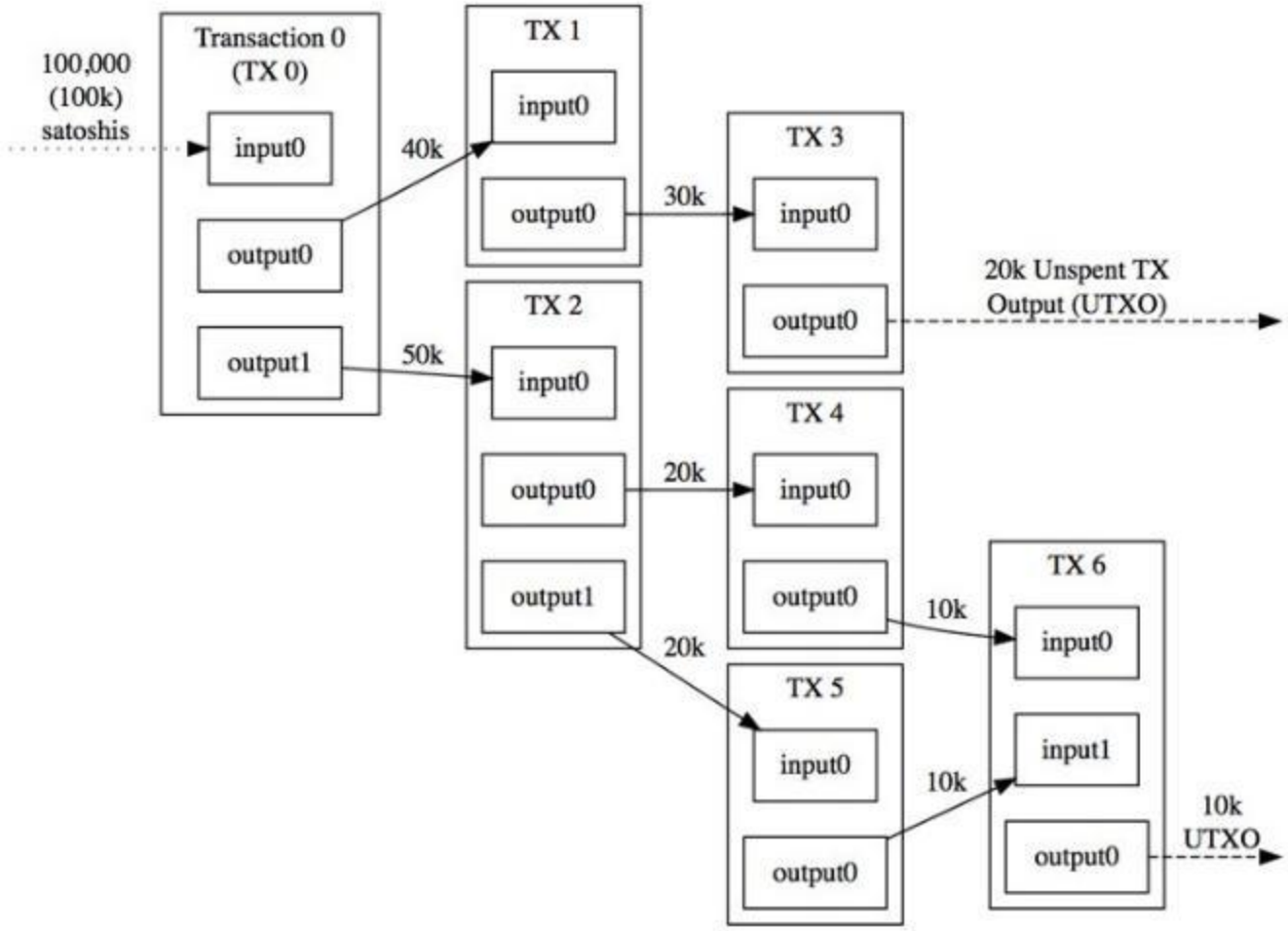
UTXO vs Account Based Model

UTXO

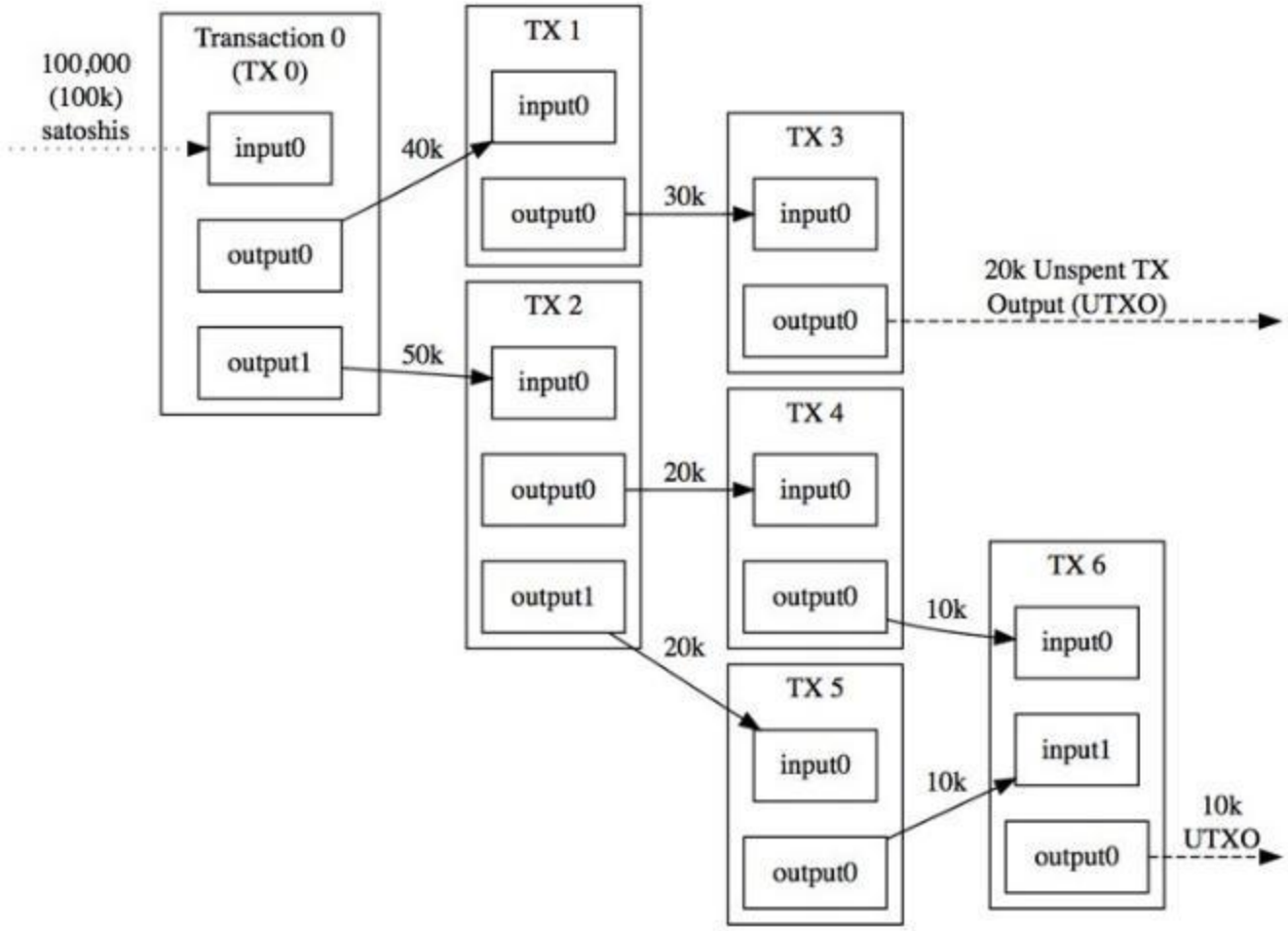
Transactions are created by consuming existing UTXOs and producing new ones in their place.

Any input bitcoins not redeemed in an output is considered a transaction fee.



UTXO

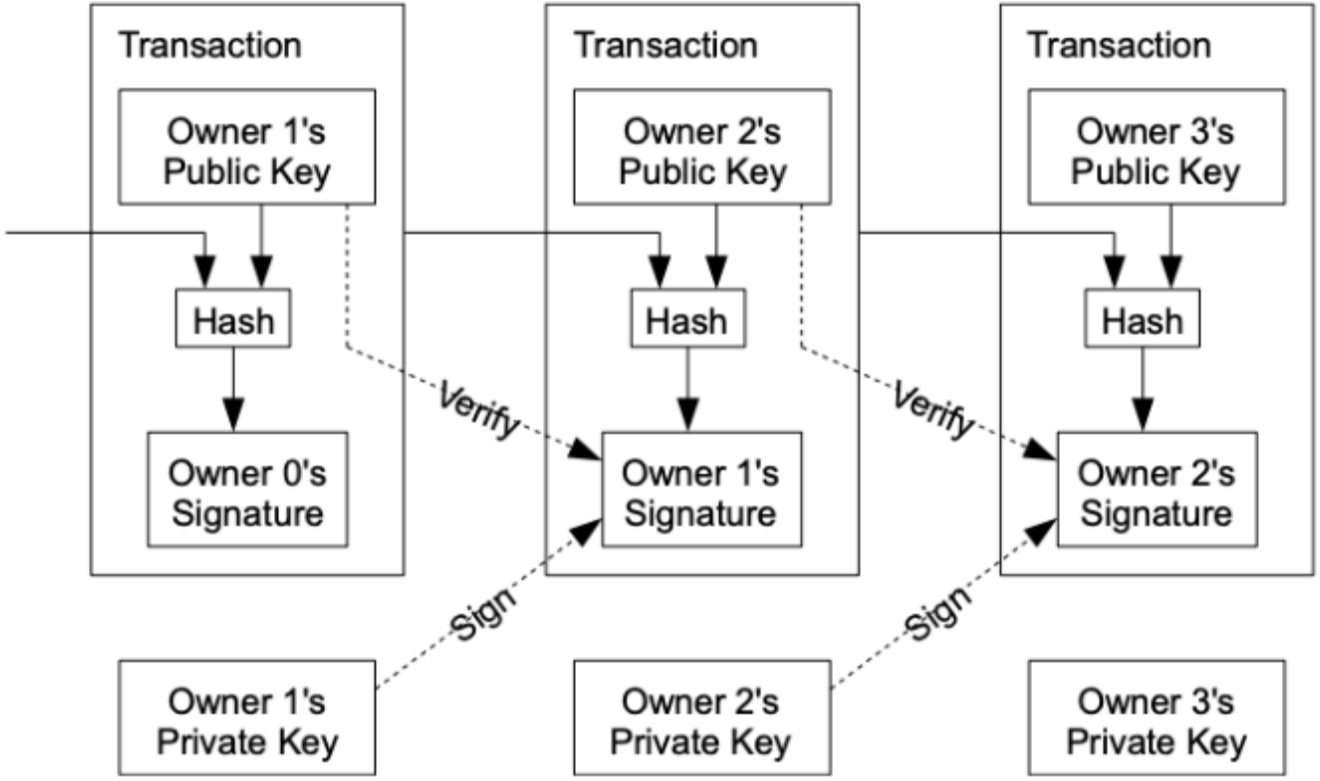
The global state of Bitcoin is all the currently unspent UTXO sets.



UTXO

Each token owner transfers a coin they own to someone else by:

- Generating the hash of a previous transaction and the public key (address) of the next owner.
- Digitally signing the hash.
- Adding these to the end of the coin



UTXO

On the protocol layer there are no accounts or wallets.

Coins are stored as a list of UTXOs.

Another way of referring to a balance of coins (e.g. BTC) controlled by a specific address.

A user's "balance" in the system is thus the total value of the unspent UTXO set for which the user has a private key capable of producing a valid signature.

UTXO - Validity Constraints

1. Every referenced input must be valid and not yet spent
2. The transaction must have a signature matching the owner of the input for every input
3. The total value of the inputs must equal or exceed the total value of the outputs

UTXO - Advantages

Simplicity. Spending a UTXO is all or nothing. Since UTXOs are uniquely referenced and completely consumed upon spending, there is no chance for a transaction to be replayed.

Transactions can be trivially **verified in parallel**. It is impossible for two transactions to affect the same UTXO. This is due to the stateless nature of UTXO transactions. Transactions do not refer to any input outside of the UTXOs consumed and corresponding signatures.

Privacy preserving behavior is encouraged in the UTXO model. Users are encouraged to generate a new address for every incoming transaction including change addresses. By using a new address each time, it is difficult to definitively link different coins to a single owner.

UTXO – Weaknesses

Since there is **no concept of an account** within Bitcoin, it is up to the wallet provider to manage a potential set of addresses and sum the corresponding balances.

Some **complex logic can not be realized**, and the programmability is poor. For complex logic or contracts requiring state preservation, it is difficult to implement, and the utilization of state space is relatively low.

When there are **more Inputs**, there will be **more validation**. Signature itself consumes CPU and storage space.

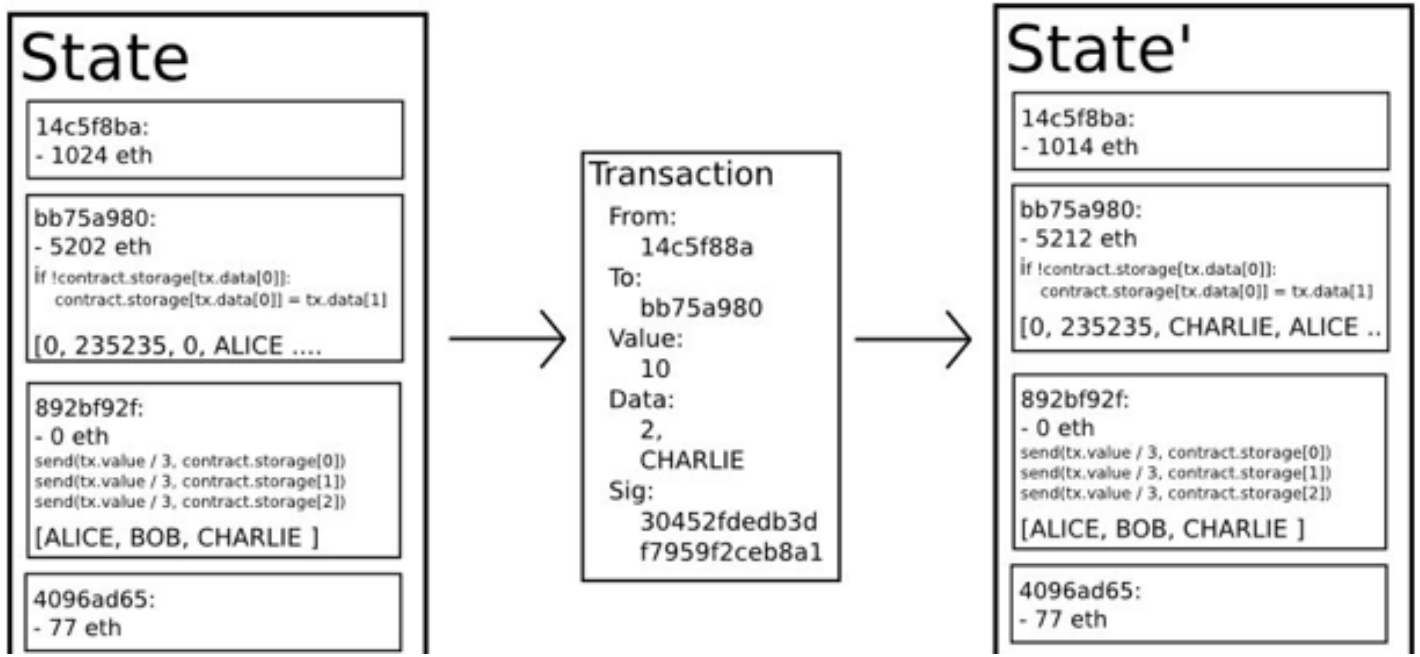
Account-Based Model

State stores a **list of accounts** where each account has a **balance**, as well as Ethereum-specific data (**code** and **internal storage**).

A transaction is valid if the sending account has enough balance to pay for it.

If the receiving account has code, the **code runs**, and **internal storage** may also be changed, or the code may even create **additional messages** to other accounts which lead to further debits and credits.

Account-Based Model



Account-Based Model - Advantages

Large space savings: if an account has 5 UTXO, then switching from a UTXO model to an account model would reduce the space requirements from

(20 + 32 + 8) * 5 = 300 bytes (20 for the address, 32 for the txid and 8 for the value)

to

20 + 8 + 2 = 30 bytes (20 for the address, 8 for the value, 2 for a nonce).

In reality savings are not nearly this massive because accounts need to be stored in a Patricia tree but they are nevertheless large.

Additionally, transactions can be smaller (eg. 100 bytes in Ethereum vs. 200-250 bytes in Bitcoin) because every transaction need only make one reference and one signature and produces one output.

Account-Based Model - Advantages

Greater fungibility: since there is no blockchain-level concept of the source of a specific set of coins, it becomes less practical, both technically and legally, to institute a redlist/blacklisting scheme and to draw a distinction between coins depending on where they come from.

Simplicity: easier to code and understand, especially once more complex scripts become involved.

Constant light client reference: light clients can at any point access all data related to an account by scanning down the state tree in a specific direction.

Account-Based Model - Weaknesses

In order to prevent replay attacks, every transaction must have a “**nonce**”.

An account keeps track of the nonces used and only accepts a transaction if its nonce is 1 after the last nonce used.

This means that even no-longer-used accounts can never be pruned from the account state.

Tries in Ethereum

All of the merkle tries in Ethereum use a Merkle Patricia Trie. Merkle Patricia Trie is a data structure that stores key-value pairs, just like a map. It combines the properties of three foundational structures:

1. Trie (Prefix Tree): A tree-like structure used for efficient key-value retrieval, leveraging common prefixes.
2. Radix Trie: Optimizes space usage by compressing nodes with single-child paths, reducing the overhead of storage and lookup operations.
3. Merkle Tree: Integrates cryptographic hashing, enabling succinct proofs of existence or inclusion, enhancing security and integrity.

The resulting hybrid, the Merkle Patricia Trie, provides an optimized balance of storage efficiency, verification speed, cryptographic integrity, and decentralized consensus. From a block header there are 3 roots from 3 of these tries.

1. stateRoot
2. transactionsRoot
3. receiptsRoot

State Trie

There is one global state trie, and it updates over time. In it, a path is always: **sha3(ethereumAddress)** and a value is always: **rlp(ethereumAccount)**

More specifically an ethereum account is a 4 item array of

[nonce, balance, *storageRoot*, codeHash]

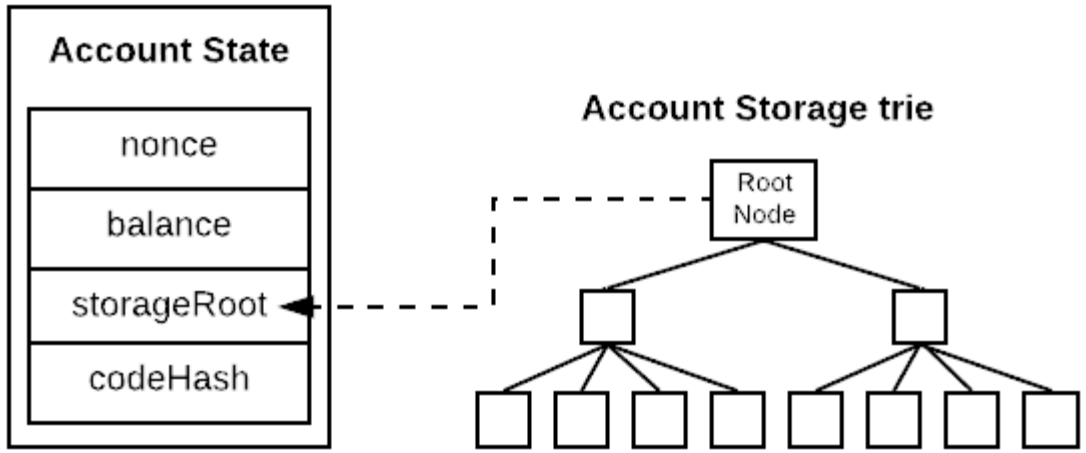
storageRoot is the root of another patricia trie.

Storage Trie

Storage trie is where all **contract data** lives.

There is a separate storage trie for each account.

To calculate a ‘path’ in this trie first understand how solidity organizes a variable’s position. (more info)



```
nonce: Transaction nonce is a sequence number of transactions sent
from a given address.

Gas Price: price you are offering to pay

Gas Limit: Gas Limit is a limit of the amount of ETH the sender is
willing to pay for the transaction

Recipient: The recipient is the destination of Ethereum address.

Value: The value field represents the amount of ether/wei from the
sender to the recipient.

Data: Data field is for contract related activities such as
deployment or execution of a contract.

v,r,s: This field is components of an ECDSA digital signature of the
originating EOA.
```

Transactions Trie

There is a separate transactions trie for every block.

A path here is: **rlp(transactionIndex)**.

transactionIndex is its index within the block it's mined.

The ordering is mostly decided by a miner so this data is unknown until mined.

After a block is mined, the transaction trie never updates.

```
blockHash: String, 32 Bytes - hash of the block where this
transaction was in.

blockNumber: Number - block number where this transaction was in.

transactionHash: String, 32 Bytes - hash of the transaction.

transactionIndex: Number - integer of the transactions index position
in the block.

from: String, 20 Bytes - address of the sender.

to: String, 20 Bytes - address of the receiver. null when its a
contract creation transaction.

cumulativeGasUsed: Number - The total amount of gas used when this
transaction was executed in the block.

gasUsed: Number - The amount of gas used by this specific transaction
alone.

contractAddress: String - 20 Bytes - The contract address created, if
the transaction was a contract creation, otherwise null.

logs: Array - Array of log objects, which this transaction generated.

status : String - '0x0' indicates transaction failure , '0x1'
indicates transaction succeeded.

cite: https://ethereum.stackexchange.com/questions/6531/structure-of-
```

Receipts Trie

Every block has its own Receipts trie.

A path here is: **rlp(transactionIndex)**

transactionIndex is its index within the block it's mined.

After a block is mined, the Receipts trie never updates.

← ***This is a field transaction receipt***

Radix tree

Every node looks as follows:

[i0, i1 ... in, value]

i0 ... in represent the symbols of the alphabet (often binary or hex)

value is the terminal value at the node

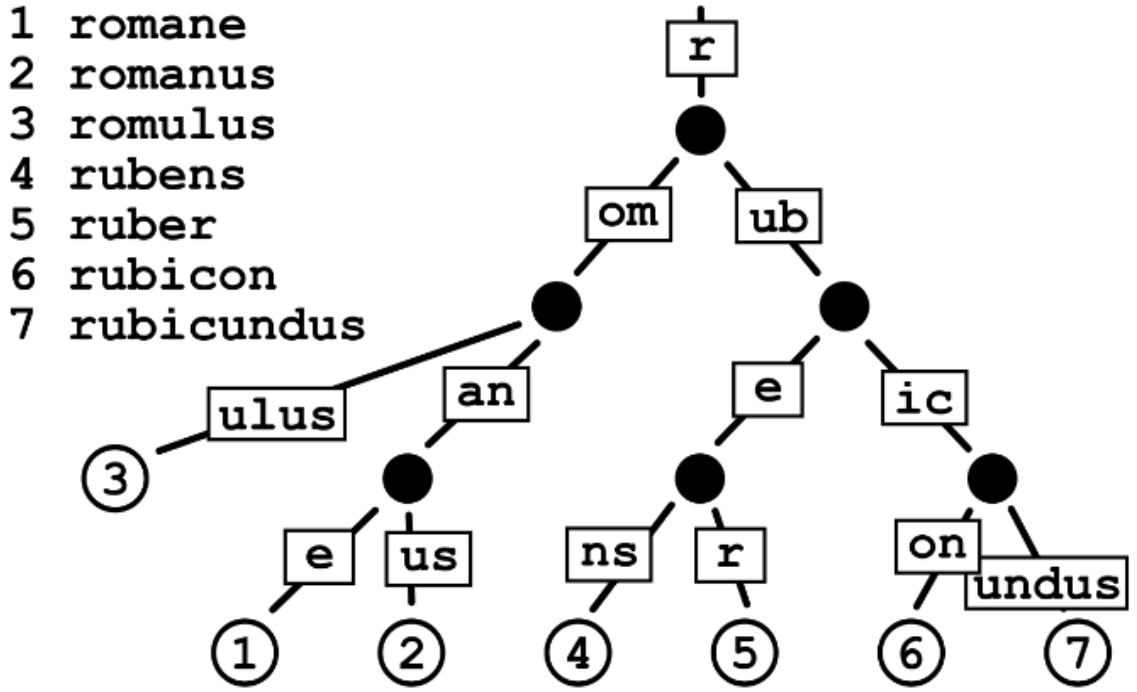
Values in i0 ... in slots are either NULL or pointers to (in our case, hashes of) other nodes.

Radix tree

[i0, i1 ... in, value]

Example: find the value currently mapped to “dog” in the trie

1. Convert dog into letters of the alphabet (giving 64 6f 67)
2. Descend down the trie following that path until at the end of the path you read the value.
3. Once you followed the path: root -> 6 -> 4 -> 6 -> 15 -> 6 -> 7, you look up the value of the node that you have and return the result.



Radix tree

Radix tries have one major limitation: they are inefficient.

If you want to store just one (path,value) binding where the path is (in the case of the ethereum state trie), 64 characters long, you will need over a kilobyte of extra space to store one level per character, and each lookup or delete will take the full 64 steps.

The **Patricia trie** introduced solves this issue.

Merkle-Patricia trie

Merkle Patricia tries solve the inefficiency issue by adding some extra complexity to the data structure.

A node in a Merkle Patricia trie is 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]

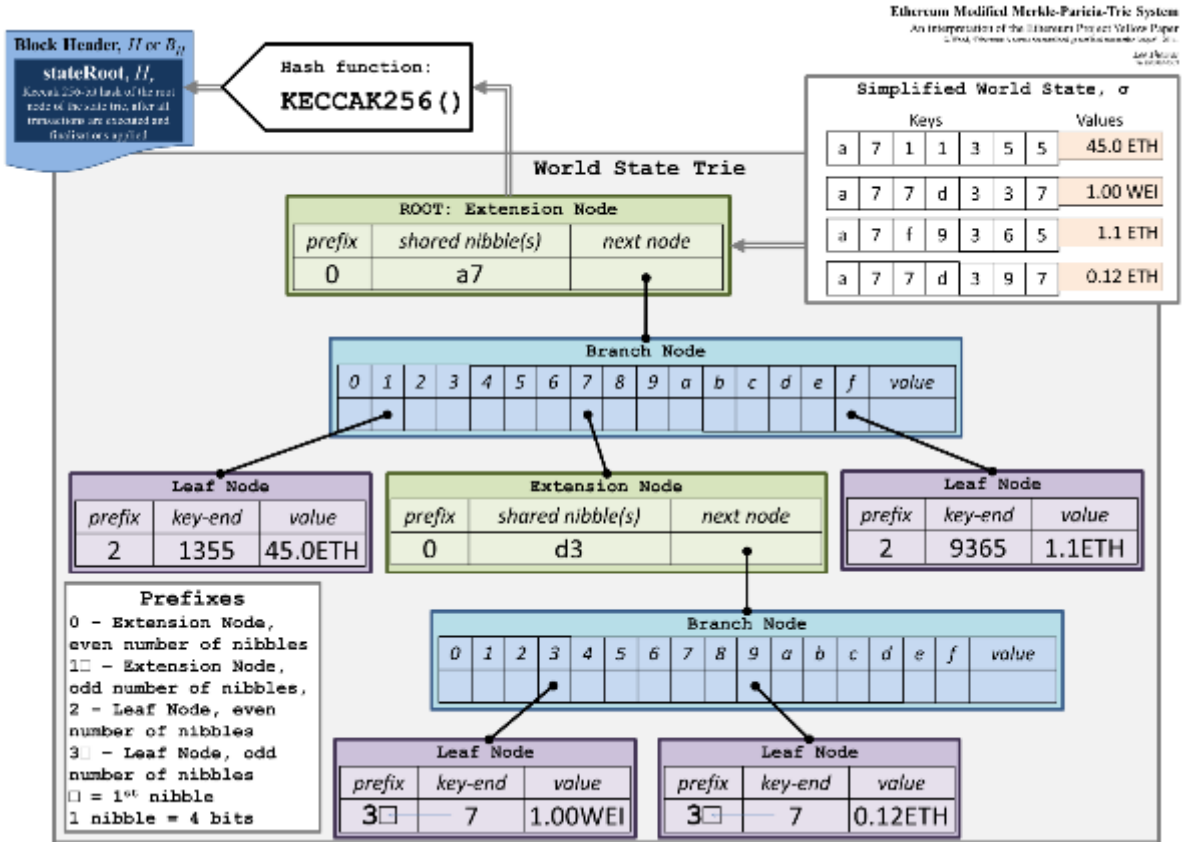
Merkle-Patricia trie

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]

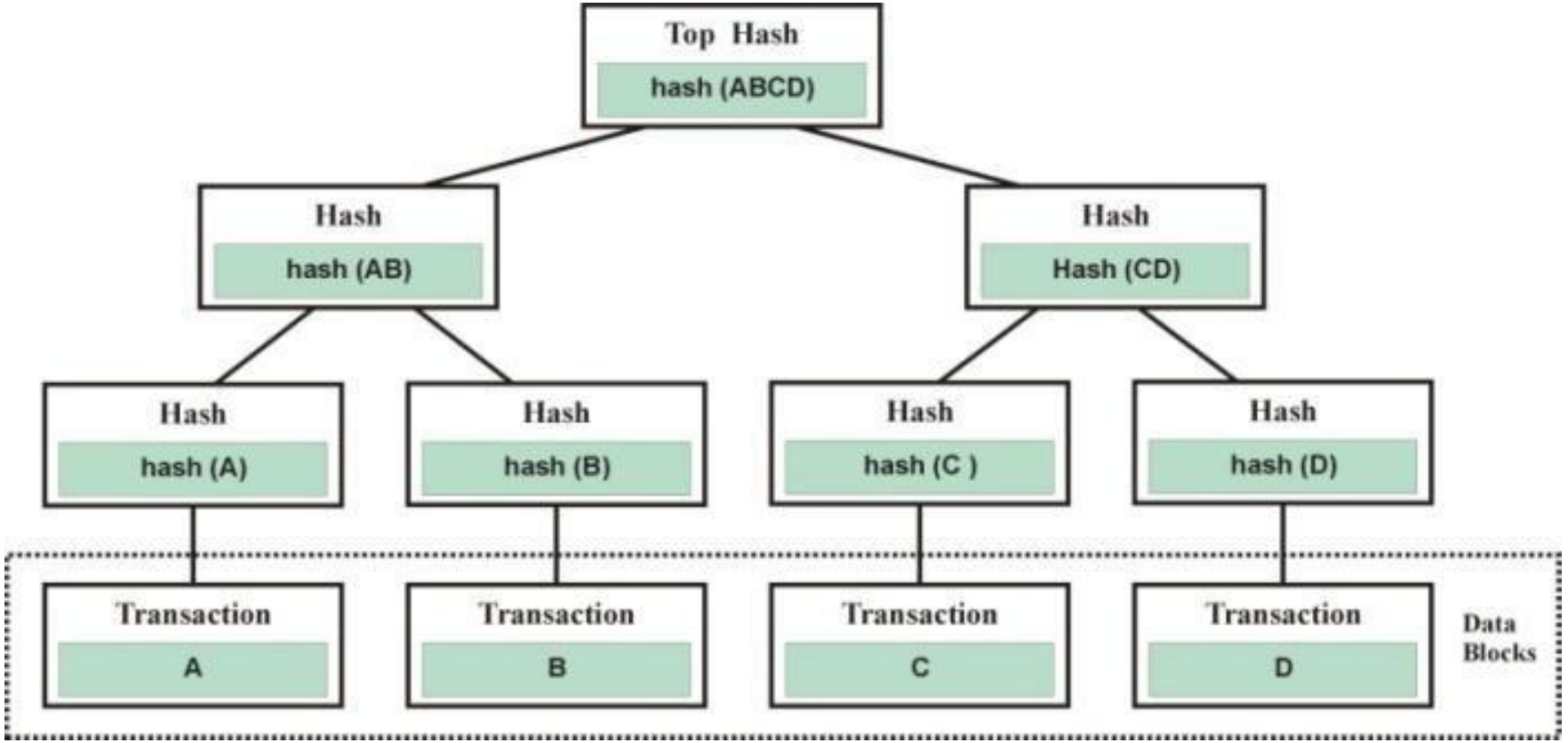
An extension node of the form [encodedPath, key] can shortcut the descent, where encodedPath contains the “partial path” to skip ahead, and the key is for the next db lookup.

In a leaf node, the situation above occurs and also the “partial path” to skip ahead completes the full remainder of a path. In this case value is the target value itself.

Merkle-Patricia trie



Merkle tree


Top Hash
hash (ABCD)
Hash
hash (AB)
Hash
Hash (CD)
Hash
hash (A)
Hash
hash (B)
Hash
hash (C)
Hash
hash (D)
Transaction
A
Transaction
B
Transaction
C
Transaction
D
Data
Blocks

Merkle-Patricia trie

Figure F		Example		
Row	**Node Type**	**Path Length**	**Path Before Encoding (IN HEX)**	**Path After Encoding (IN HEX)**
A	Extension	EVEN LENGTH	[0, 1, 2, 3, 4, 5]	`00 01 23 45`
B	Extension	ODD LENGTH	[1, 2, 3, 4, 5]	`11 23 45`
C	Leaf (has terminator)	EVEN LENGTH	[0, f, 1, c, b, 8, 10]	`20 0f 1c b8`
D	Leaf (has terminator)	ODD LENGTH	[f, 1, c, b, 8, 10]	`3f 1c b8`