# State Compression

On Solana,State Compression is the method of creating a "fingerprint" (or hash) of off-chain data and storing this fingerprint on-chain for secure verification. Effectively using the security of the Solana ledger to securely validate off-chain data, verifying it has not been tampered with.

This method of "compression" allows Solana programs and dApps to use cheap blockchainledger space, instead of the more expensiveaccount space, to securely store data.

This is accomplished by using a special binary tree structure, known as aconcurrent merkle tree , to create a hash of each piece of data (called aleaf ), hashing those together, and only storing this final hash on-chain.

## What is State Compression?#

In simple terms, state compression uses "tree " structures to cryptographically hash off-chain data together, in a deterministic way, to compute a single final hash that gets stored on-chain.

Thesetrees are created in this "deterministic " process by:

- taking any piece of data
- creating a hash of this data
- storing this hash as aleaf
- the bottom of the tree
- eachleaf
- pair is then hash together, creating abranch
- eachbranch
- is then hash together
- continually climbing the tree and hashing adjacent branches together
- once at the top of the tree, a finalroot hash
- is produced

Thisroot hash is then stored on chain, as a verifiableproof of all of the data within every leaf. Allowing anyone to cryptographically verify all the off-chain data within the tree, while only actually storing aminimal amount of data on-chain. Therefore, significantly reducing the cost to store/prove large amounts of data due to this "state compression".

## Merkle trees and concurrent merkle trees#

Solana's state compression used a special type ofmerkle tree that allows for multiple changes to any given tree to happen, while still maintaining the integrity and validity of the tree.

This special tree, known as a "concurrent merkle tree ", effectively retains a "changelog" of the tree on-chain. Allowing for multiple rapid changes to the same tree (i.e. all in the same block), before a proof is invalidated.

### What is a merkle tree?#

Amerkle tree , sometimes called a "hash tree", is a hash based binary tree structure where eachleaf node is represented as a cryptographic hash of its inner data. And every node that isnot a leaf, called abranch , is represented as a hash of its child leaf hashes.

Each branch is then also hashed together, climbing the tree, until eventually only a single hash remains. This final hash, called theroot hash or "root", can then be used in combination with a "proof path" to verify any piece of data stored within a leaf node.

Once a finalroot hash has been computed, any piece of data stored within aleaf node can be verified by rehashing the specific leaf's data and the hash label of each adjacent branch climbing the tree (known as theproof or "proof path"). Comparing this "rehash" to theroot hash is the verification of the underlying leaf data. If they match, the data is verified accurate. If they do not match, the leaf data was changed.

Whenever desired, the original leaf data can be changed by simply hashing thenew leaf data and recomputing the root hash in the same manner of the original root. Thisnew root hash is then used to verify any of the data, and effectively invalidates the previous root hash and previous proof. Therefore, each change to thesetraditional merkle trees are required to be performed in series.

This process of changing leaf data, and computing a new root hash can be avery common thing when using merkle trees! While it is one of the design points of the tree, it can result in one of the most notable drawbacks: rapid changes.

### What is a Concurrent merkle tree?#

In high throughput applications, like within theSolana runtime , requests to change an on-chaintraditional merkle tree could be received by validators in relatively rapid succession (e.g. within the same slot). Each leaf data change would still be required to performed in series. Resulting in each subsequent request for change to fail, due to the root hash and proof being invalidated by the previous change request in the slot.

Enter, Concurrent merkle trees.

AConcurrent merkle tree stores asecure changelog of the most recent changes, their root hash, and the proof to derive it. This changelog "buffer" is stored on-chain in an account specific to each tree, with a maximum number of changelog "records" (akamaxBufferSize ).

When multiple leaf data change requests are received by validators in the same slot, the on-chainconcurrent merkle tree can use this "changelog buffer" as a source of truth for more acceptable proofs. Effectively allowing for up tomaxBufferSize changes to the same tree in the same slot. Significantly boosting throughput.

## Sizing a concurrent merkle tree#

When creating one of these on-chain trees, there are 3 values that will determine the size of your tree, the cost to create your tree, and the number of concurrent changes to your tree:

1. max depth
2. max buffer size
3. canopy depth

### Max depth#

The "max depth" of a tree is themaximum number of hops to get from any dataleaf to theroot of the tree.

Since merkle trees are binary trees, every leaf is connected toonly one other leaf; existing as aleaf pair .

Therefore, themaxDepth of a tree is used to determine the maximum number of nodes (aka pieces of data orleafs ) to store within the tree using a simple calculation:

nodes_count = 2 ^ maxDepth Since a trees depth must be set at tree creation, you must decide how many pieces of data you want your tree to store. Then using the simple calculation above, you can determine the lowestmaxDepth to store your data.

#### Example 1: minting 100 nfts#

If you wanted to create a tree to store 100 compressed nfts, we will need a minimum of "100 leafs" or "100 nodes".

// maxDepth=6 -> 64 nodes 2^6 = 64

// maxDepth=7 -> 128 nodes 2^7 = 128 We must use amaxDepth of7 to ensure we can store all of our data.

#### Example 2: minting 15000 nfts#

If you wanted to create a tree to store 15000 compressed nfts, we will need a minimum of "15000 leafs" or "15000 nodes".

// maxDepth=13 -> 8192 nodes 2^13 = 8192

// maxDepth=14 -> 16384 nodes 2^14 = 16384 We must use amaxDepth of14 to ensure we can store all of our data.

#### The higher the max depth, the higher the cost#

ThemaxDepth value will be one of the primary drivers of cost when creating a tree since you will pay this cost upfront at tree creation. The higher the max tree depth depth, the more data fingerprints (aka hashes) you can store, the higher the cost.

### Max buffer size#

The "max buffer size" is effectively the maximum number of changes that can occur on a tree, with theroot hash still being valid.

Due to the root hash effectively being a single hash of all leaf data, changing any single leaf would invalidate the proof needed for all subsequent attempts to change any leaf of a regular tree.

But with aconcurrent tree , there is effectively a changelog of updates for these proofs. This changelog buffer is sized and set at tree creation via thismaxBufferSize value.

### Canopy depth#

The "canopy depth", sometimes called the canopy size, is the number of proof nodes that are cached/stored on-chain for any given proof path.

When performing an update action on aleaf , like transferring ownership (e.g. selling a compressed NFT), thecomplete proof path must be used to verify original ownership of the leaf and therefore allow for the update action. This verification is performed using thecomplete proof path to correctly compute the currentroot hash (or any cachedroot hash via the on-chain "concurrent buffer").

The larger a tree's max depth is, the more proof nodes are required to perform this verification. For example, if your max depth is14 , there are14 total proof nodes required to be used to verify. As a tree gets larger, the complete proof path gets larger.

Normally, each of these proof nodes would be required to be included within each tree update transaction. Since each proof node value takes up32 bytes in a transaction (similar to providing a Public Key), larger trees would very quickly exceed the maximum transaction size limit.

Enter the canopy. The canopy enables storing a set number of proof nodes on chain (for any given proof path). Allowing for less proof nodes to be included within each update transactions, therefore keeping the overall transaction size below the limit.

For example, a tree with a max depth of14 would require14 total proof nodes. With a canopy of10 , only4 proof nodes are required to be submitted per update transaction.

#### The larger the canopy depth value, the higher the cost#

ThecanopyDepth value is also a primary factor of cost when creating a tree since you will pay this cost upfront at tree creation. The higher the canopy depth, the more data proof nodes are stored on chain, the higher the cost.

#### Smaller canopy limits composability#

While a tree's creation costs are higher with a higher canopy, having a lowercanopyDepth will require more proof nodes to be included within each update transaction. The more nodes required to be submitted, the larger the transaction size, and therefore the easier it is to exceed the transaction size limits.

This will also be the case for any other Solana program or dApp that attempts to interact with your tree/leafs. If your tree requires too many proof nodes (because of a low canopy depth), then any other additional actions another on-chain programcould offer will belimited by their specific instruction size plus your proof node list size. Limiting composability, and potential additional utility for your specific tree.

For example, if your tree is being used for compressed NFTs and has a very low canopy depth, an NFT marketplace may only be able to support simple NFTs transfers. And not be able to support an on-chain bidding system.

## Cost of creating a tree#

The cost of creating a concurrent merkle tree is based on the tree's size parameters:maxDepth ,maxBufferSize , andcanopyDepth . These values are all used to calculate the on-chain storage (in bytes) required for a tree to exist on chain.

Once the required space (in bytes) has been calculated, and using thegetMinimumBalanceForRentExemption RPC method, request the cost (in lamports) to allocate this amount of bytes on-chain.

### Calculate tree cost in JavaScript#

Within the@solana/spl-account-compression package, developers can use thegetConcurrentMerkleTreeAccountSize function to calculate the required space for a given tree size parameters.

Then using thegetMinimumBalanceForRentExemption function to get the final cost (in lamports) to allocate the required space for the tree on-chain.

Then determine the cost in lamports to make an account of this size rent exempt, similar to any other account creation.

```
// calculate the space required for the tree const requiredSpace = getConcurrentMerkleTreeAccountSize( maxDepth,
maxBufferSize, canopyDepth, );
```

```
// get the cost (in lamports) to store the tree on-chain const storageCost = await
connection.getMinimumBalanceForRentExemption(requiredSpace);
```

### Example costs#

Listed below are several example costs, for different tree sizes, including how many leaf nodes are possible for each:

Example #1: 16,384 nodes costing 0.222 SOL

- max depth of14
- and max buffer size of64
- maximum number of leaf nodes:16,384
- canopy depth of0
- costs approximately0.222 SOL
- to create

Example #2: 16,384 nodes costing 1.134 SOL

- max depth of14
- and max buffer size of64
- maximum number of leaf nodes:16,384
- canopy depth of11
- costs approximately1.134 SOL
- to create

Example #3: 1,048,576 nodes costing 1.673 SOL

- max depth of20
- and max buffer size of256
- maximum number of leaf nodes:1,048,576
- canopy depth of10
- costs approximately1.673 SOL
- to create

Example #4: 1,048,576 nodes costing 15.814 SOL

- max depth of20
- and max buffer size of256
- maximum number of leaf nodes:1,048,576
- canopy depth of15
- costs approximately15.814 SOL
- to create

## Compressed NFTs#

Compressed NFTs are one of the most popular use cases for State Compression on Solana. With compression, a one million NFT collection could be minted for~50 SOL , vice~12,000 SOL for its uncompressed equivalent collection.

If you are interested in creating compressed NFTs yourself, read our developer guide forminting and transferring compressed NFTs .