All You Need to Know About Compression on Solana

What's this Article About?

Would you believe me if I told you that you could mint a million NFTs right now for less than $150 USD? Preposterous! It would cost upwards of a million dollars to mint that many NFTs depending on the blockchain! Wouldn't it?

State compression is a novel primitive that takes advantage of Merkle trees and Solana's ledger to cut down storage costs dramatically, while inheriting the security and decentralization of Solana's base layer. This article is designed to be a comprehensive deep dive regarding compression on Solana. It covers everything from common misconceptions to transferring compressed NFTs. If you're looking to learn about state compression as well as how to fetch, mint, or transfer compressed NFTs then this is the only

article you'll need to get started.

This article assumes you've already read our article

[Cryptographic Tools 101 - Hash Functions and Merkle Trees Explained

](https://www.helius.dev/blog/cryptographic-tools-101-hash-functions-and-merkle-trees-explained#what%E2%80%99s-a-merkle-tree). It is important to read it before reading this article as we assume knowledge on Merkle trees. This article also expands upon concurrent Merkle trees and goes into more depth on sizing and creating them.

This article uses both the Bubblegum SDK and Umi to demonstrate the various approaches for creating concurrent Merkle trees, as well as minting and transferring compressed NFTs. Familiarity with both tools is valuable as you're likely to encounter each in various codebases. The Bubblegum SDK is included to facilitate learning specifically since the workflow makes the underlying mechanisms more transparent whereas Umi provides a more succinct workflow that streamlines these processes.

Common Misconceptions

We need to clear a few things up before we delve into state compression and the intricacies of compressed NFTs:

Compression on Solana is the same as traditional compression

This is false

. Traditionally, compression is used to reduce the size of files and data. Its primary goal is to store or transmit data in fewer bits than its original file. There are two broad types of compression algorithms:

- Lossless Compression, where the original data can be reconstructed from the compressed data
- Lossy Compression, where "less important" information is removed to reduce file size

A compressed NFT is not an NFT that has gone through some sort of lossless or lossy compression algorithm to make its data smaller. It's not about reducing the quality or dimensions of the art, music, or metadata associated with the NFT either. The concept takes on a different form entirely in the context of Solana. Rather, it's about optimizing how the underlying blockchain ledger stores information related to that NFT. From the context of an account, we're compressing that into the ledger by aggregating multiple accounts (in this case NFTs) into a single Merkle root that is stored in state. This process significantly reduces storage costs while retaining verifiability.

Storing compressed data off-chain is risky and leads to vulnerabilities

This is wrong - you can securely store data off-chain by hashing it and storing its Merkle root on-chain. Technically, compressed NFTs are not stored off-chain. The data is still on-chain as anything that can be re-derived by the ledger is deemed on-chain. The difference is that accounts are state incentivized to be held in memory by validators whereas the ledger needs to be accessed via archival nodes. State compression merges the two to provide verification of the ledger data via state in an account, which still retains the security and decentralization of Solana itself. We'll go over what the ledger is and why it's safe in another section.

I can lose my concurrent Merkle tree if the indexer or RPC provider I'm using to store my tree goes down

You will not lose your tree - the entire tree can be reconstructed by replaying the tree's history by anyone with access to the ledger.

Concurrent Merkle trees can handle parallel updates

A common misconception is that the use of the word “concurrent” implies that multiple updates to an on-chain Merkle tree can occur in parallel. While concurrent Merkle trees can accommodate multiple leaf replacements within the same block, these updates are handled sequentially by validators. When a validator receives a batch of transactions that affect an on-chain concurrent Merkle tree, the validator can process in the same slot. The data per slot, however, is not produced concurrently. We expand upon this in the following section What’s State Compression?

A tree is the same thing as a collection

Concurrent Merkle trees are not the same thing as a collection. A single collection can use any

number of concurrent Merkle trees. It is important to note that groupings of NFTs can be orthogonal to their storage. NFTs can be in accounts or compressed into the ledger, across any number of trees, one or many. Although, it is recommended for concurrent Merkle trees to be used for one collection only to reduce complexity.

What’s State Compression?

State compression serves to optimize storage by creating a cryptographic hash of ledger data and storing this hash in an account. This approach leverages the inherent security and immutability of the ledger while providing a robust framework for verifying data stored in the ledger.

This is a cost-effective solution for applications built on top of Solana. Developers can now use ledger storage space instead of pricier account-based storage. Thus, state compression not only assures data integrity but also stands as a cost-effective solution for resource allocation on Solana.

The secret behind Solana’s state compression is the use of concurrent Merkle trees. Concurrent Merkle trees are optimized to process multiple transactions in rapid succession such that its proofs can be fast-forwarded. This differs from traditional Merkle trees where the tree’s proofs are invalidated on every update. Concurrent Merkle trees store a secure changelog of their most recent changes along with their root hash and the proof required to derive it. This changelog is stored on-chain in an account that is dedicated to the tree. Each concurrent Merkle tree has a maximum buffer size. This value represents the highest number of changes that can be made to the tree while its Merkle root is still valid. Think of this as how “stale” a calculated set of proofs can be before they need to be updated.

Thus, when a validator receives multiple requests to update an on-chain Merkle tree within the same slot, the validator can use the tree’s changelog as a source of truth. This permits up to the maximum buffer size number of concurrent changes to the Merkle tree. While this does not directly reduce the amount of data stored on-chain, it enhances efficiency by allowing multiple updates to be processed simultaneously. This means that the system can maintain the “proof of inclusion” integrity that Merkle trees offer, even in a high throughput environment. Here, proof of inclusion simply means the ability to demonstrate that a specific data element is indeed part of a set of data that has been hashed together into a Merkle root.

This ingenious combination of state compression and concurrent Merkle trees offers an extremely cost-effective solution for applications building on Solana. It is essential to discuss the difference between Solana’s state and its ledger to fully appreciate the impact of these technologies.

State vs Ledger

The ledger is a historical record of all transactions signed by clients that have occurred on Solana since its genesis block. It is an append-only data structure, meaning a transaction cannot be modified or removed once it's added. Validators validate the transactions that are added to the ledger. The ledger is stored by multiple nodes across the network to ensure fault tolerance. A validator’s copy of the ledger, however, may only contain newer blocks to reduce storage as older blocks are not necessary to validate future blocks.

The state represents the current snapshot of all accounts and programs on Solana. The state is mutable and changes when transactions are processed. Think of the state as a highly optimized database that can be queried for token balances, programs, and accounts.

Here’s an easy way to differentiate the two: Say Alice has a balance of 100 SOL and Bob also has a balance of 100 SOL. Alice sends a transaction to give Bob 10 SOL. Once verified, the transaction is added to a block and the block is appended to the ledger. The ledger now has an immutable record stating that Alice sent 10 SOL to Bob. Concurrently, the state would update Alice’s and Bob’s accounts to 90 and 110 SOL, respectively.

The key differences between the two can be summarized as follows:

- The ledger is immutable and append-only whereas the state is mutable and changes constantly
- The ledger is a historical record of all transactions whereas the state reflects the current status of all accounts and programs
- The ledger is used for verification whereas the state is used for executing transactions and running programs

While the ledger acts as an immutable historical record to ensure that every transaction is verifiable and traceable, the state functions as a dynamic snapshot of the ledger, adjusting to real-time operations such as transfers and program execution. Importantly, both are subject to consensus of the chain itself. Together, the state and the ledger form the backbone of Solana, enabling it to operate efficiently while upholding decentralized trust.

What are Compressed NFTs?

Compressed NFTs (cNFTs) use state compression and concurrent Merkle trees to reduce storage costs. Compressed NFTs store their metadata on the ledger instead of storing each NFT in a typical Solana account. This allows for reduced storage costs while inheriting the security and immutability of the ledger.

Compressed NFTs still follow the exact same metadata schema as their uncompressed counterparts. Thus, NFTs and cNFTs are defined the same way.

The key differences of NFTs and cNFTs are as follows:

- A compressed NFT can be converted to a regular NFT but a regular NFT cannot be converted to a compressed NFT
- Compressed NFTs are not

native Solana tokens - they do not have a token account, mint account, or metadata. They do, however, have a stable identifier (the asset ID). Upon decompression, the NFT retains the same identifier. Thus, NFTs in their compressed state are not native tokens but can be made into them if needed

- A single concurrent Merkle tree account can hold millions of NFTs
- A single collection is able to span across multiple tree accounts
- All NFT modifications occur through the Bubblegum program
- A DAS API call is recommended to read any information about a compressed NFT

Interestingly, we need to use the DAS API to get information about a compressed NFT. Why is that? And more importantly, what is that?

Reading Compressed NFTs Metadata with the DAS API

We need the help of indexers since a cNFT's metadata is stored on the ledger instead of in a traditional account. Although you can derive the current state of a compressed NFT by replaying relevant transactions, providers like Helius do this for your convenience. Developers can use the Digital Asset Standard (DAS) API, an open-source specification and system to fetch an asset's information. The DAS API supports both compressed and traditional, or uncompressed, NFTs. Thus, you can use the same endpoint for both NFT types.

Helius currently supports the following DAS API methods:

- getAsset
- get a specific asset by its id
- getAssetBatch
- get multiple assets by their IDs
- getAssetProof
- get a Merkle proof for a compressed asset by its id
- getAssetProofBatch
- get multiple asset proofs by their IDs
- getAssetsByOwner
- get a list of assets owned by an address
- getAssetsByAuthority
- get a list of assets with a specific authority
- getAssetsByCreator

- gets a list of assets created by an address
- getAssetsByGroup
- get a list of assets by a group key and value
- searchAssets
- search for assets by a variety of parameters
- getSignaturesForAsset
- get a list of transaction signatures related to a compressed asset
- Pagination - support for page-based and keyset pagination for fetching more than 1000 records at a time

See the Helius DAS API documentation to learn more about each individual method. As an example, if you wanted to fetch a list of all assets owned by an address you can make the following POST request with getAssetsByOwner

:

It's convenient to retrieve compressed assets, but what if we want to create our own? Before embarking on our minting journey, it is crucial to compute the size and associated cost of building a concurrent Merkle tree that will store these assets.

Sizing and Costs of Creating a Concurrent Merkle Tree

Calculating Size

When creating a concurrent Merkle tree on-chain, there are three important metrics that will determine the size of the tree, the cost to create the tree, and the number of concurrent changes that can be made to the tree while keeping the Merkle root valid:

- Max depth
- Max buffer size
- Canopy depth

Max depth refers to the maximum number of hops to get from any leaf to the root of the tree. Every leaf is connected to only one other leaf, existing as a leaf pair for pairwise hashing. You can calculate the maximum number of leaf nodes a tree can accommodate using the formula: numberOfNodes = 2 ^ maxDepth

. Tree depth must

be set at creation so you need to use this formula to determine the lowest possible max depth to store your data. For example, if you aim to store around 100 compressed NFTs in a tree, a maxDepth

of 7 would suffice since 2^7 = 128

and 2^6 = 64

. Max depth is a significant cost determinant when constructing a concurrent Merkle tree on-chain. These costs are incurred upfront during the tree's creation and scale with higher values of maxDepth

.

Max buffer size refers to the maximum number of changes that can occur to a tree with its Merkle root still being valid. The changelog buffer is sized and set at tree creation for concurrent Merkle trees using the maxBufferSize

value. Thus, when a validator receives multiple change requests to a tree in the same slot, they can use the changelog and allow for up to maxBufferSize

changes with the root still being valid.

It is vital to note that there are only a specific number of valid maxDepth

and maxBufferSize

pairs for creating a new concurrent Merkle tree account. The@solana/spl-account-compression package exports the constant ALL_DEPTH_SIZE_PAIRS

, which is an array of number arrays containing all valid combinations. The minimum is a maxDepth

of 3 and a maxBufferSize

of 8 whereas the maximum is a maxDepth

of 30 and a maxBufferSize

of 2048.

Canopy depth refers to a subset of the Merkle tree that is stored in an account. These cached proofs are used to supplement the proofs that are being transmitted over the wire, since they are subject to transaction limits. The complete path must be used to verify original ownership of a leaf when trying to alter its data, such as when you transfer an NFT. The larger the tree's max depth, the more proof nodes required for verification. The canopy allows for a reduced proof size and avoids using a proof size of maxDepth

to verify the tree.

The canopy depth can be calculated by subtracting your desired proof size from the max depth. So, if you had a max depth of 14 and wanted a proof size of 4, you'd have a canopy depth of 10. This means that you would only have to submit 4 proof nodes per update transaction. Canopy depth is also a significant cost determinant when constructing an on-chain concurrent Merkle tree. These cost are incurred upfront during the tree's creation and scale with higher values of canopyDepth

. While a lower canopyDepth

results in a lower upfront cost, having a low canopyDepth

can limit composability. This is due to the fact that each update transaction will have to require a larger proof size, thus putting constraints on transaction size limits. If, for example, your tree with a low canopyDepth

is being used for compressed NFTs then an NFT marketplace might only be able to support simple transfers for your collection. In general, maxDepth - canopyDepth

should be less than or equal to 10 for maximum composability. This is outlined inTensor's specification of the max proof length for Tensor cNFTs.

Calculating Costs

Different methods exist for determining the size and cost of a concurrent Merkle tree. The simplest approach is to use the Compressed NFT Calculator and input the number of compressed NFTs intended for storage in that tree:

The site offers a detailed breakdown of the optimal tree depth needed for the desired number of assets to be stored, along with various cost options based on composability. The figure, for example, shows that to create a highly composable tree that stores 10 million compressed NFTs would only cost ~7.67 SOL. Taking into account the transaction costs of ~50 SOL to mint 10 million NFTs, the total cost would be around ~57.67 SOL.

Developers can also use the@solana/spl-account-compression package to calculate the required space for a given tree size and the cost to allocate the required space for the tree on-chain. This can be achieved with the following script:

Here, we are importing the necessary modules from@solana/web3.js and @solana/spl-account-compression. We need a connection to mainnet

, which we can establish using a Helius API key. The function calculateCosts

logs to the console the maxDepth

, maxBufferSize

, canopy

, number of NFTs that could be stored in this tree, as well as the cost of rent

in SOL. Thus, when we call calculateCosts

using our desired proof size, we can see all the possible tree combinations in our console.

Note that some of the logs may output: Unable to fetch minimum balance for rent exemption

. This is because the account with your specified maxProofSize

would be too big to create and we therefore cannot fetch a minimum balance that'll make the account rent exempt.

Creating a Concurrent Merkle Tree

We need to create two accounts when creating a concurrent Merkle tree:

- A concurrent Merkle tree account
- A concurrent Merkle tree config account

The tree account holds the Merkle tree that is used for data verification. We create this using our desired max depth, max buffer size, and canopy depth as mentioned in the previous section. This account is owned by the Account Compression program, which is created and maintained by Solana. It is used to verify the authenticity of compressed NFTs.

The tree config account is a PDA derived from the address of the concurrent Merkle tree account. This is used to store additional configurations such as the tree's creator and the number of compressed NFTs minted.

Metaplex refers to concurrent Merkle trees with an associated tree configuration account as a "Bubblegum tree".

Full Code

Breakdown of the Code

This serves as a sample function for creating a concurrent Merkle tree on Solana. To invoke this sample function, createTree

, the following parameters must be passed:

- connection
- a connection to a full node JSON RPC endpoint, which is of type Connection
- payer
- the account that will be paying for the transaction, which is of type Keypair
- treeKeypair
- the key pair address for the tree, which is of type Keypair
- maxDepthSizePair
- the valid maxDepth

and maxBufferSize

pair, which is of type ValidDepthSizePair

- canopyDepth
- the canopy depth for the tree, which is of type number

and is set to a default value of 0

First, we import @solana/web3.js, @solana/spl-account-compression, and @metaplex-foundation/mpl-bubblegum with the necessary modules.

Here, we define the function createTree

with the aforementioned parameters.

createAllocTreeIx

is a helper function we use to create the concurrent Merkle tree account. The SPL Account Compression package recommends using this method to initialize a concurrent Merkle tree account because these accounts tend to be quite large and may be over the limit for what can be allocated via CPI. Here, we create the instruction to allocate the tree's account on-chain. This also computes the space needed to store the tree on-chain, as well as the cost, so we don't need to worry about this later.

We need to derive the tree configuration account with its authority owned by the Bubblegum program. This is required for the createCreateTreeInstruction

, the instruction that creates the tree, as we need to pass in the treeAuthority

as an argument. Here, we derive the PDA with the findProgramAddressSync

method using the tree's publickey and the Bubblegum program ID. We need to destructure the treeAuthority

since both the authority and the bump are returned. I have omitted the bump since it is not necessary for our function. Change the destructuring to [treeAuthority, bump]

to save the bump, if needed.

We use the createCreateTreeInstruction

from the Bubblegum SDK to build the instruction that builds the concurrent Merkle tree. This creates the tree on-chain with the Bubblegum program as its owner. createCreateTreeInstruction

has three parameters. The first is an object containing accounts to set up properties such as the creator of the tree. The second object pertains to the max depth and max buffer sizes. It also includes a public

parameter of type boolean

. Setting public

to true

will allow anyone to mint compressed NFTs from the tree. Otherwise, only the tree creator or the tree delegate will be able to mint compressed NFTs from the tree. A delegated account can perform actions on behalf of the tree owner, such as transferring or burning a compressed NFT. As an aside, you can assign a tree delegate using the createSetTreeDelegateInstruction

from the @metaplex-foundation/mpl-bubblegum

package as so:

We also pass in the program ID of the Bubblegum program. Now back to the rest of our code:

We add the two instructions we've just built to a transaction and send them off. We ensure that both the treeKeypair

and payer

sign the transaction. The successful transaction signature is then logged to the console. We wrap this process within a try-catch

block so if, for whatever reason, an error occurs it is logged to the console via console.error

.

Creating a Concurrent Merkle Tree with Umi

Using the Bubblegum SDK, Solana's account compression program, and Solana's web3.js packages can be quite confusing for newer developers and tedious to set up every time. Luckily, the Bubblegum SDK provides a createTree

operation that handles everything for us, which pairs nicely with Umi. The code is as follows:

Umi is a modular framework for building and using JavaScript clients for Solana programs. It provides a zero dependency library with a set of core interfaces that other libraries can rely on without being constrained to a specific implementation. Umi is provided by Metaplex and its documentation can be found here.

We use our Umi instance to generate a signer, create our Merkle tree, and send and confirm our built transaction. By default, the tree creator is set to the Umi identity and the public

parameter is set to false. These parameters can be customized, allowing for a custom tree creator and a public value of true

to also be passed in. This is a much faster way to create an on-chain concurrent Merkle tree.

Note that Bubblegum is agnostic to the canopy size. This is because Solana's Account Compression Program will determine the canopy size based on the available account space. It is only necessary to allocate sufficient space so that the program can accurately discern the appropriate canopy size to use.

Minting cNFTs by Interacting with Bubblegum Directly

Creating a Collection

NFTs are traditionally grouped together into a collection using the Metaplex standard. This is true for both compressed and "regular" NFTs. To create a collection:

- Create a new token "mint"

- Create an associated token account for the mint
- Mint a single token
- Store the collection's metadata in an account on-chain

While not directly related to the topic of state compression or compressed NFTs, and thus beyond the scope of this article, a script has been provided as a reference for creating your own collection. You can access that script here.

Minting an NFT to Our Collection

With your newly created collection you'll need the following to start minting:

- collectionMint
- the collection's mint address
- collectionAuthority
- the account with authority over the collection
- collectionMetadata
- the collection's metadata account
- editionAccount
- the account that holds additional attributes, like a master edition account

Full Code For Minting to a Collection

Breaking Down the Minting Process

First, we import @solana/web3.js, @solana/spl-account-compression, @metaplex-foundation/mpl-bubblegum, and @metaplex-foundation/mpl-token-metadata with the necessary modules.

We define mintCompressedNFT

, which takes in quite a few parameters:

- connection
- the connection object used to interact with Solana
- payer
- the account that will pay for the transaction fees
- treeAddress
- the account of the concurrent Merkle tree
- collectionMint
- the collection's mint address
- collectionMetadata
- the collection's metadata account
- collectionMasterEditionAccount
- the master edition account
- compressedNFTMetadata
- the metadata specific to the cNFT to be minted
- receiverAddress
- an optional public key address where the newly minted cNFT will be sent

Here, we are finding the necessary PDAs and disregarding their bumps. First, we derive the PDA for the tree's authority,

and then we derive a PDA to act as the signer for the compressed minting. We need to include collection_cpi

as it is a custom prefix required by the Bubblegum program.

We set mintInstructions

to an empty array of TransactionInstruction

. This allows us to mint multiple cNFTs at the same time, if desired.

metadataArgs

makes sure our compressedNFTMetadata

is formatted correctly. Minting an NFT into a collection using createMintToCollectionV1Instruction

requires the verified field to be set to false

for the transaction to succeed, despite it verifying the collection automatically.

We add a single mint to our instruction. We could add multiple mints in the same transaction as long as the transaction remains within the byte size limits. Here, we use the createMintToCollectionV1Instruction

to mint our compressed NFT from our collection. This instruction takes in two objects, one with the accounts needed to process the instruction and one to provide instruction data to the program. Most of these parameters should look familiar from previous sections. Note that you can set any delegate address at mint, but it should normally be the same as leafOwner

. Regardless, the delegate is auto-cleared when transferring the cNFT. We set the payer as the delegate since they will also be the one to receive the cNFT if a receiverAddress

isn't provided.

We then build our transaction, set payer

as feePayer

, and send our transaction off. We surround this logic in a try-catch

block in case of any errors in sending and confirming the transaction. If any errors do occur, we log them to the console with console.error

.

Minting cNFTs with Umi

The Bubblegum program offers two minting processes via Umi:

- Minting an NFT without associating them with a collection
- Minting an NFT to a given collection.

Minting Without a Collection

Bubblegum's MintV1

instruction allows minting compressed NFTs from a Bubblegum Tree without a collection. If the tree is public then anyone can mint to this tree. Otherwise, only the tree creator or delegate can use this instruction. Here's how to mint a compressed NFT without a collection:

This code snippet is from the Metaplex docs on minting cNFTs with Bubblegum Here, we are using an instance of Umi to mint a cNFT. The other parameters for the mintV1

instruction are as follows:

- The leafOwner

is the owner of the cNFT to be minted

- The merkleTree

is the concurrent Merkle tree account address from which the cNFT will be minted from

- The metadata

is an object that contains the metadata of the cNFT to be minted. This includes the name of the cNFT, its URI, its collection which we've invoked to none, as well as its creators. It is possible to provide a collection object but set the verified field in the creators to false

since the collection authority is not requested in the instruction. Creators can also verify themselves by setting the verified field to true and provide the creator as a signer in the remaining accounts.

The mintV1

instruction also contains a number of optional fields as the function's input is of type MintV1InstructionAccounts

& MintV1InstructionArgs

. These types are defined as follows:

MintV1InstructionArgs

is an elusive type within types that boils down to an object with a metadata field. This metadata field is of type MetadataArgsArgs

and is defined as follows:

The full function definition for mintV1

and all of its associated types can be foundhere. But, at the very least with an instance of Umi, you can mint a cNFT without a collection if you pass in the requisite metadata, leaf owner, and concurrent Merkle tree account.

Minting With a Collection

Bubblegum provides the mintToCollectionV1

as a convenient way to mint a cNFT to a given collection directly. This instruction's input is of types MintToCollectionV1InstructionAccounts

and MintToCollectionV1InstructionArgs,

which is ultimately an object of type MetadataArgsArgs

. The type definition for MintToCollectionV1InstructionAccounts

is as follows:

The key parameters are the collection mint, the collection authority, and the collection authority record PDA. A delegate record PDA must be provided when using a delegated collection authority to ensure the authority is allowed to manage the collection NFT. The metadata parameter must

contain a collection object where its address field matches the collection mint parameter and its verified field is set to false

. Creators can also verify themselves by signing the transaction and adding themselves as remaining accounts.

Here's how to mint a compressed NFT with a collection:

This code snippet can be found in the Metaplex documentation on minting cNFTs with Bubblegum Again, we are using an instance of Umi to mint the compressed NFT. Like mintV1

, we pass in the leafOwner

and merkleTree

. This time, however, we pass in the collectionMint

. In the metadata field we pass in a collection

object where the key matches the collectionMint

and the verified field is set to false

. Note that the Umi identity is set to the default collection authority. This can be changed by setting the optional collectionAuthority

field to a custom collection authority.

Minting cNFTs with Helius

At Helius, we provide a Mint API that allows you to mint compressed NFTs without additional headaches. We cover Solana fees, Merkle tree creation, and upload your off-chain metadata to Arweave. We also ensure the transaction was submitted successfully and confirmed by the network, so you don't have to worry about polling yourself. We also parse the asset ID from the transaction, allowing you to use it with the DAS API immediately.

For Helius to mint an NFT into your collection, it must be delegated the collection authority. The authority must be delegated to one of the following accounts, depending on your cluster:

- Devnet: 2LbAtCJSaHqTnP9M5QSjvAMXk79RNLusFspFN5Ew67TC
- Mainnet: HnT5KVAywGgQDhmh6Usk4bxRg4RwKxCK4jmECyaDth5R

Here's how to mint a cNFT using the Helius Mint API:

This code snippet and a further breakdown of the request's schema can be found in our documentation.

Note that if you do not fill in the uri

field, we will build a JSON file and upload it to Arweave on your behalf. The file will adhere to the v1.0 Metaplex JSON standard and will be uploaded via Irys (formerly known as Bundlr).

Transferring cNFTs

The general steps for transferring a compressed NFT are as follows:

- Get the cNFT's asset data from the indexer
- Get the cNFT's proof from the indexer
- Get the concurrent Merkle tree account from Solana
- Prepare the asset proof
- Build and send the transfer transaction

Using Umi and Metaplex greatly simplifies this process, but this section will demonstrate what occurs behind the scenes. Below we'll outline how to transfer a compressed NFT using both web3.js and Metaplex.

Transferring by Interacting with Bubblegum Directly

We need to fetch some information about our compressed NFT before we use our script to execute our transfer. First, we need to use the getAsset

method on the DAS API to retrieve the compressed NFT's metadata. Here, we're looking for the data_hash

, creator_hash

, owner

, delegate

, and leaf_id

:

This is what part of the successful response will look like:

Once we have the necessary information, we need to use the getAssetProof

method to retrieve the proof

and tree_id

(the tree's address). Here's an example call:

This is what the successful response will look like:

Now with the root, proof, and tree_id we can jump into our transfer script.

Full Code

Breaking Down the Code

We import @solana/web3.js, @metaplex-foundation/mpl-bubblegum, and @solana/spl-account-compression with the necessary modules.

We define our transferCompressedNFT

function where we'll parse the proof path, build the transfer instruction, and execute it.

We fetch the concurrent Merkle's tree account from the blockchain and extract the tree authority and canopy depth. These values are needed to build the transfer instruction.

To put it simply, we parse the list of proof addresses into a valid array of type AccountMeta

. AccountMeta

is the account metadata used to define transactions. This includes the account's public key, whether an instruction requires a transaction signature that matches the public key, and whether the public key can be loaded as a read-write account.

We take a slice of our full proof, starting from the beginning of the array and ensure we only have proof.length - canopyDepth

number of proof values. We do this to remove the portion of the tree that is already cached in the on-chain canopy. Then we structure each remaining proof values as a valid AccountMeta

. This is done since the proof is submitted on-chain in the form of "extra accounts" within the transfer instruction.

We then set the leafOwner

to the owner

parameter and the leafDelegate

to the delegate

parameter.

We build the transferInstruction

using the createTransferInstruction

helper function from the Bubblegum SDK. Note that the root

, dataHash

, and creatorHash

are returned from the DAS API as a string so we must convert them to type PublicKey

and then an array of bytes.

With our built instruction, we add it to a new transaction and send it off to Solana. If there are any errors we then log it to the console using console.error

.

If you are running into errors regarding the concurrent Merkle tree, it is possible that your RPC is providing stale or incorrect data for the concurrent Merkle tree proof. This can happen from time to time due to caching issues. To remedy this, you can try client-side verification of the proof that was provided by the RPC:

Note you'll also need to use the leaf

value returned by our getAssetProof

DAS API call. This is not required because actual proof validation is performed on-chain, however, this may help with error handling.

And with that, you can make another getAsset

call to see that the leafDelegate

is an empty value and the leaf has a new owner!

Transferring with Umi

This code is from the Metaplex documentation on transferring compressed NFTs.

Bubblegum provides a transfer

instruction that is very straightforward to use. First, it takes in an instance of Umi. Then it accepts an object that contains the asset with information on its proof, the leaf owner, and the new leaf owner. To get the asset with its requisite proof, we can use the getAssetWithProof

method also provided by Bubblegum. Note that the leaf delegate can be used in place of the leaf owner - only an account with authority to authorize the transfer is needed. With the .sendAndConfirm()

method we're sending off the transaction that initiates the transfer and then confirming it with our instance of Umi.

Conclusion

Congratulations! We explored state compression and compressed NFTs on Solana in a very comprehensive way. We've navigated the complexities of concurrent Merkle trees, demystified common misconceptions, and dove deep into Solana's ledger. Theory aside, we've learnt how to fetch, mint, and transfer cNFTs using the power of Solana's web3.js, Metaplex, and Helius!

Solana's state compression is revolutionary in a landscape where transaction and storage costs can be restrictive. Compression drastically slashes costs without compromising security or decentralization. This is a paradigm shift that opens up unprecedented possibilities for artists, collectors, and developers alike.

If you've made it this far, thank you anon! You are well-equipped to contribute to this exciting frontier. Go forth - mint a ten million NFT collection for your on-chain MMORPG, build a decentralized app that utilizes the power of the ledger, or simply share your newfound knowledge with the community. The best way to predict the future is to create it.

Additional Resources / Further Reading

- Solana Documentation on State Compression
- Account Compression Program
- Metaplex's Bubblegum Documentation
- Compressed NFTs on Solana Are The Future • Helius Explains
- Case Study on Dialect and cNFTs
- Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree
- Parallelized C++ Implementation of a Merkle Tree