget 127.0.0.1:6060/debug/metrics` , όπου ανακτούμε τις μετρικές του Geth από εκεί που τις σερβίρει.

Β΄.2 Χαρακτηριστικά Μηχανημάτων

Στην υποενότητα αυτή παρουσιάζονται τα χαρακτηριστικά των μηχανημάτων που χρησιμοποιήθηκαν για την εκτέλεση των πειραμάτων. Συγκεκριμένα έχουμε :

SSD

- `CPU Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz`
- `SSD Size : 372.6G`
- `Byte Order: Little Endian`
- `Address sizes: 46 bits physical, 48 bits virtual`
- `CPU(s): 40`
- `On-line CPU(s) list: 0-39`
- `Thread(s) per core: 2`
- `Core(s) per socket: 10`
- `Socket(s): 2`
- `NUMA node(s): 2`

HDD

- `Intel(R) Xeon(R) CPU X5650 @ 2.67GHz`
- `HDD Size : 1.8T`
- `Byte Order: Little Endian`
- `CPU(s): 24`
- `On-line CPU(s) list: 0-23`
- `Thread(s) per core: 2`
- `Core(s) per socket: 6`
- `Socket(s): 2`
- `NUMA node(s): 2`

Βιβλιογραφία

[1] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. *Decentralized Business Review*, 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf.

[2] D. Vujicic, D. Jagodić και S. Ranđić. *Blockchain technology, bitcoin, and Ethereum: A brief overview*. *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, σελίδες 1–6, 2018.

[3] A. Mohammed, A. Abdulateef και I. Abdulateef. *Hyperledger, Ethereum and Blockchain Technology: A Short Overview*. *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, σελίδες 1–6, 2021.

[4] S. Ferretti και G. D'Angelo. *On the Ethereum blockchain structure: A complex networks theory perspective*. *Concurrency and Computation: Practice and Experience*, 32(12):e5493, 2020.

[5] S. King και S. Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. 2012.

[6] G. Wood και others. *Ethereum: A secure decentralised generalised transaction ledger*. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[7] S. Underwood. *Blockchain beyond bitcoin*. *Communications of the ACM*, 59(11):15–17, 2016.

[8] M. Zochowski. *Why proof-of-work is not viable in the long-term*. https://medium.com/logos-network/why-proof-of-work-is-not-viable-in-the-long-term-dd96d2775e99. Ημερομηνία πρόσβασης: 19-08-2022.

[9] P.O.A. Network. *Proof of Authority: consensus model with Identity at Stake*. https://medium.com/poa-network/proof-of-authority-consensus-model-with-identity-at-stake-d5bd15463256. Ημερομηνία πρόσβασης: 20-08-2022.

[10] A.M. Antonopoulos, G. Wood και G. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018.

[11] *The Ethereum world - how its data is stored*. https://github.com/tpmccallum/ethereum_database_research_and_testing/blob/master/leveldb/leveldb.md. Ημερομηνία πρόσβασης: 22-08-2022.

[12] S. Kushwaha, S. Joshi, D. Singh, M. Kaur και H. Lee. *Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract*. *IEEE Access*, 2022.

[13] T. McCallum. *Diving into Ethereum's world state*. https://medium.com/cybermiles/diving-into-ethereums-world-state-c893102030ed. Ημερομηνία πρόσβασης: 21-08-2022.

[14] K. Kiyun. *Modified Merkle Patricia Trie — How Ethereum saves a state*. https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd. Ημερομηνία πρόσβασης: 24-08-2022.

[15] J. Cook. *Patricia Merkle Trees*. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/. Ημερομηνία πρόσβασης: 25-08-2022.

[16] *Go Ethereum*. https://geth.ethereum.org/. Ημερομηνία πρόσβασης: 22-08-2022.

[17] P. Szilágyi. *Geth v1.10.0. Ethereum foundation blog*. https://blog.ethereum.org/2021/03/03/geth-v1-10-0/. Ημερομηνία πρόσβασης: 23-08-2022.

[18] P. Szilágyi. *Ask about Geth: Snapshot acceleration. Ethereum foundation blog*. https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration/. Ημερομηνία πρόσβασης: 24-08-2022.

[19] P. Robinson. *The merits of using Ethereum MainNet as a Coordination Blockchain for Ethereum Private Sidechains*. *The Knowledge Engineering Review*, 35, 2020.

[20] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau και Remzi H. Arpaci-Dusseau. *WiscKey: Separating Keys from Values in SSD-conscious Storage*. *14th USENIX Conference on File and Storage Technologies (FAST 16)*, σελίδες 133–148, Santa Clara, CA, 2016. USENIX Association.

[21] Smallnest. *Go K/V Databases Benchmark*. https://medium.com/@smallnest/go-k-v-databases-benchmark-cd051279ef22. Ημερομηνία πρόσβασης: 28-08-2022.

[22] *Pebble*. https://github.com/cockroachdb/pebble. Ημερομηνία πρόσβασης: 05-09-2022.

[23] *Official Go implementation of the Ethereum protocol*. https://github.com/ethereum/go-ethereum. Ημερομηνία πρόσβασης: 25-08-2022.

Συντομογραφίες - Αρκτικόλεξα - Ακρωνύμια

HDD	Hard Disk Drive
SSD	Solid-State Drive
CPU	Central Processing Unit
π.χ	παραδείγματος χάριν
PoS	Proof of Stake
PoW	Proof of Work
PoA	Proof of Authority
MPT	Merkle Patricia Trie
EVM	Ethereum Virtual Machine
RTT	Round Trip Time
IO	Input/Output
IOPS	Input/Output Operations per Second
DoS	Denial of Service
GB	Gigabyte
ACID	Atomicity, Consistency, Isolation, Durability
SSI	Serializable Snapshot Isolation

Απόδοση ξενόγλωσσων όρων

Απόδοση	Ξενόγλωσσος όρος
άδειος	empty
ακεραιότητα	integrity
αληθές	true
αλυσίδα	chain
αμετάβλητος	immutable
ανάγνωση	read
απόδειξη	proof
απόδειξη	receipt
απόδειξη συμμετοχής	proof of stake
απόδοση	performance
αποθήκη	store
απομόνωση	isolation
απόρριψη	denial
αρχείο	file
αρχή	start
ατομικότητα	atomicity
βάση δεδομένων	database
γεμάτος	full
γρήγορος	fast
δείκτες αναφοράς	benchmarks
δέντρο	tree
δεσμεύω	commit
δέσμη τεμάχιων	batch
δημόσιος	public
διαδοχικός	sequential
διακίνηση	throughput
διαπροσωπεία	interface
διάρκεια	duration
διάσχιση	iteration
διευθηντήριο	directory
δίκτυο	network
δίσκος	disk
δουλειά	work
εγγραφή	write

εικονικός	virtual
εκτίναξη	spike
ενίσχυση	amplification
ενσωματωμένο	built-in
εξουσία	authority
εξυπηρέτηση	serving
εξυπηρετητής	server
έξυπνα συμβόλαια	smart contracts
επέκταση	extension
επιμελητεία	logistics
επίμονος	persistent
επίπεδος	flat
εργασία	paper
εύρος ζώνης	bandwidth
ημερολόγιο	log
ιδιοκτήτης	owner
καθολικός	global
καθυστέρηση δικτύου	latency
καθυστέρηση	delay
κατακερματισμός	hash
κατάσταση	state
καύσιμο	gas
κεφαλή	header
κλαδί	branch
κλειδί	key
κλήση	call
κόμβος	node
κορυφαίος	top
κρυπτογραφία	cryptography
κώλυμα	bottleneck
λήψη	download
λειτουργία	operation
λεπτομέρεια	detail
μάρκες	tokens
μεταλλωρύχος	miner
μεταφόρτωση	upload
μηχανή	machine
μνήμη	memory
μονοπάτι	path
ομότιμος	peer-to-peer
πελάτης	client
πίνακας	table
πλήρης	complete

ποσοστό	percentage
προγραμματιστής	developper
προηγούμενος	previous
προκαθορισμένος	default
πρόσβαση	access
πρόθεμα	prefix
ρίζα	radix
σειριοποιήσιμο	serializable
σημαία	flag
στάδιο	stage
σύγκριση	comparison
συγχρονισμός	sync
συμβιβασμός	trade-off
συμπίεση	compaction
συναίνεση	consensus
συναλλαγή	transaction
σύστημα ψηφοφορίας	voting system
σφάλματα	errors
σώματα	bodies
ταξίδι μετ΄ επιστροφής	round trip
ταξινομημένος	sorted
ταυτότητα	id
τιμή	value
τροποποιώ	modify
τύπος	type
τυχαίος	random
υπεράνω κόστος	overhead
υπογραφή	signature
υπόλοιπο	balance
φόρτος εργασίας	workload
φύλλα	leaves
χαρακτήρας	string
χρόνος	time
ψευδές	false