Onchain Social Graph Storage with Concurrent Merkle Trees: An Integrated Approach

Nicholas Oxford[*]
n@primitives.com

David Gabeau[†]
d@primitives.xyz

Nehemiah Blackburn[‡]
nemo@primitives.xyz

Abstract

The Primitives Protocol proposes a method to utilize advancements in storage efficiency to dramatically lower the storage costs associated with fully onchain digital identity and social graphs[1]. The protocol aims to improve the current state of decentralized social graphs in which storing the graph onchain is cost-prohibitive, non-standardized, and difficult to scale in a decentralized manner. By applying Merkle tree technology—known for its role in optimizing storage through data verification and integrity checks—to the storage of social graphs, we can achieve near-zero marginal storage cost for each additional identity or connection added to the network[2] while keeping the entire social graph onchain. The integration of Merkle tree technology and compressed NFT technology introduces a new primitive to the ecosystem: Non-Fungible Graphs, completing the abstraction of our analog world onchain (e.g., wallets, NFTs, social connections).

This paper explores integrating Compressed NFT technology through Remote Procedure Calls (RPCs), which monitor blockchain activities and index blockchain data within Merkle Trees. This mechanism not only ensures the efficient and secure management of decentralized identities but also enables full composability within and across decentralized applications. The protocol supports an efficient architecture where identities and connections are broadcast and managed on a peer-to-peer basis. This ensures that the network remains robust and flexible, with the ability to adapt as participants join or leave the ecosystem. This novel approach to digital identity and social graph management heralds a new era of decentralized application development, in which the cold-start problem is significantly mitigated, and user engagement is enhanced through streamlined onboarding and social connectivity.

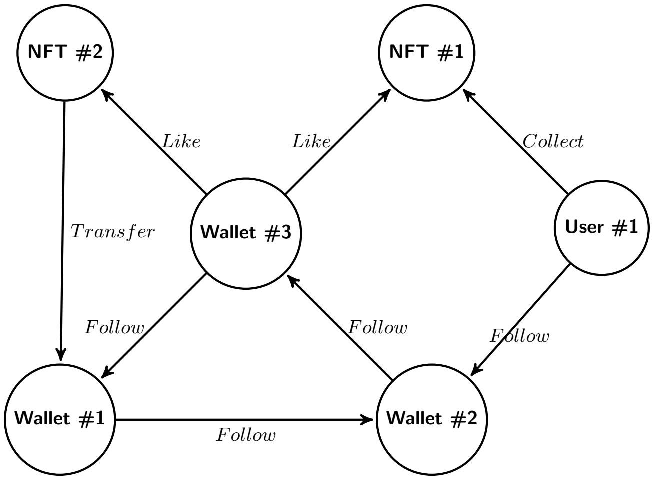
[*] Engineer at Primitives.xyz
[†] Founder at Primitives.xyz
[‡] Product Manager at Primitives.xyz

Introduction

The introduction of blockchain technology has significantly changed how digital assets are stored (i.e., money in the form of cryptocurrency). Building on this transformation, new breakthroughs in the Solana ecosystem have enabled the storage of graph data on the blockchain at scale, prompting a reevaluation of how we store, manage, and leverage social connections. As we navigate this transformation, the role of a social graph—a graphical representation of interpersonal relationships—has significant implications for the way we understand and interact with blockchain entities, such as wallets and non-fungible tokens (NFTs). However, the contemporary state of crypto social graphs face two primary challenges: decentralization and scalability. Traditional methods of representing these graphs are either overly reliant on off-chain abstractions or are constrained by the limitations of onchain storage solutions.

In response to these challenges, we propose the Primitives Protocol, a blockchain-first approach to constructing a social graph that is inherently decentralized, scalable, and composable. By utilizing graph database principles and integrating these with the unique properties of blockchain technology — decentralization, immutability, and permissionlessness — the Primitives Protocol enables a flexible representation of relationships between nodes in the crypto ecosystem. This paper outlines the theoretical foundation, implementation strategies, and potential impacts of such an approach, highlighting the advantages of using Merkle Trees for efficient onchain storage and the protocol's compatibility with existing blockchain infrastructures like Solana.

Through this exploration, we aim to demonstrate that the Primitives Protocol not only addresses the pressing issues of decentralization and scalability in crypto social graphs but also offers a robust framework for the future development and integration of decentralized social networks.



A Blockchain First Social Graph

A modern graph database consists of a set of nodes and edges, in which each individual node and edge possesses a map of properties. A vertex, representing something like a wallet address or token collection, is the basic object of a graph. These objects can exist independently of everything else in the graph. An edge creates a directed connection between two nodes.

For web3, a social graph can represent the relationships between wallets, NFTs, and other assets. However, not everyone will agree on how and even if nodes in a social graph are connected. Because of this, composability is paramount for the design of a crypto social graph. Other parties must be able to write their own nodes and edges to the graph. Or even, one could prescribe to a set of nodes but have a totally different set of edges. This property enables the crypto social graph to adhere to the blockchain principle of permissionlessness.

Current crypto social graphs are designed in two ways, both of which have deficiencies. In the first method, the graph is constructed via off-chain abstractions built on top of minted NFTs. This method falls short because it relies on centralization to achieve its aims. In the second, social data is written to a Layer 2, and transactions rolled up the main chain. This method falls short because the fragmented nature of Layer 2s makes it difficult to decentralize and scale the social graph. Each company, then, has their own protocol or is facing the realities of storing a sizeable social graph onchain. Storing a change log of a graph database in the nodes of a Merkle tree solves this issue.

Compressing a Social Graph

The goal of the decentralized social graph is for the ability of any party to rebuild a graph database from scratch. This also means anyone maintaining an index of a crypto graph database will have to continually listen to appends to the Merkle tree and ingest the changes.

Table 1: Nodes Hash Seed

Seed	Type	Size (B)	Description
Label	String	8	Label of the node
Properties	String	256	Properties of node - *key:value*

Table 2: Edge Hash Seed

Seed	Type	Size (B)	Description
Start ID	PublicKey	32	Address or Id of starting node
End ID	PublicKey	32	Address or Id of ending node
Properties	String	256	Properties of node - *key:value*

Unlike traditional onchain storage, in this architecture the hashes of the vertex and edge data are stored. The goal of these hashes is to be able to create commands to send to a graph database. The mass adoption of compressed assets suggests that companies and developers are okay with the compromise of only storing the hash onchain, thus relying on a third party (RPC) to effectively store and serve the data. For example, when interacting with compressed NFTs (cNFTs), you call the cNFT by its AssetID. As Solana points out in their documentation, this AssetID value differs slightly from traditional NFTs and compressed NFTs:

> For traditional/uncompressed NFTs: this is the token's address for the actual Account onchain that stores the metadata for the asset. [1]
>
> For compressed NFTs: this is the id of the compressed NFT within the tree and is NOT an actual onchain Account address. While a compressed NFT's assetId resembles a traditional Solana Account address, it is not. [1]

The Primitives Protocol will follow a similar pattern, where one can retrieve information about any node or edge by calling its AssetID. This follows the spirit of the white paper that proposed storing digital assets in Merkle trees. The authors wrote:

> The natural solution is to store a compressed fingerprint of the digital asset data on the blockchain while maintaining the actual data with traditional database solutions. The off-chain data cannot be maliciously tampered with as it would be easy to verify that the onchain fingerprint is mismatched. Additionally, it should always be possible to reconstruct the full uncompressed state of the world by processing the ledger sequentially. [2]

Cost Comparisons

Not only does the Primitives Protocol integrate well with existing Solana infrastructure and ideals, but it is also extremely cost-competitive.

Table 3: Cost of Storing a Social Graph on Solana (Compressed and Uncompressed) and Ethereum

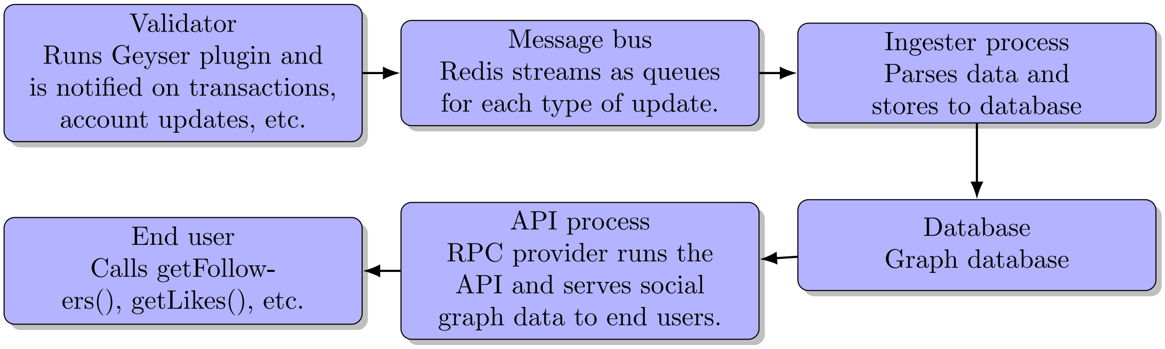
# of Nodes	# of Edges	Compressed Cost	Solana NFT Cost	ETH Cost
10,000	250,000	$887	$727,864	$1,300,000-13,000,000
100,000	2,500,000	$894	$7,278,648	$13,000,000-130,000,000
1,000,000	25,000,000	$1214	$72,786,480	$130,000,000-1,300,000,000
10,000,000	250,000,000	$1267	$727,864,800.	$1,300,000,000-13,000,000,000
250,000,000	750,000,000	$1267	$2,799,480,000	$50,000,000,000-500,000,000,000

Integration with existing Solana Infrastructure

It is one thing to have a theoretically perfect design for a social graph; it is another to have mass adoption of this protocol. When creating the Primitives Protocol, a survey of the current Solana ecosystem was conducted. It was found that Metaplex already standardized how one would listen to the blockchain and index the data of the Merkle tree. Instead of reinventing the wheel, the Primitives Protocol builds on these existing decisions to give an easy path for other RPC providers to save and store onchain social graphs.

Before we expect others in the ecosystem to begin serving the Primitives Protocol, Primitives.xyz will be the first to serve the social graph data from the Merkle trees. This will be done by executing the following steps:

1. Set up a no-vote validator
2. Create a Geyser plugin to listen and notify social graph related spl-account-compressions transactions occur
3. Write a producer client for the redis cluster used to queue updates
4. Create ingestor adapters for different graph databases
5. Store the data in a graph database
6. Create APIs for developers to interact with the graph database


Validator
Runs Geyser plugin and is notified on transactions, account updates, etc.
Message bus
Redis streams as queues for each type of update.
Ingester process
Parses data and stores to database
Database
Graph database
API process
RPC provider runs the API and serves social graph data to end users.
End user
Calls getFollowers(), getLikes(), etc.

Concurrency

To achieve global scale, the underlying web3 social graph protocol needs to not only provide competitive data storage costs and fast reads - the protocol also needs to enable applications to write nodes and edges concurrently (and without polling) for the latest state of the graph. These fast writes are enabled by two things: Solana's short block times and the ability to do concurrent updates on the Merkle tree. As of February 2024, Solana has an average block time of 400ms and an average block size of around 2MB. Although Ethereum has a similar block size, their average block time is 13-15 seconds. This means that Solana is processing around 3,000 TPS while ETH is processing around 30 TPS. These faster writes give developers an experience more similar to a traditional database.

Furthermore, the ability to do concurrent updates on the Merkle tree means multiple authorities can write nodes and edges to the same block, even if they would theoretically cause a conflict. A real life example of this is that in the same block a wallet follows and unfollows another wallet. The way the trees are stored onchain means that not only is the hash of the root of the tree stored, but also a tree change log. This means if first the follow is written and then the unfollow, the program will fast forward to the unfollow! The number of change logs available is determined by the tree's Max Buffer Size. Lastly, it is important to note that the rightmost proof of the tree is stored onchain. This is what enables developers not to need to fetch the current state of the tree.

Social Authority

Similarly to how traditional and compressed NFTs can have one or many authorities, the Primitives Protocol will enable multiple key pairs to write and/or alter the social graph. When you create the Merkle tree onchain, the wallet or key pair that signed the transaction is the tree creator. By default, this wallet can perform any action on this tree. If one wants to give another wallet the ability to write to the tree, one can delegate the authority to another wallet. This is done by creating a new tree operation, Delegate, and signing it with the tree creator's key pair.

Table 4: An example list of actions and corresponding tree operations for compressed social graphs.

Action	Tree Operation	Tree	Authority
Create Node	Append	Nodes	Social Authority or Delegate
Follow	Append	Edge	Social Authority or Delegate
Like	Append	Edge	Social Authority or Delegate
Collect	Append	Edge	Social Authority or Delegate
Delete	Replace Leaf	Edge	Social Authority

We expect that most people using Primitives Protocol to build and interact with social graphs will not know deep technical details about social authorities. In fact, we designed the Protocol to ensure that those developing on it and using it will not need deep technical blockchain knowledge to build on top of it. We expect these users will be using APIs built on top of the protocol, and the providers will be the ones interacting with the trees. However, as crypto becomes more widespread, perhaps it will be commonplace to sign something with your crypto wallet, and people will expect to have the ability to do so.

Spam Protection

One of the biggest issues facing crypto adoption today is spam in wallets. If you use a wallet to interact with any popular tokens or NFTs, there is an extremely high chance you will get spam trying to get you to click on malicious links. There is so much spam because a fundamental value proposition of the blockchain is permissionless airdrops. Meaning if you try to stop spam via pricing people out via transaction fees, you also push out people who are using the cheap permissionless airdrops for good. That means you cannot and should not censor transactions on the write, you should instead rely on RPC and client-facing applications to filter out the junk.

If you look at the nodes graph, you can conceive it as a trusted network. If Primitives were an RPC, they would not only serve social graph reads and writes but also NFT data. Developers and companies could subscribe to a value-added service ("Primitives Premium"), including an opinionated view of the crypto world. In other words, when asked for NFT data, Primitives can also serve metadata about the trust level of this NFT. Furthermore, when fetching details about many NFTs, the client can pass in filter parameters to only get NFTs trusted by the network. The client could also tell the RPC to subscribe to another social graph when serving NFT data.

Blockchain Agnostic

It is easy to only think of the Primitives Protocol as representing the relationship between Solana wallets and NFTs, but the protocol is flexible enough to store which blockchain an entity is on. Nothing is stopping a developer from writing a Solana wallet node and an Ethereum wallet node and connecting them with an edge. Technically, these graphs are agnostic to whether the data is blockchain-related at all, just as we are witnessing with NFTs, the things people will build with a new technology go beyond the imagination of first iterations of builders. The fact that the Primitives Protocol both has familiar developer experience and is flexible enough to store any social graph data means that it is likely to be the first choice for any developer building a social graph on or even off a blockchain.

Build Applications on Top of Non-Fungible Graphs

Non-Fungible Graphs present the opportunity to build social graphs that the public can verify. While the security, or the promise your data won't be tampered with, is one of the biggest draws to storing info on a blockchain. However, the Primitives Protocol is meant to give developers the ability to build social applications faster than ever before. No longer do blockchain developers have to worry about building abstractions on top of NFT metadata, now they can build their application in a way that accepts already created graphs.

maxDepth, maxBufferSize, and canopyDepth

When you create a Merkle tree onchain, you are deciding the maxDepth of the tree by how many nodes or edges you want to store. This does require the developer minting the tree to decide how big they think the tree could possibly get. Because we are storing a change log in the tree, you could link multiple trees together, introducing complexity. Like developers creating apps on top of compressed NFTs, although not absolutely necessary, it is also recommended to think about potential concurrency. If concurrency is not a concern, the cost of the tree dramatically drops. To calculate the number of nodes or edges a tree can store, you can use the following formula:

$$nodes\_count = 2^{maxDepth} \tag{1}$$

Beyond maxDepth you are also deciding the max buffer size and canopy depth. As we learned in the Concurrency section, the max buffer size determines how many concurrent writes can happen in the same block. This value is important if you think you will build a global network on top of the Primitives Protocol. The canopy depth is the size of cached proofs stored in the account. For example, you need to submit some proofs when you want to write to the tree. A larger canopy depth means you need to include fewer proofs at the time of write. Although is the biggest contributor to the cost of the tree, and you pay it all upfront, a small canopy size means more proofs need to be included in the transaction which limits the amount of data that can be written to the tree.

Simply put, a developer interacting with the protocol directly would need to decide the maxDepth, maxBufferSize, and canopyDepth. The following table is a list of recommended values for these parameters based on the size of the social graph.

Table 5: Recommend values for maxDepth, maxBufferSize, and canopyDepth based on the size of the social graph.

Total # of Nodes/Edges	maxDepth	maxBufferSize	canopyDepth	Cost (SOL)
1 Million	20	64	14	7.61
10 Million	24	64	14	7.66
1 Billion	30	64	14	10.87

Conclusion

Non-fungible graphs are the representation of the relationships between wallets, NFTs, and other assets onchain, which is both novel and necessary. Up until this point, companies have stored crypto social graphs off-chain. This means if that company ceases to exist, the relationship data is lost. These companies need profit motivation to continually store and serve data. The idea that there is a single point of failure is antithetical to the decentralized values in crypto. The importance of the Primitives Protocol is that the relationships represented in these Merkle trees will outlive the lifespan of any single company. The Primitives Protocol turns the vision of decentralized social networks into reality. It allows you to transfer your social connections across different applications and even other blockchains. In this paper we spoke mostly about these graphs representing relationships between wallets and NFTs, but you can write any social graph data to these trees. A family tree is a social graph, a company's org chart is a social graph. It is clear that the relationship between people, physical objects, digital assets, and even organizations defines who we are. It can be considered completing the trifecta of abstracting our analog world into the crypto space.

Appendices

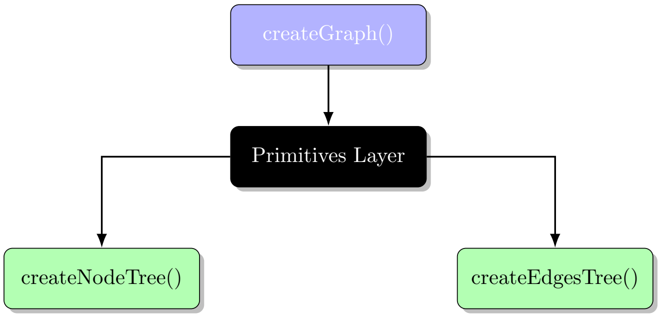
1 Monetizing the Primitives Protocol

The Primitives Protocol can be serviced similarly to other software-as-a-service (SaaS) products. While developers can interact with the blockchain directly, mint their own trees, and append graph data; most developers will find it easier to use an API provider. Just like with traditional SaaS products, different customers have different needs. Most customers probably want to get to writing and reading data as fast as possible, and maybe they don't care if their data is isolated in their own tree. Next, the storage cost gets cheaper the larger the tree is. As of February 2024, a million node tree is $784.21, and a billion node tree is $1120.60. This means it makes sense to store as many nodes as possible in one tree. This opens up the opportunity for Primitives to offer something like a free tier, where you are able to write 10,000 changes to a social graph. This is because we don't pay recurring fees to store the data onchain. The cost, minus the small Solana transaction fee, is when you mint the tree.

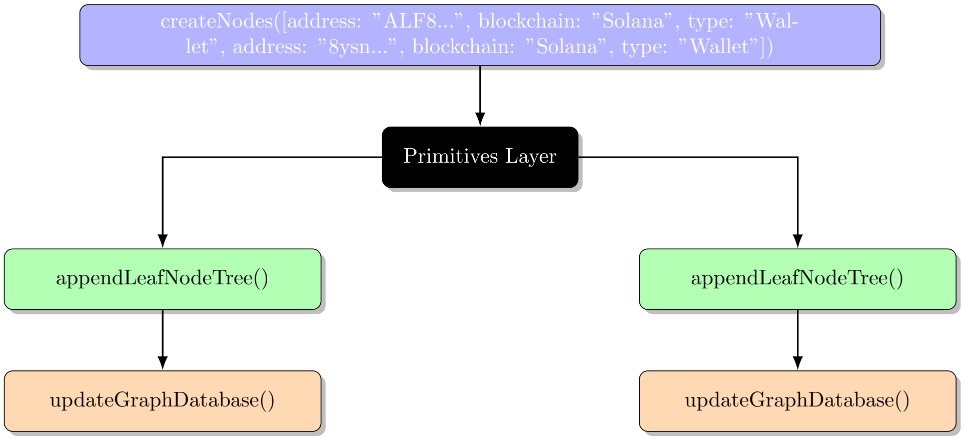
Because we can store any arbitrary key-value property in each node and edge, for customers sharing trees, we would record some identifier for the customer. On the backend, we would most likely set up a small graph database to index the onchain data and serve performantly.

2 Typical API Flow

For a developer beginning to build on top of the Primitives Protocol, they would need to do the following:

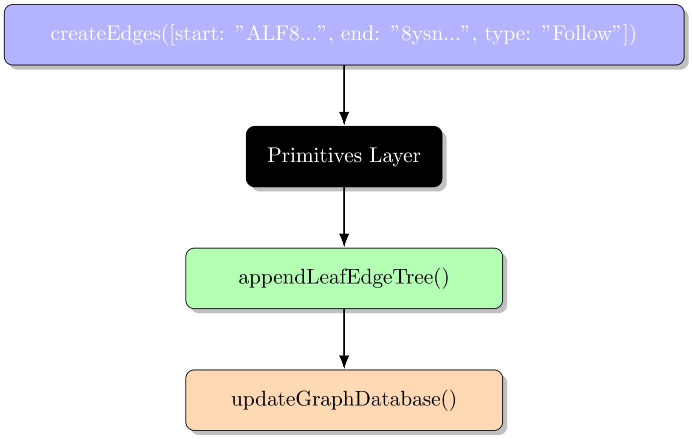


Now that they have a graph database and the tree has been provisioned, they can begin to write nodes and edges.


createNodes([address: "ALF8...", blockchain: "Solana", type: "Wallet", address: "8ysn...", blockchain: "Solana", type: "Wallet"])
Primitives Layer
appendLeafNodeTree()
appendLeafNodeTree()
updateGraphDatabase()
updateGraphDatabase()

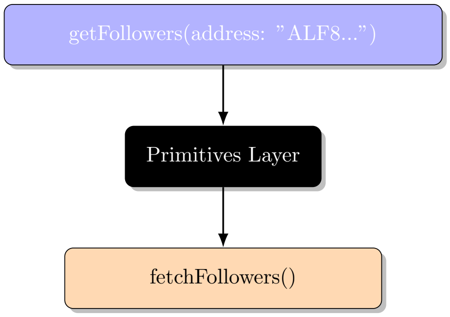
Note: The updateGraphDatabase() happens because we are listening to the blockchain and updating the graph database with the latest changes. A developer not using our API would have to both build out the infrastructure for writing to the blockchain, but also listening and ingesting changes to their own graph database.

If the developer wants to have two nodes connected by an edge, they would do the following:


createEdges([start: ”ALF8...”, end: ”8ysn...”, type: ”Follow”])
Primitives Layer
appendLeafEdgeTree()
updateGraphDatabase()

Even though when you create the nodes, the address of those are nodes correspond to a leaf in a tree. Instead of forcing the user to keep track of leaf ids, they can just pass in the blockchain address of the nodes, and the ingestor will figure it out.

Now that we have created a graph, stored nodes, and connected them. Let's call the Primitives API to get the followers of a wallet.



To grab any data from the graph, we aren't calling the blockchain directly. Instead, we are fetching the info from the provisioned graph database. This is part of what gives the developer a web 2 like experience with the security and portability of the blockchain.

3 Storing Data On Chain Efficiently

Traditional NFT Storage

Compared to Ethereum, it is a fraction of the cost to mint an NFT on Solana. However, if your company or project reaches a significant scale, the cost of minting can become a limiting factor. When you mint an NFT you are both paying the transaction fee to send the mint commands, and also you are "renting" space on the blockchain. For Solana, the data of the NFT is stored in a token account. Furthermore, the cost of minting stays consistent in SOL terms, but if the price of SOL goes up, the cost of minting goes up as well.

Compressed NFTs

To get around the restrictive onchain storage cost, Solana and Metaplex introduced Compressed NFTs.

> Instead of storing an NFT's metadata in a typical Solana account, compressed NFTs store the metadata within the ledger. This allows compressed NFTs to still inherit the security and speed of the Solana blockchain, while at the same time reducing the overall storage costs. [1]

Switching to storing the NFTs metadata in a Merkele tree introduced the dependecy of your Remote Procedure Call (RPC) provider has to set up certain infrastructure to and endpoints to proivde their uses the ability to read compressed NFTs via the address. An advantage of this is you can use the new API endpoint, get asset, to read information for both compressed and uncompressed NFTs.

References

[1] Solana Labs. Compressing nfts with solana, 2024. Accessed: 2024-02-13.

[2] Jarry Xiao, Noah Gundotra, Austin Adams, and Anatoly Yakovenko. Compressing digital assets with concurrent merkle trees, 2023.