National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science & Machine Learning

Study and Resource Analysis of Ethereum Execution Client Bootstrapping

Diploma Thesis

of

KONSTANTINOS TSIARAS

Supervisor: Nectarios Koziris
Professor

Athens, March 2023

National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science & Machine Learning

Study and Resource Analysis of Ethereum Execution Client Bootstrapping

Diploma Thesis

of

KONSTANTINOS TSIARAS

Supervisor: Nectarios Koziris
Professor

Approved by the examination committee on 16th March 2023.

(Signature)	*(Signature)*	*(Signature)*
....	....	....
Nectarios Koziris	Georgios Goumas	Ioannis Konstantinou
Professor	Associate Professor	Assistant Professor

Athens, March 2023

National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science & Machine Learning

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

. . . .
Konstantinos Tsiaras
16th March 2023

Abstract

The Ethereum blockchain, beyond its status as the second largest cryptocurrency by market cap, is a decentralized platform which is currently employed for a variety of use cases. Its programmable nature has allowed its users to develop a plethora of applications on it using smart contracts and it has seen an explosive growth and adoption in the few years since its creation. However, in order to keep such a decentralized network online, numerous nodes around the globe are required to run some specific client software. This client software in Ethereum is responsible both for deciding upon the current canonical chain ("consensus client") but also for downloading and storing the chain's data, locally preserving and maintaining an up-to-date copy of the network's state ("execution client"). The latter in particular requires a lengthy and resource-intensive process to bootstrap, one that has been the subject of much debate but also the catalyst for several innovations.

The purpose of this thesis is to thoroughly understand the workloads an Ethereum execution client is required to handle, and the different ways in which the different client implementations have opted to approach it. We will initially research Ethereum's architecture, the data structures used to persist chain and state data on disk, and we will also be exploring the different sync modes each client implementation utilizes to initially synchronize the network. Finally, we will be performing a comparative resource analysis on them, focusing on the system resources and time needed for their bootstrapping and we will be evaluating these results.

Keywords

Ethereum, Blockchain, Node, Client, Benchmark, Resource Analysis, Bootstrapping, Synchronization, Execution, Storage

Περίληψη

Το Ethereum blockchain, πέρα από τη θέση του ως το δεύτερο μεγαλύτερο κρυπτονόμισμα βάσει κεφαλαιοποίησης, είναι μια αποκεντρωμένη πλατφόρμα η οποία σήμερα αξιοποιείται για ένα μεγάλο πλήθος χρήσεων. Στα λίγα χρόνια ύπαρξής του έχει δει εκρηκτική ανάπτυξη, με την προγραμματιζόμενη φύση του να έχει δώσει τη δυνατότητα στους χρήστες του να αναπτύξουν πληθώρα εφαρμογών πάνω σε αυτό μέσω έξυπνων συμβολαίων (smart contracts). Ωστόσο, η εύρυθμη λειτουργία ενός τέτοιου δικτύου απαιτεί πολυάριθμους κόμβους ανά τον κόσμο να εκτελούν κάποιο προκαθορισμένο λογισμικό-πελάτη (client). Το λογισμικό αυτό είναι υπεύθυνο τόσο για να αποφασίζει για την ορθότητα της τρέχουσας αλυσίδας ("consensus client") όσο και για το κατέβασμα και αποθήκευση των δεδομένων της αλυσίδας, διατηρώντας ένα ενημερωμένο τοπικό αντίγραφο της τρέχουσας κατάστασης του δικτύου ("execution client"). Η εκκίνηση του λογισμικού εκτέλεσης συγκεκριμένα χρήζει μιας χρονοβόρας διαδικασίας με μεγάλες απαιτήσεις σε πόρους συστήματος, η οποία έχει αποτελέσει συχνό θέμα διαλόγου αλλά έχει επίσης δράσει καταλυτικά για αρκετές σχετικές καινοτομίες.

Ο σκοπός αυτής της διπλωματικής εργασίας είναι η διεξοδική κατανόηση του φόρτου εργασίας που το λογισμικό εκτέλεσης στο Ethereum καλείται να διαχειριστεί και τους διαφορετικούς τρόπους με τους οποίους οι διαφορετικές υλοποιήσεις αυτού έχουν επιλέξει να τον προσεγγίσουν. Αρχικά θα διερευνήσουμε την αρχιτεκτονική του Ethereum, τις δομές δεδομένων που χρησιμοποιούνται για να αποθηκεύσουν τα δεδομένα αλυσίδας και κατάστασης στον δίσκο και επίσης θα εξερευνήσουμε τις διαφορετικές μεθόδους τις οποίες κάθε υλοποίηση αξιοποιεί για να συγχρονιστεί αρχικά με το δίκτυο. Τέλος, θα πραγματοποιήσουμε μια συγκριτική μελέτη μεταξύ αυτών, εστιάζοντας στους πόρους συστήματος και συνολικούς χρόνους εκτέλεσης που έκαστη υλοποίηση χρειάζεται για την εκκίνησή της και θα αποτιμήσουμε τα αποτελέσματα αυτής της μελέτης.

Λέξεις Κλειδιά

Ethereum, Blockchain, Κόμβος, Πελάτης, Συγκριτική Μελέτη, Πόροι, Εκκίνηση, Συγχρονισμός, Εκτέλεση, Αποθήκευση

Σύνοψη

Το Ethereum blockchain αποτελεί μια πλατφόρμα ανάπτυξης αποκεντρωμένων εφαρμογών για ένα μεγάλο εύρος τομέων μέσω της χρήσης έξυπνων συμβολαίων (smart contracts). Το νόμισμά του, *Ether* (ETH), επιτρέπει τη μεταφορά αξίας εντός ενός κρυπτογραφικά ασφαλούς δικτύου και αποτιμάται σε εκατοντάδες εκατομμύρια δολάρια, όντας το δεύτερο μεγαλύτερο κρυπτονόμισμα ανά κεφαλαιοποίηση.

Ένα τέτοιου είδους ομότιμο (peer-to-peer) δίκτυο απαιτεί έναν ικανό αριθμό από χρήστες ανά τον κόσμο οι οποίοι να λειτουργούν κόμβους του που θα τρέχουν κάποιο συγκεκριμένο λογισμικό-πελάτη (client). Μετά τη μεγάλη αναβάθμιση που ολοκληρώθηκε το Σεπτέμβριο του 2022, γνωστή ως "The Merge", το λογισμικό αυτό για το Ethereum χωρίστηκε σε 2 κομμάτια. Το προϋπάρχον, γνωστό ως "execution client", παρέμεινε υπεύθυνο τόσο για το κατέβασμα και την αποθήκευση των δεδομένων της αλυσίδας όσο και για τη διατήρηση ενός ενημερωμένου τοπικού αντιγράφου της τρέχουσας κατάστασης του δικτύου. Το καινούργιο, που ονομάστηκε "consensus client", είναι πλέον υπεύθυνο για να αποφασίζει την ορθότητα της τρέχουσας αλυσίδας με βάση το νέο αλγόριθμο συναίνεσης που ακολουθείται πλέον στο Ethereum, το *Proof-of-Stake*.

Έχοντας ορίσει στο Κεφάλαιο 2 κάποιες απαιτούμενες ορολογίες, στο Κεφάλαιο 3 αρχικά εστιάζουμε στις δομές δεδομένων που χρησιμοποιούνται στην αρχιτεκτονική του Ethereum όπου κυρίαρχη θέση έχουν τροποποιημένες δενδρικές δομές *MPT* (Merkle Patricia Tries). Αυτή η δομή αποτελεί μια βελτιστοποιημένη εκδοχή του κλασικού *trie*, με τα δεδομένα να βρίσκονται αποθηκευμένα στα φύλλα αυτού ενώ οι ενδιάμεσοι κόμβοι περιέχουν κρυπτογραφικές αποδείξεις *Merkle* έκαστος για το υποδένδρο του. Τα MPT βρίσκουν εφαρμογή στο Ethereum τόσο συνολικά για την αποθήκευση της τρέχουσας κατάστασης του δικτύου, όσο και επιμέρους για τις δομές αποθήκευσης των πληροφοριών κάθε λογαριασμού αλλά και για την οργάνωση των συναλλαγών εντός του κάθε μπλοκ και των αποδείξεων αυτών. Αξιοποιώντας ιδιότητες των συναρτήσεων κατακερματισμού (και ειδικότερα στην προκειμένη της *Keccak-256*) τα MPT παρέχουν ένα πλήθος από πλεονεκτήματα τα οποία μεταξύ άλλων περιλαμβάνουν τη δυνατότητα επιβεβαίωσης της ορθότητας επιμέρους υποδένδρων αλλά και την εγγύηση πως 2 ίδιες ρίζες δένδρων συνεπάγονται απαραίτητα ίδια δένδρα στην ολότητά τους, λόγω του ντετερμινιστικού τρόπου κατασκευής τους.

Μνεία γίνεται βέβαια και στις βάσεις δεδομένων αποθήκευσης των προαναφερθέντων δομών στο δίσκο που χρησιμοποιούνται από τους διάφορους clients, με προεξέχουσες αυτές που βασίζονται στη χρήση πολυεπίπεδων *LSM δέντρων*, τις LevelDB και RocksDB. Αυτές κάνουν κατάλληλη αξιοποίηση της μνήμης RAM κάθε φορά που χρειάζεται κάποια εγγραφή προκειμένου οι εγγραφές στο δίσκο να γίνονται περιοδικά σε πακέτα, επιτυγχάνοντας ιδιαίτερα υψηλές ταχύτητες εγγραφών αν και παρουσιάζουν χαμηλότερες αποδόσεις όσον αφορά

τις τυχαίες αναγνώσεις δεδομένων.

Στη συνέχεια, στο Κεφάλαιο 4, αναλύουμε τα διαφορετικά είδη κόμβων που υπάρχουν στο Ethereum, τους *full*, *archive* και *light* κόμβους. Οι full είναι το πλέον συνηθέστερο είδος κόμβων που, αρμοδιότητα του οποίου είναι να κατεβάσει το σύνολο των μπλοκ της αλυσίδας, να επιβεβαιώσει την ορθότητά τους και να κατασκευάσει ένα τοπικό αντίγραφο της τρέχουσας κατάστασης του δικτύου. Αφού τα ολοκληρώσει αυτά, είναι σε θέση να υποβάλλει συναλλαγές στο δίκτυο αλλά και να δεχτεί άλλες από τους ομότιμούς του, οι οποίες συναλλαγές θα προστεθούν σε κάποιο επόμενο μπλοκ από τους *validators*. Οι validators δεν αποτελούν ξεχωριστό είδος κόμβου, παρά είναι full κόμβοι που έχουν στην κατοχή τους τουλάχιστον 32 ETH και φροντίζουν για την ασφάλεια του δικτύου και τη δημιουργία νέων μπλοκ, αποκομίζοντας οικονομικά οφέλη από τη διαδικασία αυτή όπως προβλέπεται από το πρωτόκολλο Proof-of-Stake. Οι archive κόμβοι είναι επίσης full κόμβοι οι οποίοι όμως περαιτέρω έχουν την αρμοδιότητα να αποθηκεύουν ένα πλήρες ιστορικό των ενδιάμεσων καταστάσεων (μεταξύ των μπλοκ) από τις απαρχές του δικτύου. Είναι ένα είδος κόμβου με μεγάλες απαιτήσεις χώρου στο δίσκο και πολύ χρονοβόρο στο συγχρονισμό του, για τους οποίους λόγους συνήθως δεν χρησιμοποιείται παρά μόνο για συγκεκριμένες εφαρμογές. Τέλος, οι light κόμβοι έχουν ελάχιστες απαιτήσεις πόρων συστήματος αφού επεξεργάζονται μόνο τις κεφαλίδες (headers) των μπλοκ, αλλά κατά συνέπεια αδυνατούν να επιτελέσουν αρκετές λειτουργίες και εξαρτώνται από τα υπόλοιπα είδη κόμβων για την απάντηση πλήθους αιτημάτων που δύνανται να απαιτηθούν από ενδεχόμενους χρήστες τους.

Ύστερα αναφερόμαστε στην αναγκαιότητα για *ποικιλομορφία* στους clients κάθε τύπου, όπως και εμβαθύνουμε σε κάποιο βαθμό στους consensus clients, παραθέτοντας εν συντομία τις διαφορετικές υλοποιήσεις αυτών και τον τρόπο που αυτοί επιτυγχάνουν τον αρχικό συγχρονισμό τους με το δίκτυο ("bootstrapping"). Ο πλέον διαδεδομένος τρόπος που τον πραγματοποιούν, ονόματι "checkpoint sync", περιλαμβάνει την αξιοποίηση τρίτων ήδη συγχρονισμένων consensus clients και είναι μια διαδικασία που δύναται να ολοκληρωθεί εντός ολίγων λεπτών παράγοντας στο τέλος της μια έγκυρη κεφαλή της αλυσίδας (chain head). Την τελευταία ο execution client μπορεί να πάρει στη συνέχεια για να εκκινήσει τη δική του αντίστοιχη διαδικασία bootstrapping.

Καθώς το κυρίως αντικείμενο της διπλωματικής μας είναι συγκεκριμένα οι execution clients, στο Κεφάλαιο 5 εστιάζουμε ακριβώς εκεί αλλά και στις ποικίλες μεθόδους αρχικού συγχρονισμού τους. Αντίθετα με τους consensus clients, εδώ η διαδικασία αυτή είναι τόσο χρονοβόρα όσο και ιδιαίτερα απαιτητική σε πόρους συστήματος. Το κατέβασμα των μπλοκ είναι μεν και αυτό πολύωρο, ωστόσο πολύ περισσότερος χρόνος δαπανάται στην τοπική κατασκευή ενός αντιγράφου της τρέχουσας κατάστασης του συστήματος.

Ο πιο απλός τρόπος για να επιτευχθεί αυτό θα ήταν να επανεκτελεστούν σειριακά όλες οι συναλλαγές όλων των μπλοκ από την αρχή του δικτύου (γνωστός ως "full sync"), κάτι που λόγω του πλήθους αυτών των συναλλαγών δύναται να διαρκέσει ακόμα και εβδομάδες, καθιστώντας τον εμφανώς μη πρακτικό για το μέσο χρήστη. Μια διαφορετική προσέγγιση συγχρονισμού θα ήταν να προσπαθήσει ο χρήστης να κατεβάσει κατευθείαν την κατάσταση του δικτύου από τους ομότιμούς του, ξεκινώντας από τη ρίζα του σχετικού MPT και διατρέχοντάς το, αιτούμενος κάθε φορά στο δίκτυο όσους κόμβους του λείπουν. Αν και η προσέγγιση αυτή, γνωστή ως "fast sync", παρείχε σημαντικές βελτιώσεις και για αρκετό διάστημα αποτελούσε προεπιλογή των περισσότερων clients, περιλάμβανε πλήθος μεμονωμένων αιτημάτων για μικρά κομμάτια του MPT κάθε φορά, κάτι που καθώς το Ethereum μεγάλωνε οδηγούσε σε ολοένα και περισσότερες καθυστερήσεις λόγω αναμονών για τα αιτήματα αυτά. Τελικώς σαν μέθοδος συγχρονισμού αντικαταστάθηκε από το "snap sync" το οποίο διατήρησε την κεντρική ιδέα επικοινωνίας της εν λόγω κατάστασης μέσω του δικτύου, αλλά ανταλλάζοντας κομμάτια ενός επίπεδου στιγμιότυπου ("snapshot") της κατάστασης το οποίο κάθε client διατηρεί και ανανεώνει δυναμικά. Ο client στη συνέχεια μπορεί και εύκολα ανακατασκευάζει το σχετικό MPT τοπικά, ενώ το στιγμιότυπο αυτό διευκολύνει και άλλες λειτουργίες του. Οι παραπάνω μέθοδοι συγχρονισμού είναι οι πλέον διαδεδομένες στους περισσότερους clients, αν και στο κείμενό μας αναφερόμαστε και στις υπόλοιπες που χρησιμοποιούνται εν γένει.

Επιπλέον παραθέτουμε τις διαφορετικές υλοποιήσεις των execution clients, πλαισιώνοντάς τες και με ιστορικές πληροφορίες όπου αυτό κρίνεται χρήσιμο. Στις υλοποιήσεις αυτές προεξέχουσα θέση έχει ο Geth, που αποτελεί την παλαιότερη υλοποίηση και χρησιμοποιείται σήμερα από περίπου τα δύο τρίτα των κόμβων του δικτύου, ενώ επίσης περιγράφουμε τους νεότερους Nethermind και Besu με τους οποίους θα αντιπαραθέσουμε τον Geth στα μετέπειτα πειράματά μας. Καταληκτικά αναφερόμαστε στον Erigon, που ακολουθεί σημαντικά διαφορετική αρχιτεκτονική από τους προηγούμενους ενώ υλοποιεί μόνο μια σειριακή εκδοχή του full sync ονόματι "staged sync" ως τη μοναδική του μέθοδο συγχρονισμού.

Η διπλωματική μας καταλήγει σε πειραματικό κομμάτι το οποίο αφορά μια συγκριτική ανάλυση μεταξύ των execution clients ως προς το χρόνο συγχρονισμού αλλά και τους πόρους συστήματος που καταναλώνουν. Η ανάλυσή μας αυτή εκτείνεται σε 2 άξονες, τόσο μεταξύ εκτελέσεων διαφορετικών clients με τη χρήση του ίδιου sync mode όσο και μεταξύ εκτελέσεων του ίδιου client με διαφορετικές παραμέτρους. Για το πρώτο σκέλος των μετρήσεων, το sync mode που επιλέχθηκε ήταν το προαναφερθέν snap sync καθώς είναι το μόνο που υποστηρίζεται στην πλειοψηφία των clients (στους τρεις εκ των τεσσάρων, εξαιρουμένου μόνο του Erigon), ενώ για το δεύτερο ακολουθήθηκαν διαφορετικές προσεγγίσεις ανά client.

Οι μετρικές πάνω στις οποίες πραγματοποιήσαμε τις μετρήσεις μας αφορούν τη χρήση CPU (ποσοστό), τη χρήση RAM (GB), το μέγεθος των δεδομένων στο δίσκο (GB), τις αναγνώσεις και εγγραφές στο δίσκο (MB/δευτερόλεπτο) όπως και τις αποστολές και λήψεις από το δίκτυο (MB/δευτερόλεπτο). Αντλήσαμε αυτές τις μετρικές καταγράφοντας τους πόρους της διεργασίας του εκάστοτε client εκτελώντας ένα Bash script[1] το οποίο αξιοποιεί διάφορα εργαλεία γραμμής εντολών στα Linux.

Τα αποτελέσματα αυτών των μετρήσεων παρατίθενται στο Κεφάλαιο 6. Αν και τα γραφήματα εκεί προσφέρουν σίγουρα μια καλύτερη οπτική κατανόηση των συμπερασμάτων μας, εν συντομία παρατηρήσαμε πως ως προς το πρώτο σκέλος της ανάλυσής μας ο Nethermind αποδείχθηκε ανώτερος των Geth και Besu όσον αφορά το χρόνο συγχρονισμού καθώς ήταν σε θέση να αξιοποιήσει καλύτερα τις υψηλές ταχύτητες εγγραφών του SSD μας. Καταληκτικά, διαπιστώσαμε πειραματικά πως η παροχή περισσότερης μνήμης RAM στον Geth αναμενόμενα βελτιώνει την απόδοση του καθώς μειώνει την ανάγκη για εγγραφές στο δίσκο ανά δευτερόλεπτο, ενώ τέλος σε μια ενδεικτική σύγκριση μεταξύ fast και snap sync modes στον Nethermind αποτυπώθηκε η ανωτερότητα του snap sync από κάθε άποψη.

[1]Διαθέσιμο στο GitHub: `https://github.com/TsiarasKon/Ethereum-Client-Metrics`

"Don't trust, verify"

Acknowledgements

Firstly, I would like to thank my supervisor Professor Nectarios Koziris for giving me the opportunity to pursue this thesis, on the innovative field of blockchain technologies. Furthermore, I would like to thank postdoctoral researcher Katerina Doka for her guidance and support during both the research and the experimental part of this thesis. Lastly, I would like to thank my family and all the people near and dear to me for their support during this period.

Athens, March 2023

Konstantinos Tsiaras

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Ethereum blockchain is a decentralized Turing-complete platform that enables the creation of decentralized applications through the use of smart contracts. It seeks to provide a trustless, cryptographically-secure value transfer network, utilizing its native token Ether which is currently the second largest cryptocurrency with a market cap in the hundreds of billions of USD. As its community of developers and users has grown and its adoption increased, Ethereum has enabled decentralized solutions for a wide range of use cases, from financial services and digital identity to supply chain management and gaming.

Being a peer-to-peer network, Ethereum is comprised of thousands of nodes communicating with one another, each running two pieces of client software — a consensus and an execution one. Bootstrapping an execution client in particular is a resource-intensive and often time-consuming process, but is a necessary step for any user wanting to participate in the network. This synchronization process (also known as "initial sync") seeks to create and maintain a local copy of the entire Ethereum blockchain which can then be used to execute smart contracts and retrieve information regarding the network's latest state. This initial sync generally involves downloading and validating all the chain's blocks since the genesis one and re-executing all the transactions included in them, though not all these steps are necessarily always executed.

Our purpose here will be to thoroughly understand the workloads Ethereum execution clients are required to handle, and the different ways in which the different client implementations approach them. This will be done both from a research perspective, where we will explore data structures used and sync modes employed by each client but also experimentally, by benchmarking on the system resources and time needed for their initial sync and then evaluating these results.

Most of the research for this thesis was done in mid-2022, when the major Ethereum upgrade known as "The Merge" was still upcoming. Contrary to some pessimist predictions, the long-awaited Merge was indeed completed on September 15, 2022 and the Ethereum network transitioned from the Proof-of-Work consensus algorithm to Proof-of-Stake. The fact that this transition occurred during the writing of this thesis was seen as an opportunity to incorporate it to the extent that it was relevant. Since it naturally affects the nature of our study's focus — what was formerly simply "a client" split into a consensus and an execution client — we of course need to cover each client type and their responsibilities. As part of that we will be able to examine, albeit briefly, new challenges introduced by Proof-of-Stake and how Ethereum has opted to address them.

In general, while the focus of this thesis is on execution clients and their bootstrapping, it was deemed preferable to approach this subject from a broader perspective. For this reason, in Chapter 2 we will first go over some necessary Ethereum terminology accompanying it with some historical background wherever applicable. In Chapter 3 we will explore the Merkle Patricia Tries and other data structures that are used by Ethereum, along with the storage engines most widely used to persist these structures on the disk (such as LevelDB or RocksDB). Subsequently, in Chapter 4 we will provide some definitions on the different types of nodes and clients, while in Chapter 5 we will be delving deep into execution client implementations and focusing on the different modes they use to initially synchronize the Ethereum network. Finally, our analysis will culminate in our benchmarks in Chapter 6, where we will explain our methodology and present the results of our metrics from several runs of the aforementioned execution clients. These results lend themselves both for inter-client but also intra-client comparisons (using different configurations) and will form the basis of an overall assessment and overview of potential future work in Chapter 7.

The intended contribution of this thesis is manifold. On one hand, a comprehensive analysis of the resource usage of each execution client can prove useful to identify possible bottlenecks in the sync process and is the first step towards designing and implementing architectural improvements to them. Moreover, this resource analysis along with the sync times of each client can help potential node operators make a decision on which to prefer and be better prepared for their hardware requirements.

On the other hand, the exploratory part of this thesis aspires to provide a source of insight on the current state of Ethereum execution clients and everything related to them. In a constantly evolving field as that of blockchains, relevant information is often scattered between outdated documentations, stack exchanges and blog posts, with little cross-referencing to help the reader comprehend the flow of information. While some of the contents of this thesis will inevitably become likewise outdated, a concentrated work detailing their evolution up to the time of this writing is likely to be valuable to any researchers seeking to expand on Ethereum clients, sync modes' architecture or anything else relevant.

Chapter 2

The Ethereum Blockchain

2.1 Historical Background

A Blockchain is a shared, immutable ledger that facilitates the process of recording transactions and tracking assets in a decentralized manner. Bitcoin's creation in 2009 by the pseudonymous Satoshi Nakamoto marked the first successful application of blockchain technology as a finite-supply decentralized currency and ushered in a new asset class, that of cryptocurrencies, currently at a total market capitalization of around a trillion US dollars.

While Bitcoin still remains the undisputed leader among the various cryptocurrencies that have emerged since its creation, its lack of Turing-completeness along with its UTXO (Unspent Transaction Outputs) architecture has meant that it cannot effectively be used for more than a means of transactions or a store of value. Ethereum, taking its name after *ether* (the hypothetical fifth element and invisible medium that permeates the universe according to ancient and medieval science), was conceived in 2013 by Vitalik Buterin as a way to expand on Bitcoin's core concepts by providing a programmable platform on which anyone can create decentralized applications covering a plethora of potential needs. It was developed throughout 2014 and early 2015 by Buterin along with a long list of co-founders (Gavin Wood, Charles Hoskinson, Anthony Di Iorio, Joseph Lubin, Mihai Alisie, Amir Chetrit, Jeffrey Wilcke) some of whom later distanced themselves from Ethereum and went on to develop other prolific blockchains.

Its development has been largely overseen by the *Ethereum Foundation*, a Swiss non-profit organization which, by its own admission[1], does neither own nor control Ethereum but is rather dedicated to supporting it and related technologies. Its philosophy revolves around advocating Ethereum to the outside world and supporting its decentralized ecosystem through the proper allocation of resources so as to maximize its potential to achieve long-term success.

Following a crowd sale in July 2014 an initial number of 72 million coins were sold to individuals who payed in Bitcoin raising a total of $18.3 million and, a year later, Ethereum was officially launched with the genesis block being mined in July 30 2015.

The basics of what initially made up Ethereum and the way it should operate are all described in the *whitepaper* originally published by Buterin in 2014 [1], albeit not in much

[1]Source: `https://ethereum.foundation/philosophy/`

technical detail. The first version of its more technical document (called *yellowpaper*) which contains the formal definitions of the protocol and its building blocks was written by Gavin Wood in the same year [2]. In this chapter we will be going over a few of those Ethereum building blocks, presenting many of the updates that have happened since and adding historical context wherever necessary.

2.1.1 Upgrades and Major Milestones

Despite its widespread popularity, it could be argued that Ethereum is still in its infancy. In the few years since its release, it has undergone several upgrades to its architecture with quite some more underway (see Section 7.1). Changes to any aspect of the network are initially proposed in the form of Ethereum Improvement Proposals (*EIPs*) and thoroughly discussed in the community. Several of those end up being withdrawn or discarded through this process, while others are implemented usually in the form of some *fork* in the network.

While not having a strict definition, a fork is generally described as what happens when a blockchain "diverges into two potential paths forward", something that can be caused either inadvertently or by a change in its protocol. Forks are categorized as either *soft* or *hard*. Soft forks preserve backwards compatibility, with old nodes not needing to upgrade as they can continue to accept new blocks as valid. Hard forks on the other hand mandate all network participants to upgrade their software as otherwise they will perceive any new blocks as invalid. The latter can effectively split a blockchain into two new ones causing different nodes to work on different chains, based on whether they have performed some specific update. Naturally, soft forks are overall less disrupting and new changes tend to be implemented using soft forks whenever possible.

Perhaps the most tumultuous time period in the Ethereum community was the "DAO hack" and everything that followed it. A great lot has been written on that topic and the full extent of the disputes of that time needs not be analyzed at length here[2]. In summary, a Decentralized Autonomous Organization (*DAO*) naming itself "The DAO" was launched in April 2016, gathering funds from numerous users. On June 17 2016 an attacker exploited a security vulnerability of The DAO to transfer around 3.6 million ETH to themselves — valued at around 50 million USD at the time and amounting to about a third of the Ether that had been committed to The DAO thus far. The Ethereum community was split on how to tackle the highest-profile (at the time) attack in the cryptocurrency space with the majority eventually agreeing on the difficult decision to essentially nullify it by performing a hard fork which reverted the chain to a state where the attack had not happened. The DAO hard fork, as it was named, was strongly opposed by a part of the Ethereum community that viewed "code as law" and thus refused to follow the fork, opting to instead continue mining on the original chain. The once original chain was renamed to *Ethereum Classic* (with a much devalued coin, named *ETC*) while the forked one kept the name "Ethereum" as it was supported by the majority of Ethereum developers and users alike.

[2]A detailed timeline of the events around the DAO hard fork along with expanded explanations can be found here: `https://cypherpunks-core.github.io/ethereumbook/appdx-forks-history.html`

Since then there has been a long list of both minor and major upgrades to Ethereum [3], with another notable one being the *London hard fork* which we will put into context in Section 2.2.5.2. However, the most anticipated upgrade to the Ethereum was undoubtedly "The Merge" which was executed on September 15 2022 after years in the making. While largely uncontroversial and well-received, it too caused a fork by a group of users that opposed the network's transition to Proof-of-Stake thus spawning the EthereumPoW (ETHW) coin. ETHW and ETC alike are supported by drastically smaller communities than those of ETH, both in terms of developers and users, and as such our study will be focusing solely on the Ethereum mainnet and its Ether.

The majority of what we will be going over in this thesis, was not affected by The Merge. Wherever this is not the case we may also shortly present the state of things as they were in the pre-Merge era in order to allow for comparisons. As for the details of the changes that The Merge introduced in Ethereum, we will expand on them further on Section 2.3, right after we define some necessary terminology for our study.

2.2 Terminology

2.2.1 Coins and Tokens

Ethereum's native token (or coin) is *Ether* (ETH), sometimes annotated by the Greek uppercase Xi character (Ξ). It is generated by the Ethereum protocol as a reward for the building of new blocks, and it also the only currency accepted to pay for transaction fees.

The smallest and most commonly used denomination of Ether is called a *Wei*, named after cryptocurrency pioneer Wei Dai, and is equal to 10^{-18} ETH. Other named yet obscure denominations include a *Szabo* (10^{-6} ETH) and a *Finney* (10^{-3} ETH).

At the time of writing there are currently more than 122 million Ether in circulation and, contrary to Bitcoin of which only 21 million will ever be mined, there is no cap on the maximum supply of Ether. New Ether is issued every time a new block is created, but The Merge significantly decreased that amount. Additionally, EIP-1559 [4] introduced the burning of part of the Ether used as fee per transaction (which used to entirely be transferred to the miners), a mechanism which effectively decreases the issuance rate of Ether and can even cause it to become deflationary during periods of high network congestion. In our study we will not be focusing on economics of supply and demand and potential economic repercussions that Ether's inflation has on its price.

Furthermore, Ethereum allows the creation of additional custom tokens according to specific *ERC* (Ethereum Request for Comments) standards. ERC-20 is the most widely used for fungible tokens, which can then be used for transactions and interact with smart contracts in much the same way as Ether.

Another common type of tokens is defined in ERC-721, which is the standard for non-fungible tokens (*NFT*s). The uniqueness that NFTs provide (no two NFTs can be identical in the same way that, for instance, two Ether tokens are) has caused them to gain popularity in gaming as well as for representation of art pieces and other types of collectibles, having thus birthed a distinct market for them than the rest of the cryptocurrency space with a daily trading volume in the tens of millions of USD.

2.2.2 Accounts

In order for an entity to send and receive transactions on the Ethereum blockchain they require an account, which is represented by a 42-character hexadecimal address.

There are two types of accounts on Ethereum: externally owned accounts (*EOAs*) and contract accounts. Both of these contains the same following 4 fields:

- `nonce`: A simple counter which is incremented each time the account sends a transaction in order to ensure that each one is only processed once. Also useful to guarantee order of execution if multiple transactions are sent at once (regardless of order of receival, each node will execute the transactions of an account by increasing order of nonce). For contract accounts, this field represents the number of contracts created by the account.
- `balance`: The amount of ETH (in Wei) the account owns.
- `codeHash`: For contract accounts, this is the hash of the code of the account that gets executed on the EVM. For EOAs, this field is always the hash of the empty string. Unlike the rest of the account's fields, `codeHash` is immutable following the account's creation.
- `storageRoot`: A 256-bit hash of the root node of a Merkle Patricia trie that encodes the storage contents of the account. This trie encodes the hash of the storage contents of this account and it is empty by default. We will be investigating the data structures used here in the next chapter.

2.2.2.1 Externally owned accounts

An EOA is made up of a cryptographic pair of a public and a private key. The ownership of this private key is what allows the signing transactions on behalf of the account and, consequently, is what grants an individual who possess it custody over the funds associated with the related account.

To create such an account, a private key is usually made up by randomly generating 64 hexadecimal characters (possibly encrypted with a password) and then a public key is generated from that using the Elliptic Curve Digital Signature Algorithm (ECDSA). The account's address is obtained by concatenating the prefix `0x` (the hexadecimal identifier) with the rightmost 20 bytes (40 hexadecimal digits) of the `keccak256` hash of the public key. It is possible to derive additional public keys (and, consequently, addresses) from a single private key, but it is naturally impossible to derive a private key from public keys.

In order to prevent a frequent terminology confusion, it is worth mentioning that this keypair is not itself a *wallet*. A wallet is simply an interface (e.g. an application) that facilitates the interaction between an individual and their Ethereum account. Lastly, from the above it should be evident that an individual technically does not ever hold any funds themselves - these always reside solely on the blockchain - instead, they hold private keys that are associated with accounts that own funds and they can transfer it to others by again using that same private key.

2.2.2.2 Contract accounts

Smart contracts in the Ethereum blockchain exist as a type of account, one controlled not by private keys but by its own contract code. Contract accounts also have 42-character hexadecimal addresses which is deterministically computed from data of the account which created them. More specifically, the contract address is produced by RLP encoding the creator's address along with their `nonce` and then hashing them with Keccak-256.

Contrary to creating an EOA, creating a contract account is not free because it makes use of network storage and the cost required is paid by the creator account. Once this is done, the smart contract code is deployed on the network and cannot be altered. Furthermore, that code will only be run whenever a transaction is sent to its account — either by an EOA or by another contract account. A smart contract's code can include several different actions, including the transferring of tokens between two accounts and even the creation of new contracts.

2.2.3 Smart Contracts

First conceptualized by Nick Szabo in 1997 [5], a *smart contract* is a merely a computer program which implements some contractual clauses between two entities without the need for a third-party intermediary. Smart contracts achieve trustless and automatic execution and control of whatever on-chain asset was agreed between the parties once some pre-defined condition is met. In that way, the parties' agreement can be securely and reliably carried out, cutting potential commission costs and eliminating accuracy errors that can be caused by involving intermediaries. Additionally, since smart contracts are deployed in advance of an expected condition to be determined and are automatically executed following that, any need for trust between the parties themselves is also eliminated as there is no chance for either one to back down and refuse to honor their part of their agreement in case of an undesirable outcome.

Despite the name, smart contracts are neither particularly "smart" nor legally binding in the way conventional contracts tend to be. In computer science contexts "smart" is typically associated with artificial intelligence and machine learning algorithms, whereas smart contracts codify comparatively simple programs and are not intended to carry out any amount of deep analytics.

Meanwhile, in most cases the legal status around smart contracts remains unclear, with countries generally lacking a legal framework to process anything regarding this innovative type of contractual agreement. Virtually all smart contracts that are employed today in blockchain applications could not realistically be enforced by any court or tribunal[3].

[3]There does, however, exist a classification for smart contracts that have all the elements of a legally enforceable contract in some jurisdiction, called "smart legal contracts" [6].

Several uses for smart contracts have been suggested, some of which have already been implemented by existing applications at least to some degree, including the following:

- Encoding financial agreements: a deal between a person and an insurance company or a mortgage could be made significantly more efficient for all parties involved via the use of a smart contract.
- Elections: while gaining public trust at a large scale could be an issue, smart contracts can allow to reliably record votes without revealing voters' identities, essentially eliminating electoral fraud risks.
- Speculation platforms on financial (related to sports), betting (sports, political) or similar.
- Decentralized Finance (DeFi) applications: Financial applications that provide most of the services that banks traditionally support including but not limited to borrowing and lending peer-to-peer (P2P), interest earning, asset and derivative trading.
- Multisignature accounts: accounts owned by multiple people where moving funds can require a predefined percentage of its owners agreeing.

While in no way exhaustive, the above list should prove that the potential for smart contract applications is undeniable.

Smart contracts are nowadays supported by most major blockchains (though, notably, not Bitcoin) but they were first introduced in Ethereum where they are still most widely adopted. Programmers can write Ethereum smart contracts in a handful of languages but they almost invariably use *Solidity*, a statically typed object-oriented programming language with similarities to C and JavaScript.

Requiring the use of real-world, off-chain data for some of the above applications poses an obvious problem regarding the source of this data. Some trustworthy entity is needed to reliably provide this data and any entity of this kind would in itself result in some degree of centralization, which blockchain developers generally strive to avoid. Furthermore, the issue of deterministic execution arises: all network nodes must somehow receive the same data (regardless of when they requested it) in order to reach a consensus.

Third-party services called *oracles* have been developed to connect smart contracts with the outside world, allowing them to access real-world data quickly on-demand. An oracle acts as a layer that queries, verifies and authenticates external data sources, usually via trusted APIs or reliable data feeds and then relays that information. The oracle of choice for the majority of applications currently on Ethereum is *Chainlink*, created in 2017 along with its own native cryptocurrency, LINK. Chainlink is comprised of a network of thousands of oracles that independently collect the necessary off-chain data, which is then aggregated so that the system can come to a deterministic value of truth for the data requested [7].

2.2.4 Ethereum Virtual Machine

The Ethereum Virtual Machine (*EVM*) is the runtime environment where code execution happens on the Ethereum blockchain. While programmers generally write Ethereum contracts in Solidity, in order for them to be run on the EVM they are first compiled to a low-level, stack-based bytecode language, referred to as "EVM code".

Code execution is a rather simple process and happens in an infinite loop where operations are sequentially executed until the end of the code is reached, an instruction to halt execution is detected or some error occurs. The operations have access to the following three types of storage for any data they might wish to store: a *stack* (a LIFO container in which values are pushed and popped), *memory* (an expandable byte array) and the contract's own long-term *storage* (a trie storing key-value pairs) which, unlike the other two, persists long-term even after the code execution completes.

2.2.5 Transactions

An Ethereum transaction refers to an action initiated by an EOA (i.e. not a contract) and causes an update to the state of the Ethereum network. The simplest type of transaction is transferring ETH from one account to another, which results in a state change by debiting the sender's account and crediting the receiver's account with the respective ETH amount.

All transactions need to be broadcast to the whole network which any node can do, and then validators (which we will detail in Section 2.3.1) are responsible to execute them on the EVM. After one of them does that, they propagate the resulting state change to rest of the network. A transaction may not always be successful and the wait time between its submission and its being processed by a validator can vary significantly, both usually depending on the gas fee set.

The structure of an Ethereum transaction object is the following:

- `from`: The sending address.
- `to`: The receiving address. If an EOA, the transaction will transfer value - if a contract account, the transaction will execute the contract code.
- `value`: Amount of ETH to transfer from sender to recipient (in Wei).
- `nonce`: A sequentially incrementing counter, issued by the originating EOA, which indicate the transaction number from the account. It is used to prevent transaction replay.
- `data`: Optional field which may contain code or a message to the recipient.
- `gasLimit`: The maximum amount of gas units that can be consumed by the transaction.
- `maxPriorityFeePerGas`: The maximum amount of gas to be included as a tip to the validator (formerly miner).

- `maxFeePerGas`: The maximum amount of gas willing to be paid for the transaction (including `baseFeePerGas` and `maxPriorityFeePerGas`).
- `v`, `r`, `s`: The three components of an ECDSA digital signature of the originating EOA. This signature is generated when the sender's private key signs the transaction and confirms the sender has authorized this transaction.

The lifecycle of a transaction begins once an EOA creates the transaction object. The sending account then signs the transaction, thus getting a *transaction hash* (also known as "transaction ID") which acts as a unique identifier for it. Next up, the transaction is broadcast across the Ethereum network, waiting for a validator (formerly miner) to pick it up and verify it - the waiting time for that depends on the network's current traffic as well as the gas fee set for the transaction. Once it gets picked up and added to a block, the transaction is completed and considered successful, otherwise (e.g. due to insufficient gas provided or "bad instructions" in case of a contract deployment) it may be considered *failed*. Other possible transaction states during or after the process we have just described include *pending*, *queued*, *cancelled* and *replaced*.

2.2.5.1 Types of Transactions

The different types of transactions supported in Ethereum are the following:

- Regular transactions: a transaction (of either Ether or some other token) from one account to another.
- Execution of a contract: a transaction that interacts with a deployed smart contract, in which case the recipient address is the smart contract address.
- Contract deployment transactions: a transaction without a recipient address, where the data field is used for the contract code. This transaction creates a smart contract account as described in Section 2.2.2.2.

The first two of these are sometimes referred to as *message call* transactions while the last one is also known as *contract creation* transaction.

On the more technical side, while Ethereum originally only had one format for transactions, it eventually evolved to support multiple types of transactions and to allow for new features without affecting legacy transaction formats. EIP-2718 [8] defines the typed transaction envelope that is currently used and is defined as `TransactionType || TransactionPayload` (where `||` is the byte concatenation operator). `TransactionType` in this context represents a number between `0` and `0x7f` allowing for great future extensibility up to theoretical maximum of 128 possible transaction types. EIP-2718 does not itself define any transaction types, but proposals that use this new standard include the aforementioned EIP-1559 as well as EIP-2930 [9].

2.2.5.2 Gas and Fees

Gas is the fuel of Ethereum. Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee. Gas refers to the fee required to conduct a transaction on Ethereum successfully. Gas fess are paid in Ethereum's native currency, Ether, and are denominated in Gwei (10^9 Wei = 10^{-9} ETH).

The way transaction fees on the Ethereum network are calculated changed with the London Upgrade on August 5 2021 which introduced EIP-1559 to the network [4]. Before the London Upgrade, Ethereum had fixed-sized blocks which, in times of high network demand, regularly operated at total capacity with users having to wait long queues in order to get their transactions included in some block. EIP-1559 introduced variable-sized blocks to Ethereum, each having a target of 15 million gas on average but with that being dependant on network demand. To understand this process better, the following parameters need to be explained:

- `baseFeePerGas`: The bare minimum fee required to send a transaction on the network, set by the network itself based on how full the latest block was. The Ether provided for this fee will be burned.
- `maxPriorityFeePerGas`: This fee is intended as a "tip" to the validator (formerly miner) and acts as an incentive for them to introduce that transaction to the current block. The greater this tip is, the more likely it is that the transaction will be included in the next block.
- `maxFeePerGas`: The maximum fee the user is willing to pay for that transaction and is generally equivalent to `baseFeePerGas` + `maxPriorityFeePerGas`. Should a user manually sets this gas limit higher than it is needed to be, any Ether unused by the EVM will be refunded to the their account.

By introducing the concept of `baseFeePerGas` the London Upgrade resulted in making gas fees more predictable, a common grievance of Ethereum users in the period leading to it. Furthermore, by introducing the concept of fee burning, despite diminishing the (then) miners' profits at least in the short-term, it diminishes Ether's inflation — even causing it to become deflationary in periods of high network demand when `baseFeePerGas` is increased — which was deemed advantageous to Ethereum's long-term prospects.

Ultimately, gas fees help keep the Ethereum network secure. Since a fee is required for any computation executed on the network, bad actors are disincentivized to spam the network. On top of that, accidental or hostile infinite loops in code are avoided since each transaction is required to set a limit to how many computational steps of code execution it can use, with the fundamental unit of computation being "gas".

2.2.6 Blocks

A block consists of a batch of transactions organized in a trie structure (that we will examine in the next chapter), along with a few other fields helping to identify it but also to facilitate the work of the nodes that will be required to process it. A short overview of all the fields in a block can be seen in Figure 2.1.

Blocks are strictly ordered, each pointing to its previous one through `parentHash`, all the way back to the first block ever mined — which is known as the *genesis* block and

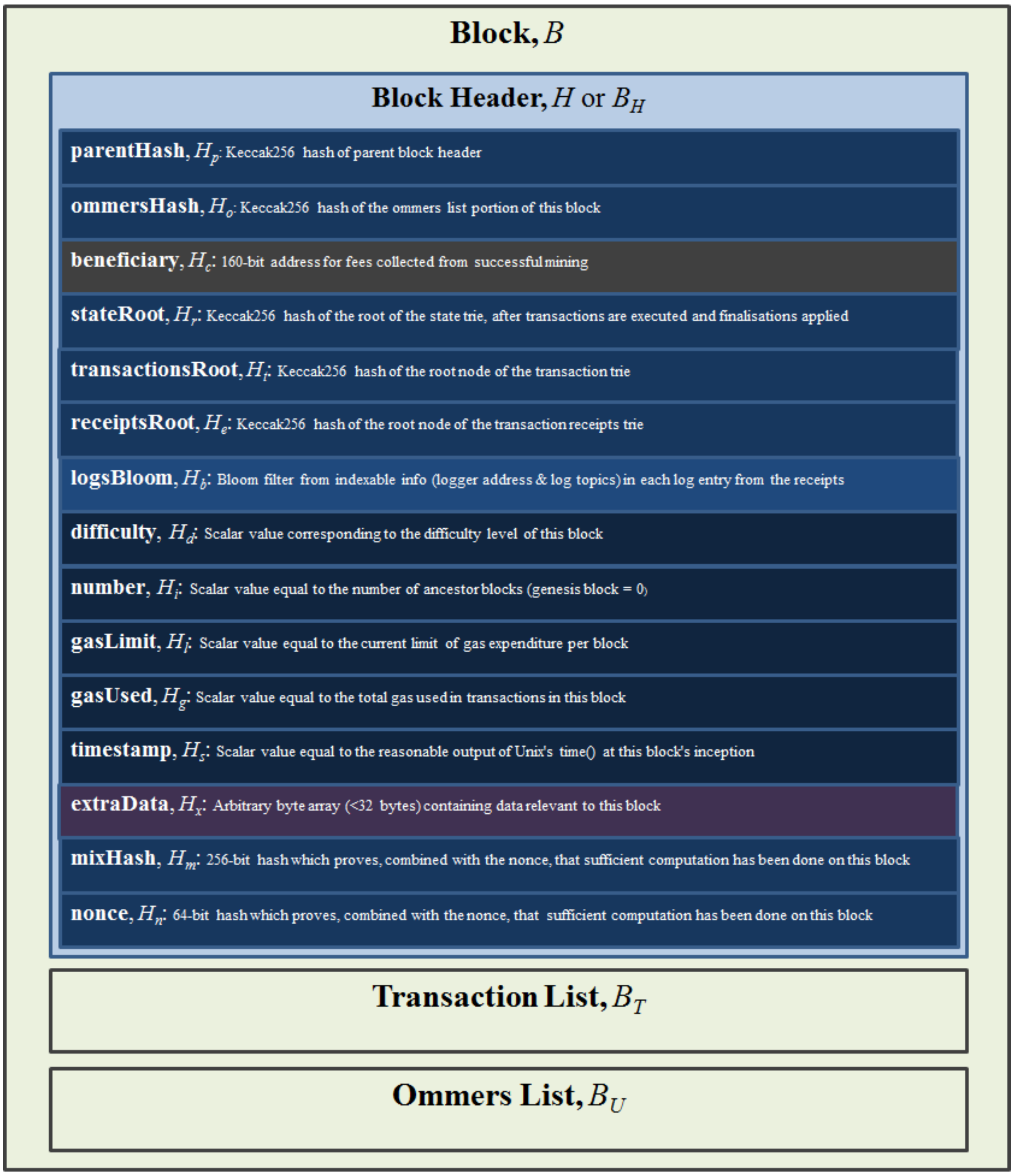

Block, B
Block Header, H or B_H
parentHash, H_p: Keccak256 hash of parent block header
ommersHash, H_o: Keccak256 hash of the ommers list portion of this block
beneficiary, H_c: 160-bit address for fees collected from successful mining
stateRoot, H_r: Keccak256 hash of the root of the state trie, after transactions are executed and finalisations applied
transactionsRoot, H_t: Keccak256 hash of the root node of the transaction trie
receiptsRoot, H_e: Keccak256 hash of the root node of the transaction receipts trie
logsBloom, H_b: Bloom filter from indexable info (logger address & log topics) in each log entry from the receipts
difficulty, H_d: Scalar value corresponding to the difficulty level of this block
number, H_i: Scalar value equal to the number of ancestor blocks (genesis block = 0)
gasLimit, H_l: Scalar value equal to the current limit of gas expenditure per block
gasUsed, H_g: Scalar value equal to the total gas used in transactions in this block
timestamp, H_s: Scalar value equal to the reasonable output of Unix's time() at this block's inception
extraData, H_x: Arbitrary byte array (<32 bytes) containing data relevant to this block
mixHash, H_m: 256-bit hash which proves, combined with the nonce, that sufficient computation has been done on this block
nonce, H_n: 64-bit hash which proves, combined with the nonce, that sufficient computation has been done on this block
Transaction List, B_T
Ommers List, B_U

Figure 2.1. *Block structure* [4]

was mined on July 30 2015. Except in rare cases, all participants on the network are in agreement on the exact number and history of blocks at any given time and are working to batch the current live transaction requests into the next block.

New blocks are constantly being put together by batching transactions and the prioritization of transactions is being done by considering the gas the transactions' senders are willing to spend for their transactions to be included in the block, among other factors. This optimization process has long been an important aspect for maximizing profitability and this maximal possible profit is called *maximal extractable value* (MEV, formerly

[4]Source: Lee Thomas, https://ethereum.stackexchange.com/a/6413/100602

meaning *miner* extractable value).

Pre-merge those new blocks were created and propagated to the network by miners every 13 seconds on average, while now this responsibility rests on validators with a block time of (almost) exactly 12 seconds. Furthermore, blocks need to be prevented from being arbitrarily large, since that would cause less performant nodes to gradually be unable to keep up with the network due to space and speed requirements. As briefly mentioned in Section 2.2.5.2, this is currently achieved in Ethereum by having a block limit not in the number of transactions but rather the gas expended by them. That limit is 30 million gas per block (in cases of high network demand), with the target size being 15 million gas.

2.2.7 Networks

What is generally referred to simply as "the Ethereum network" is also known as "mainnet". This term is sometimes used to differentiate it from various test networks, called "testnets". Testnets are of great value to application developers who can test their smart contracts there, being akin to a staging environment in the traditional development cycle (with the mainnet respectively corresponding to the production environment). Test Ether in testnets is given freely to experiment with, often through applications that directly sent a fixed amount of it to a requesting wallet, called *faucets*. Naturally, Ether (or any other token) in a testnet cannot be transferred to the mainnet and does not hold any monetary value. Testnets are not required to share many of their network parameters with the mainnet including block sizes, block times and even consensus algorithm (more on these next up). The most popular testnets at the time of writing are Sepolia, Goerli and Rinkeby, though new ones are regularly created to account for new testing needs.

Our focus in this thesis will be the Ethereum mainnet which is a public (or permisionless) network, as are all the aforementioned testnets. It is noted that private (or permissioned) networks can also be created in Ethereum, which can likewise be useful in the process of developing an application before deploying it to the mainnet. Such a network can be accessed by Ethereum nodes by parameterizing them with its configuration consisting of a specific *chain ID* and a custom genesis block.

2.2.8 Consensus Algorithms

A fundamental concept in distributed systems is that of consensus and how to achieve it, i.e. the way in which all the network's participants agree on what its global state is. In blockchain systems these are generally Proof-of-*X*, the most widely used of which are the following.

2.2.8.1 Proof-of-Work

Proof-of-Work (*PoW*) is a form of cryptographic proof in which one party proves to others that a certain amount of computational effort has been expended to perform a specific task. This concept was originally invented in 1993 by Moni Naor and Cyuntia Dwork [10] in the context of deterring denial-of-service attacks and combating spam but was first formalized in 1999 by Markus Jakobsson and Ari Juels [11]. Its popularization however unarguably came when Satoshi Nakamoto used it as a foundation for Bitcoin's permissionless decentralized network [12] and PoW has served as a model consensus algorithm for several of the blockchains that have since been created.

In PoW, network participants called *miners* compete with each other in order to solve some arbitrary mathematical puzzle. Whoever solves it first gets to add a block to the blockchain, containing the transactions of his choice — most usually just based on the amounts of "tips" the senders of those transactions have included for the miner. An additional *block reward* transaction is added crediting the miner for a fixed amount (which used to be 2 ETH in pre-merge Ethereum) and that is how new coins are minted.

In the case of Ethereum, the PoW algorithm *Ethash* required miners to iteratively increment the block's nonce so that the hash of the resulting entire block (including both that nonce and the miners' transactions of choice) would be lower than a certain target value. Due to the one-way nature of hash functions, the best and only way to achieve that is through trial and error by attempting to hash the block with different nonce values. Bitcoin's PoW also works in a similar way, using the SHA-256 hash function and requiring that the resulting block's hash begins with a certain number of zeroes.

A useful characteristic of PoW is that the difficulty of the problem requiring solving by the miners can be tweaked as the network's computational power fluctuates so as to approximately achieve a desired outcome, for example that of a constant average block time. As such, the exact target value that a block's hash needs to be lower than (or the number of zeroes at the start of the block's hash in the case of Bitcoin) is tunable through a `difficulty` network parameter, communicated through the block header and automatically updated whenever necessary based on the speed on which the latest blocks have been produced. But while the solution itself may take however long based on the parameters specified, confirming that the solution is indeed correct (and thus that the resulting block is valid) is effectively instantaneous by anyone, a necessary property for the entire system to function.

Security in PoW derives from the immense computational effort required to defraud the chain. While a malicious actor could theoretically randomly get to mine some single block, consistently creating malicious yet valid blocks would require 51% of the network's mining power (also known as a *Sybil Attack*), a feat realistically impossible and definitely unprofitable for any entity given a large enough network (such as Ethereum or Bitcoin).

Lastly, the mathematical problem in question for both Bitcoin and Ethereum used to originally require little overall computational power, even being solvable by a home PC of average hardware specifications. Gradually however, it became infeasible to do so — at a rate to be able to keep up with the rest of the miners — and users had to *pool* together in order to remain profitable. This was possible in pre-merge Ethereum for users with high-end GPUs, but Bitcoin mining has long been unprofitable on consumer-grade hardware and is being performed only on ASICs (Application-specific integrated circuits). This ever-increasing necessity for more advanced hardware is widely regarded as the largest drawback of PoW, as it involves growing amounts of energy being "wasted" with the sole purpose of securing the network.

2.2.8.2 Proof-of-Stake

Proof-of-Stake (*PoS*) is a consensus algorithm that centers around randomly selecting network participants called *validators*, in proportion to their quantity of holdings. First introduced by Sunny King and Scott Nadal in 2012 and applied to created Peercoin [13], PoS is currently used by most major blockchains (notably excluding Bitcoin), especially newer ones and those that provide smart contract support.

In a blockchain using PoS, validators are expected to operate constantly and are chosen at random to construct blocks, propagate them through the network and receive a reward for doing so. In order for an entity to become a validator they need to pledge or "stake" a minimum amount of coins that they own which consequently provides them with an economic incentive to behave honestly.

Furthermore, most PoS blockchains implement a concept known as *slashing*, where validators lose a portion of their stake, usually triggered if they are unjustifiably offline for long periods of time or if they are found to have behaved dishonestly. In most blockchains a validator can possibly lose even the entirety of their stake, depending on the severity of their misbehaviour that caused them to be slashed. Even if a group of validators ever assembled to corrupt the chain, the amount of coins that they risk to lose (and also devalue, buy discrediting the network's reliability) all but ensures that it would be in their own self-interest to not act maliciously.

Slashing is also used to combat *Nothing-at-Stake*, an issue exclusive to PoS blockchains where a validator is incentivized to work on forks of the main chain at the same time [14]. While in PoW working on separate network forks equates to wasting computing resources, in PoS working for multiple forks does not require splitting one's stake nor does it inherently incur any other meaningful cost whatsoever. Consequently, as long as it is even remotely possible that a fork will eventually result in the longest chain, working on it too would maximize a validator's likely payout. A punishment is thus introduced so as to prevent such behavior, which is applied to a validator if it is found to be working on several forks at the same time or simply if it is found to be validating the wrong chain — the latter can also occur inadvertently to an honest validator but it nonetheless acts as an incentive to always be working on the chain with the highest likelihood of being the longest.

While energy waste is indeed minimal in PoS, there are other aspects in which it is inferior to PoW. To begin with, due to its recency as a consensus mechanism, PoS' security is not as proven as that of PoW. Additionally, from a validator's standpoint, participating in the network may appear unappealing as not only may their stake be subject to a lock period during which they cannot access ("unstake") their funds, but it can also be entirely at risk due to the aforementioned concept of slashing.

More crucially however, tying the network's security to the validators' stake ensures that the wealthiest users will posses more influence on it and also *passively* be getting most of the rewards, while the less affluent ones may outright be prevented from becoming a validator due to not meeting certain minimum criteria. While this last issue is not exclusive to PoS (especially if one considers the costs of PoW mining equipment), it is certainly more evident here and that is why PoS has been widely criticized as causing a decrease in the network's decentralization, considered by many an end in itself.

These arguments against it of course do not lessen the fact that PoS is a more scalable consensus algorithm, which solved the long-term sustainability issue of constantly increasing energy waste in PoW and is rightfully credited with ushering in a new era for blockchains. As for the more technical security concerns, these can all be sufficiently tackled in the confines of a well-designed protocol as we will later examine is the case with Ethereum.

2.2.8.3 Proof-of-Authority

Proof-of-Authority (*PoA*) is a comparatively new consensus algorithm (first coined as a term by Ethereum co-founder Gavin Wood in 2015), quite similar to PoS. It likewise involves validators who are responsible for the construction and propagation of blocks, but these validators are entities with known identities, placing their reputation at stake instead of — or along with — their funds.

PoA is considered to provide even greater scalability than PoS (and much more than PoW), being able to securely provide very high throughput. However, since the validators here have to be identified, trusted and selected by the network, their number tends to be relatively small. This fact causes PoA networks to be even more centralized and hence more susceptible to corruption and even manipulation due to the identifiable nature of the validators involved — there have even been cases of a malicious actor taking control of a PoA network by directly hacking a few (just over half) publicly known validators.

PoA is much less widespread than the previous two consensus mechanisms (few widely adopted blockchains use it) and has never been relevant to the Ethereum mainnet. We are however mentioning it here for the sake of completeness since it is often preferred by Ethereum testnets, where there would not be enough miners to apply PoW but PoS would also be impractical to implement due to the testnet's coins' lack of monetary value.

2.3 The Merge

The most major upgrade in the history of the Ethereum network was labelled "*The Merge*" and was completed on September 15 2022. Long anticipated as "Ethereum 2.0" (or "Eth2"), that naming has since been deprecated so as to clarify that the new upgrade does not constitute some new chain separate from "Eth1", but merely its continuation into a new era.

2.3.1 Consensus algorithm change

On December 1 2020, a PoS blockchain was created by the Ethereum developers called the *Beacon Chain*, running alongside the original PoW Ethereum mainnet. Its purpose was to ensure that the PoS consensus logic was sound and sustainable before enabling it on Ethereum mainnet. When almost two years later this was assured, the Beacon Chain was instructed to accept transactions from the original Ethereum chain, bundle them into blocks and then organize them into a blockchain using a PoS based consensus mechanism. At the same moment the original Ethereum clients turned off block propagation and consensus logic, handing all that over to the Beacon Chain. This event is known as "The Merge", after which there were no longer two blockchains but only the single PoS Ethereum chain.

By far the most cited advantage of this change was its environmental friendliness, since PoS manages to secure the network without necessitating miners to waste electricity on "useless" calculations. It is estimated that by switching to PoS, Ethereum achieved an energy consumption reduction of over 99.9%.

In PoS Ethereum all functions regarding the network's operation and security are completed by *validators*. A validator is a virtual entity that participates in the consensus of the Ethereum protocol by staking 32 ETH and is represented by a public key, a balance, and some other properties. The rules that all validators must abide by and the exact way in which validators communicate but can also be slashed for misbehaving so as to address security concerns such as those described in Section 2.2.8.2 were all initially presented in a protocol called *Casper* [15]. Combining Casper with *GHOST*, a fork choice algorithm originally developed for PoW, led to the development of the *Gasper* protocol, which is the one currently used by PoS Ethereum [16].

Time is now divided in 12 second units called "slots" and in each one a single validator is randomly selected to propose a block — aptly called *proposer*. A period of 32 slots is also called an "epoch", after each of which validators are shuffled for security purposes into "committees" in a pseudorandomized manner by the same process that chooses proposers, called RANDAO. Every epoch the validators will propose an "attestation" (vote) to the network so as to help reach a consensus for the current state of the Ethereum network, for which they are rewarded. Assuming all validators are online and fully functional, there will be a block created in every slot resulting in a block time of exactly 12 seconds. In practice, occasionally a validator might be offline when called to propose a block, meaning that a slot can sometimes go empty and thus the average block time is ever so slightly more than 12 seconds. This is in contrast to pre-merge Ethereum and PoW-based blockchains in general, where block times are probabilistic and tuned by the mining difficulty.

Under Gasper, a block must pass through a two-step upgrade procedure in order to be considered *finalized*, that is when the block and the transactions it contains are for all intents and purposes considered valid and thus part of the "correct" (*canonical*) chain. Firstly, two-thirds of the total staked ETH must have voted in favor of block's A inclusion in the canonical chain, which upgrades the block's status to "justified" (or "safe"). A justified block is unlikely to be reverted, but that may still happen under certain conditions such as to restore the network's state following a large-scale coordinated attack. Then when a block B is also justified on top of A, block A is upgraded to "finalized" which is in itself a strong commitment that it will keep being considered as part of the canonical chain. It must be noted that these block upgrades do not happen in every slot but rather periodically every few blocks which are known as "checkpoints".

The only way a finalized block can be reverted is by creating an alternative finalized chain which requires two-thirds majority. An attacker could only ever achieve that either

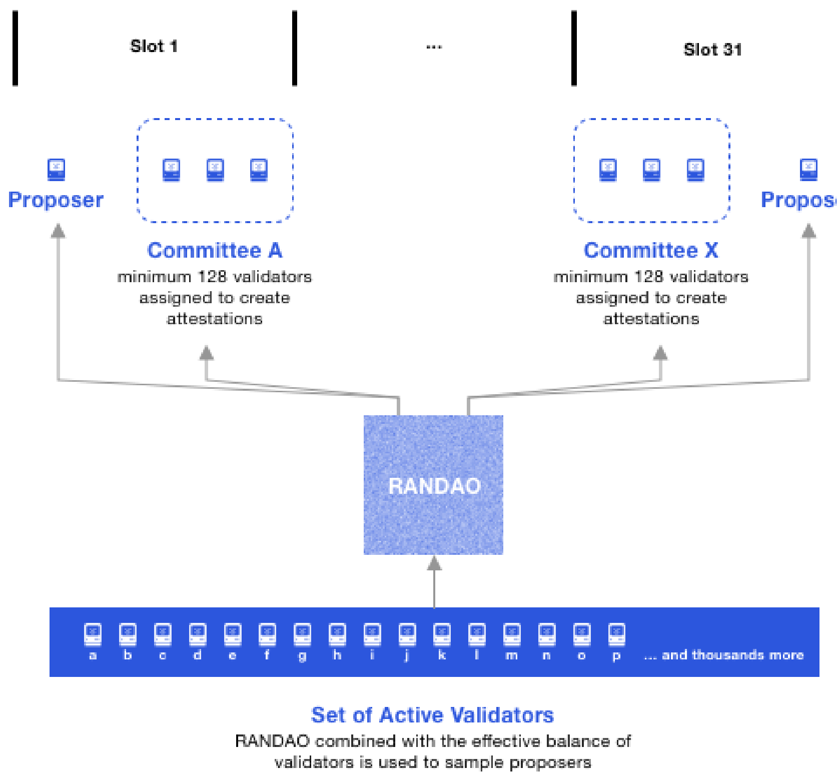

Slot 1
...
Slot 31
Proposer
Committee A
minimum 128 validators assigned to create attestations
Committee X
minimum 128 validators assigned to create attestations
Proposer
RANDAO
a b c d e f g h i j k l m n o p ... and thousands more
Set of Active Validators
RANDAO combined with the effective balance of validators is used to sample proposers

Figure 2.2. *Committee and Proposer selection through RANDAO* [5]

by controlling two-thirds of the total staked ETH or by owning and destroying one-third of the total staked ETH (by using it to double-vote, a behaviour that is maximally punished through slashing). The former is already an improvement from the usual "51% rule" by raising the bar for a catastrophic attack to the network to two thirds, while the latter could only be performed once and would require the attacker to burn an amount of ETH worth billions of USD, a feat unreasonable enough in any real-world scenarios.

An interesting property deriving from the above is that we now have an in-protocol definition of *finality*. By contrast, finality in PoW is necessarily probabilistic and based on the fact that the older a block is in the currently longest chain the more likely it is that it is valid and thus, after enough blocks, its contained transactions are agreed upon to be probabilistically final.

At the time of writing there are over half a million Ethereum validators and their number is monotonically increasing as they are still not able to unstake their ETH. With The Merge having been successfully completed, stake withdrawals are intended to be enabled with the upcoming Shanghai upgrade, planned for April 2023. Even then, the exit rate of validators will be limited for security reasons, in order to prevent a potential mass exodus.

[5]Source: `https://ethos.dev/beacon-chain`

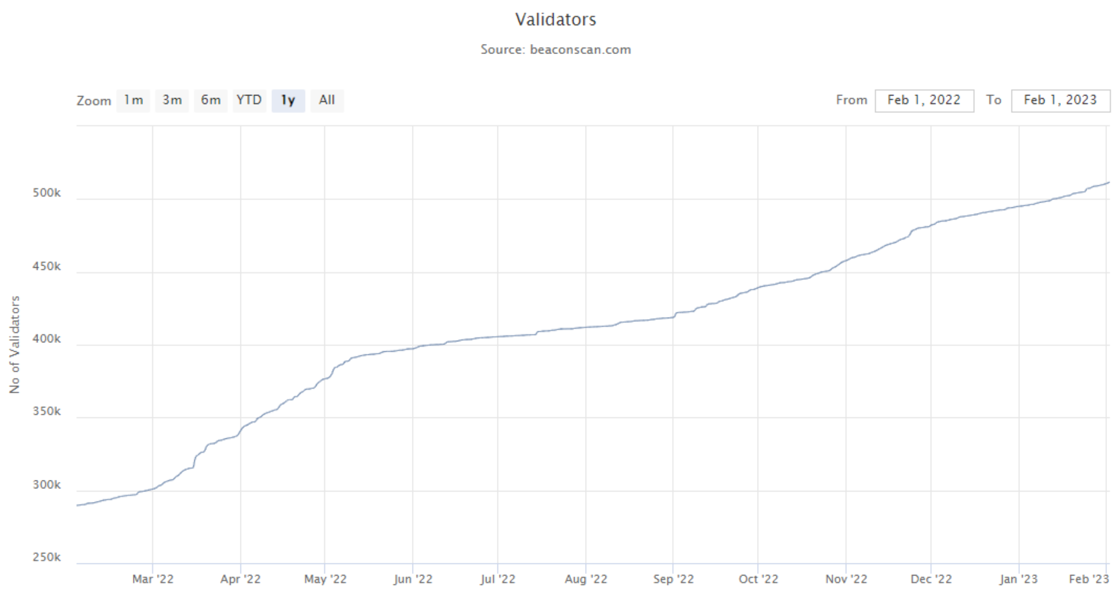


Figure 2.3. *Number of Ethereum mainnet validators over time* [6]

2.3.2 Consequences on Ether's economics

A major consequence of The Merge is a drastic decrease to Ether's issuance rate. By replacing the ∼13,000 ETH per day mining rewards with a mere ∼1,600 ETH per day staking rewards, the new ETH issuance is reduced by roughly 90% compared to pre-Merge levels. Furthermore, taking into account the burning of ETH introduced in the London upgrade, on an average day ETH can have close to zero net inflation and even become deflationary under periods of moderate network use [17].

Furthermore, the existence of staking on the network caused a variety of DeFi apps to see their attraction crucially diminished (if not outright deprecated) as their business model was largely centered around on gaining passive income on Ether. Even for the majority of users that lack 32 ETH to stake, they can stake however little ETH they have through various third parties including some of the largest cryptocurrency exchanges — albeit with slightly reduced rewards. As such, for a majority of network users and/or investors who want some (relatively) riskless passive income on their Ether, directly or indirectly staking seems a solid choice and will inevitably be the obvious one, once validators' stakes withdrawals are enabled.

It is lastly noted that the amount of staking rewards in PoS Ethereum is inversely correlated with the number of validators (or, more specifically, the amount of total staked ETH). This is an apt mechanism to ensure the network's health should the number of validators ever get too small since in that scenario the staking rewards would increase and new users would be financially incentivized to stake their ETH.

[6] on February 1 2023; Source: `https://beaconscan.com/stat/validator`

Chapter 3

Storage Architecture

3.1 Data Structures

In order to fully comprehend the workload of and the challenges faced by an Ethereum client, we first need to be aware of what kind of data it needs to store but also the way in which it stores it. Ethereum's main data structure, which is used for multiple different purposes, is a version of a *trie* called modified Merkle Patricia Trie (*MPT*). In this chapter we will go over the architecture of MPTs and then examine how Ethereum takes advantage of them.

3.1.1 Radix Tries

A trie, taking its name from the word re-*trie*-val, is a tree data structure that is used to store and easily retrieve a set of keys — most usually strings which will be our focus here. Unlike conventional (binary) search trees, a node in a trie does not store its associated key but rather a single character. By concatenating nodes' characters in the path from the trie root to any given leaf node (i.e. traversing the trie depth-first) we can retrieve a string that has been stored in the trie. A simple example of such a trie can be seen in Figure 3.1.

An obvious issue with the tries described above is that they are quite inefficient for strings of any decent length. Storing a single 64 character-long string (a frequent use case in Ethereum) would necessitate a trie of equal depth, with a single node at each level for each character and each lookup or delete would require traversing the full 64 nodes. While this issue would naturally become less pronounced as we continued to add strings to our trie, it is evident that an optimization of some kind could significantly improve efficiency and it comes in the form of *radix tries*.

A radix trie represents a space-optimized trie (or *prefix trie*) in which only-child nodes are merged with their parents. This results in internal nodes having at most r amount of children, with the radix $r \geq 2$ of the radix tree being a power of 2. This radix represents all the possible sorting combinations of the binary data that can be stored in the radix trie. In the simplest case, a binary radix tree would only ever have two children for any given node, while a radix tree used to store English words (i.e. strings from an alphabet of 26 characters) would have $r = 32 = 2^5$ (the nearest power of 2) and, consequently, up to 5 children per node.

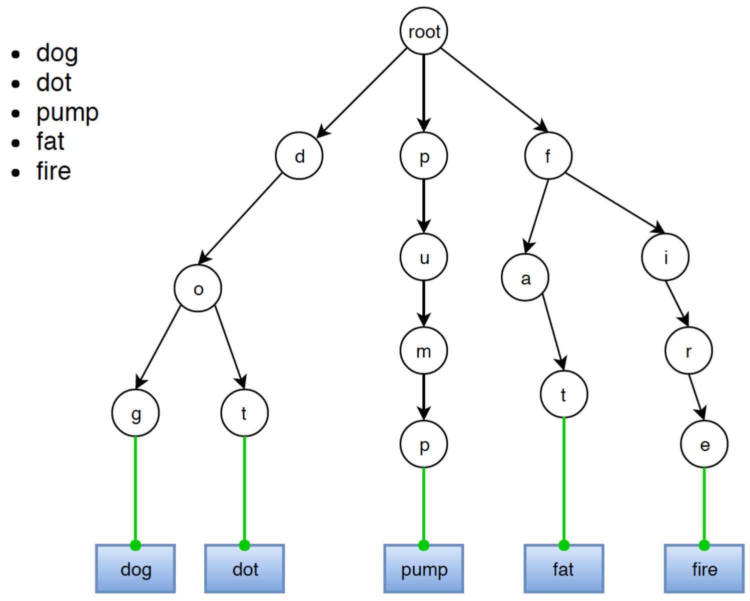


Figure 3.1. *An example of a trie*

A *Patricia trie* (sometimes stylized as "PATRICIA" which stands for "Practical Algorithm To Retrieve Information Coded in Alphanumeric"), despite often being used as another equivalent term for "radix trie" by some sources, is in fact a variant of the binary ($r = 2$) radix trie. Its difference lies in that rather than explicitly storing every bit of every key, the nodes in a Patricia tree store only the position of the first bit which differentiates two sub-trees and, as such, there are only ever n amount of nodes to contain n items [18]. The Ethereum whitepaper never mentions radix tries by name but instead solely focuses on Patricia ones. This semantic ambiguity is not particularly relevant to our study so henceforth we will likewise only be focusing on "Patricia tries".

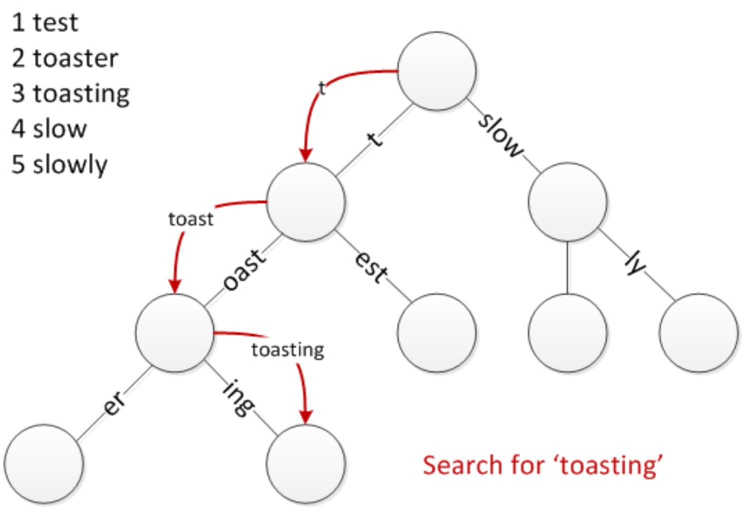


Figure 3.2. *An example of a patricia trie*

3.1.2 Merkle trees

A *Merkle tree* (also commonly known as *hash tree*) is a tree in which the data is stored in the leaf nodes and every non-leaf node contains the cryptographic hash of the contents of its child nodes. It was named after Ralph Merkle who patented it in 1979 [19] and is regarded as the first of a type of data structures that are known as *authenticated data structures* (ADS) [20]. Its utility derives from the fact that it allows parties to verify the consistency of a given dataset without needing to exchange the dataset itself in its entirety.

While the usefulness of comparing hashes to protect against maliciously or unintentionally corrupted data is evident, that same functionality could be achieved by simply using a *hash list*. That would entail simply hashing the concatenation of the hashes of all the data in our dataset and the result would serve the same purpose as a hash tree's root node. However, the tree structure further introduces a mechanism called *Merkle proof*, comprised of a leaf node and the tree's branch consisting of all of the hashes going up along the path from that node up to and including the root node. Someone reading such a proof can verify that the hashing, at least for that branch, is consistent going all the way up the tree, and therefore that the given leaf actually is at that position in the tree [21]. Using the above process one can efficient verify the membership of a value in a Merkle tree but the potential non-membership of it can similarly and equally efficiently be verified as well.

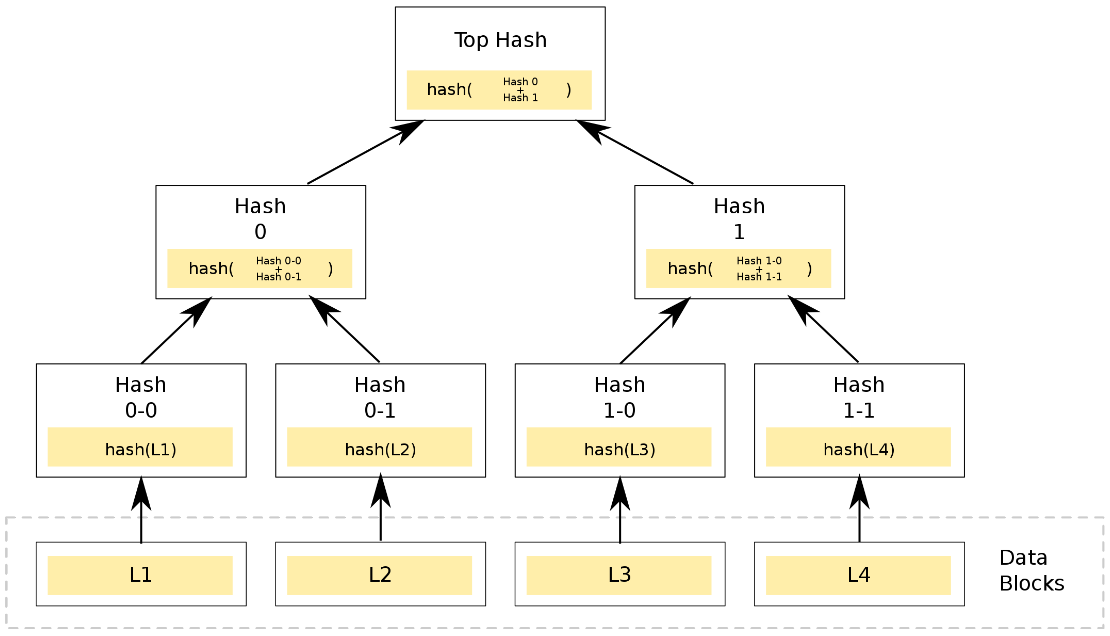


Figure 3.3. *An example of a hash tree*

A practical benefit of Merkle proofs is that they permit integrity checks of tree branches before even the entire Merkle tree is available, let alone the data itself. For example, let's assume that our application is currently downloading the tree depicted in Figure 3.3 and would like to verify the integrity of data block `L2`. This can be verified in logarithmic time if the tree already contains hashes `0-0` and `1` by firstly hashing `L2`, then hashing the concatenation of `0-0` and `L2`'s hash and iteratively repeating this process until it reaches the top hash. If at any point the produced hash differs from the downloaded tree's one then we have identified an invalid data block in the corresponding subtree (though not necessarily due to an erroneous `L2`), otherwise we can be certain that `L2` is indeed valid. In the latter case and in a real-world application (where the hash tree would be vastly deeper) this would mean that `L2`'s download could begin *simultaneously* with the download of the rest of the tree. This is of utmost importance in distributed applications where exchanging data can be costly and time-consuming and so it would be valuable to know to discard invalid data before downloading them.

Furthermore, when dealing with blocks of data that change over time in a distributed system, nodes can preserve a Merkle tree to facilitate data synchronization. Whenever updating, instead of comparing entire datasets to figure out what changed, a node can simply perform a hash comparison of Merkle trees. This way, only subtrees whose root node changed hashes need to be requested and sent over the network, saving on potentially valuable network bandwidth.

Effectively all major blockchains make use of Merkle trees in some way but these trees had been used in a vast variety of applications long before blockchain technology. Indicatively, they are used by the Git version control system as well as a number of NoSQL distributed database systems such as Apache Cassandra and Amazon DynamoDB.

3.1.3 Ethereum's Modified Merkle Patricia Trie

Contrary to the simplified descriptions of tries we have given so far where only singular elements (keys) where stored, Ethereum needs to store not merely keys but key-value pairs. A common use case for that — though not the only one, as we'll examine in the next section — would be storing an account's address (key) and its associated balance in ETH (value). As such, both leaf and intermediary nodes need to include a "value" field to fulfil that requirement.

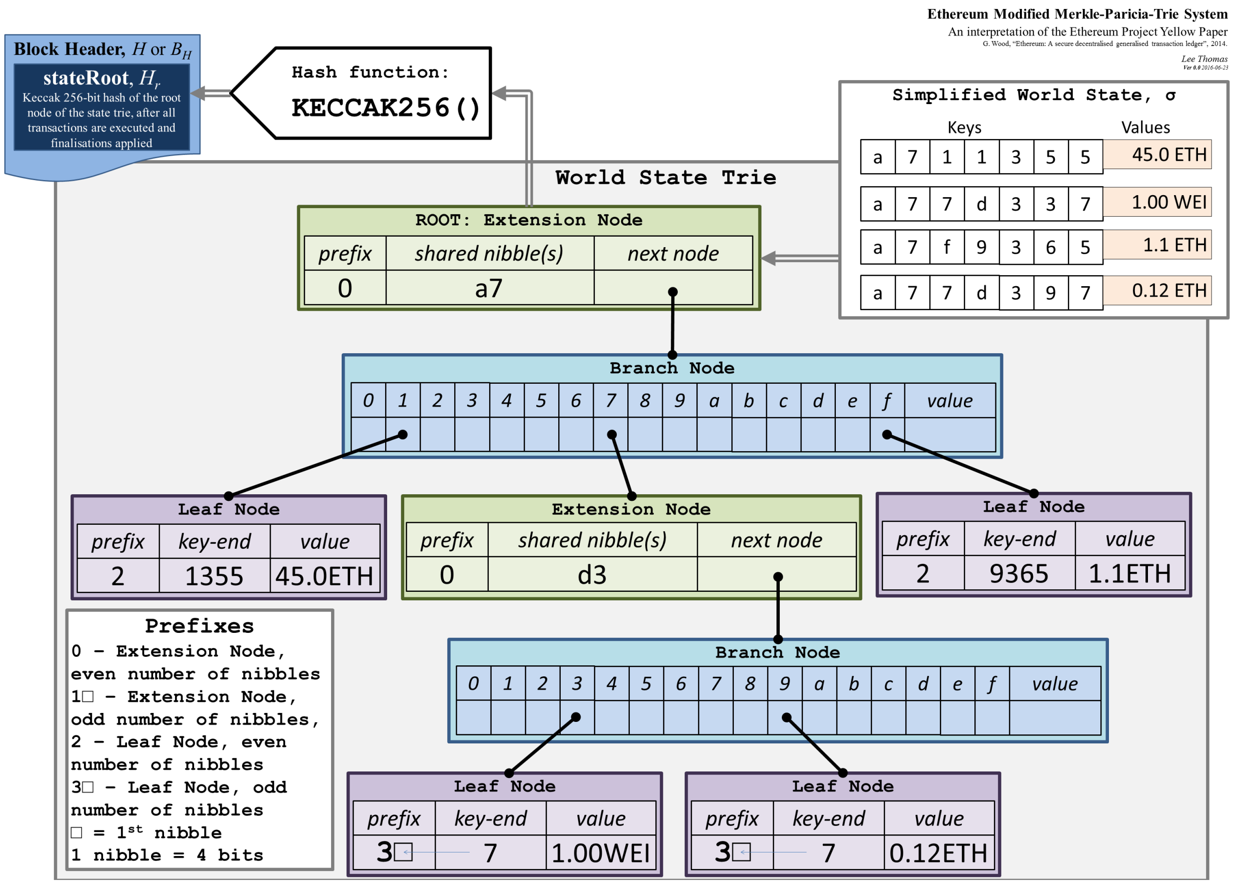
The modified MPT that Ethereum uses is, in fact, not a binary trie as "Patricia" would suggest but instead a hexadecimal one. Furthermore, the modification comes from introducing some additional complexity to the data structure, since here a node can be one of the following:

1. `NULL` (represented as the empty string)
2. `branch`: A 17-item node `[v0 .. v15, vt]`
3. `leaf`: A 2-item node `[encodedPath, value]`
4. `extension`: A 2-item node `[encodedPath, key]`

Branch nodes effectively act as routers, providing a child node at every *nibble* (hexadecimal digit) wherever they are found, in addition to a value (the 17th item) in case a key ends at that node in its traversal. An extension node is interjected after a branch node wherever there is a shared prefix between multiple keys at some nibble and saves space in the way that radix tries in general are useful for. Lastly, if after some branch node's nibble there is only a single key to be stored a leaf node is used, containing the non-shared suffix of that key along with the value corresponding to it.

This data structure, albeit somewhat complicated, is greatly space-efficient as well as scalable, designed to be able to accommodate vast amounts of data. Moreover, it is optimized for updating by requiring a minimal amount of nodes to be created or altered as new elements are added to the trie, with the vast majority of nodes usually remaining intact.

While this optimization serves several functions, it does adversely introduce some ambiguity. When traversing paths in nibbles, we may end up with an odd number of nibbles to traverse, but because all data is stored in bytes, it is not possible to differentiate between, for instance, the nibble `1`, and the nibbles `01` (both must be stored as `01`). This is solvable by prepending a `prefix` nibble in all 2-item nodes (i.e. extension and leaf ones) during their encoding to signify the difference between odd and even partial path lengths.



Ethereum Modified Merkle-Paricia-Trie System
An interpretation of the Ethereum Project Yellow Paper
G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.
Lee Thomas
Ver 0.0 2016-06-23
Block Header, H or B_H
stateRoot, H_r
Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied
Hash function:
KECCAK256()
Simplified World State, σ
Keys
Values
a 7 1 1 3 5 5
45.0 ETH
a 7 7 d 3 3 7
1.00 WEI
a 7 f 9 3 6 5
1.1 ETH
a 7 7 d 3 9 7
0.12 ETH
World State Trie
ROOT: Extension Node
prefix
shared nibble(s)
next node
0
a7
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f
value
Leaf Node
prefix
key-end
value
2
1355
45.0ETH
Extension Node
prefix
shared nibble(s)
next node
0
d3
Leaf Node
prefix
key-end
value
2
9365
1.1ETH
Prefixes
0 – Extension Node, even number of nibbles
1□ – Extension Node, odd number of nibbles,
2 – Leaf Node, even number of nibbles
3□ – Leaf Node, odd number of nibbles
□ = 1st nibble
1 nibble = 4 bits
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f
value
Leaf Node
prefix
key-end
value
3□
7
1.00WEI
Leaf Node
prefix
key-end
value
3□
7
0.12ETH

Figure 3.4. *Simplified example of Ethereum's modified MPT* [1]

All that we have described here in words can be better understood by examining the simplified Ethereum use example in Figure 3.4, where one such modified MPT is constructed to contain 4 key-value pairs with some prefix overlap.

[1]Source: Lee Thomas, https://ethereum.stackexchange.com/a/6413/100602

From all these we can infer a property of MPTs that is of paramount importance to the structure of Ethereum, that of determinism. In other words, two MPTs containing the same key-value pairs are guaranteed to be identical down to the last byte. As such, it becomes easy to communicate MPT roots across different nodes for validation purposes because it is a certainty that if, for example, two nodes have independently locally constructed the same state MPT then they must also have ended up with the same MPT root.

Lastly, it is worth noting that this structure is not set in stone as explicitly stated in the yellowpaper [2]: "The core of the trie, and its sole requirement in terms of the protocol specification is to provide a single value that identifies a given set of key-value pairs, which may be either a 32-byte sequence or the empty byte sequence. It is left as an implementation consideration to store and maintain the structure of the trie in a manner that allows effective and efficient realisation of the protocol". Nonetheless, modified MPTs have been widely adopted as part of the Ethereum standard given that they provide unique advantages in storage of key-value data as well as fast verifiability through Merkle proofs.

3.1.4 Use in Ethereum

The modified MPT is used in four different interrelated places in Ethereum's architecture and we will examine each one separately.

3.1.4.1 World State Trie

There is one global state in Ethereum and it is represented by the singular world state trie which is updated over time. This trie is not contained in any block's data but is rather constructed by every node separately during their initial sync to the network and is then constantly kept up-to-date upon the receival of additional blocks and transactions from other nodes. That way, it can be queried at any time to retrieve any and all data associated with any Ethereum account.

The Ethereum world state is a mapping between addresses and account states. More specifically, the key-value pairs that comprise the world state trie are always `keccak256(ethereumAddress)` keys paired with `rlp(EthereumAccount)` values. Ethereum accounts, as we have examined in the previous chapter, consist of a 4-item array of `[nonce, balance, storageRoot, codeHash]` with `storageRoot` in turn being the root of another MPT.

Accounts in Ethereum are only added to the state trie once a transaction involving them has taken place. Simply creating a new account *A* will not cause it to be appended to the state trie — this will only happen after a successful transaction is recorded with *A* as the recipient. This behaviour constitutes a protective measure against malicious attackers who could otherwise costlessly create new accounts and bloat the state trie.

3.1.4.2 Account Storage Trie

The account storage trie is where the data associated with an account is stored. This is only relevant for contract accounts, as for EOAs the `storageRoot` is empty (and the `codeHash` is the hash of an empty string). Values contained in this trie can be retrieved by querying with the integer position of the stored data and the related block ID.

3.1.4.3 Transactions Trie

Each Ethereum block has its own separate transactions trie wherein all the transactions included in that block are encoded. The choice of which transactions to be included in it used to be decided by the miner who assembled that block, while post-Merge this responsibility belongs to validators as explained in Section 2.3.1.

The key-value pairing here is an `rlp(transactionIndex)` key (the index within the block the transaction is included in) paired with the contents of the transactions which we presented in Section 2.2.5. Contrary to the aforementioned tries, the Transactions trie is immutable and once the block is validated the transactions it contains can never be changed nor included in some other block. As such, a finalized transaction in the network can always be reliably and uniquely located by knowing the block in which it was included and its position in that block.

3.1.4.4 Receipts Trie

Similarly to the transactions trie, every block contains the root of a receipts trie using the same type of key (`rlp(transactionIndex)`). The difference here however is that, instead of the transactions themselves, the receipts trie records the outcomes (i.e. receipts) of these transactions and these are not included in each block but are rather recreated by each client locally. In order to validate a transaction's T receipt one simply needs to re-execute T on the state that the network had when T was originally executed. The fields that can be found in such a receipt include the resulting MPT root, the gas that ended up being used, a set of logs created as a result of the transaction's execution as well as a Bloom filter composed of information in those logs (more on Bloom filters' use in Section 3.1.8). Each log record can include up to 4 indexed topics in it for the type of event, the sender and receiver addresses, as well as for any other related (but short-form) data.

It is worth noting that the transactions and receipts trie, due to the nature of the keys they contain as well as their lack of updating over time, do not particularly take advantage of the Patricia optimizations we went over earlier. Still, these tries are too coded as MPTs for uniformity purposes, as Ethereum has deemed it less maintainable to implement and optimize a separate Merkle tree data structure in every client just for them.

3.1.5 Advantages of MPT use

Having explored how MPTs are used by Ethereum, we should also expand a bit on what are the benefits they provide that have rendered them so integral to Ethereum's architecture.

One such benefit we already went over in Section 3.1.2 is the ability for partial verifications. Given that everything in Ethereum is ultimately representable as simple key-value

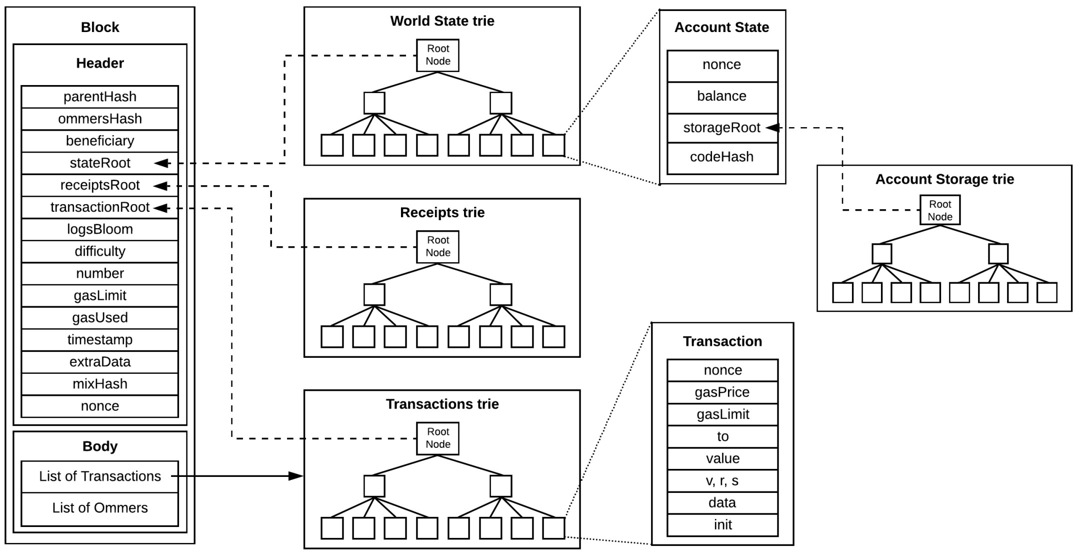


Figure 3.5. *MPTs in Ethereum* [2]

pairs, it might seem that both read and write times could be improved by a flat structure instead of a tree one. However, a vital aspect of blockchain applications is their ability to verify each and every piece of data they receive from the network before accepting it as authentic. Taking the state MPT as an example, were we to instead try to preserve its (hundreds of millions of) contained accounts in a flat key-value storage then any modification of any kind at any account would necessarily always result in the need to rehash the entire dataset. These problems are solved by the MPT data structure as trie branches are self-contained (in that their hashes are unaffected by changes in different trie branches) and much more quickly verifiable by any peer even before the have downloaded the rest of the MPT. Lastly, modifications in the trie leaves only require a logarithmic number of hashes to be recalculated, a major improvement in efficiency when dealing with the colossal number of data in the Ethereum blockchain.

Furthermore, the very existence of light nodes — lightweight Ethereum nodes that do not download the entirety of the blockchain but merely their headers, which we will analyze in Section 4.1.3 — is only possible thanks to the clever use of MPTs. A sufficiently advanced light node protocol can easily get verifiable answers to a variety of queries. For instance "has transaction `T` been included in block `B`?" can be answered by checking `B`'s transactions trie while questions like "does account `A` exist?" or "what is `A`'s current balance?" are handled by the state trie. More complex queries (e.g. "what would be the output of running transaction `T` on contract `C`?") require computations that are out of the scope of our analysis, but it is sufficient to point out that many of them can likewise be handled by the state trie [21].

[2]Source: Lucas Saldahna, `https://www.lucassaldanha.com/ethereum-yellow-paper-walkthrough-2/`

3.1.6 Bonsai Tries

As already mentioned, MPTs are the standard but not the sole way of storing Ethereum data. *Bonsai* tries are an additional data storage format to the traditional MPTs and, at the time of writing, are implemented as an optional alternative to them only in the Besu execution client (more on clients in the following chapters).

Their goal is to improve access speeds to the current state through piles of leafs and trie logs to quickly access them, where in traditional MPT one must traverse all the branches by hash in order to read a leaf value. Instead of keeping those large MPTs within storage, Bonsai keeps only the most recent trie in its storage as well as a trie log layer. This log layer provides a small store of changes (a diff between the states of a parent block and a new one) that, when needed, can be used to construct the complete history of the tries. This "flatter" approach effectively reduces storage and offers much faster times for nodes to read any data about Ethereum's current state, such as $O(1)$ account lookups.

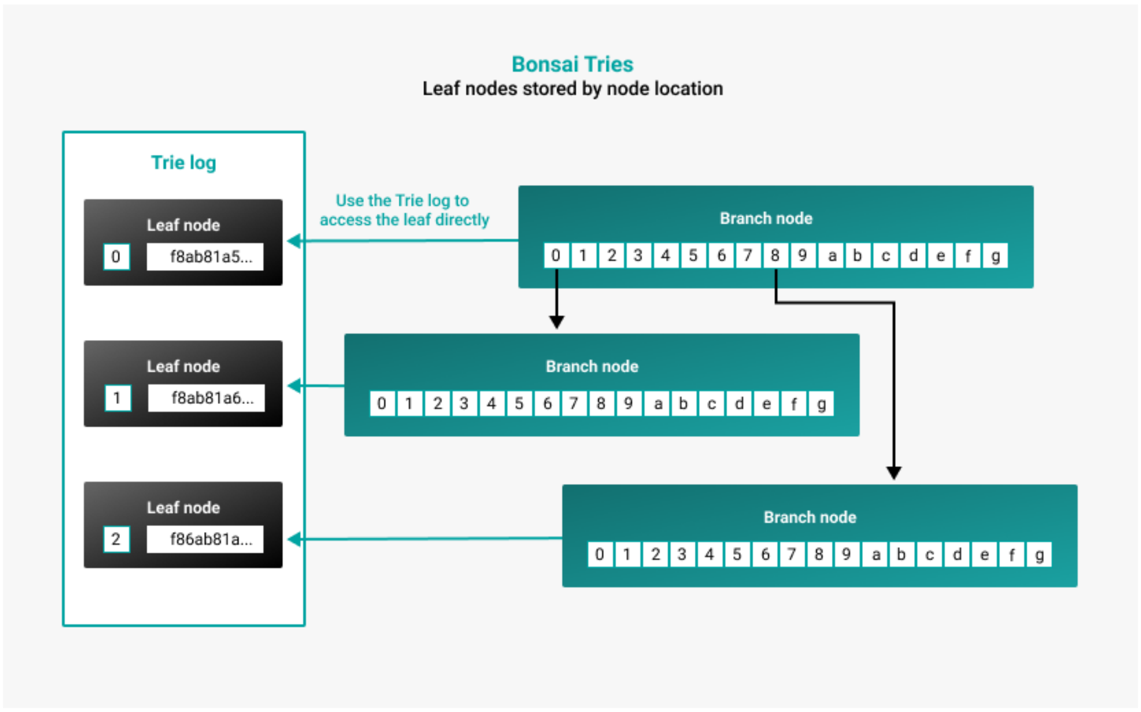

Bonsai Tries
Leaf nodes stored by node location
Trie log
Leaf node
0
f8ab81a5...
Leaf node
1
f8ab81a6...
Leaf node
2
f86ab81a...
Use the Trie log to access the leaf directly
Branch node
0 1 2 3 4 5 6 7 8 9 a b c d e f g
Branch node
0 1 2 3 4 5 6 7 8 9 a b c d e f g
Branch node
0 1 2 3 4 5 6 7 8 9 a b c d e f g

Figure 3.6. *Bonsai trie visualization* [3]

The main drawback of Bonsai tries when used to store Ethereum state is that, while they do make accessing recent blockchain data much faster, it becomes increasingly more resource-intensive the further in history we try to read data. If this is a common use case then using MPTs instead remains the optimal choice.

3.1.7 Verkle Trees

In 2018, John Kuszmaul proposed a new alternative data structure to Merkle trees, called Verkle trees [22]. Verkle trees are constructed with a significantly larger branching factor k than Merkle trees do (where typically $k = 2$) and leverage that fact by using a

[3]Source: https://besu.hyperledger.org/en/stable/public-networks/concepts/data-storage-formats/

special type of hashing function, *vector commitment* [23], instead of conventional cryptographic hash functions. Since proof size directly depends on the tree's depth, these shallower Verkle trees (with a proposed $k = 256$) naturally produce significantly smaller proofs.

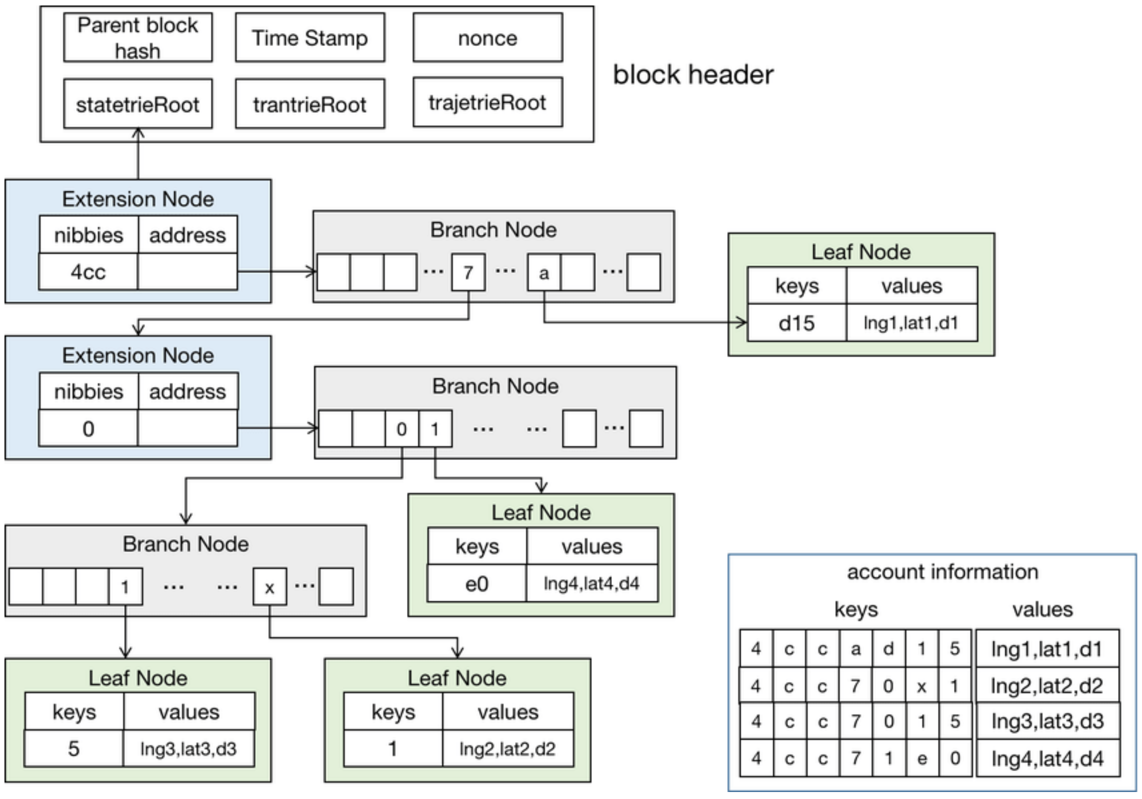


Figure 3.7. *An example of a Verkle tree* [4]

We will not be going over their technical details here since Verkle trees are currently still in a proposal phase with no indication of urgency by Ethereum developers to implement them in the short-term. The key takeaway is that while a Verkle tree is more expensive to initially construct, its proof size complexity is only $O(\log_k n)$ as opposed to a Merkle tree's $O(k\log_k n)$, while preserving the same $O(k\log_k n)$ time complexity for updating a file (where k is the tree's branching factor and n the number of files to be stored in the tree).

While the developers' intention is to sometime transition Ethereum to exclusively relying on Verkle trees to store execution state, this is quite a major upgrade and is not meant to begin until a number of other unrelated Ethereum upgrades have been successfully deployed. Nevertheless, a draft EIP has been written which would see the introduction of Verkle state trees alongside the existing Patricia ones [25]. According to it, the use of Verkle trees allows proof sizes to decrease by a factor of ~6-8 compared to ideal Merkle trees, and by a factor of over 20-30 compared to the hexary Patricia trees that Ethereum uses today. Although fully integrating them to Ethereum is not trivial — a hard fork would likely be required — and they require more complex cryptography to implement, Verkle trees show serious potential and furthermore present large measurable

[4]Source: [24]

gains to scalability.

3.1.8 Bloom filters

While tries are rightfully in the spotlight when discussing Ethereum's data structures, for the sake of completeness it must be noted that they are not the only data structure used in Ethereum clients. Another structure known as *Bloom filter* is used to enable efficient querying of information related to accounts involved in transactions.

A *Bloom filter* is a space-efficient probabilistic data structure, named after Burton Howard Bloom who conceived it in 1970 [26] and is used to check for element membership in a set. False positive matches may occur (and increase in likelihood as elements are added to the set) but false negatives are impossible. It is implemented using a simple bit array of length m (with all elements initialized to 0) and some k independent hash functions with each mapping to some of the array's elements so as to ideally jointly approximate a uniform random distribution. Insertion and search time complexities are $O(k)$ while space complexity is naturally $O(m)$ — the bit array itself. It is noted that a bloom filter can be made using $k = 1$, though in that case a sufficiently large m is required to preserve a low false positive rate.

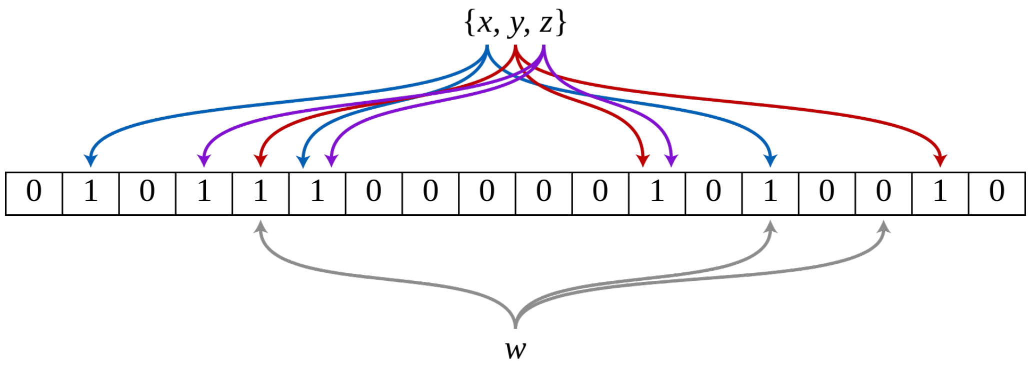


Figure 3.8. *A Bloom filter example*

Insertion of new elements to the filter can easily be done by hashing each new element and setting to 1 all the bits corresponding to its hashes, while the removal of elements is not supported. A trivial example of how element inclusion checks are done in Bloom filters can be understood through the Bloom filter depicted in Figure 3.8. This filter includes the elements $\{x, y, z\}$ which were inserted by hashing each one separately and updating the array's bits that belonged to its hashes. When querying for a new element w we first hash it too and then check whether *all* the hashes' corresponding bits in the filter have already been set to 1. In this case there is at least one bit for which this condition is not satisfied so we can be certain that w has not been included in this filter. Had that not been the case, we would not have proved the existence of w in the filter but rather merely have an indication that it *may* have been included in it, which would then prompt us to perform an actual search for it in our underlying data.

In Ethereum, since we want the various events that can occur as a result of transactions to be easily searchable, it would be a significantly time-consuming process for an application to be examining all the transactions in each and every block regardless of what it is searching for. To prevent this, every block contains a `logsBloom` filter which consists of a simple 2048-bit string constructed from hashing (using `keccak256`) parts of the log records found in the receipts trie.

A common use case of them would be trying to retrieve some historical information regarding a particular account *A*, such that would require identifying whether *A* is involved in the transactions included in some block *B*. Instead of tediously going over all the transactions in *B*, we can instead take the `keccak256(A)` hash and compare it against *B*'s `logsBloom`. Through this process, a potential application will need to check far fewer blocks for such a query than it would otherwise have to. While the "positive" blocks will each still require further individual checking as there is a risk of false positives, there is a certainty that *A* was not involved in any transaction included in any other block.

Furthermore, by leveraging the up to 4 indexed topics included in each log record and integrating them in our Bloom filter (each of these will set some different bits to 1) we can further narrow our search to some specific fields, e.g. find only the blocks that include a transaction where *A* was the recipient. Evidently, this use of Bloom filters can drastically increase performance of both applications like block explorers but also of nodes themselves, which are able to quickly scan over the headers of blocks and quickly determine which ones are relevant depending on their workload at any given time.

3.2 Helper Functions

When we earlier described the use of MPTs we mentioned certain functions like `keccak256()` and `rlp()` without providing any explanation for them. Let's explore them a little bit here before we shift our focus to other aspects of Ethereum.

3.2.1 Hash Functions

A fundamental architectural characteristic of MPTs and Merkle trees at large is the hash function chosen. A hash function can be any function that can map data of arbitrary size to fixed-size values[5] (called *hashes*), but a valuable one should ideally meet some additional criteria. A *cryptographic* hash function (*CHF*), which is what interests us here, ought to make it impossible to deduce an input value from its hash, while also additionally ensuring that there are no collisions, i.e. `chf(input1)` ≠ `chf(input2)` ∀ `input1` ≠ `input2`. Lastly, any small change to an input value should drastically change the output of a CHF.

A number of CHFs have found various applications over the years, including *SHA-256* (Secure Hashing Algorithm 256) in Bitcoin. For its purposes, wherever a hash is needed Ethereum uses *Keccak-256* (often referred to as simply "Keccak", though the Keccak family includes hash functions other than Keccak-256) which is the original implementation of SHA-3, before SHA-3 was officially standardized following the 2012 Cryptographic

[5]Though unrelated for us here, it is noted that there also exist hash functions that can generate hash values of arbitrary lengths such as *RC6* [27]

Hash Algorithm Competition by the American National Institute of Standards and Technology (NIST) [28]. While both these CHFs have proven security, the newer Keccak-256 is considered to be even more secure than SHA-256 albeit comparatively slower to run in non-specialized hardware due to some additional precautionary measures taken.

Keccak-256 allows for arbitrary inputs and an infinite input space, thus being able to provide a singular 256-bit hash for arbitrarily large files or byte-arrays (as is the case with our trie nodes). Its implementation details are beyond the scope of this thesis, but it should be noted that so far Keccak-256 as used by Ethereum has exhibited perfect collision resistance (though collisions have been observed in round-reduced versions of Keccak [29]).

3.2.2 Encoding Functions

Since everything is stored in key-value pairs, in the effort to decrease the size of chain data on disk it would be desirable to encode the value part of those pairs in some space-efficient way. Thus, a necessity arises to encode complex structures into easily storable values, while of course also being able to recover our original structures from the chosen encoding whenever needed.

3.2.2.1 Recursive Length Prefix (RLP)

Recursive Length Prefix (*RLP*) is an encoding function that was created specifically to be used in Ethereum where object encoding was needed, including transaction fields, nodes in MPTs and blocks in their entirety. We have already mentioned some of its uses, such as in the world state MPT where a `keccak256(ethereumAddress)` key is used to store an `rlp(EthereumAccount)` value but also in the transactions and receipts MPTs where even the keys themselves are encoded as `rlp(transactionIndex)`.

RLP can be used to encode nested sequences of bytes into flat sequences of bytes that can be decoded back into the original nested sequences. RLP is self-describing, meaning that decoding any output of it can be performed without any other prerequisite knowledge. This is achieved by having the first byte of the serialised content indicate the type of data that follows. For example, a single byte the value of which is in the $[0x00, 0x7f]$ range is itself its own RLP encoding, while a string's encoding will begin with a byte in the range $[0x80,\ 0xb7]$ based on its length, followed by the length itself and the string's bytes in a sequence. A list is encoded similarly (with the first byte in the range $[0xc0,\ 0xff]$) as are its contents which accounts for nested elements. This approach makes RLP decoding fairly straightforward, though a limitation of it is that it cannot support payloads (neither strings nor arrays) with length higher than 2^{64}. A detailed strict definition of RLP can be found in the yellowpaper [2].

As we established, RLP is used abundantly in Ethereum on account of the several nested objects throughout its architecture that necessitate some encoding in order to be stored efficiently. RLP was originally the sole encoding function of Ethereum and was used in all cases where it was applicable, but that changed after The Merge additionally introduced *SSZ*.

3.2.2.2 Simple Serialize (SSZ)

Simple Serialize (SSZ) was first proposed by Vitalik Buterin in 2017 and soon after became the canonical serialization format of Ethereum's consensus layer, where it has completely replaced RLP except in the peer discovery protocol. Its goal was to be simpler than RLP, efficient to traverse, and also support Merkle-proof schemes by design. SSZ is also bijective, meaning that serializing the same object of the same type will always yield the same result, while the serialized representations of two different objects are guaranteed to be different, an essential characteristic for its use in Ethereum's consensus.

Unlike RLP, the newer SSZ is not self-describing but rather relies on a schema that must be known in advance before decoding it where the precise layout of the serialized data (type, value, size, and position) is defined. Its goal is again to represent objects of arbitrary complexity as strings of bytes which merely consists of a straightforward conversion for basic types but becomes more complex in the case of composite types of potentially variable lengths. We will not go over its specification[6] here, but the gist of it revolves around constructing a byte sequence with offset values in the place of the actual data while adding the latter to a heap at the end of the serialized object.

SSZ is less space-efficient than RLP but enables indexing, i.e. accessing inner values of the data structure without fully deserializing it. Requiring a schema could itself be considered a downside of SSZ but it also means that it is extensible so that new data types could potentially be added to the serialization format without breaking backwards compatibility. Moreover, its ability to "merkleize" (i.e. transform into a Merkle tree representation of the same data) efficiently is regarded as a major architectural advantage. For these reasons, the Ethereum development team has expressed its intention to eventually retire RLP entirely in favour of SSZ.

3.3 Storage Engines

It will be useful for our later analysis to not only have a high-level overview of the data structures chosen in Ethereum but also of the way they are actually stored on the disk at the fundamental level. This is generally done through the use of some LSM (Log-Structured Merge) tree based storage engine (a key-value store without schema constraints) organized on disk in multiple layers. This kind of stores differ from SQL databases in that they do not have a relational data model, nor do they support indexes. They are recommended for write-intensive scenarios but less so for read-intensive ones, making them ideal for the average expected workload of an Ethereum node.

In order to minimize the amount of random writes on disk, when new records are ready to be written to a LSM-based storage they are first batched in memory (in a data structure called *MemTable*) and are only inserted to a level on the disk once the size of that batch reaches a certain threshold. Furthermore, they periodically perform a *compaction* which consists of selecting multiple files and merging them together — a process that can

[6]As is often the case in cutting edge blockchain developments, this specification is not yet part of some published research paper but is rather maintained in the official GitHub repository: https://github.com/ethereum/consensus-specs/blob/dev/ssz/simple-serialize.md

be done efficiently since files in disk levels are kept sorted (by key). This is a crucial part of their design, since through compaction a handle can be kept on their read performance which degrades as the number of files increases. Compactions are by default performed in some background thread in most implementations but may also be invoked manually in the foreground if that is deemed desirable.

Given this architecture and that LSMs are optimized for multiple writes, locating some particular key would likely require accessing several files because the requested one could be found in any of the levels. To minimize redundant reads, a bloom filter is used across levels as a memory-efficient way of working out whether a file contains some key.

Of course, there are some limitations in the use of LSMs to store Ethereum data. One such is that in Ethereum everything is identified via hashes which are, by definition, uniformly randomly distributed. For an LSM that keeps everything sorted by identifier (key), this fact evidently makes accessing values associated with hash keys very expensive. However, without a specific database schema for Ethereum MPTs by design, this is a challenge without a direct optimization solution.

While a comparative analysis of storage engines is beyond the scope of our study, we can briefly mention the main open-source ones that are currently used by the various Ethereum clients along with some noteworthy information for each:

- **LevelDB**: The most popular of its kind and developed by Google, LevelDB is the storage engine used by a variety of non-blockchain and blockchain applications alike, including Bitcoin. It does not provide a server nor a CLI (command-line interface) but rather applications are expected to use it simply as a library.

  LevelDB employs a 7-level architecture in addition to up to 2 in-memory tables, with each level containing multiple files called *SSTables* (Sorted String Tables). As their name implies, an SSTable is a simple abstraction to efficiently store large numbers of key-value pairs while optimizing for high throughput.

  Each level's capacity is approximately an order of magnitude greater than that of the previous level, which also comes with a comparatively slower access time. When a new key-value pair is to be added, as described above, it is first inserted to a MemTable and when that is filled an SSTable file is created which is then merged with the SSTables in Level 0 in disk. As the number and size of SSTables in Level 0 increases they are compacted into Level 1 and so on for the rest of the levels. This write flow is depicted on a high-level in Figure 3.9.

- **RocksDB**: Developed by Facebook, RocksDB began as a fork of LevelDB and still shares many similarities to it. It likewise uses the same leveled approach with a MemTable, storing disk data in SSTable files. RocksDB is optimized for high-concurrency and multi-threaded workloads where it can offer greater read and write performance than LevelDB, though by using comparatively more disk space [30].

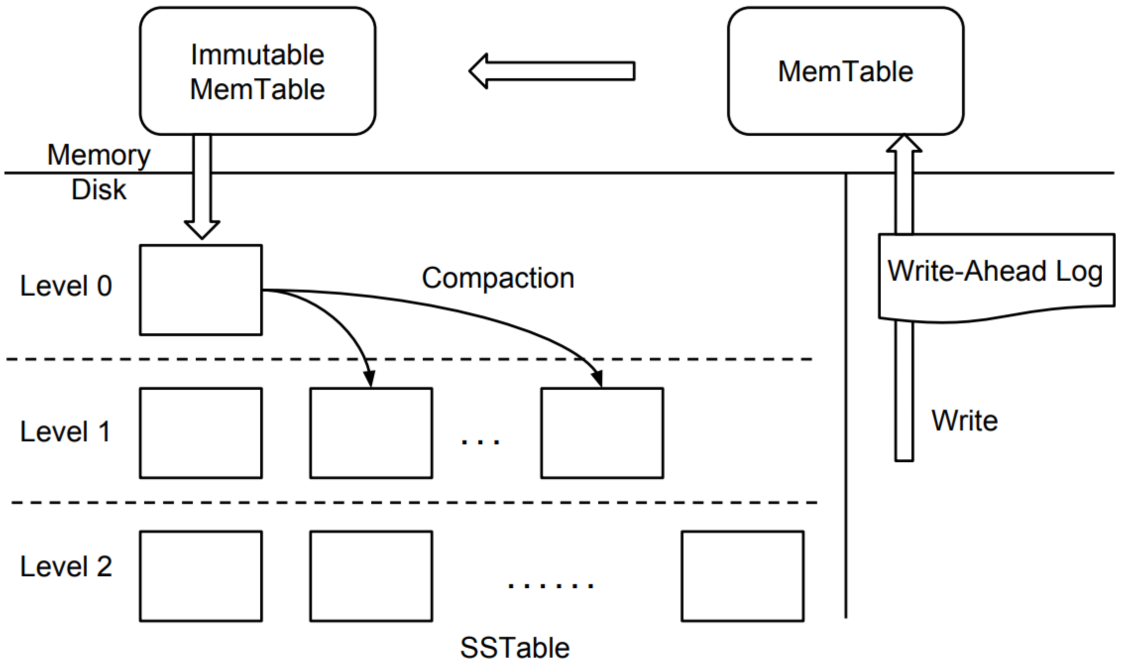


Figure 3.9. *A high-level overview of LevelDB write flow*

- **MDBX**: MDBX is a community-maintained data store that, contrary to the previous two, is based on B+ trees instead of LSMs. We need not expand more on B+ trees here, especially since MDBX is solely used by Erigon (see Section 5.4.4) out of all the clients in our study, but it should be noted that they offer a more balanced performance between read and writes, making them preferable to LSMs when more reads (especially random ones) are expected. Lastly, MDBX is additionally ACID-compliant[7], making it more resilient than its alternatives.

[7]ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee data validity despite application crashes or other failures

Chapter 4

Nodes and Clients

The terms “node” and “client” are often used interchangeably but they are not equivalent. A *node* represents the computer that connects to the network and performs certain actions such as receiving, storing and sending data, which is done by running *client* software. Therefore, while we often generically refer to network participants as "nodes", we ought to turn to "clients" whenever we are interested in any and all implementation details.

The Ethereum network is made up of nodes, which are the only method to access it. Nodes communicate with one another in order to validate transactions and record data about the status of the blockchain while each keeps its own copy of the blockchain and strives to guarantee that it matches the copies kept by the majority of other nodes. This network of continually communicating nodes allows users to avoid relying on a single source of truth and all of the challenges that entails.

Participating in the Ethereum network or, as it is commonly referred to, "running a node", involves running client software (which we will imminently examine) on some computer while maximizing its uptime. An Ethereum node can be run on an average consumer-grade home computer, but most users opt to run their node on dedicated hardware to eliminate the performance impact on their machine and minimize node downtime.

Many nodes are operated by large organizations and crypto-related companies (e.g. cryptocurrency exchanges), but truly anyone can run a node. While network validators, as described in Section 2.3, are naturally required to run a node, one does not need to be a validator in order to do so — in fact, no ETH whatsoever is needed for that. While in that case the node operator will not have any financial rewards, there are other benefits to running an Ethereum node including privacy, security and improving the decentralization of the network by reducing the user’s need for third-party services. The latter is actually the cornerstone of decentralized blockchain systems commonly summed up in the phrase "don’t trust, verify", i.e. using one’s own node instead of relying to any third parties for information about the state of the network.

4.1 Node Types

Fundamentally, there are 3 distinct types of Ethereum nodes that differ in the way they consume block data and the utility they provide. These are *light*, *full* and *archive*, with the various client implementations choosing to support any number of them.

4.1.1 Full Node

Full nodes store the current and the most recent blockchain state (by default, up to the last 128 blocks in most implementations) and participate in verifying and validating each and every transaction that takes place on the Ethereum network. When a smart contract transaction occurs, full nodes execute all of the instructions in it and determine whether or not the smart contract execution is producing the expected results. Full nodes also participate in block validation, verifying all blocks and state and are thus the conventional node type that validators use.

A full node will usually take a lengthy time to initially sync, the reasons for which will be our focus in Section 5.1. As that syncing progresses, the node does not necessarily retain historical blockchain states (also known as "pruning", which we will likewise inspect in the next chapter) but these can nonetheless be recreated on demand since they are derived from the blocks' data. Such a request would certainly be resource-intensive and would take a longer time to be fulfilled than one depending only on the current blockchain state but it is important to point out that no information is lost by pruning as it can always be retrieved if necessary.

4.1.1.1 Bootstrap Node

A *bootstrap* node (or *bootnode*) is not a separate type of node, but rather a — usually stripped down — version of a full node which provides initial configuration information to new nodes joining the network. In most P2P networks, since nodes can join and leave at any time, new users may struggle to find peers. Bootnodes are useful in combating this issue by being highly available and providing a newly joining node with vital resources and information regarding the network itself and mainly guiding them to discover other peers.

The connection information (addresses) of several Ethereum bootnodes is hard-coded in the source files of most client implementations so that they can begin syncing without requiring user configuration, at least in regards to the network node discovery protocol. Despite that, additional bootnodes can generally be manually provided as an argument when starting a node or even during its runtime.

For disambiguation purposes, it is noted that node "bootstrapping" in the context of this thesis refers to the process of the initial synchronization of a node and is not related to bootstrap nodes — though this process naturally does internally initially involve requesting information from them.

4.1.2 Archive Node

An archive node does everything that a full node does while also preserving an archive of all historical states. Where a full node will prune those interim states to save on resources and decrease sync time, an archive node will disregard such constraints and store as much information as possible for quick querying capabilities.

An archive node will verify all downloaded blocks, re-execute all transactions, and write all intermediate states to the disk. The trade-off for this costlier initial sync and significantly greater storage requirements (over 13TB using Geth at the time of writing) is a much faster response time to questions regarding the blockchain's history, e.g. "What was X's account balance on this day 2 years ago?" or "At which block did X's account exceed Y ETH?".

At times, even full nodes that have the ability to rebuild past states find it quicker to simply query archive nodes when posed with historical questions. Rebuilding historical blockchain states is a costly operation for those and it is generally expected that some archive node will be quick to fulfil such requests. For these reasons, archive nodes are of limited use to an average user but have proven effective in certain applications such as block explorers and chain analytics.

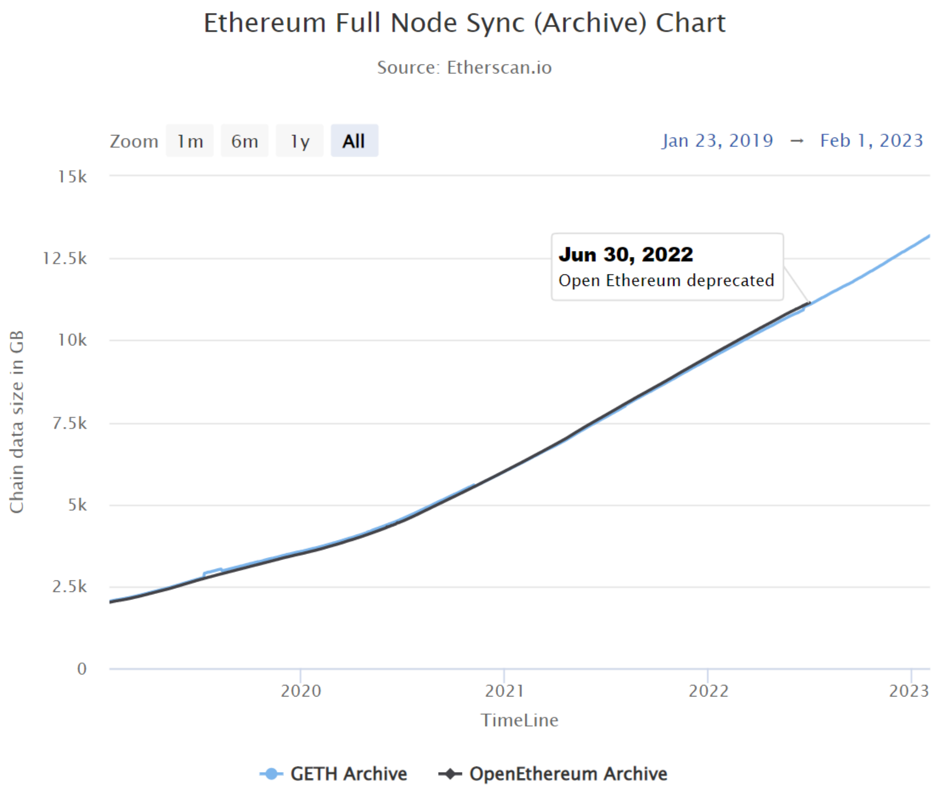


Figure 4.1. *Ethereum mainnet archive node disk size over time* [1]

4.1.3 Light Node

Instead of downloading every block in its entirety, light nodes achieve their lightweightness by merely downloading and storing the blocks' headers. They do not even hold the complete current blockchain state since those headers only contain summary information about the blocks' contents but are able to request such data on-demand from full nodes. Light nodes use the minimum amount of data possible to interact with the Ethereum

[1] on February 1 2023; Source: https://etherscan.io/chartsync/chainarchive

blockchain but are still able to respond to more intricate requests by in turn requesting additional information from full nodes.

Naturally, light nodes are suitable for low memory and computational devices and maintaining a light node involves the least investment in both hardware and technical skills. Eventually, it is envisioned that light nodes could even run on mobile phones or embedded devices. Unsurprisingly however, these conveniences are accompanied by their own set of drawbacks.

Light nodes' heavy reliance to full nodes necessitates the existence of enough full nodes willing to altruistically serve data to them (which is generally done by setting a related flag and dedicating a maximum amount of system resources to handling light nodes' requests). In practice few full nodes opt to do that, leading to a struggle to find peers and, consequently, noticeable delays in data retrieval. But even in ideal condition with enough peers willing to serve light node data, it is undeniable that even the smallest of network delays would add up to a significant degree if data retrieval was a frequent requirement. For that reason, if a node operator anticipates such a use case from their node, the optimal choice would be to forego light nodes and instead operate a full node to be able to access this kind of information directly.

Lastly, light nodes cannot participate in consensus (block validation) and are thus of no use to validators who wish to stake ETH and receive financial rewards for that.

4.2 Client Types

In PoW Ethereum there was only one type of client, formerly simply referred to as "Ethereum Clients" and now as *Execution (Layer) Clients*. After The Merge however, additional client software was needed to support the consensus upgrade, called *Consensus (Layer) Clients*.

An execution client runs the code pertaining to the *execution layer*, which is where the transactions actually get executed and enact changes on the state of the blockchain. It is thus tasked with maintaining an up-to-date copy of the latest state by validating and handling all the transactions that are communicated through the network. On the other hand, a consensus client runs the code for the *consensus layer*, i.e. is responsible for all the logic that enables the node to stay in sync with the Ethereum network. It achieves that by keeping up with the canonical order of blocks and transactions, as defined by the PoS criteria described in Section 2.3.1. Each client has its own distinct networking stack — sometimes referred to as P2P or *network layer* — through which communication with peers of the same type can be established and which is used to gossip transactions (in the case of execution clients) or blocks (in the case of consensus clients).

These types of clients were originally called "Eth1" and "Eth2" clients respectively. This naming scheme was deprecated as it gave the false impression that execution clients would be discontinued in PoS Ethereum. In fact, consensus clients now operate alongside and complement the execution ones and after The Merge every node is required to run them both together in order to gain access to the Ethereum network. An authenticated connection is required between the consensus and the execution clients which is established through a JWT (JSON Web Token) file. This signed token acts as a shared secret used to check that information is being sent to and received from the correct peer.

There are several implementations of both types of clients in various programming languages maintained by different teams of developers. One could justifiably wonder why is that the case, when surely securing and updating a single implementation of client software should be easier than doing so for multiple ones. A simple reason is that not all implementations are focusing on the same uses, with some opting for the highest efficiency and speed possible, while others aiming to minimize resource consumption which allows them to run even on low specification computers. However, the main reason for promoting what is known as "client diversity" is indeed network security. Bugs or security holes are bound to eventually spring up somewhere in any decently complicated software, no matter the amount of auditing or the number of programmers involved in preventing them. When that happens, it is necessary that there exist enough nodes running healthy client implementations on the network so as for the faulty clients to not pose any threat to its stability. While this broadly applies to both types of clients it is of paramount importance to consensus clients in particular since a bug in these could result in double spending and invalid blocks being perceived as valid. An accidental fork of that nature is not implausible and did, in fact, happen in PoW Ethereum in August 2021, when a bug in an older version of Geth clients caused several mining pools to split from the main chain[2]. This thankfully only had minimal impact because the majority of miners happened to have had already updated their clients and the longest chain was indeed the bug-free one. Evidently, it is crucial for the health of the network that no single client implementation possesses a dominant share of the network, hence eliminating a potential single point of failure.

A thorough analysis of the various execution client implementations and their architecture is going to be the basis of the following chapters of this thesis. But before moving on, we should first briefly go over the most popular consensus clients and their bootstrapping which will also be relevant in our later benchmarks.

4.3 Consensus clients

4.3.1 Implementations

All client implementations mentioned here — as well as the 4 major execution client ones in the following chapter — are free, open-source and cross-platform (available on Windows, Linux and MacOS).

- **Lighthouse**: Written in Rust and maintained by Sigma Prime, it aims to be secure, performant and interoperable in a wide range of environments, from desktop PCs to sophisticated automated deployments. Lighthouse has been production-ready since Beacon Chain genesis and is currently one of the most widely used consensus clients along with Prysm.

[2]Source: https://www.theblock.co/post/115822/bug-impacting-over-50-of-ethereum-clients-leads-to-fork

- **Prysm**: Written in Golang by Prysmatic Labs, it prioritizes user experience, documentation, and configurability for both solo stakers and institutional users.
- **Teku**: Written in Java by ConsenSys, it is dedicated to building enterprise-ready clients and tools for interacting with the core Ethereum platform but is comparatively more resource-intensive that the alternatives.
- **Nimbus**: Written in Nim and maintained by the Status.im team, it strives to be as lightweight as possible, allowing it to perform well even on embedded systems or resource-restricted devices.

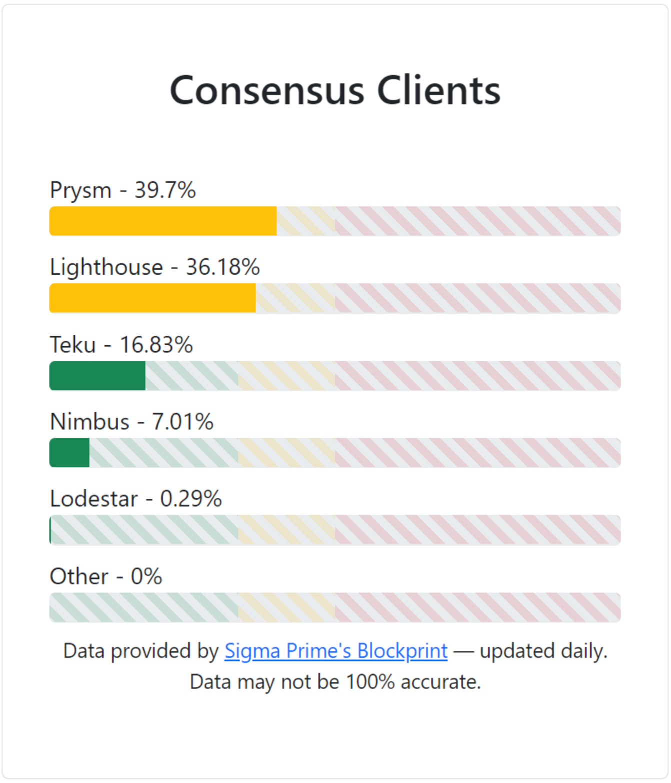


Figure 4.2. *Consensus client distribution on Ethereum mainnet* [3]

4.3.2 Checkpoint Sync

Before an Ethereum node can perform any meaningful work, both its consensus and its execution clients must initially get in sync with the network. The execution client in particular requires to know the head of the chain as a target to begin syncing towards and, post-Merge, it is the responsibility of the consensus client to provide that correct chain head. Contrary to execution clients where the initial sync is a seriously time-consuming process — and which will be our focus for most of the rest of this thesis — it is a much more effortless matter in consensus clients. While they too need to initially sync, there exists a quick and easy way of achieving that by employing a sync mode called *checkpoint sync* (also known as *weak subjectivity sync*).

[3]on February 1 2023; Source: `https://clientdiversity.org/`

Of course, there exist alternatives to bootstrap a consensus client, most notably *optimistic sync* which involves "optimistically" assuming that downloaded blocks are valid so that the execution client can commence syncing with some presumably valid chain head while the consensus client is still in the process of verifying it. But by contrast, checkpoint sync is completed in a matter of minutes, upon which the consensus client is considered synced and can immediately be used to guide the initial sync of an execution client, without any potential risk of later invalidating the presumed chain head.

Checkpoint sync is performed by bootstrapping a consensus client with a parameter containing the URL of another, trusted and already synced node which will provide a (weak subjectivity) checkpoint block to act as the ground truth of the client's chain. Once that is completed, a further step ought to be taken by manually verifying that the legitimacy of the received checkpoint by validating their chain head against another known source.

Following its checkpoint sync, a consensus client may also commence a *backfill sync* so as to download the blocks from the checkpoint block back to genesis. All the consensus clients listed previously have implemented checkpoint sync — though not necessarily backfilling too, which is not a requirement to participate in the network or run a validator.

It must be emphasized that from a checkpoint-synced client's point of view, the checkpoint block has all the characteristics of a genesis block (except that it is found at a non-genesis position in the chain). Even *finalized* blocks do not receive such a treatment, since when a node encounters two conflicting finalized blocks then it is considered to be experiencing a consensus failure and thus being unable to identify a canonical fork. Conversely, if a checkpoint-synced client sees a block conflicting with a weak subjectivity checkpoint, then it immediately rejects that block [31].

Given this description, it is clear that checkpoint sync involves "asking" and then completely *trusting* an off-chain source about information regarding the network's state, an approach which objectively seems inherently insecure for a decentralized trustless network. But somewhat counter-intuitively, checkpoint sync is actually considered *more* secure than simply syncing from the genesis block. In order to comprehend the reasoning behind that we ought to examine a PoS concept that we had omitted so far, that of "weak subjectivity" and the perils it introduces in the form of "long-range attacks".

4.3.3 Weak Subjectivity

Subjectivity in blockchains refers to reliance upon social information to agree on the current state — i.e. there may be multiple valid forks that are chosen from according to information gathered from other peers on the network. The converse of that would be *objectivity*, a case where there is only one possible valid chain that all nodes will necessarily agree upon by applying their coded rules. But there can also be a third state, known as *weak subjectivity*, which refers to a chain that can progress objectively after some initial seed of information is retrieved socially.

Weak subjectivity naturally ensues from the fact that a node that comes online for the first time (or even just after a long offline period) will necessarily have to ask a trusted source what the head of the valid chain is. This seemingly completely undermines the trustless nature of blockchains and is an issue intrinsic to all PoS blockchains, where selecting the correct chain from multiple forks is done by counting historical votes.

Weak subjectivity also enables a PoS-exclusive attack vector, known as "long-range attack". When the newcomer node requests to learn about the canonical chain, a group of validators that had been online for a longer amount of time can present any alternative version of the blockchain beginning from the genesis block (or just the latest block that the requesting node is aware of as being part of the chain) as the canonical chain. Since there are no PoW proofs to be independently verified, as long as the presented chain is internally consistent and passes some basic sanity checks, the newcomer node has no way of knowing that it is being deceived [32].

Checkpoints embolden initial sync security by severely limiting the extent to which a newcomer node can be presented with a fabricated chain. These checkpoints form a chain in and of themselves (with each one pointing to its predecessor) and anyone connecting to the Ethereum network for the first time knows a priori that the canonical chain must include them, being thus effectively protected from the worst of long-range attacks.

These checkpoints may offer a way to prevent long-range attacks, yet still need to be trustworthy themselves. It is not entirely impossible to imagine ways which could help minimize trust for them even under the PoS framework. For instance, a couple of recent papers proposed reusing Bitcoin mining to enhance PoS security by "anchoring" PoS checkpoints to its mined blocks [33] [34], though this approach would necessarily create a probably undesirable dependency between any blockchain that chose to implement it and Bitcoin. Instead, Ethereum has opted for a more pragmatic approach using community checkpoints.

As Buterin explained in 2014 [35] — Ethereum's transition to PoS was indeed envisioned even before its genesis block was mined — the kind of situation in which weak subjectivity by itself would compromise a blockchain's security is one where a powerful malicious entity can convince the entire community that the hash of some block is different than it truly is, even despite a number of honest participants having been online during that block's creation and having its correct version saved on their computers. Understandably, any such entity (such as a hypothetical oppressive government or a malevolent corporation, as a usual argument goes) that could infiltrate several centralised entities to alter the community checkpoints could just as easily corrupt the client software itself or trick users into downloading a compromised version of it. Even in the extreme hypothetical scenario of a user that was willing to write the client software themselves (a daunting task, to say the least) they too would need to get the protocol's specifications from some, likewise compromisable, external off-chain source.

In conclusion, while a decentralized system always seeks to minimize trust, in practice one will always need to put some amount of trust to a party outside of the system, be that to a software provider or to request social information about a checkpoint. Again, all these worrying considerations only become relevant in the very unlikely event that a majority of validators conspire to produce an alternate fork of the blockchain. Under any other circumstances, there is only one Ethereum chain to choose from and weak subjectivity checkpoints only facilitate and improve the users' experience.

Chapter 5

Execution Clients

In this chapter we will be going over the main focus of this thesis and the subject of our upcoming benchmarks, Ethereum execution clients. We will be detailing the characteristics of the different initial sync modes, state pruning but also how each of the major clients (Geth, Nethermind, Besu and Erigon) implements these and differs from one another. Our overview will primarily be from the perspective of Geth which at the time of writing represents around two thirds of the total execution clients of the network — a clear sign of insufficient client diversity, which the different client implementation seek to mend. Regardless, attention will be given to all important aspects in which those implementations have distinguished themselves from Geth and the consequences of their approaches.

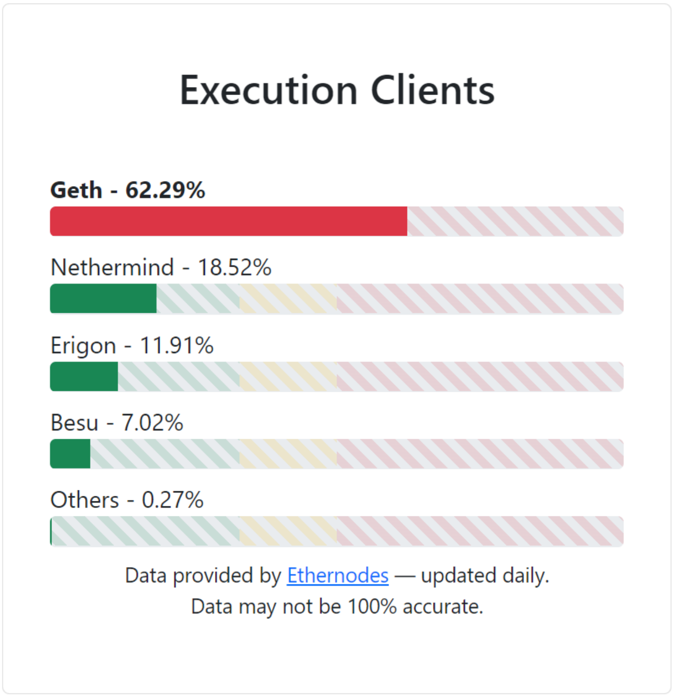


Figure 5.1. *Execution client distribution on Ethereum mainnet* [1]

[1]on February 1 2023; Source: https://clientdiversity.org/

5.1 Initial Synchronization

As we explained in the previous chapter, bootstrapping an execution client in post-Merge Ethereum requires a target chain head, meeting the consensus criteria defined by Gasper. Assuming that an already synced consensus client has provided such a sync target, the execution client can commence syncing with a few specific objectives to strive towards.

First and foremost, it needs to download all the past blocks that are part of the chain, which it naturally obtains by requesting them from rest of the network's nodes. Both internet speed and hardware (especially disk write speeds) can be an impediment in that process. Then these blocks ought to be verified, confirming that block headers are structured correctly, containing the hash of the previous block, a valid timestamp and that the gas limit specified in it is not exceeded by the sum of the total gas used by all the transactions contained in the block (this process additionally included checking PoW proofs in pre-Merge Ethereum). Should any of the above fail then the entire block is rejected as invalid.

A common misconception is that if a client has completed downloading the blocks then it must also be in sync with the network. This is not the case because, by themselves, the blocks do not automatically provide any information about the network's state. State data is *implicit* data, i.e. it is not gathered from the blocks' information communicated but rather needs to be produced through calculations. However, extracting state information from the downloaded blocks is a process more time-consuming in itself than the block download phase and we will be examining the different approaches to this process next up.

But before going over the various sync strategies, let's briefly point out that it is technically feasible to directly download any information necessary from some off-chain source and then import it to one's client of choice. While such a functionality is implemented by the various clients (mainly for backup purposes), if it were to be used in such a way it would evidently undermine the purpose of a decentralized blockchain network as it relies on trusting a single party as the source of all information. Even in a consensus client's checkpoint sync (see Section 4.3.2) where we do rely on some other party for the chain head, it is still assumed that we subsequently verify the blocks leading to it and the transactions within them. So for security purposes, the undisputed standard procedure for a client to get up-to-date with the network is to first connect to it and then request any data needed from its peers, using one of the following sync modes.

5.2 Synchronization Modes

5.2.1 Full Sync

The first sync mode that we ought to analyse is certainly *full sync* so as to properly understand the workload that an execution client is called to fulfill. A full sync consists of downloading and fully verifying all the blocks received from the client's peers, and then executing every transaction in every block one by one starting from genesis. In addition to what was earlier described regarding the block's header, a block's full verification process additionally incorporates checking if its contained transactions are valid in themselves (the sender have enough balance to pay for it along with the gas indicated and their nonce is correct) and that the transactions' receipts included in the block match the receipts that the client computed by executing the transactions. Having completed all these validations itself, the client can always be confident about the integrity of the blockchain data it maintains locally. It is for this reason that this sync mode is necessarily implemented by all clients.

Transaction execution is done on the EVM that all clients also implement and it is through their execution that the world state MPT (as described in Section 3.1.4) is gradually constructed, a process that is actually significantly more time-consuming than the block download phase. By sequentially re-executing transactions, including smart contract execution on the EVM, state balances across all Ethereum accounts (EOAs and contract ones alike) are updated, resulting in a different global state after processing each block. Full sync provides the functionality to optionally save all these intermediate states on the disk, with this also referred to as *archive sync* as it is the requirement to run an archive node.

At the time of writing there exist almost 2 billion transactions in the Ethereum mainnet, steadily increasing by around a thousand per day. It is therefore no wonder why sequentially re-executing all these transactions as described would decidedly be the longest part of the initial full sync process. There is also inherently no way to parallelize it as each state $s+1$ is dependent on state s. Naturally, in all client implementations full sync is by far the most time-consuming, requiring weeks to complete (except in Erigon, which we will later be presenting in detail).

5.2.2 Fast Sync

Fast sync begins similarly to full sync by first connecting to a few peers and downloading all the blocks (first their headers and then later their bodies) from them. While a client syncing using fast sync will too check the validity of the downloaded blocks' headers and will also download the transaction receipts (which contain information about their outcomes), it will not re-execute the transactions themselves.

Instead, the client remains oblivious to the world state until it reaches a "recent" enough block to the current chain head, called the *pivot* block. The pivot block is chosen so that its state trie is recent enough to be close enough to the network's current state, yet far enough back in the blockchain that it is unlikely to change in the future. This allows the node to quickly synchronize with the current state of the blockchain while minimizing the risk of downloading a stale or outdated state trie. In most implementations its default value is 64 blocks behind the chain head.

At that point, the client uses the pivot block's world state root with the purpose of replicating this state locally by iteratively requesting any data it is missing by its peers, commencing what is known as the *state trie download phase*. This root node, as detailed in Section 3.1.3, may contain up to 16 branches to other nodes which will too initially be unknown to the fast-syncing client and will require a separate request to its peers. Then each of these nodes may in turn contain up to 16 branches which likewise must be downloaded from the peers with this traversal and consequent requests to download missing trie nodes from peers continuing until the client completes the reconstruction of the entire state MPT, down to the leaves.

The state trie download phase takes a significant amount of time to be completed, again longer than the block download phase. This is because the world state MPT consists of hundreds of millions of nodes and it is estimated that about 1000 are deleted and 2000 new ones are added at every block. Given that the block time is at roughly 12 seconds, this means that the client is attempting to synchronize a dataset that is changing more than 200 times per second. Moreover, the pivot block will inevitably become *stale* several times during the course of this process, meaning that enough blocks have been added to the chain so as for the client's current pivot block to not be considered as representative enough of the current state. Whenever that occurs the client needs to *pivot*, i.e. pick a more recent pivot block and start syncing again. Pivoting does not mean that the entire process is started from scratch (as there will be enough overlap between the already downloaded state trie and that of the new pivot block) yet it obviously increases the time spent downloading and verifying state. It is for this reason that during this phase a fast network connection with low latency becomes a more important factor that CPU or RAM, although a fast SSD is still of the essence.

While a fast-synced client will not be able to quickly respond to historical queries (i.e. ones further back than the pivot block), it is still used to sync a *full* node as it possesses all the data necessary to respond to them — and can reconstruct the respective historical states if required. Furthermore, once the syncing process is completed and the client has successfully locally reconstructed the world state, it switches to full sync mode. All the above ensure that a fast-synced client can practically guarantee the security of a full client, but at a fraction of the time that is needed for a full sync.

Nonetheless, the once ubiquitous fast sync is clearly and steadily being phased out, with Geth having already dropped it entirely, Besu actively advising against using it for mainnet sync, and Nethermind seemingly likewise heading in the same direction. The cause of that is the emergence of another sync mode called "snap sync".

5.2.3 Snap Sync

Geth's release of v1.10.0 in early 2021 led to a paradigm shift in the bootstrapping of execution clients by introducing the *snap sync* (or *snapshot sync*) mode. In snap sync, while the client once again downloads and verifies all the block headers since genesis from its peers, similarly to fast sync it will not re-execute transactions. But in contrast to it, it will not seek to reconstruct the MPT of the current state either but will instead merely request a *snapshot* of it by its peers.

To understand state snapshots, we must first examine the reasons that led to their development. For all its benefits, fast sync eventually hit an originally unforeseen bottleneck in the form of network latency ultimately caused by the Ethereum's data model. Downloading the hundreds of millions of nodes one-by-one, even batching requests wherever possible, resulted in millions of request to peers and even a few ms of waiting time per request added up to several hours of the fast-syncing node doing nothing. To make matters worse, a serving peer wanting to fulfil such a request had to traverse its state MPT and locate some arbitrary node, requiring to touch up to 7 files (in the average implementation using LevelDB as detailed in Section 3.3) *for every node requested* and with no meaningful way to retrieve these nodes batched as they are stored by hash. Subpar SSDs used by some nodes as well as a probably low upload speed for most further compounded these issues.

All these led to the reexamination of whether communicating the entire state MPT is truly necessary during a client's sync. In essence, the meaningful segment of the state trie is only its leaves, which ultimately contain all the Ethereum accounts, i.e. the "value" part of the key-value pairs that comprise it. While Merkle proofs are certainly vital for verification purposes and the MPT structure enables a number of architectural benefits (as we detailed in Section 3.1.5), in the case of the initial sync this very structure also poses a significant impediment. Since a client obviously trusts the data it has already verified, traversing the world state MPT over and over again is an unnecessary overhead for reads. Ideally, a flat key-value data structure, one containing a *snapshot* of the state trie's leaves would solve all these issues.

Of course we would not want to entirely get rid of the state MPT for reasons already described in Section 3.1.5, but that does not mean that it needs to be used for all purposes. Maintaining such an additional snapshot and serving it to snap-syncing peers makes the syncing process much more appealing for both the syncing and the serving peers. The syncing peer can download this snapshot in contiguous chunks of useful state data, without needing to consistently perform requests and await responses for individual trie nodes, in addition to entirely skipping the download of all the intermediate state MPT nodes. Crucially, since these chunks consist of sequential Merkle trie leaves, any range of them can be individually validated and thus allow for quick detection of potential fraudulent data. On the serving peer's side, since the data requested is no longer randomly keyed (in the form of hashes, as in fast sync), the client needs only perform a significantly smaller amount of contiguous disk reads to serve syncing requests, removing disk I/O delays.

Upon completion of the snapshot download and before the client can be considered synced, it must of course reconstruct the state MPT locally, a straightforward process given that it has all the trie's leaves. However, while this is happening the blockchain is naturally progressing, meaning some of the regenerated state MPT becomes invalid. Therefore, a final *state heal phase* is needed in order to correct any errors in the state MPT. This is, once more, a phase were a performance SSD is vital since in order for the client to catch up with the current state the healing must be able to outpace the growth of the blockchain.

Simultaneously with the state trie download phase, block bodies and receipts are being downloaded to be preserved in the disk as is the requirement for every full node.

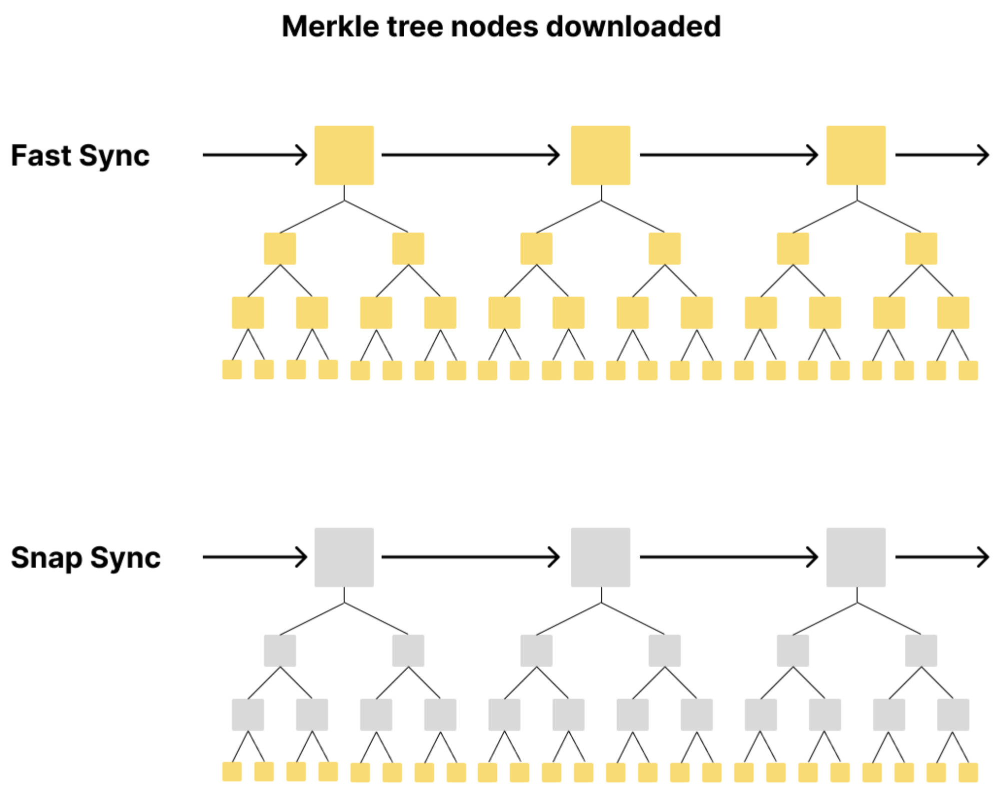


Figure 5.2. *Fast vs snap sync visualization*

Lastly, once snap sync is completed then the client switches its sync mode to full, a switch which helps ensure the long-term stability of the network.

It should not be understated that, once locally constructed, the state snapshot can serve other purposes other than serving syncing requests. Having successfully reduced read time complexity from $O(logN)$ to $O(1)$ (always multiplied by the storage engine's overhead), it makes sense that the client would use this snapshot in any operation that requires such reads. One example could be EVM smart contract execution which, aside from state writes, may often involve a decent number of state reads.

The only meaningful drawback of this approach is that all clients need to maintain a state snapshot at all times to satisfy requests from newcomer nodes. This is maintained live without noticeable performance impact in regards to system resources, partly due to in-memory diff layers that help prevent redundant reorgs of the in-disk snapshot [36]. The state snapshot also does occupy an additional few tens of GB on the disk as it is comprised of duplicate information from the state MPT, an admittedly negligible overhead when compared to the hundreds of GB already needed to sync a full node.

In the patch notes of Geth's v1.10.0 [37], where snap sync was originally introduced following years of development (cautiously, as obviously no client had snapshots ready at the time and needed to generate them for the first time), it was denoted that synchronizing the mainnet state with it against a mere 3 serving peers took a fifth of the time that fast sync required, with an over 99% reduction of both the amount of reads those serving peers had to perform on their disks and the amount of packets needed to be exchanged between them.

Snap sync is currently on its way to replace other sync modes, especially fast sync which it directly succeeded. Following its popularization in Geth, both Besu (since v22.4.0) and Nethermind (since v1.13.0) implemented support for snap sync. In all 3 of these clients, snap sync has become the default sync mode.

5.2.4 Checkpoint Sync

Not to be confused with the consensus client's sync mode namesake, *checkpoint sync* operates using the same principle in execution clients too. Only available in Besu as an early access feature at the time of writing, this sync mode behaves exactly like snap sync, but instead of syncing from the genesis block it syncs from some other manually provided "checkpoint" block.

Checkpoint sync is faster and occupies less disk space than snap sync, though this is achieved by altogether ignoring some historical data (e.g. receipts) of earlier blocks. While the earlier blocks' bodies are always themselves downloaded (as is a requirement for every full node), since the checkpoint block acts as a ground truth there are certain security considerations regarding older blocks' validation (or rather the lack thereof). These are the ones that we went over when we explored consensus clients' checkpoint sync in Section 4.3.2.

5.2.5 Staged Sync

In all previous sync modes the various aspects of workloads (including downloading block headers and bodies, executing transactions or downloading state data) are executed in parallel wherever possible. While this intuitively seems to be beneficial to the client's efficiency, it turns out that doing so also prevents various optimizations. A simple example of such an optimization is inserting data in large pre-sorted (in-memory) batches versus inserting each element as it is received (effectively at random) into a database that keeps its contents sorted at all times. On this basis, the Erigon team devised a rearchitected version of full sync, called *staged sync*, that seeks to redefine how client syncing can be done more efficiently.

As its name suggests, staged sync is separated in stages — 10 on a high level — that are executed sequentially. Initially, block headers are downloaded, verified and stored, before proceeding to download the blocks' bodies. Senders' signatures (the addresses contained in the `from` of every transaction) are then located and likewise persisted in the database. The majority of the total sync time is spent on the next stage which is the re-execution of all transactions since genesis which produces a `PlainState` (simple key-value store of accounts and their contents) along with receipt and event logs. Hashes and trie roots are calculated next up, which are followed by the (optional) creation of the call trace, history and log indexes. The final stage consists of creating a `TxLookup` mapping between transaction hashes and the height of the block in which they were included for quick future retrieval.

This sequence of stages will be executed once upon bootstrapping the node until its completion, at which point it will restart from its first stage. That second time, of course, there will be a much smaller amount of blocks to process meaning that the entire loop will last significantly less. Eventually, these repetitions converge to processing 1 block at a time, as they are being produced by the network. This flow (along with the block receival flow in Erigon) as well as the exact stages of this sync mode are depicted in Figure 5.3, which is further expanded upon in Erigon's extensive documentation [38].

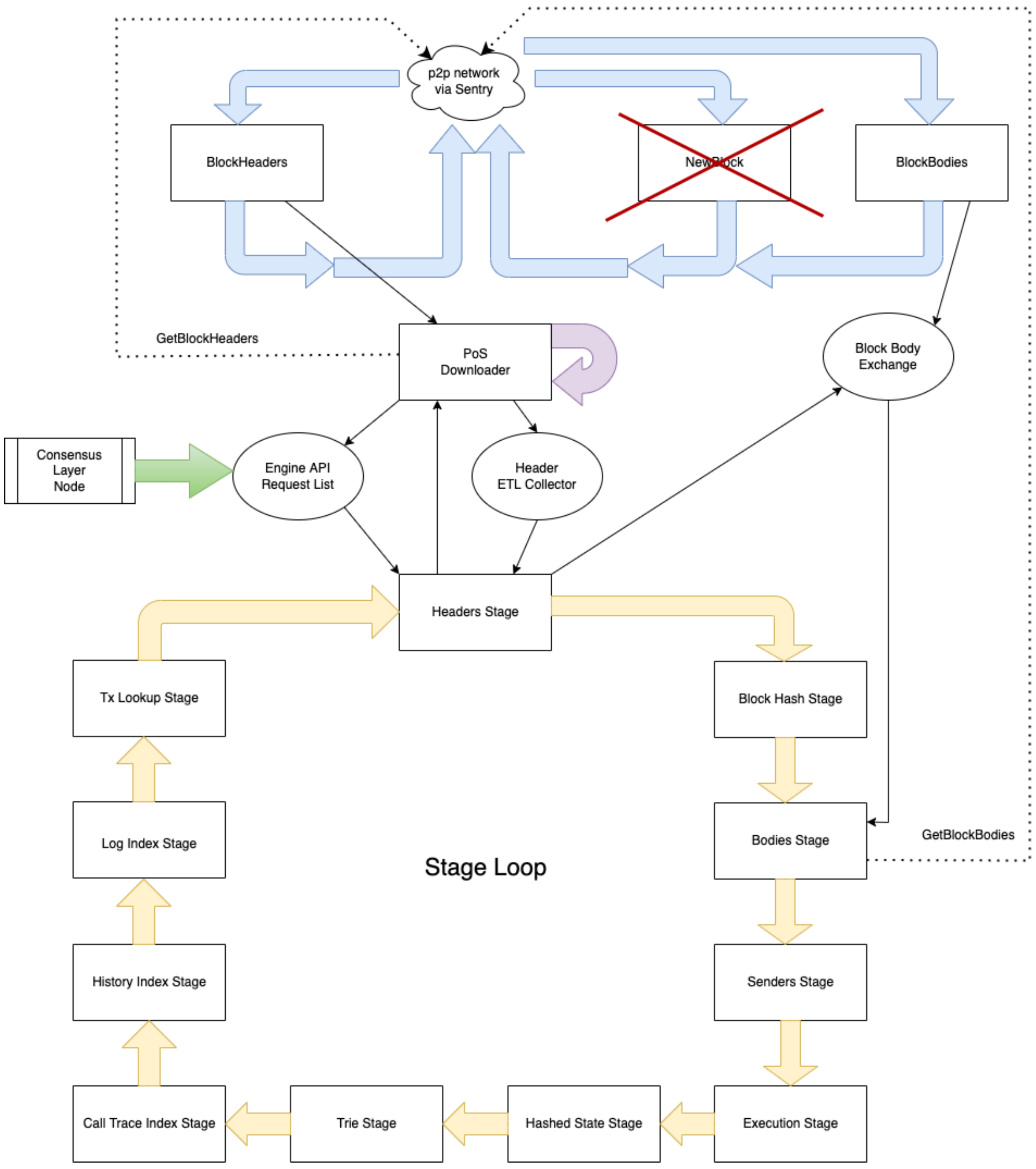


Figure 5.3. *Erigon's control flows (staged sync loop in yellow arrows)* [2]

[2]Source: [38]

5.2.6 Light Sync

Light syncing is used exclusively as part of a light node's bootstrapping and relies merely on downloading and validating block headers, which was relatively straightforward in PoW Ethereum. There exists a mathematical function which a block header must satisfy in order to be valid, and it would be computationally very intensive for an attacker to produce such a valid header. A light client could always easily look for the longest chain of valid block headers, and assume that the resulting chain is the canonical one.

In PoS, however, the same process is not that simple. For a block to be considered valid the client must first confirm that the validator who created it is indeed the one who was meant to do that and verify their signature against the public keys. While the latter part is done in constant time, simply retrieving that validator's public keys from the state trie has a logarithmic complexity. On the surface, this shows that light clients can exist on PoS, albeit with some additional computational overhead. In fact, protocols involving checkpoints have been proposed (such that the validity of every n blocks needs to be checked instead of that of every block) exhibiting that PoS could potentially prove to be even more light-client friendly than PoW was [39]. Nonetheless, at the time of writing none of these proposals have yet been implemented by any execution client and as such light sync is not currently operational on the Ethereum mainnet.

5.3 State Pruning

As Ethereum increases its adoption and use, its state inevitably grows larger and archiving it becomes increasingly challenging. As we mentioned in Section 4.1.2, the total (archive) chain data size indicatively on Geth surpasses 13TB and is constantly increasing at a somewhat steady pace of around 300GB per month. While one could constantly upgrade their storage to keep up with that growing demand — and archive nodes must indeed do exactly that as they are by definition meant to preserve all historical data — such a requirement would be prohibitive for the average user.

Of course, the average user does not need the entire history of states since genesis. Already, when an execution client is bootstrapped using any non-archive sync mode, it does not store intermediate historical states — i.e. it already downloads a *pruned* version of the Ethereum state. Once it has finished synced however, it always switches to full sync mode, necessarily saving new states to the disk as the blockchain progresses. An issue that then arises when a client has been running for a considerable amount of time is the inevitable filling of the available disk space.

At a high level, all data in Ethereum can be separated in two types, *permanent* and *ephemeral*. The blocks themselves as well as the transactions contained in them are examples of permanent data. It must be emphasized that permanent data is never pruned as they are vital for guaranteeing the long-term stability of the network. By contrast, state data (including anything mutable like an account's balance at some block) is considered ephemeral and may be stored separately. Since ephemeral data can be reconstructed using the permanent data whenever necessary, client implementations have a great degree of freedom on how to store it and how much of it they wish to have readily available at any time.

The practical solution to the growing disk space occupied by the client is to delete older state data, which are not necessary for its operation. However, it turns out that deleting these concurrently with inserting new pieces of state data block-by-block as a node receives them is quite a difficult problem. Since state in Ethereum is stored in a trie data structure — and since most blocks only change a small fraction of the state — two such tries will share huge portions of the data with one another. It can easily be decided whether the root of an old trie is stale and can be deleted, but it is exceedingly costly to figure out if a node deep within an old state is still referenced by anything newer or not. Several pruning solutions had been proposed over the years, but consistently broke down as the size of the Ethereum state kept growing.

As a result of the above, in v1.10.0 Geth introduced *offline pruning*. This kind of pruning takes advantage of the state snapshots introduced in the same Geth version by constructing a bloom filter which helps identify and delete stale trie nodes. Upon completing that, Geth performs a database compaction and manages to reclaim free disk space[3].

Offline pruning is not done automatically but is instead meant to be periodically manually initiated after, as its name suggests, shutting down the client. When running a validator post-Merge, taking one's node offline for any amount of time can result not only in missed profits but also to some limited slashing. In these cases, the standard way of approaching state pruning is by syncing a secondary execution client in a separate disk and linking one's consensus client to it while the primary execution client is pruning.

State pruning is of course not exclusive to Geth. In fact, Nethermind has pioneered a *full pruning* approach that does not necessitate shutting down the client [40]. This involves creating a separate empty MPT and a period of duplicate writes (both to the old and the new MPTs) while the client is operating as usual, following which the old state MPT can be safely deleted. Full pruning is only possible in Nethermind because, contrary to other clients that keep all kinds of data (state, blocks headers and transactions) in a single database, Nethermind has a separate one for each, allowing for easy and targeted deletion of state data without affecting the stored permanent data.

Nethermind's pruning approach effectively circumvents the challenge of locating which trie nodes should be kept and which ones are stale and thus should be deleted. A certain drawback though of it is that it causes a lot of additional disk writes which, aside from any performance impact, also cause unnecessary wear on the SSD used. It is also a reasonably time-consuming process, potentially lasting more than a day depending on the hardware used. For these reasons, it is recommended not to run this pruning task more than once every few weeks.

In conclusion, there is no silver bullet solution to pruning state data. Despite Nethermind's aforementioned approach, its team acknowledges that there exist better solutions together with different storage models and with which it is currently experimenting at

[3]As a side note, initiating this process requires a few additional tens of GB of free disk space meaning that it cannot be used to salvage a hard drive that has already been completely filled.

the time of writing. Alternative storage data structures (like Besu's Bonsai tries or the currently under development Verkle trees) can provide different approaches to pruning as well. Consequently, novelties in pruning methods are to be expected in the near future.

5.4 Implementations

5.4.1 Geth

Go Ethereum (usually referred as *Geth*) is the original Ethereum execution client implementation, written in Golang and maintained by the Ethereum Foundation. It has been the most widespread client with the biggest user base for the entirety of Ethereum's history. In addition to interacting with the Ethereum network manually through a console, Geth also has built-in support for JSON-RPC based APIs, which can be exposed via HTTP among other ways, and allow user programs to easily access Ethereum information.

At present, Geth uses LevelDB where it stores both state and chain data. The latter are separated internally on the basis that older blocks are less likely to be needed for retrieval and — as of v1.9.0 — can thus be stored on slower, cheaper storage. Offline pruning of state data is supported, as previously described.

Geth solely supports full and snap sync modes. Snap sync was made the default sync mode over fast sync in v1.10.4, the support for which was ultimately dropped in v1.10.14. Despite that, Geth continues to serve the relevant requests to other client implementations that still rely on fast sync.

Up until The Merge, a user could opt to run a light client using Geth's light mode syncing, with the benefits and detriments that such a client has compared to a full one, as detailed in Section 4.1.3. Additionally, Geth supported an even lighter sync mode which resulted in an *ultra light client* (ULC). Its difference during syncing as opposed to light mode syncing was that a ULC did not even check the PoW in block headers, getting its data from one or more trusted light servers (the addresses of which had to be provided upon starting the client). Neither of these Geth light clients currently work on PoS Ethereum, but new PoS light clients are being developed at the time of writing.

5.4.2 Nethermind

Founded by a small team in 2018 of the same name, Nethermind is an Ethereum implementation created with the C# .NET tech stack. Like Geth, Nethermind can be used both to sync the Ethereum mainnet as well as various testnets and even private networks. Nethermind uses RocksDB for its storage but, unlike Geth, does not save all Ethereum data in the same database but splits them in multiple ones which allows it to perform live full pruning, as we detailed in Section 5.3.

In regards to sync modes, Nethermind implements full, fast and snap sync. Snap sync has been supported since v1.13.0, though Nethermind can only download the state snapshots but not serve it to other clients at the time of writing. Due to this fact, snap sync in Netherming can be used only for the networks (mainnet and testnets) that are also supported by Geth. However, contrastingly to Geth, in fast and snap sync modes

Nethermind does not begin blocks' data download until after the state download has been completed. Especially in snap sync, its initial exclusive focus on state syncing enables Nethermind to take advantage of sequential disk write operations, consequently managing to complete it in remarkably fast speeds. This approach additionally allows users to quickly get up to speed with the network and present them with the *option* to download block bodies and receipts[4].

5.4.3 Besu

Hyperledger Besu is an enterprise-grade Ethereum client written in Java. Formerly known as *Pantheon* by ConsenSys, its first official release was in 2019. It is now maintained by the Hyperledger Foundation, an umbrella project of open source blockchains and related tools, started by the Linux foundations but with contributions from a long list of member organizations and companies. Apart from the Ethereum mainnet and its public testnets, Besu is often run on private permissioned networks. As with previous clients, it supports smart contract and Dapp development, deployment, and operational use cases as well as common JSON-RPC API methods over HTTP.

Besu uses a RocksDB key-value store to persist chain data locally. Since v21.1.0 in early 2021, Besu supports optionally syncing using the newer Bonsai tries for storage (Bonsai mode) instead of the traditional MPTs (Forest mode). This data format (which we described in Section 3.1.6) offers noticeably faster sync speeds but also provides implicit tree pruning which results in reduced disk usage regardless of sync mode. These improvements become even more pronounced in full archive sync, where using Bonsai achieves up to an order of magnitude lower storage compared to Forest mode (an estimated 1.2TB with Bonsai as opposed to ~13TB otherwise).

In addition to full sync, Besu supports fast, snap and checkpoint sync modes. Checkpoint sync, which we explained earlier, is still an early access feature. Moreover, fast syncing is being discouraged by Besu's developers who state that it might eventually become impossible to fast sync the Ethereum mainnet in the future. Snap syncing Besu using Bonsai is their latest (stable) recommendation for both lower sync times and lower storage requirements.

5.4.4 Erigon

Formerly known as *Turbo-Geth*, Erigon began as a fork of Geth with its first public alpha version being released in mid-2020 after more than two years in development. Soon after that however, the Erigon team realized that its plans to provide a faster, more modular and better optimized Ethereum implementation required a radical overhaul of the entire architecture which led to a rewrite of the database structure, data model, and sync process.

Erigon is undoubtedly the most unique out of the clients of our study. Contrary to the more monolithic designs of other clients, it boasts a modular design which enable paral-

[4]Nethermind presents the download of these as distinct options of its sync modes: `https://docs.nethermind.io/nethermind/ethereum-client/sync-modes`

lelized development of the different components. These components include the client's core, the RPC API, the P2P sentry and the `TxLookup` mentioned in Section 5.2.5 among others and can each be compiled into a separate executable and run as a standalone application. It additionally includes an embedded consensus client that is sufficient to optionally complement Erigon post-Merge, but not yet replace a dedicated consensus client when running a validator node.

Throughout its history Erigon changed its storage engine several times, going from *BoltDB* to *LMDB* before finally settling on *MDBX*. MDBX is largely different from the storage engines used by other clients, first of all architecturally in that it is based on B+ trees instead of LSM ones, as explored in Section 3.3. The Erigon team has provided several reasons for this choice, among which the need for faster and more predictable times in random disk reads for their implementation. In addition to that, traditional LSM-based databases are not ACID-compliant meaning that a potential crash or power failure could corrupt them, something that was deemed a non-starter for a client which only provides a full sync mode. While all these applied equally on LMDB too, the switch from it to MDBX happened both for performance reasons as well as a number of desirable features available only on the latter.

As will also be evident from our benchmarks in the following chapter, Erigon has different resource usage patterns than the rest of the clients. Most notably, the resources it requires vary depending on the stage it is currently executing but in general it comparatively uses up far more RAM which (along with a fast SSD) does significantly affect sync times. The reason for that is its ability of preprocessing grater amounts of data in-memory due to its staged sync, rendering the eventual write operations to the disk faster by an order of magnitude according to its developers.

Erigon's sole sync mode is the aforementioned staged sync, which can be used to sync either simply a full node or an archive one. The latter can be completed using less than 3TB of disk space and, more impressively, in less than 5 days on reasonably fast hardware. Nevertheless, Erigon's lack of a faster sync mode drives off some potential users and also means that any potentially disruptive changes will always require a, growingly impractical, full replay of all blocks from genesis. This is acknowledged as a problem by its developers and the implementation of snapshot sync is regarded as a priority for a future major release.

Finally, it is worth mentioning that at a time there was a full (and faster) implementation of Erigon in Rust, named "Akula" which was however short-lived (see Section 5.4.5.2). Furthermore, a C++ version of Erigon called "Silkworm" is under development, though by a significantly less active ecosystem. Not even having reached an alpha phase of release yet, Silkworm is at the time of writing unable to actually sync a blockchain from genesis but relies on an already synced database by Erigon to fulfill other execution client tasks.

5.4.5 Discontinued clients

Even though the above 4 are the only Merge-ready Ethereum execution clients at present, a few other abandoned client implementations merit a mention for historical reasons as well as insight on how they affected our contemporary ones.

5.4.5.1 Parity - OpenEthereum

Originally introduced in 2015 as *Parity Ethereum* by Parity Technologies this client written in Rust had been the second most widely used (after Geth) for most of Ethereum's history. In 2019 Parity announced the client's transition to *OpenEthereum*, with Gnosis-DAO taking primary oversight of the project [41] before eventually in 2021 announcing their intention of ending support for it in favour of Erigon. What led to that decision was OpenEthereum's huge legacy codebase, the managing of which was proving to be increasingly difficult, especially with Ethereum's transition to PoS fast approaching. As such, OpenEthereum has been officially deprecated since The Merge, with the team behind it recommending using Erigon instead [42].

Parity's historical importance also stems from an incident in September 2016, when a bug in Geth's v1.14.11 caused Geth clients to run out of memory and crash, thus preventing the mining of new blocks[5]. While Geth's developers quickly worked out the root cause and deployed a fix within hours, the Ethereum network would have completely halted in the meantime had it not been for Parity, the only other client implementation at the time. Parity clients were unaffected by the bug and continued to produce blocks as normal, keeping the network online and once again proving the need for client diversity.

Despite not being used anymore, Parity also introduced an innovative mode of syncing worthy of discussion, named *warp sync*. Warp sync was a sync mode that preceded snap sync and from which the Geth team took many design ideas to develop it. It likewise involved snapshots created by each client which could be served to newcomer clients. The most significant difference between it and snap sync was that warp sync relied on static snapshots created periodically by the clients, in contrast to snap sync's dynamic snapshots that are kept updated in real-time. This meant that every 30000 blocks (or about 5 days) — as was the interval chosen at the time — the client would have to regenerate the snapshots practically from scratch by continuously iterating the state MPT, something that was even then seen as unsustainable long-term given the Ethereum state's rate of growth. Moreover, instead of following the Merkle trie layout, the data format of warp sync snapshots was comprised of a manifest (metadata about the snapshot) followed by raw block data about the blocks. The obvious drawback of that approach was that chunks of such data could not be individually validated, forcing syncing nodes to download the entire snapshot (of several GB) before they could verify it. In conclusion, while warp sync was certainly a novelty at the time and faster than Geth's fast sync, it was admittedly wholly outclassed by snap sync which it helped inspire.

5.4.5.2 Akula

Following the abandonment of OpenEthereum, there was a large amount of interest in a high-performance Ethereum implementation written in Rust. Akula was such a client that grew out of an internal project in Erigon's team at the end of 2021, where it

[5]Source: `https://blog.ethereum.org/2016/09/18/security-alert-geth-nodes-crash-due-memory-bug`

was originally conceived as a helper library for Erigon's database. It had lower storage requirements and was up to twice as fast as Erigon in full sync mode using the same staged sync approach, even achieving to bootstrap an archive node in a record time of under 3 days.

Despite the significant progress done on Akula during 2022 and it beginning to gain traction in the community, in November of the same year it was unexpectedly announced that the Erigon team was winding down support for it [43]. The reason cited for this decision was the impending release of another, at the time unnamed, execution client written in Rust with many similarities and nearly identical scope as Akula. Predicting that it would soon be surpassed and that it would prove challenging to secure funds for it in the future, the Erigon team deemed it unsustainable to keep spending resources and development effort on Akula.

That unnamed project was revealed in December 2022, when Web3 investment firm Paradigm announced that it was developing a new execution client written from scratch in Rust named *Reth*. According to what little is known about it at the time of writing [44], Reth will indeed also be using the staged sync architecture pioneered by Erigon and the MDBX storage engine, with its first release being expected in early 2023.

5.4.5.3 Smaller projects

- Ruby-Ethereum: A Ruby implementation of Ethereum. Little development was done for it, mainly in 2016, and there has not been an Ethereum client written in Ruby ever since.

- Aleth: Part of a collection of official C++ Ethereum libraries and tools by the Ethereum foundation, formerly known as "cpp-ethereum". Its last release was in December 2019.

- Mana-Ethereum: Built using Elixir, it sought to take advantage of the Erlang Virtual Machine to provide a massively scalable Ethereum client. Despite its GitHub repository not having been explicitly archived, it has not received any updates since 2019.

- Trinity: A client written in Python by a small development team within the Ethereum foundation. It never left the alpha release stage and was officially deprecated in mid-2021, having served mostly as a research and educational tool for the community. A large number of python-based modules related to Trinity yet continues to be maintained by the same team, including a Python implementation of the Ethereum Virtual Machine, called "Py-EVM".

Chapter 6

Benchmarks

6.1 Methodology

In the experimental chapter of this thesis we will be running a full node on the Ethereum mainnet, bootstrapping different execution clients with various configurations and recording certain metrics throughout their initial sync (and up to few hours following that). Our goal in performing these benchmarks is twofold:

- A comparative analysis across different clients using the same sync mode in an attempt to quantifiably evaluate their different approaches to optimizations.
- A comparative analysis between different configurations of the same client so as to better evaluate the performance impact some parameters can have on a client's bootstrapping.

Firstly, let's justify our selection of snap sync for our benchmarks. When initially conceptualizing this thesis, there was an intention to perform inter-client bootstrapping comparisons for multiple sync modes. However, Erigon merely supports its own unique staged sync mode (as we went over in Section 5.2.5) and the only 3 sync modes that used to be supported by all Geth, Nethermind and Besu are snap, fast and full. A full client sync takes weeks (often upwards of a month) to bootstrap, something prohibitive in the context of a thesis where we want to run several such syncs. On the other hand, fast sync is clearly and steadily being phased out, with Geth having already dropped it since v.1.10.14 and Besu actively advising against using it for mainnet sync. Consequently, with snap sync becoming the de facto "standard" way of bootstrapping an Ethereum execution client, it made sense to only base our inter-client comparisons there.

A further goal is to additionally perform some limited intra-client comparisons, i.e. between different configurations of the same client. These focus mainly on the cache memory used which, as previously emphasized when exploring storage engines and how the clients use them during syncing, can prevent redundant disk writes and as such provide significant performance improvements. Furthermore, as fast sync is still usable in Nethermind, an additional run was executed using it so as to juxtapose it with snap sync. Lastly, in the case of Besu, there was both a choice between snap and checkpoint sync modes but also a data storage one (optionally using the experimental Bonsai tries) to consider. We performed runs using Bonsai tries for both of these sync modes and we intended to juxtapose these results with the respective ones from runs using the traditional Forest data storage. The latter, however, was not possible due to a related bug in Besu[1] that remained unresolved throughout the weeks of our benchmarks and, as such, we were only able to perform Besu runs using Bonsai tries.

6.1.1 Metrics

The focus of our comparisons will be on initial sync time but also on the system resources each client consumes over time during that sync (and up to a few hours after its conclusion, to juxtapose it with normal use). These resources are reflected in the following 7 metrics:

- **CPU usage** (percentage)
- **RAM usage** (GB)
- Client data **disk size** (GB)
- **Disk I/O**: Reads and Writes (MB/second)
- **Network traffic**: Sent and Received (MB/second)

Peer counts are also important when syncing an Ethereum client but, as long as their number does not drop too low, increasing the number of peers does not directly translate to increased sync speeds. On the contrary, having too many peers can lead to spending more system resources (network bandwidth but also disk reads) in order to fulfill their various requests. As such, we wanted to set the `max-peers` parameter for our benchmarks to a sufficiently high value that will help our clients' bootstrapping but not cripple our network — a value of 128 was thus chosen for consistency across our clients, with the exception of Erigon where the default 100 were kept. Peer counts over time were recorded for each benchmark and will also be plotted along our aforementioned 7 metrics.

It must be emphasized that studying a client's bootstrapping is certainly not a way to categorically determine its quality or efficiency. Whatever cost it may incur, either in terms of time or system resources, is one that will be paid only once in the lifetime of a node (assuming no hard failures that force a restart from scratch). In spite of that, studying a client's initial sync is objectively the ideal way to understand the workload that it is tasked with performing in order to preserve the Ethereum network. Moreover, identifying what system resources may be underutilised or cause a bottleneck in this entire process could lead to ideas for future improvements.

An inspiration for our approach was an insightful similar resource analysis done on Ethereum consensus clients (on the Beacon chain, before The Merge) in 2021 [45].

[1] Related GitHub issue: `https://github.com/hyperledger/besu/issues/4901`; when using the user-suggested fix we failed to locally build from source. Even if we had succeeded, any benchmark results would not have been suited for comparisons as we would have been using a non-release version of Besu.

6.1.2 Data gathering

In order to gather the data required for our aforementioned metrics, a custom `Bash` script was created[2], making use of some common Linux command line tools. The majority of what we required — more specifically, CPU, RAM and disk I/O — was readily available by `pidstat`, simply by passing the client's process ID to it immediately once the client begins running. Network traffic data was gathered using `nethogs`, a useful tool that breaks down sent and received traffic by process which is ideal for our use case — unlike most others of its kind (like the popular `netstat`) which group network traffic per protocol or subnet. Lastly, the growth of the client data size on the disk (including both chain and state data) was observer by the built-in `du` tool. It is noted that, since `du` necessitates iterating and checking directories for the sizes of each file they contain (as does any other similar tool), it would occasionally fail to access some inner files in the chain data when these were created and deleted in quick succession. Errors caused for that reason were safely ignored as it was assessed that they cannot meaningfully affect our conclusions. Our script ran the above at an interval of 1 second, logging its output of all these values to a csv file, which could then be used to produce the graphs we will be presenting in the following section.

There is a number of other options to consider when seeking to monitor an Ethereum node, most notably *Prometheus-Grafana* and *Netdata*. Both of these tools provide some quite insightful dashboards for tracking various real-time metrics and can be truly useful for a node operator. However, the graphs that they display are a product of extensive custom configuration and while this has already been done to an extent for all of our execution clients, they do not all provide the metrics that we require nor do they provide what they do in a consistent manner across all clients. Furthermore, since those graphs would each be produced by a single client execution, there would be no easy way to plot metrics across different executions on the same graph against one another for easier visual comparisons. As such, it was preferable to obtain raw data in a csv format which could then easily be parsed and used to plot any and all graphs that most closely matched the focus of our analysis.

Lastly, the output (`stdout` and `stderr`) logs of all runs for every client, after appropriately setting the `verbosity` parameter, were redirected to log files. These logs were necessary to provide additional context to our results and help us locate the various events that occurred throughout the syncing process as well as to keep track of peer counts in order to later plot them.

6.1.3 Consensus client selection

Post-Merge it is not possible to independently run an execution client on its own. Both an execution and a consensus client must be run together in order to gain access to the Ethereum network and the connected consensus client needs to already be synced before the execution one can begin bootstrapping.

[2]GitHub repository: `https://github.com/TsiarasKon/Ethereum-Client-Metrics`

Our only requirement for our consensus client was to be as lightweight as possible so as for its resource needs to not weigh down our execution client. As such, Nimbus was chosen, the syncing of which was completed nearly instantaneously through the use of checkpoint sync (as described in Section 4.3.2). For the purpose of uniformity, Nimbus was used in the Erigon run too, despite Erigon's capability to run with its own embedded consensus client.

Even though benchmarking our consensus client is not part of our scope, an indicative resource usage of Nimbus over a 12-hour period is presented in Figure 6.1. No commenting of these results needs to be included here other than to point out that the values across all metrics are several times lower than those we will soon examine in Section 6.2, proving that the weight of running an Ethereum node in terms of system resources undeniably lies on the execution client.

Lastly, had it not been for the need to also run an execution client (and given that no other application runs on our system along with our experiments) it would have been simpler to use some system-wide resource monitoring utility like `dstat` with sufficiently accurate results. We instead opted for the more precise way of isolating exactly the resources used by the execution client using it process ID as presented above, and these results are the ones used for our resulting graphs. This way we have successfully eliminated the parameter of the consensus client from our experiments and thus, for example, a potential spike in its network traffic will correctly not be depicted in our execution client analysis.

6.1.4 Hardware

All the experiments were run on the same computer with the following:

- **CPU**: AMD Ryzen 5 2600X Six-Core 3.6 GHz
- **RAM**: G.Skill RipjawsV 32GB DDR4 (at 2933 MHz)
- **Motherboard**: MSI B450 Tomahawk
- **Disk**: Samsung 980 Pro NVMe SSD (in PCIe 3.0)
- **OS**: Ubuntu 22.04 LTS
- **Internet**: Stable 100/100 Mbps FTTH connection

The technical requirements of running an Ethereum execution client are not all of equal importance. Since the sync process is not multi-threaded, a high-end multi-core CPU is necessarily underutilized. Most clients do not take full advantage of RAM either — with the notable exception of Erigon — and while some stable and reliable connection is required, high network speeds are not obligatory.

The most common bottleneck in Ethereum clients' syncing is usually the disk used. It has long been impossible to bootstrap a client in HDDs, while some SSDs may too be unsuitable due to their subpar speed and lack of some components. An ideal consumer SSD would at present be an NVMe (NVM Express), with DRAM, TLC (triple-level cell)

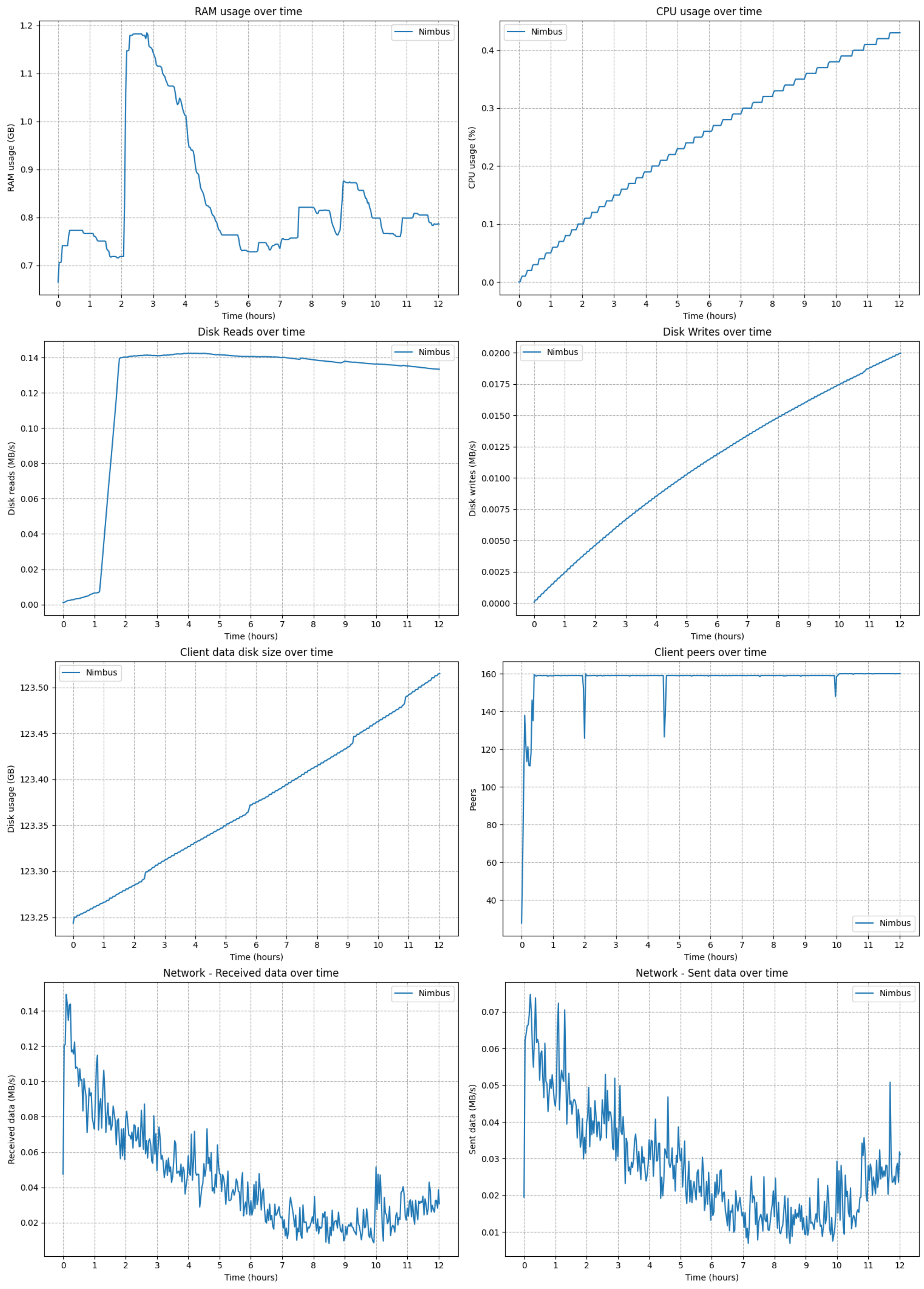


Figure 6.1. *Nimbus indicative benchmark results*

instead of QLC (quad-level cell) and an SLC (single-level cell) cache. Analyzing these terms would necessitate delving into the implementation details of SSD architecture in general, so for our experiments suffice it to say that our Samsung 980 Pro fulfills all these criteria. In regards to its size, as we will see in the results below and thanks to state pruning it is currently possible to snap sync an execution client using less than 1TB of disk space. Of course, one should also take into account the consensus client which, presumably, will too be running on the same drive and may require up to 200GB. Given the steady expansion of the Ethereum network's chain size and the common fact that an SSD performance degrades when it reaches low amounts of free space, the recommended SSD size at the time of writing in order to run a full node is 2TB.

It is of course feasible to achieve better sync times and overall results using better hardware than the above. A CPU with stronger performance in single-threaded applications is generally preferable, as would be a Gigabit or faster internet connection. Lastly, probably the most meaningful upgrade would be a CPU-motherboard combination that supports PCIe 4.0 (Peripheral Component Interconnect Express), using which with a compatible drive could yield up to twice the speeds of PCIe 3.0.

6.2 Results

All client runs were executed from mid-January to mid-February 2023 (around block 16.5M). It is noted that since the experiments for each setup were run sequentially on the same machine up until the initial syncing was completed (and a few hours beyond that), the later runs are comparatively slightly "disadvantaged" in that they need to catch up with a later block that those of the earlier runs. While technically this should result in slightly greater sync times and disk space used — the resource usage metrics would still remain unaffected — in practice that difference is small enough that it needs not be taken into account in any of our later conclusions.

Default configurations for mainnet sync were used in all of our clients' runs. Wherever adjustments were made for our intra-client comparisons or for any other purpose, the parameters changed and values used are denoted per client configuration.

All graphs were produced by custom-made `Python` scripts[3], making use of the popular `pandas` and `matplotlib` libraries. Since our input data contained 1 second interval values over several hours, these were averaged out (e.g. by using the average network traffic of 1 minute's worth of values as a single data point) so as to produce smoother and more easily readable plot line graphs.

We will be presenting the results for every metric per client, commenting on them and the depicted intra-client comparisons. The completion of the various stages in each run is represented by a marker on the plotted lines of each, as these events were retrieved from their execution logs. In all executions, the last event marked also signals the completion of the entire initial sync process. At the end we will evaluate the inter-client comparison results — produced using the best configuration for every client — and perform an overall

[3]GitHub repository: https://github.com/TsiarasKon/Ethereum-Client-Metrics

assessment of our results.

6.2.1 Geth

Geth **v1.10.26** was used. In all the configurations, aside from the cache adjustment, max peers were set to 128 and the rest of the parameters were left at defaults.

Table 6.1. *Geth benchmark configurations and results*

Label	Sync mode	Cache (MB)	Sync Duration
`Geth_1`	Snap	4096 (default)	22h 07m
`Geth_2`	Snap	10683 (max)	19h 44m

When tried to provide more cache to Geth through its parameters, it adjusted the amount down to ~10.6GB on our computer. In regards to peers numbers, we can see that there are several fluctuations (though in similar patterns) in both of our Geth runs, with them only trying to reach their `max-peers` value only after their sync was completed.

In Geth the different parts that comprise the sync process happen in parallel whenever possible. In snap sync in particular, as we detailed in Section 5.2.3, a chain of block headers is initially constructed by requesting them from peers. Following that, the state's (snapshot) download begins in parallel with the download of all the blocks' bodies and receipts. Lastly, once the state sync is completed, it requires healing. The completion times of all these events are marked in all graphs of Figure 6.2.

As we can see, the headers' sync happens very quickly (around the 20m mark in both configurations) which explains the initial spikes in network traffic. The majority of the sync's duration is spent on the parallel download of blocks and state, with the former occurring first (around the 12h45m mark, again in both configurations). The state's sync is where we see the impact of the increased RAM cache, with this phase being completed at 19h26m in `Geth_1` as opposed to 21h46m in `Geth_2`. As expected, the higher RAM usage is correlated with a lower amount of disk writes (to LevelDB) throughout the initial sync. Lastly, the state heal phase is a swift one, beginning immediately after the state has been synced and lasting about 20m.

6.2.2 Nethermind

Nethermind **v1.15.0** was used. In all the configurations, aside from the sync mode change, max peers were set to 128 and the rest of the parameters were left at defaults.

Table 6.2. *Nethermind benchmark configurations and results*

Label	Sync mode	Sync Duration
`Nethermind_1`	Snap	11h 59m
`Nethermind_2`	Fast	1d 08h 07m

Quite differently to Geth, Nethermind's state sync in snap mode was completed remarkably fast (at the 2h39m mark). While at that point it could be considered synced

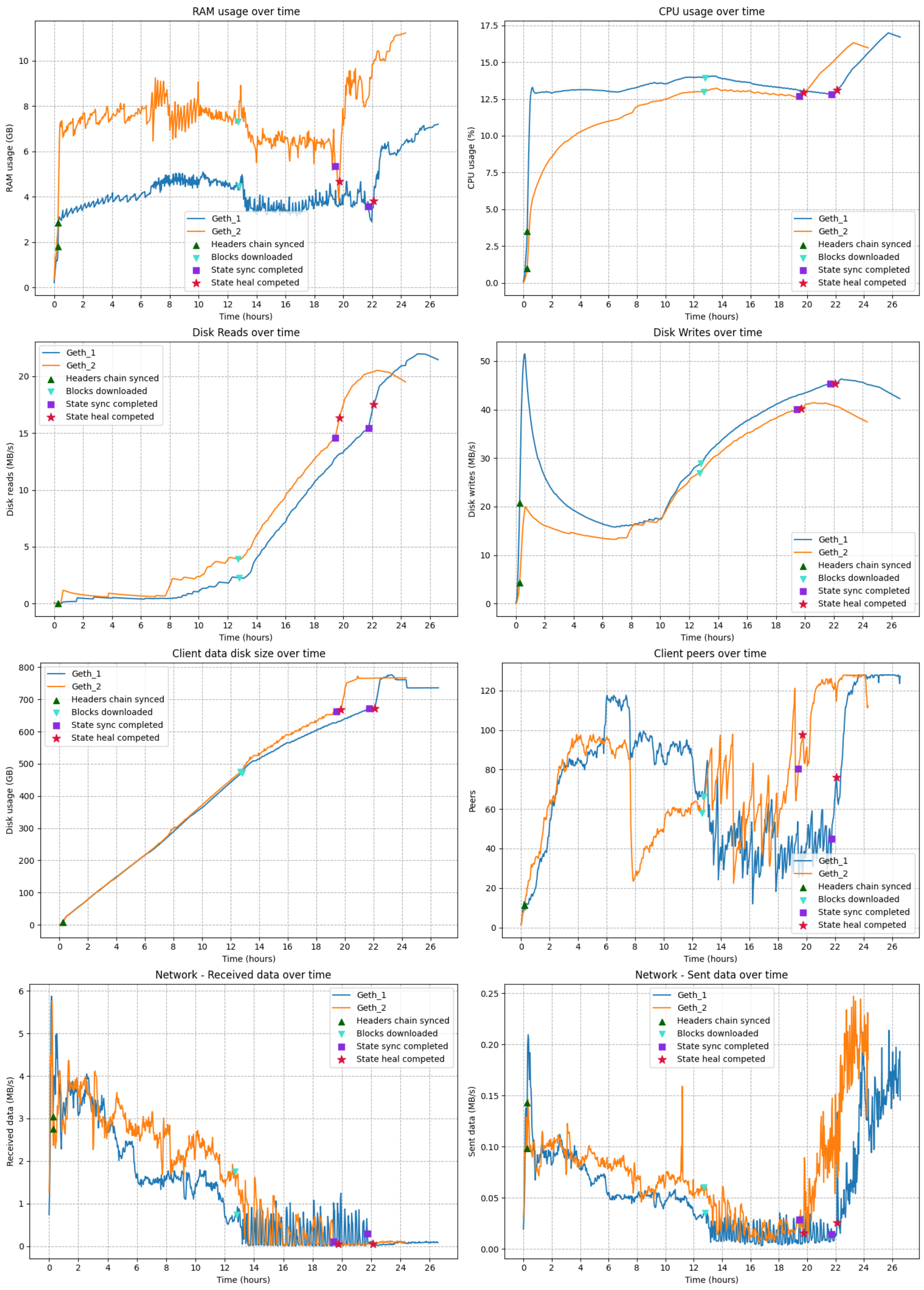


Figure 6.2. *Geth benchmark graphs*

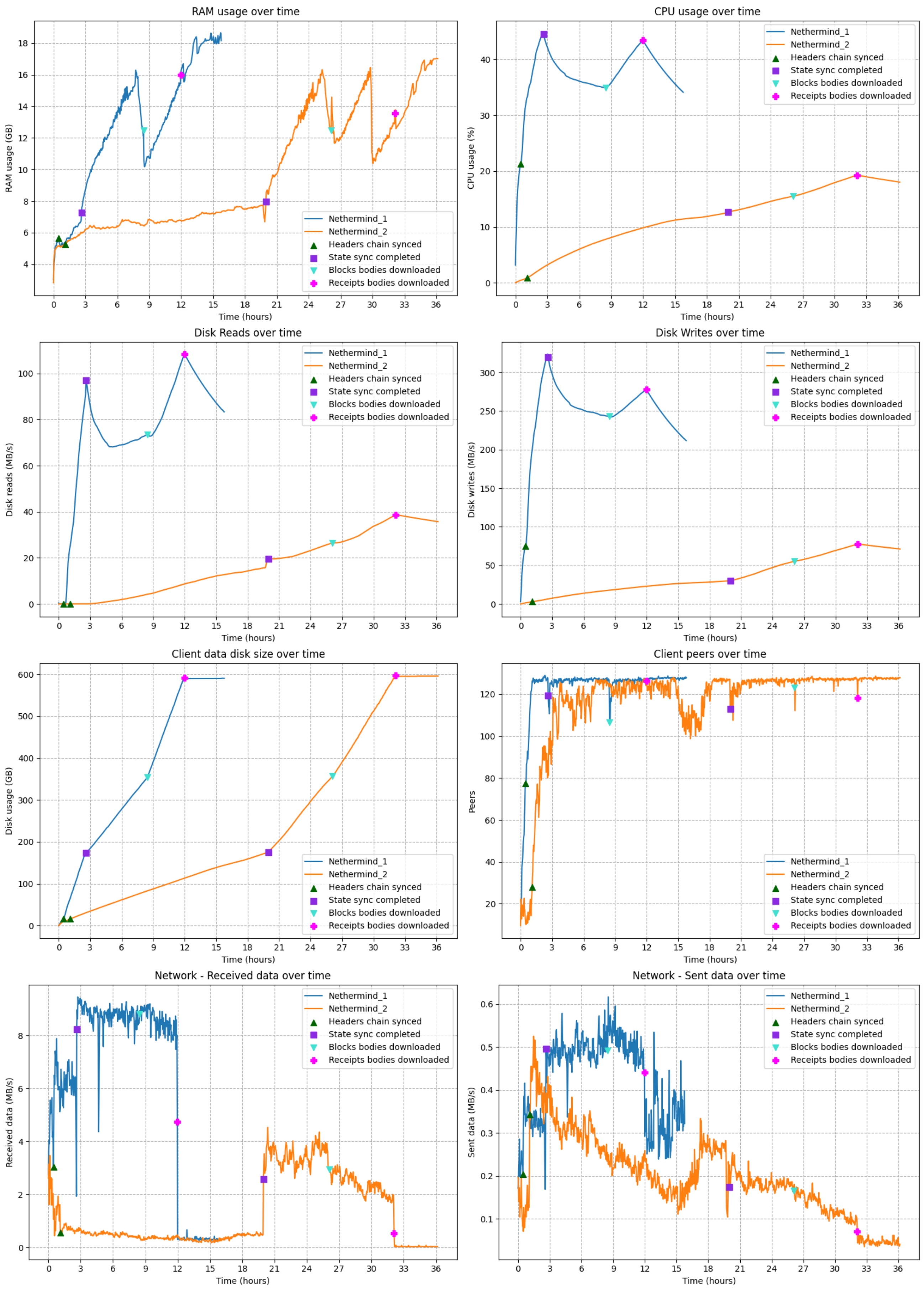


Figure 6.3. *Nethermind benchmark graphs*

for some purposes (and would be able to respond to queries regarding the network's current state), a full Ethereum node should locally preserve the blocks' data as well. Also in contrast to Geth, these begin downloading only following the state sync's completion. Once these are too downloaded (at the 8h29m mark for `Nethermind_1`), Nethermind finally commences the download of the receipts' bodies.

As `Nethermind_2` will be the only of our execution runs using fast sync, it merits some discussion here. Following the block headers' sync, we can notice a drop in the incoming network traffic on the corresponding plots in Figure 6.3. The reason for that is that, contrary to snap sync's snapshots, here separate requests are being constantly sent out for different state nodes and this procedure introduces a lot of idle time due to network latency — as a response is required before a new request can be sent. These delays are additionally evident from the comparatively much slower disk size growth rate until the state is synced, at which point an increase can be seen across our metrics as the block download phase begins. It is worth mentioning that, regardless of sync mode, both of our Nethermind runs quickly try to reach and strive to maintain their `max-peers` value.

6.2.3 Besu

Besu **v22.10.3** was used. In all the configurations, aside from the mentioned adjustments for sync mode, storage format and peers, the rest of the parameters were left at defaults.

Table 6.3. *Besu benchmark configurations and results*

Label	Sync mode	Storage format	Sync Duration
`Besu_1`	Snap	Bonsai	1d 04h 51m
`Besu_2`	Checkpoint	Bonsai	17h 30m

In Besu, as in Geth, the state's download is performed in parallel with the blocks' downloading. Contrary to Geth however, here the state download is completed first (at 10h23m and 10h42m for `Besu_1` and `Besu_2` respectively) with the blocks' download finishing much later. Moreover, state heal in both our Besu runs is logged as to have been completed mere seconds after the blocks' download, evidently waiting for the latter to complete before designated the client as synced.

Lastly, using checkpoint sync mode unsurprisingly leads to lower total disk sizes and better overall performance. A Besu client would likely prefer it over snap sync unless they have a need for information related to earlier blocks. Moreover, checkpoint sync exhibits a preference to connect to significantly less peers than it is able to through its parameters — with the exception of a peculiar spike close to 70, it mostly preserved a connection with only around 30 peers, despite having a `max-peers` value of 128.

6.2.4 Erigon

Erigon **v2.36.1** was used. A single Erigon configuration was run, with all the default options but also the inclusion of pruning due to storage constraints. It is noted that

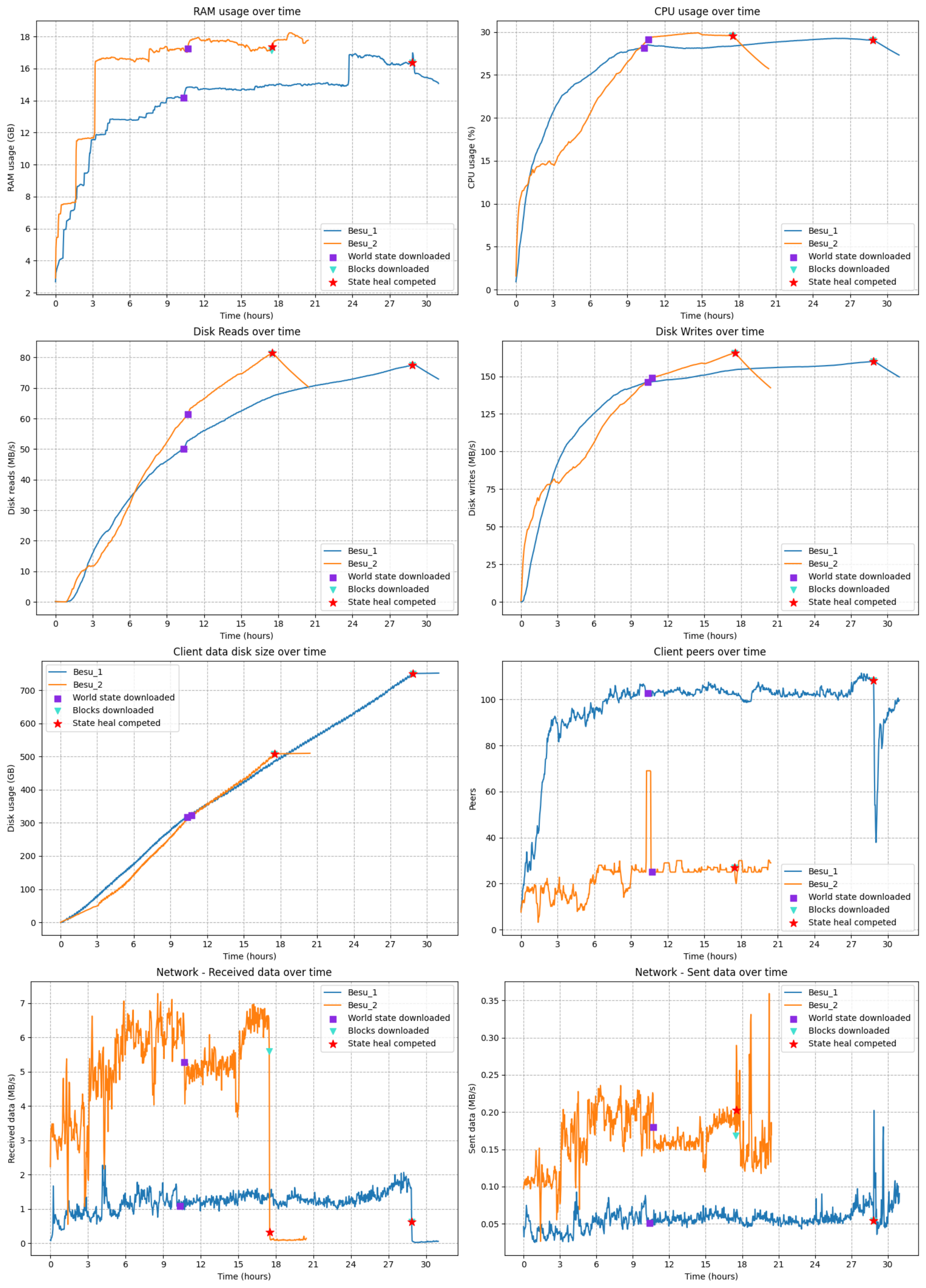


Figure 6.4. *Besu benchmark graphs*

without any pruning this same sync mode would have resulted in an archive node.

Table 6.4. *Erigon benchmark results*

Label	Sync mode	Sync Duration
`Erigon`	Staged	4d 12h 13m

The 10 high level sync stages that we went over in Section 5.2.5 are further split for a total of 15 in the execution logs[4]. We will not be detailing them all here as most of them were completed in a matter of seconds or few minutes. As such and for visibility purposes, on Figure 6.5 we will only be marking the completion of the stages that lasted over an hour.

This run is the only one in our client benchmarks that performed (a version of) full sync and that is immediately evident from the sync time required. A sync time of four and a half days is much better than other clients could hope to achieve using their full sync modes, but it is admittedly multiple times worse when compared to their snap sync times.

This is of course due to the transaction execution stage, which in itself took up the vast majority of the sync time (3d18h in particular). This stage becomes progressively slower as the local state grows in the disk but also as the blocks themselves increase in size as we approach the present day, which was evident by the execution logs that periodically logged the rates of blocks and transactions processed per second. In the beginning (close to genesis) Erigon was processing hundreds of blocks/second and tens of thousands of transactions/second, while by block 10M it had dropped to about 60 blocks/second and 7000 transactions/second, before converging to about 20 blocks/second and 2500 transactions/second when its sync was completed shortly after block 16M.

An interesting capability that the non-parallel nature of Erigon's staged sync enables is the adjustment of its resource consumption based on the stage it is currently executing. For instance, we can see that the network traffic (most notably, data received) was minimal throughout the execution stage, which is easily explainable as at this stage Erigon does not need to request data from its peers.

6.2.5 Inter-client comparisons

For our inter-client comparisons we used the best configuration for each of Geth, Nethermind and Besu (`Geth_2`, `Nethermind_1` and `Besu_2` respectively), based on the above results.

Erigon is not directly comparable to the other 3 clients since its staged sync re-executes all transactions from genesis, a process that is circumvented when using snap sync and which in itself takes significantly longer than the entire snap sync of the other 3 clients. As it would simply skew the axes of our graphs and render visual comparisons less distinct, it was thus excluded from this comparative analysis.

[4]These sub-stages are occasionally adjusted by Erigon's developers; for instance, an earlier version of Erigon with which we experimented included 17 stages

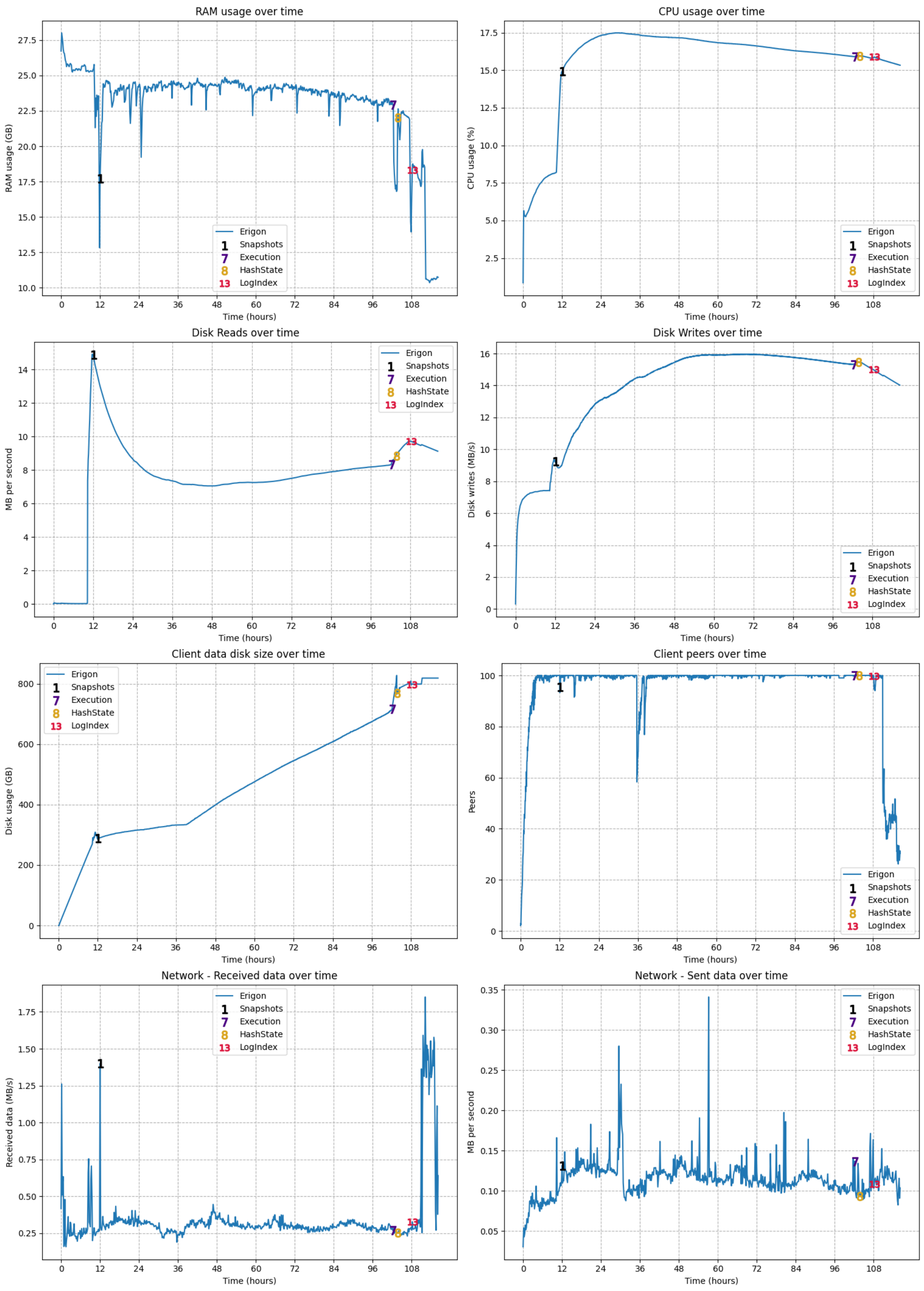


Figure 6.5. *Erigon benchmark graphs*

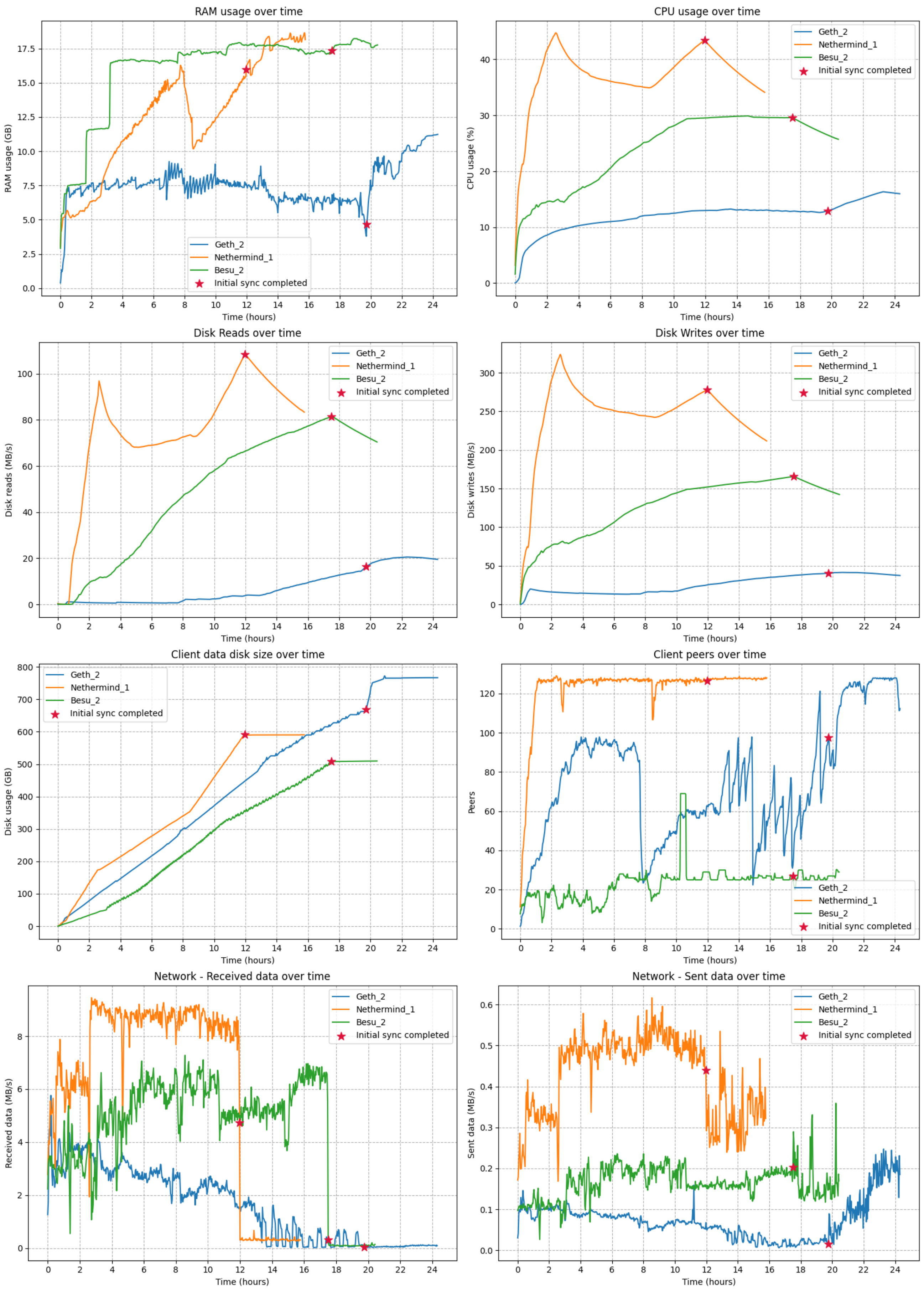


Figure 6.6. *Inter-client comparative benchmark results*

At first glance, we notice that Geth struggles to reach the disk read and write speeds exhibited by the other 2 clients. While this could be seen as proof of RocksDB superiority over LevelDB, the various other optimizations that Nethermind and Besu have implemented as well as their architectural design differences ought to be taken into account.

We treated Nethermind's initial sync completion as coinciding with the completion of its last marked event (receipt bodies downloaded) although, as mentioned when describing its results above, depending on the use case one could consider Nethermind to have already synced quite earlier than that. Even so, `Nethermind_1` outperformed both Geth with max cache options and Besu using checkpoint sync and Bonsai tries. It appears that being able to perform large uninterrupted sequential writes during state download in traditionally disk-bound applications (as Ethereum execution clients are) unsurprisingly provides an important edge when it comes to sync times. Nonetheless, on a weaker computer with a less performing SSD or on a worse network connection it is not unlikely that Nethermind's performance would most closely resemble that of Geth and Besu.

6.3 Results assessment

There are a few overall conclusions to which we can arrive from our benchmark results. First of all, the comparison between snap and fast sync in Nethermind's configurations (Figure 6.3) categorically shows how fast sync is outclassed both it terms of speed but also on how it can make use of the available system resources. It thus comes as no surprise that snap sync has currently become the default mode of syncing among execution clients, with fast sync being phased out.

Peer number fluctuations are to be expected in any P2P network. While a low number of them would surely negatively impact total sync time, any spikes that we observed in our experiments did nonetheless not meaningfully affect performance across our metrics. We also make note of the fact that during its initial sync a client will often not seek to reach its maximum allowed number of peers with which it was parameterized but it may prefer to instead preserve a lower peer count, a behaviour which was in fact exhibited in several of our client configurations.

Furthermore, granting more RAM to a client for caching purposes expectedly contributed substantially to decrease sync times and decrease the amount of writes (and, to a lesser extent, reads) to the disk, as is evident in Geth's intra-client comparison results (Figure 6.2). On the other hand, across our metrics we can see that our CPU was not maximally utilized, largely due to the fact that the sync process is inherently not suited for parallelization. While sync phases can be executed in parallel as is done in Geth and Besu, a single phase cannot meaningfully split its workload across different threads — for instance, blocks' download cannot be parallelized because each block needs to be individually validated before processing its child block.

Finally, it bears repeating that the above results are indicative of execution client runs on a particular computer but it is certainly plausible that different configurations either in terms of hardware or parameters when running each client could yield more favourable results for some of them.

Chapter 7

Conclusion

In this thesis, we explored the current state of Ethereum execution clients, the data structures that they use but also the various innovative modes they have employed to more efficiently synchronize the network. In our penultimate chapter we experimentally compared the most widely used of them in several configurations, resulting in a comprehensive comparative analysis of the different execution client implementations in terms of resources used and sync times required.

Other metrics that were mostly left out of our analysis but could provide a further insight depending on one's use case would be block and transaction processing speeds (throughput). These could, for instance, help deduce whether a client implementation takes an approximately constant amount of time to process each block or if that time varies depending on the blocks' contents (such as the gas used or the types of its included transactions) and could be grounds for further benchmarks.

It should be emphasized that preserving a healthy Ethereum client is not only important for the security of the network, but ever since staking was introduced it also has financial benefits to its operator. Benchmarks such as ours can be a first step towards initially choosing a client, but a continued resource monitoring during a client's normal operation (even after its bootstrapping) will always be of paramount importance.

Before drawing to a close, we can briefly go over what is already planned for Ethereum's future and how it relates to our study as well as explore some related work on proposed improvements and alternative client implementations.

7.1 The Future of Ethereum

The Ethereum ecosystem is constantly evolving at a rapid rate. The Merge was a long-awaited upgrade which brought with it several architectural changes, including to the subject of our thesis. Following the Merge, a number of further major network upgrades have already been planned — and have even been given their own rhyming names. In brief:

- *The Surge*: Introduction of *shard chains* to the network (63 in number, for a 64 total along with the main chain) which are intended to massively increase the scalability of the network. These will pave the way for significantly lower gas fees and enable the network to handle thousands of transactions per second. Most relevant to our focus here, shard chains will mean that a node will no longer need to locally store the entire blockchain's data, drastically lowering their workload and reducing their hardware requirements.

- *The Verge*: Incorporation of Verkle trees (as briefly described in Section 3.1.7) to the network, with the goal of replacing MPTs.
- *The Purge*: Elimination of certain historical data and technical debt. This includes the introduction of native *history expiry* and *state expiry* concepts through which a client will be able to safely ignore data past a certain date, further decreasing its load.
- *The Splurge*: Various miscellaneous changes, including EVM improvements and the implementation of *proposer-builder separation* (PBS) [46].

These are currently intended to happen in the above order, though exact dates have not been given at the time of writing. Regardless, it is undeniable that Ethereum is constantly striving to improve itself and, in all likelihood, before these upgrades come to pass some yet unpublished piece of research will enrich them in some advantageous way.

7.2 Related work on potential improvements

Undoubtedly, apart from what is already planned for upgrading the Ethereum network, related research for improvements in all its aspects continues without stopping. In terms of execution client sync implementations, this research often focuses on alternative approaches to their bootstrapping. Admittedly, many of them were conceived before the release of snap sync which has since rendered them outdated (e.g. a *turbo sync* mode, improving on Parity's *warp sync* [47]) .

An intriguing approach to efficient client bootstrapping was *Ethanos* [48]. Its researchers observed that only a fraction of the total Ethereum accounts are "active" (about 5% of accounts sent or received a transaction in the period of a month). Using that observation, they devised a syncing algorithm based around periodic *epochs*[1], at the start of which the client would sweep inactive accounts, recreating the state trie only containing the active ones. This approach necessarily adds some complexity, for instance when an account becomes active again and its state needs to be manually retrieved, but nonetheless results in MPTs of significantly smaller sizes. While the sync time gains that it originally exhibited when compared to Geth's fast sync are now too eclipsed by those of snap sync, it remains a practical concept that merits further future consideration.

A separate but necessary approach to bootstrapping improvements would be on the storage engines used. No matter the techniques employed by the clients to sync the network, a bottleneck will sooner or later be found on the underlying storage unless that is likewise optimized. A plethora of LSM-based key-value stores exist that improve upon the popular LevelDB and RocksDB ones [49] [50], but the most compelling one is probably *mLSM trees* [51].

[1]unrelated to PoS epochs

The cost of disk reads and writes in client implementations using multi-level LSM-based storage engines can be traced back to the additional read and writes that these engines need to perform internally on each such request (see Section 3.3). Merkelized Log Structured Merge trees (mLSM trees) focus specifically on making authenticated storage faster on Ethereum, by redesigning the data structure that a client's storage engine uses so as to avoid this amplification of reads and writes, through the incorporation of Merkle proof caching. While caching does seems like an obvious solution to many of our issues — and it indeed would be, should a client's workload solely involve reads — its problem lies in that a single write to any MPT leaf updates several intermediary nodes, including the root, meaning that cached Merkle proofs would be invalidated on every write. The solution that mLSM propose is replacing LevelDB's immutable SSTables on disk with mutable MPTs (keeping the multiple level implementation), while decoupling lookup and authentication. Despite only being tested on a small subset of Ethereum's total blocks as of their paper's publication, mLSM trees recorded a noteworthy decrease in redundant disk reads and writes compared to LevelDB on Geth and are certainly a promising development.

Lastly, in the blockchain landscape where thousands of developers are unceasingly working to deliver high quality services, some significant innovation may naturally first arise in some other blockchain. Ethereum can take inspiration from similar account-based cryptocurrencies and adapt any applicable and useful features of theirs into its own architecture. Some interesting relevant work — not necessarily suitable for direct application in Ethereum, but still useful for providing insights — include Cardano's *Mithril* [52] and Algorand's *Vault* [53], both of which are proposals for their respective blockchains that enable scalability through faster node bootstrapping.

Bibliography

[1] V. Buterin, “Ethereum white paper: A next generation smart contract & decentralized application platform,” 2013.

[2] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” 2014.

[3] “The history of Ethereum.” https://ethereum.org/en/history/. Date accessed: 16-09-2022.

[4] V. Buterin, E. Conner, *et al.*, “EIP-1559: Fee market change for eth 1.0 chain.” https://eips.ethereum.org/EIPS/eip-2718. Date accessed: 16-09-2022.

[5] “The idea of smart contracts.” https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html, 1997. Date accessed: 06-07-2022.

[6] ISDA, “Smart contracts and distributed ledger – a legal perspective,” aug 2017.

[7] L. Breidenbach, C. Christian, *et al.*, “Chainlink 2.0: Next steps in the evolution of decentralized oracle networks,” apr 2021.

[8] M. Zoltu, “EIP-2718: Typed transaction envelope.” https://eips.ethereum.org/EIPS/eip-1559. Date accessed: 16-09-2022.

[9] V. Buterin and M. Swende, “EIP-2930: Optional access lists.” https://eips.ethereum.org/EIPS/eip-2930. Date accessed: 28-09-2022.

[10] C. Dwork and M. Naor, “Pricing via processing or combatting junk mail,” in *Advances in Cryptology — CRYPTO’ 92* (E. F. Brickell, ed.), (Berlin, Heidelberg), pp. 139–147, Springer Berlin Heidelberg, 1993.

[11] M. Jakobsson and A. Juels, *Proofs of Work and Bread Pudding Protocols(Extended Abstract)*, pp. 258–272. Boston, MA: Springer US, 1999.

[12] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” May 2009.

[13] S. N. Sunny King, “PPCoin: Peer-to-peer crypto-currency with proof-of-stake,” 2012.

[14] G. A. F. Rebello, G. F. Camilo, L. C. B. Guimarães, L. A. C. de Souza, and O. C. M. B. Duarte, “On the security and performance of proof-based consensus protocols,” in *2020 4th Conference on Cloud and Internet of Things (CIoT)*, pp. 67–74, 2020.

[15] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017.

[16] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and Casper," *CoRR*, vol. abs/2003.03052, 2020.

[17] "How the merge impacts eth supply." https://ethereum.org/en/upgrades/merge/issuance/. Date accessed: 16-09-2022.

[18] D. R. Morrison, "PATRICIA – practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, pp. 514—-534, oct 1968.

[19] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87* (C. Pomerance, ed.), (Berlin, Heidelberg), pp. 369–378, Springer Berlin Heidelberg, 1988.

[20] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), p. 411–423, Association for Computing Machinery, 2014.

[21] "Merkling in Ethereum." https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/, nov 2015. Date accessed: 02-08-2022.

[22] J. Kuszmaul, "Verkle trees,"

[23] D. Catalano and D. Fiore, "Vector commitments and their applications," in *Public-Key Cryptography - PKC 2013* (K. Kurosawa and G. Hanaoka, eds.), (Berlin, Heidelberg), pp. 55–72, Springer Berlin Heidelberg, 2013.

[24] H. Chen and D. Liang, "Adaptive spatio-temporal query strategies in blockchain," *ISPRS International Journal of Geo-Information*, vol. 11, p. 409, 07 2022.

[25] V. Buterin, "Verkle draft EIP." https://notes.ethereum.org/@vbuterin/verkle_tree_eip. Date accessed: 22-08-2022.

[26] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, p. 422–426, jul 1970.

[27] K. Aggarwal and H. K. Verma, "Hash_RC6 — variable length hash algorithm using RC6," in *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 450–456, 2015.

[28] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Advances in Cryptology - EUROCRYPT 2013* (T. Johansson and P. Q. Nguyen, eds.), (Berlin, Heidelberg), pp. 313–314, Springer Berlin Heidelberg, 2013.

[29] I. Dinur, O. Dunkelman, and A. Shamir, "New attacks on Keccak-224 and Keccak-256," in *Fast Software Encryption* (A. Canteaut, ed.), (Berlin, Heidelberg), pp. 442–461, Springer Berlin Heidelberg, 2012.

[30] G. H. A. Jesse Hines, Nicholas Cunningham, "Performance comparison of operations in the file system and in embedded key-value databases,"

[31] A. Asgaonkar, "Weak subjectivity in Eth2.0." https://notes.ethereum.org/@adiasg/weak-subjectvity-eth2. Date accessed: 26-12-2022.

[32] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis, "A survey on long-range attacks for proof of stake protocols," *IEEE Access*, vol. 7, pp. 28712–28725, 2019.

[33] S. Azouvi and M. Vukolić, "Pikachu: Securing PoS blockchains from long-range attacks by checkpointing into Bitcoin pow using Taproot," in *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, ConsensusDay '22, (New York, NY, USA), p. 53–65, Association for Computing Machinery, 2022.

[34] E. N. Tas, D. Tse, F. Yu, and S. Kannan, "Babylon: Reusing Bitcoin mining to enhance Proof-of-Stake security," *CoRR*, vol. abs/2201.07946, 2022.

[35] V. Buterin, "Proof of Stake: How i learned to love weak subjectivity." https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity. Date accessed: 26-12-2022.

[36] P. Szilágyi, "Ask about Geth: Snapshot acceleration." https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration. Date accessed: 27-01-2023.

[37] P. Szilágyi, "Geth v1.10.0." https://blog.ethereum.org/2021/03/03/geth-v1-10-0#snap-sync. Date accessed: 27-01-2023.

[38] A. Sharp, "Erigon stage sync and control flows." https://erigon.substack.com/p/erigon-stage-sync-and-control-flows. Date accessed: 30-01-2023.

[39] V. Buterin, "Light clients and Proof of Stake." https://blog.ethereum.org/2015/01/10/light-clients-proof-stake. Date accessed: 28-12-2022.

[40] I. Darwish, "Nethermind's full pruning is here — cutting the gordian knot." https://medium.com/nethermind-eth/netherminds-full-pruning-is-here-cutting-the-gordian-knot-5e3450f02de9. Date accessed: 27-01-2023.

[41] P. Technologies, "Transitioning Parity Ethereum to OpenEthereum DAO." https://www.parity.io/blog/parity-ethereum-openethereum-dao/. Date accessed: 04-02-2023.

[42] Gnosis, "Gnosis client development team joins Erigon (formerly Turbo-Geth) to release next-gen Ethereum client." https://medium.com/openethereum/gnosis-joins-erigon-formerly-turbo-geth-to-release-next-gen-ethereum-client-c6708dd06dd. Date accessed: 04-02-2023.

[43] A. Sharp, "Winding down support for Akula project." https://erigon.substack.com/p/winding-down-support-for-akula-project. Date accessed: 04-02-2023.

[44] G. Konstantopoulos, "Introducing Reth." https://www.paradigm.xyz/2022/12/reth. Date accessed: 13-01-2023.

[45] M. Cortes-Goicoechea, L. Franceschini, and L. Bautista-Gomez, "Resource analysis of Ethereum 2.0 clients," in *2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, pp. 1–8, 2021.

[46] V. Buterin, "State of research: increasing censorship resistance of transactions under proposer/builder separation (PBS)." https://notes.ethereum.org/@vbuterin/pbs_censorship_resistance. Date accessed: 02-04-2023.

[47] X. Qian, "Improved authenticated data structures for blockchain synchronization," Master's thesis, University of Illinois at Urbana-Champaign, 2018.

[48] J.-Y. Kim, J. Lee, Y. Koo, S. Park, and S.-M. Moon, "Ethanos: Efficient bootstrapping for full nodes on account-based blockchain," in *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, (New York, NY, USA), p. 99–113, Association for Computing Machinery, 2021.

[49] P. Ktistakis, "Scaling the RocksDB key-value store via data distribution on multiple nodes," Master's thesis, National Technical University of Athens, School of Electrical and Computer Engineering, 2018.

[50] H. Huang and S. Ghandeharizadeh, "Nova-LSM: A distributed, component-based LSM-tree key-value store," in *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, (New York, NY, USA), p. 749–763, Association for Computing Machinery, 2021.

[51] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham, "mLSM: Making authenticated storage faster in Ethereum," HotStorage'18, (USA), p. 10, USENIX Association, 2018.

[52] P. Chaidos and A. Kiayias, "Mithril: Stake-based threshold multisignatures,"

[53] D. Leung, A. Suhl, Y. Gilad, and N. Zeldovich, "Vault: Fast bootstrapping for the Algorand cryptocurrency," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
ADS	Authenticated Data Structure
API	Application Programming Interface
BTC	Bitcoin
CHF	Cryptographic Hash Function
CLI	Command-Line Interface
DAO	Decentralized Autonomous Organization
DApp	Decentralized Application
DeFi	Decentralized Finance
ECDSA	Elliptic Curve Digital Signature Algorithm
EOA	Externally Owned Account
EIP	Ethereum Improvement Proposal
ERC	Ethereum Request for Comments
ETH	Ether
EVM	Ethereum Virtual Machine
JSON	JavaScript Object Notation
JWT	JSON Web Token
LSM	Log-Structured Merge
LIFO	Last-In First-Out
MEV	Maximal Extractable Value
MPT	Merkle Patricia Trie
NFT	Non-Fungible Token
P2P	Peer-to-Peer
PoA	Proof-of-Authority
PoS	Proof-of-Stake
PoW	Proof-of-Work
RPC	Remote Procedure Call
RLP	Recursive Length Prefix
SHA	Secure Hash Algorithm
SSZ	Simple Serialize
ULC	Ultra Light Client
UTXO	Unspent Transaction Output