sub0.1

# Substrate Storage

Deep dive.

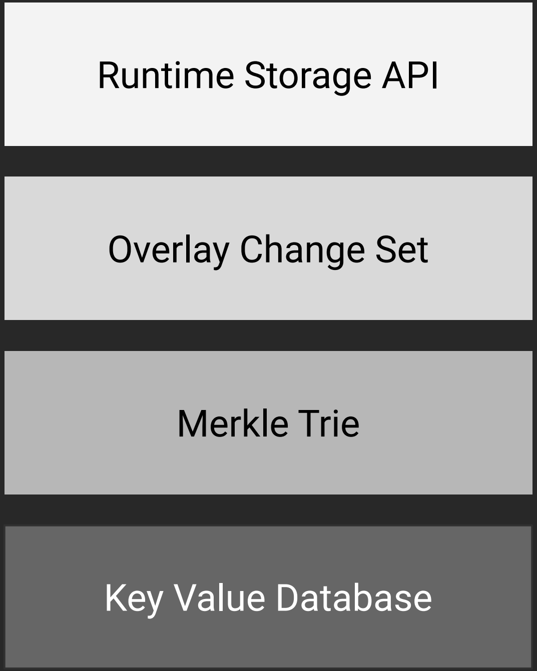
**Shawn Tabrizi**

Software developer @ Parity Technologies Ltd.

shawntabrizi@parity.io | @shawntabrizi

www.substrate.dev

# Abstractions of Substrate Storage



Runtime Storage API

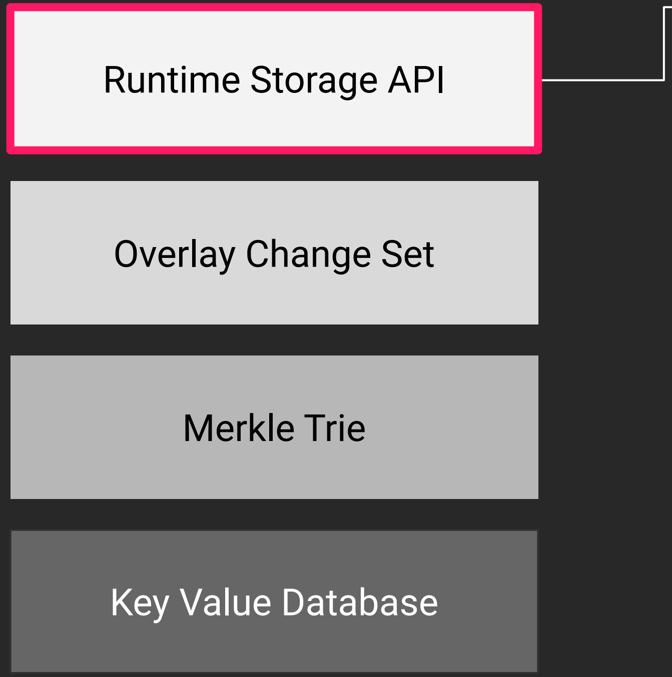
Overlay Change Set

Merkle Trie

Key Value Database

High Level Overview

# Abstractions of Substrate Storage



Runtime Storage API

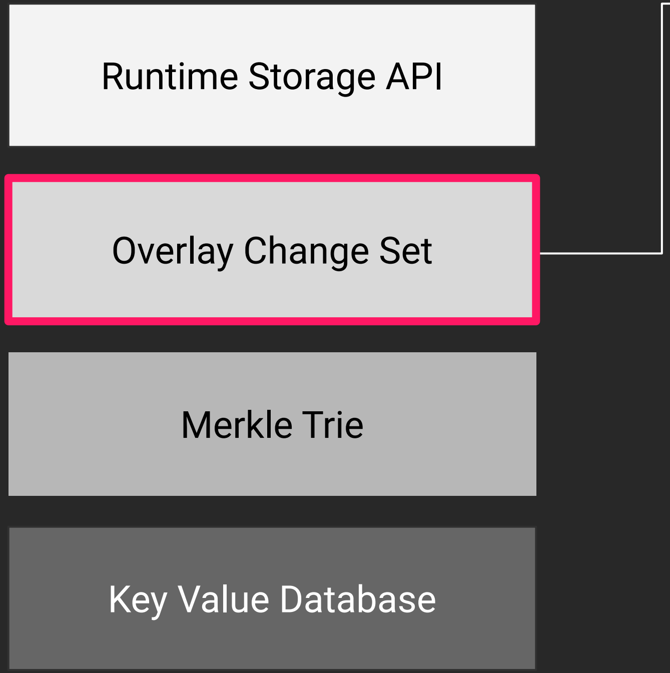
Overlay Change Set

Merkle Trie

Key Value Database

- sp-io can write to storage with a given key + value
- Easy APIs generated through decl_storage! macro
- StorageValue, StorageMap, StorageDoubleMap, etc...

# Abstractions of Substrate Storage



Runtime Storage API

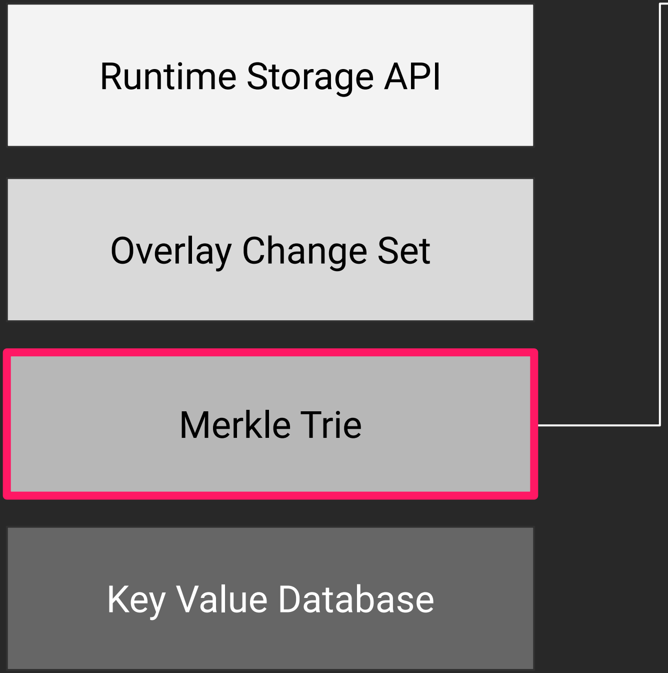
Overlay Change Set

Merkle Trie

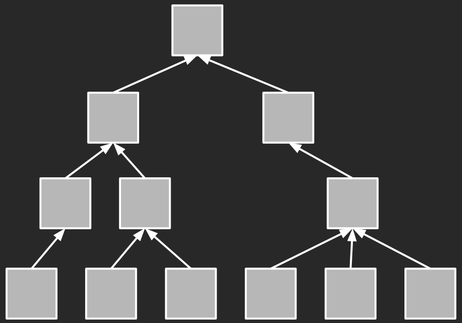
Key Value Database

- Stages changes to the underlying DB.
- Overlay changes are committed once per block.
- Two kinds of changes:
  - Prospective Changes - what may happen.
  - Committed Changes - what will happen.

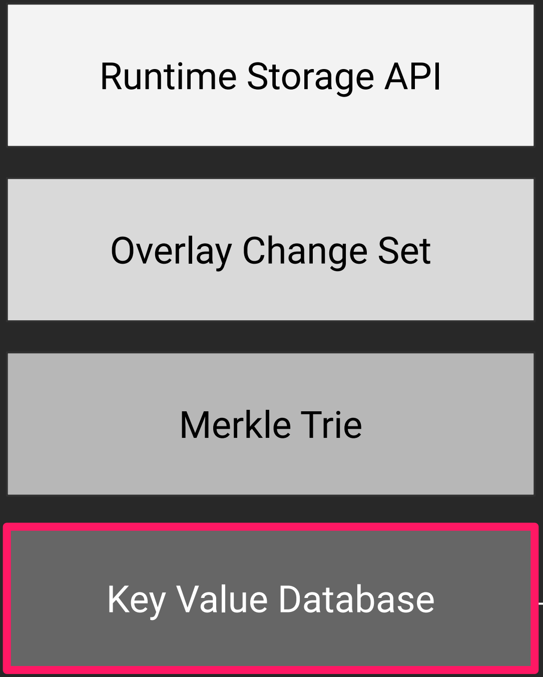
# Abstractions of Substrate Storage



- a.k.a. HashDB
- paritytech/trie
- Data structure on top of KVDB
- Arbitrary Key and Value length
- Nodes are Branches or Leaves



# Abstractions of Substrate Storage



Runtime Storage API

Overlay Change Set

Merkle Trie

Key Value Database

- a.k.a. KVDB
- Implemented with RocksDB
- Hash -> Vec<u8>
- Substrate: Blake2 256

| Key (Hash 256) | Value (Vec<u8>) |
|---|---|
| 0x0fd923ca5e7... | [00] |
| 0x92cdf578c47... | [01] |
| 0x31237cdb79... | [02] |
| 0x581348337b... | [03] |

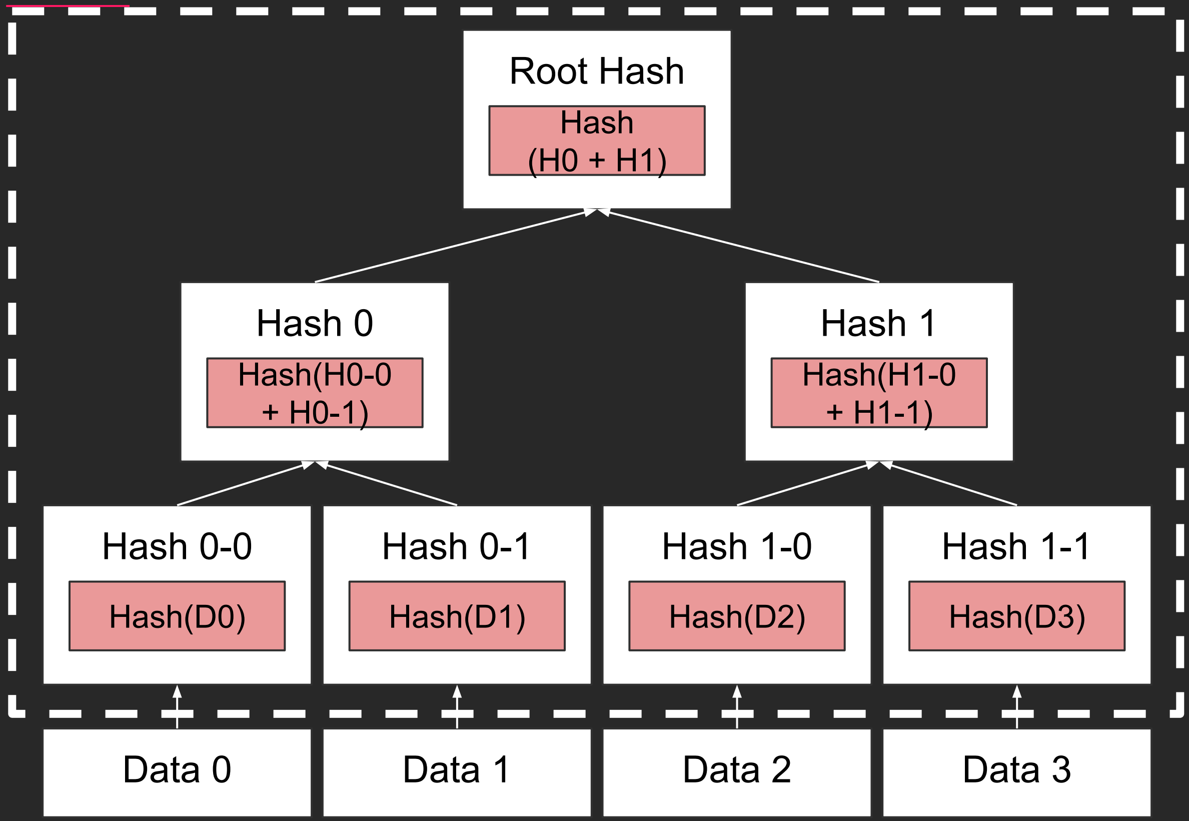
# Two Kinds of Keys!

- Trie key path
- KVDB key hash

Don’t worry, we will come back to this...

# Substrate uses a
# Base-16 Patricia Merkle Trie

# Merkle Tree



## Root Node

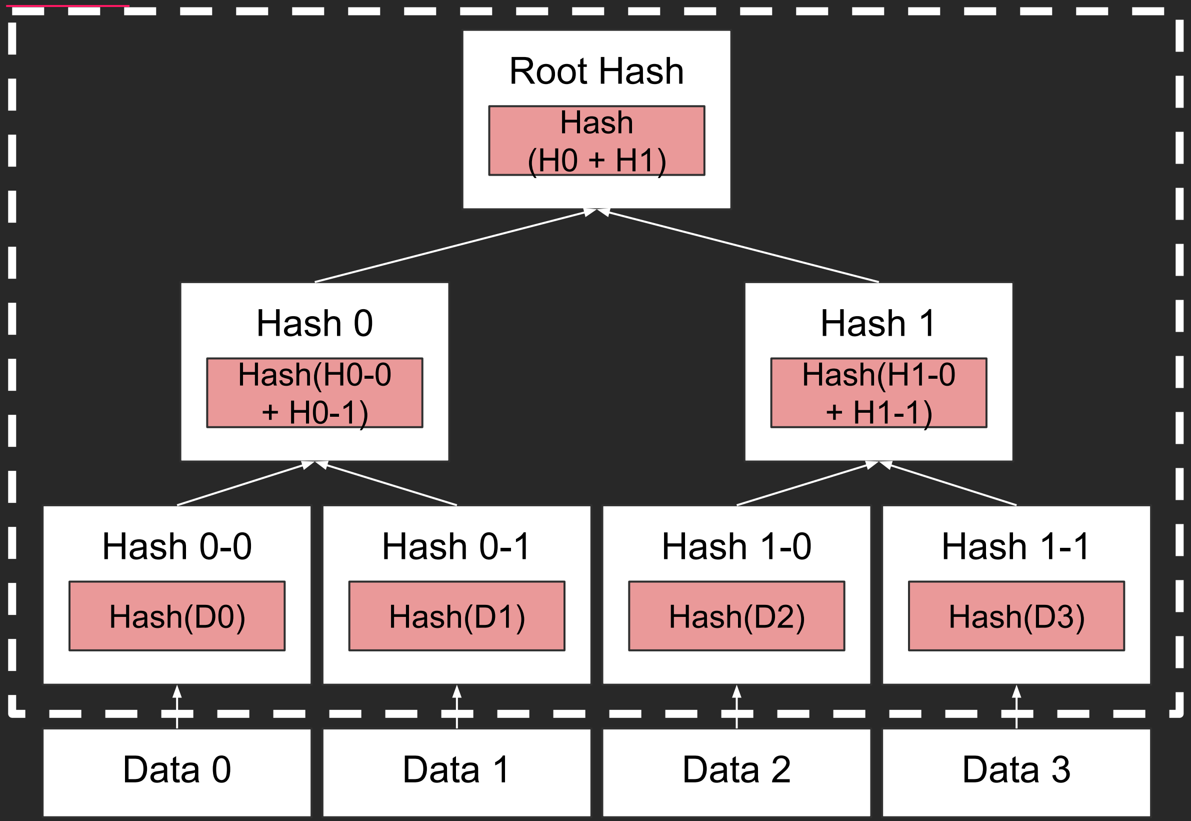
Can be used to verify two trees are the same.

## Branch Nodes

## Leaf Nodes

parity

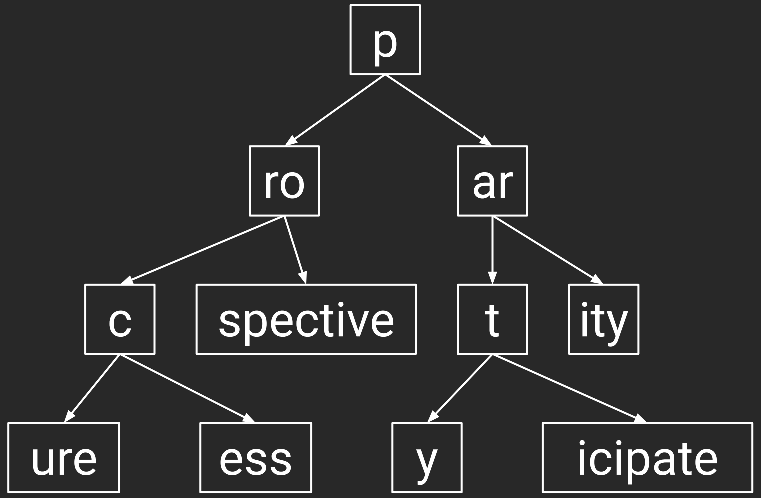
# Merkle Tree



Merkle tree allows you to more easily prove that some data exists within the tree with a “Merkle Proof”.

More about that later.

# Patricia Trie



p

ro

ar

c

spective

t

ity

ure

ess

y

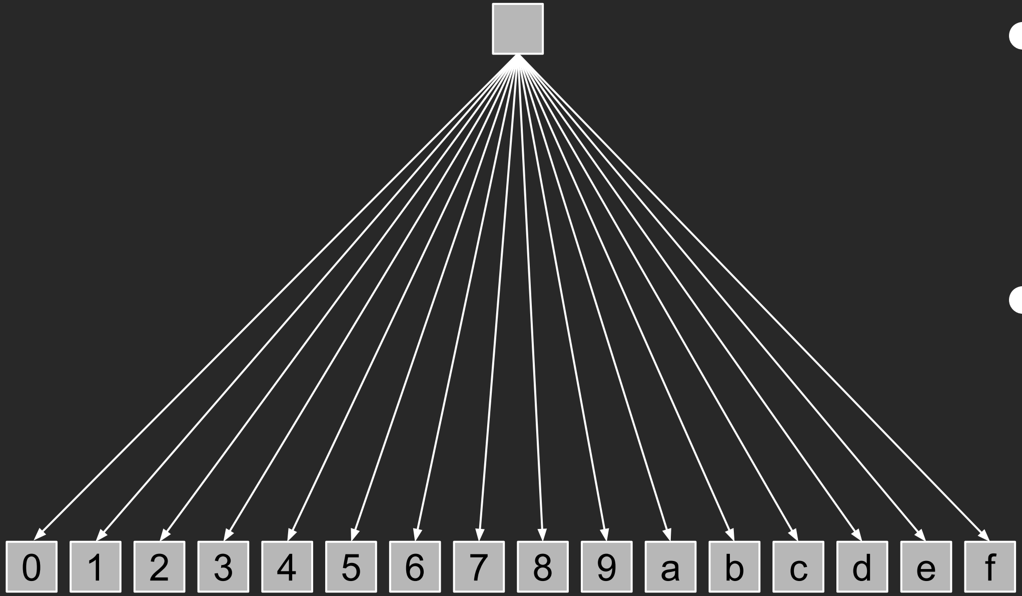
icipate

| | | |
|---|---|---|
| 1. parity | 2. participate | 3. party |
| 4. process | 5. procure | 6. prospective |

- Position in the tree defines the associated key.

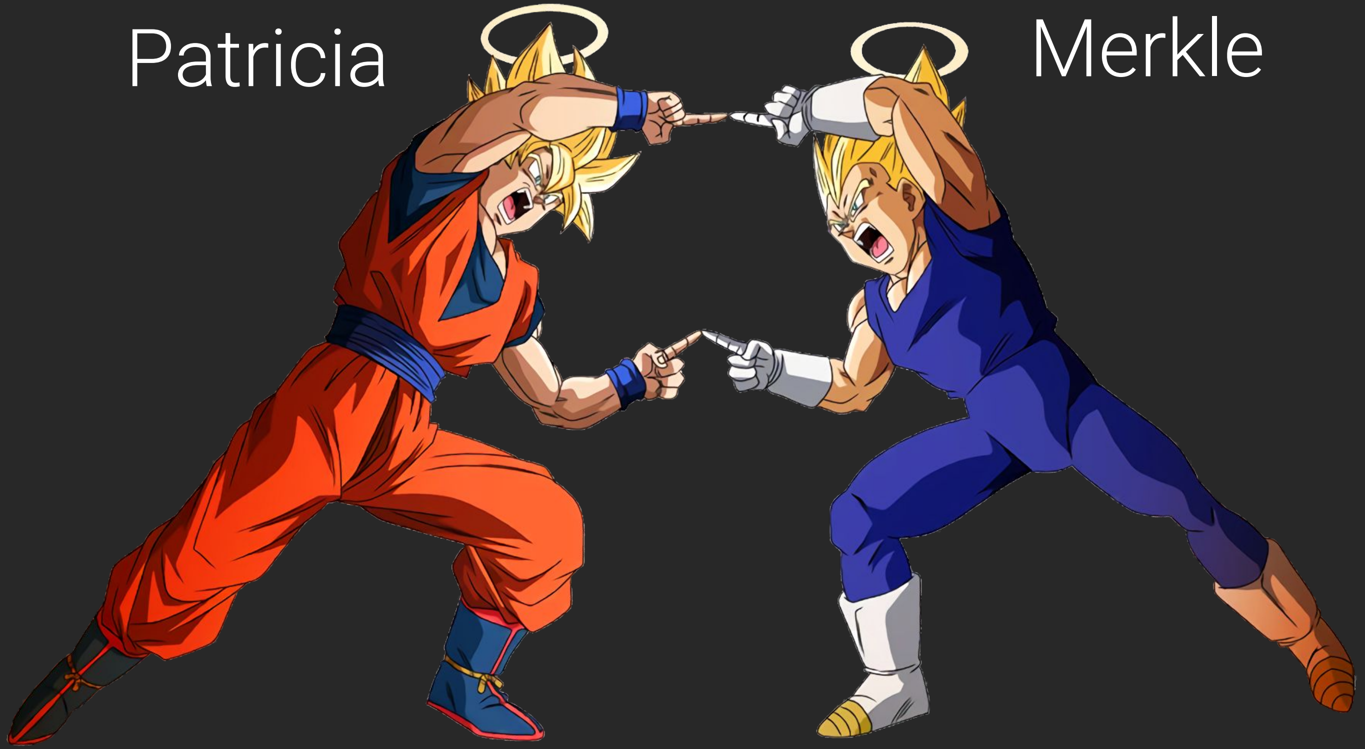
- Space optimized for elements which share a prefix.

# Beyond Binary Trees



- Branches can have more than two children.
- Everything is the same, just scaled up.

A single hex character is called a “nibble”.



## Creation of the Patricia Merkle Trie

# Let’s get visual.

# What we will be working with...

## Literal KVDB Table

| Key | Value |
|---|---|
| 0x8f35a27d9... | [BRANCH] |
| 0x2ebcd78e8... | [LEAF 00] |
| 0x27434bcd0... | [BRANCH w/ VAL 01] |
| 0x802c9c18c... | [LEAF 02] |
| 0x986d278c5... | [LEAF 03] |

## Types of Nodes

| Prefix | Type |
|---|---|
| 00 | Empty |
| 01 | Leaf |
| 10 | Branch w/o value |
| 11 | Branch w value |

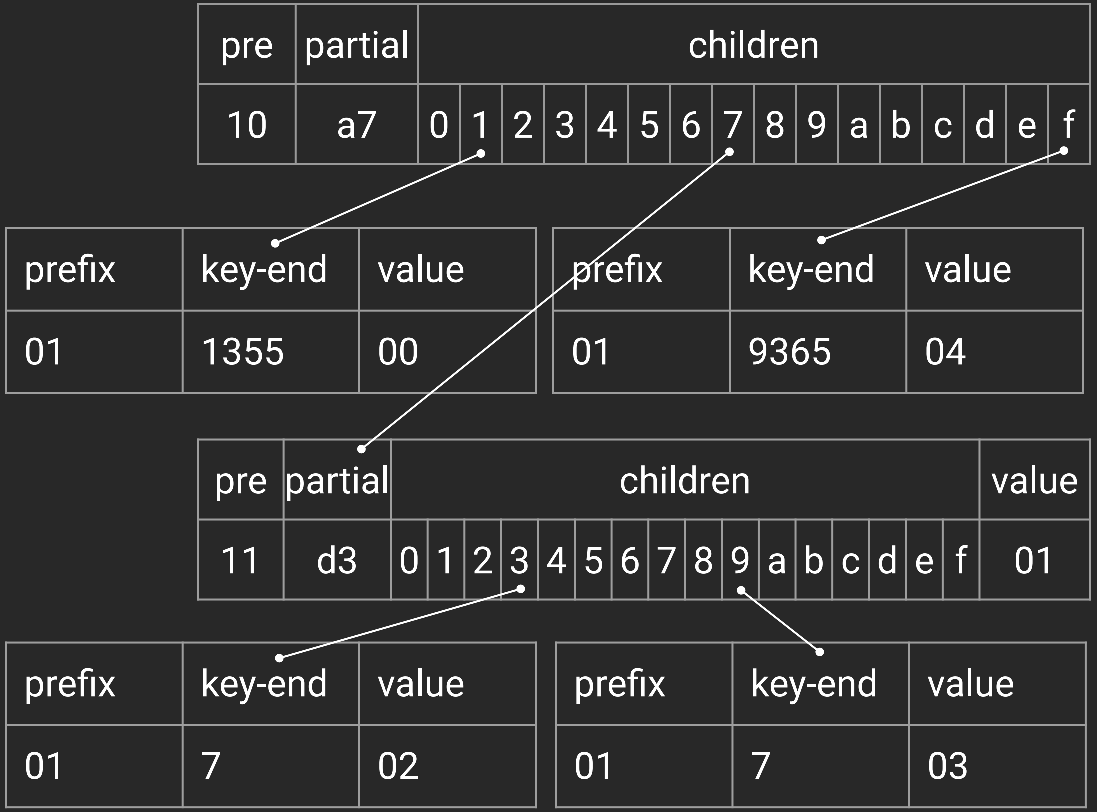
## Virtual Trie Table

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |

## Node Structure

| Trie Node | | | |
|---|---|---|---|
| header | key | children | value |

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

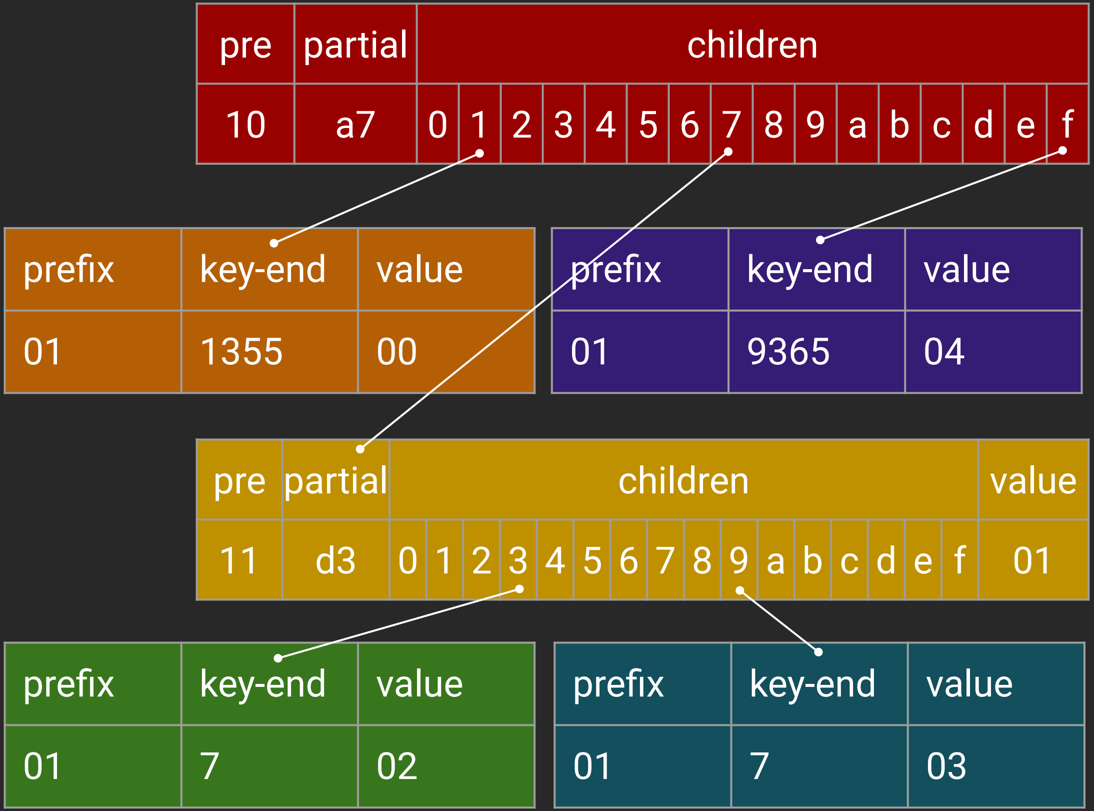
| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

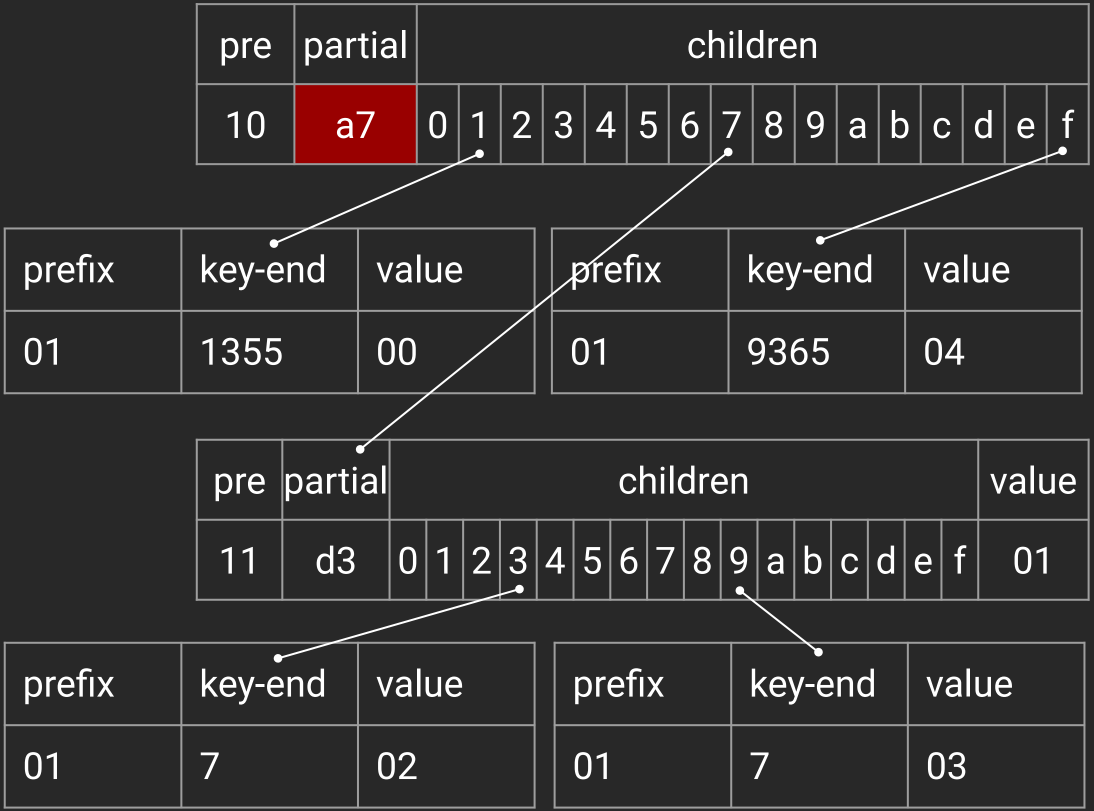
| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

All nodes are present.

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

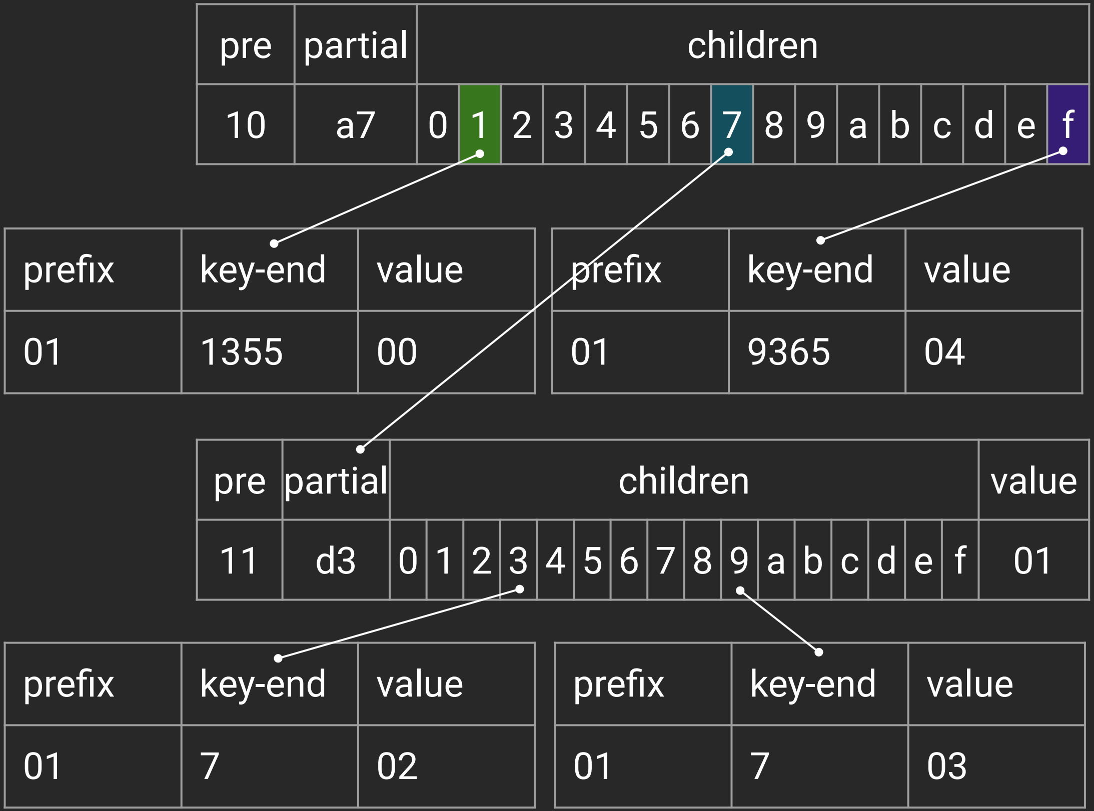
| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

Nodes with a shared path are children of a branch.

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

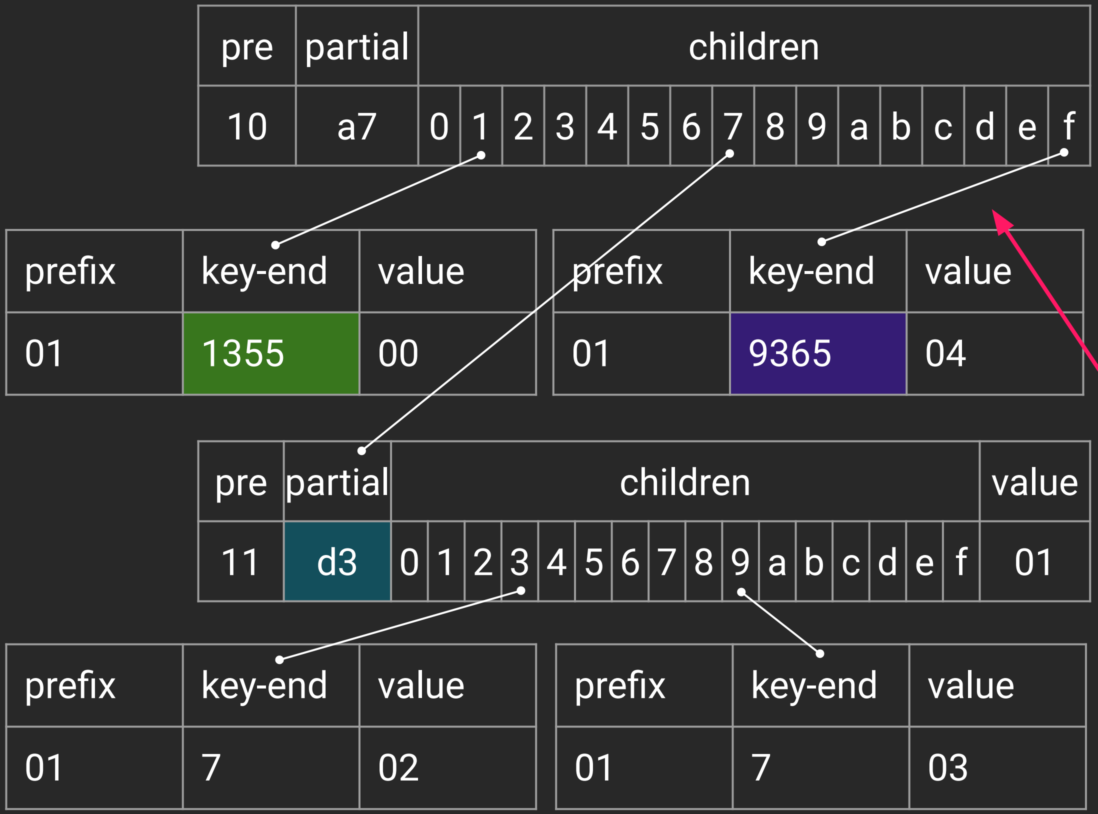
| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

You can then progress by looking at the children of the branch.

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

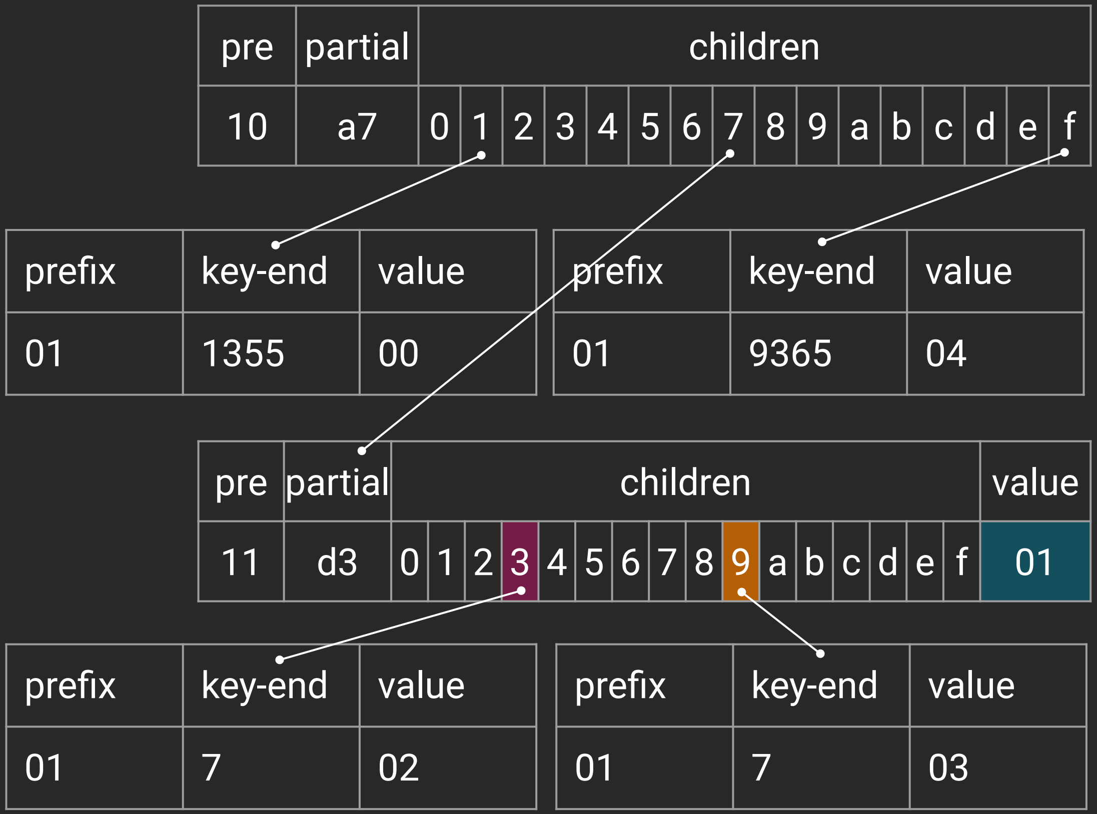
| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

This is a KVDB look up!

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

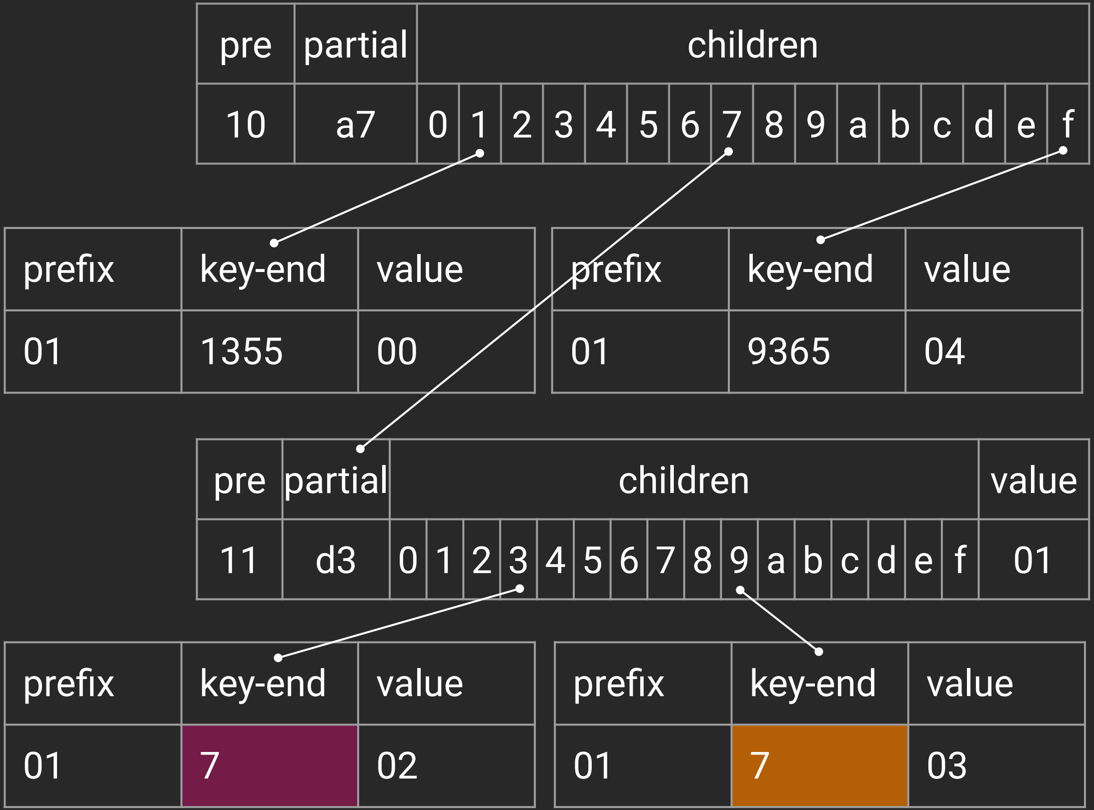
| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

You can have a branch which also contains a value!

# Visual of the Substrate State Trie



| pre | partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

| prefix | key-end | value |
|---|---|---|
| 01 | 9365 | 04 |

| pre | partial | children | | | | | | | | | | | | | | | | value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | d3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | 01 |

| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 02 |

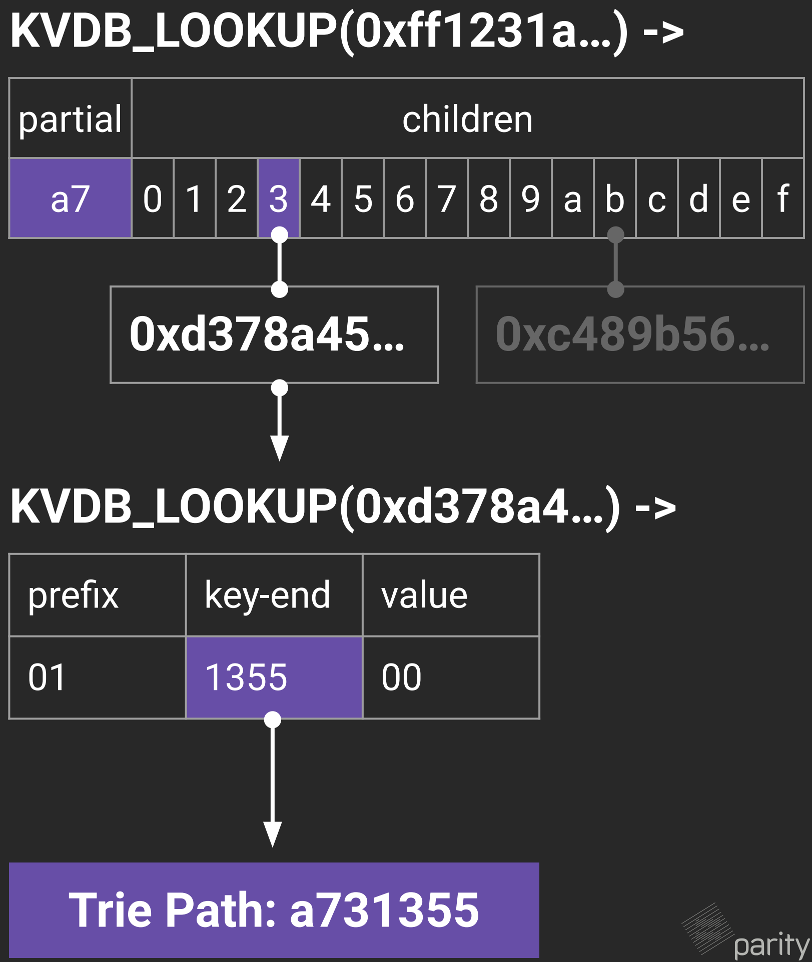
| prefix | key-end | value |
|---|---|---|
| 01 | 7 | 03 |

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

You reach the end when there are no more branches.

# What you just saw

- Patricia provides the trie path.



KVDB_LOOKUP(0xff1231a...) ->

| partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

0xd378a45...

0xc489b56...

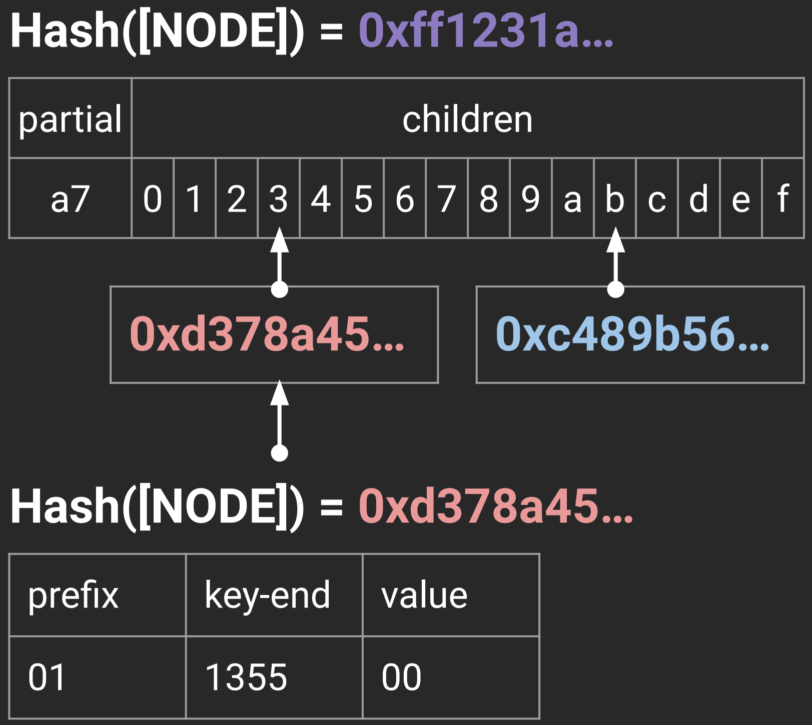
KVDB_LOOKUP(0xd378a4...) ->

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

Trie Path: a731355

# What you just saw

- Patricia provides the trie path.
- Merkle provides the recursive hashing of children nodes into the parent.



Hash([NODE]) = 0xff1231a...

| partial | children | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |

0xd378a45...

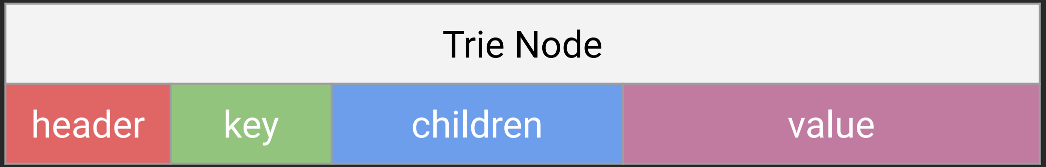
0xc489b56...

Hash([NODE]) = 0xd378a45...

| prefix | key-end | value |
|---|---|---|
| 01 | 1355 | 00 |

# Two Kinds of Keys!

1. Trie key path is set by you! (e.g. “:CODE”)
    - Arbitrary length!
    - Trie Node
        - Header Info
        - Key Info
        - Possible Children
        - Possible Value
2. KVDB key = Hash([Trie Node])

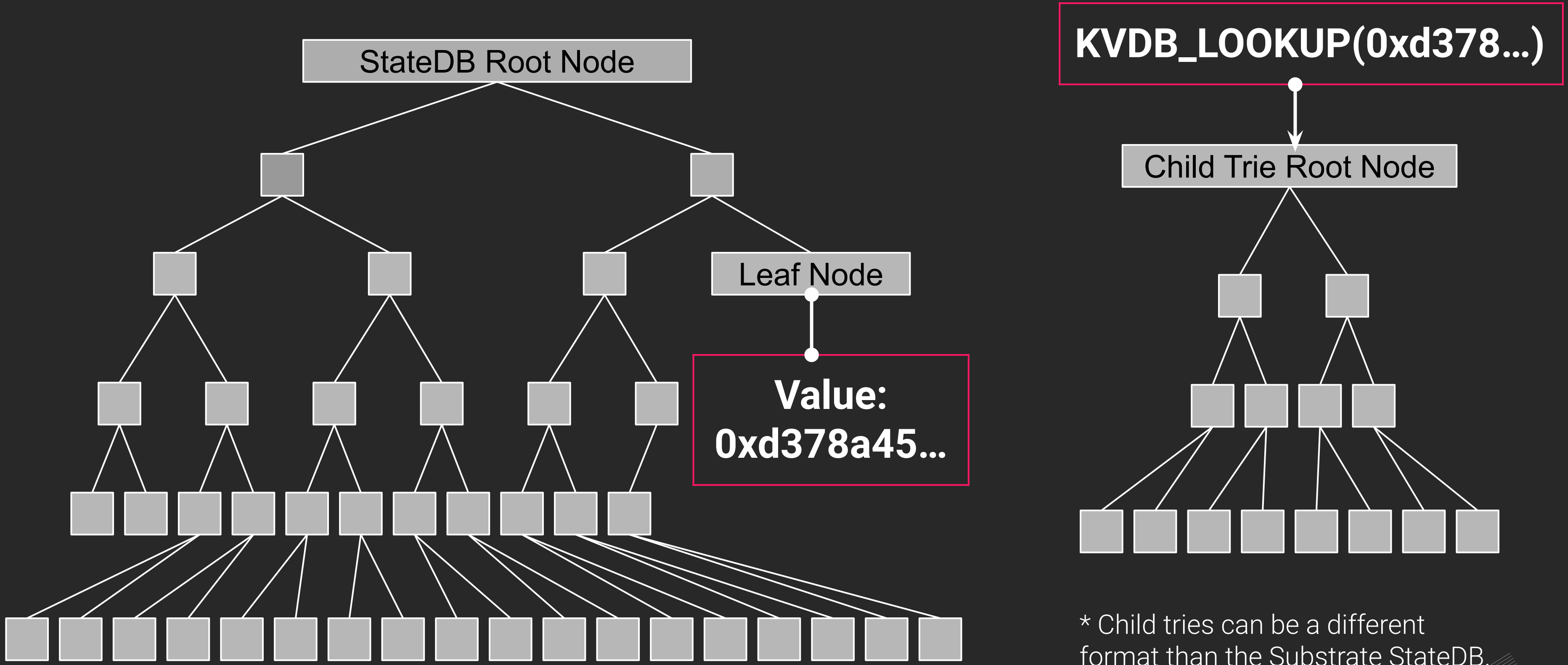


Trie Node

header key children value

# But wait… there’s more.

# Child Trie



* Child tries can be a different format than the Substrate StateDB.

# Prefix Trie

| Trie Key Path | | | | | | | Value |
|---|---|---|---|---|---|---|---|
| a | 7 | | | | | | [BRANCH] |
| a | 7 | 1 | 1 | 3 | 5 | 5 | [LEAF 00] |
| a | 7 | 7 | d | 3 | | | [BRANCH w/ VAL 01] |
| a | 7 | 7 | d | 3 | 3 | 7 | [LEAF 02] |
| a | 7 | 7 | d | 3 | 9 | 7 | [LEAF 03] |
| a | 7 | f | 9 | 3 | 6 | 5 | [LEAF 04] |

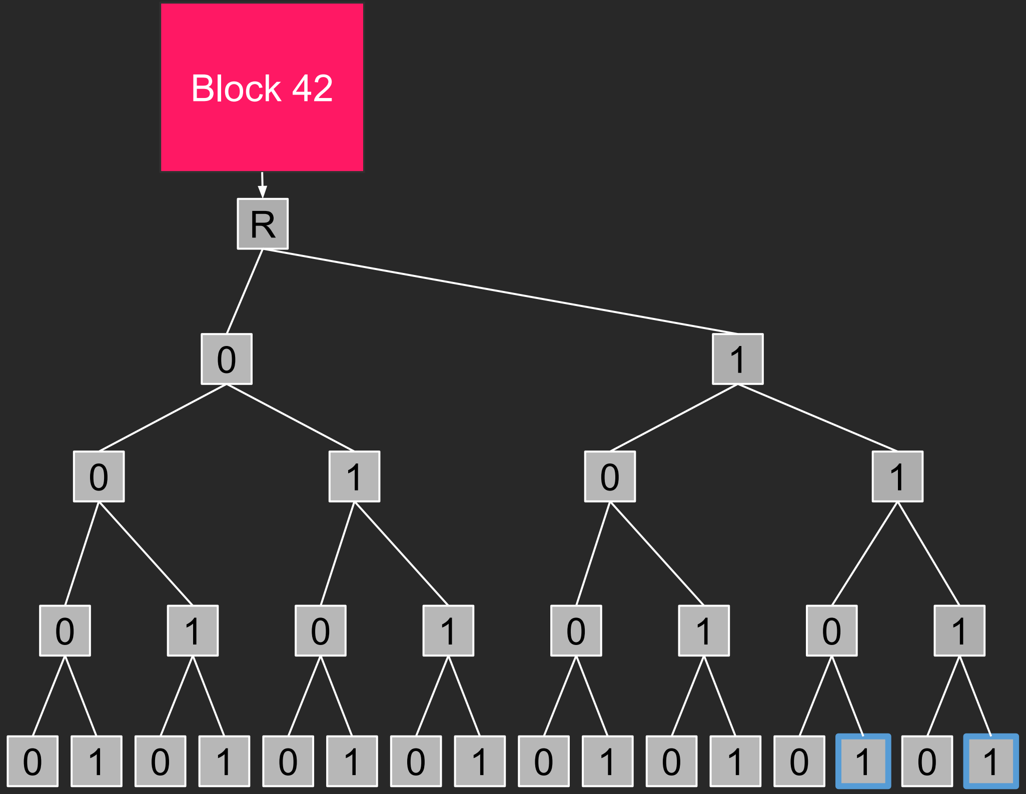
- Similar to Child Trie, but you cannot get the Root Hash.
- Probably something temporary while we fix pruning issues with child trie.

# Runtime Storage Trie Path (NEW)

All modules use a prefix trie now! (Long term, they probably become a child trie.)

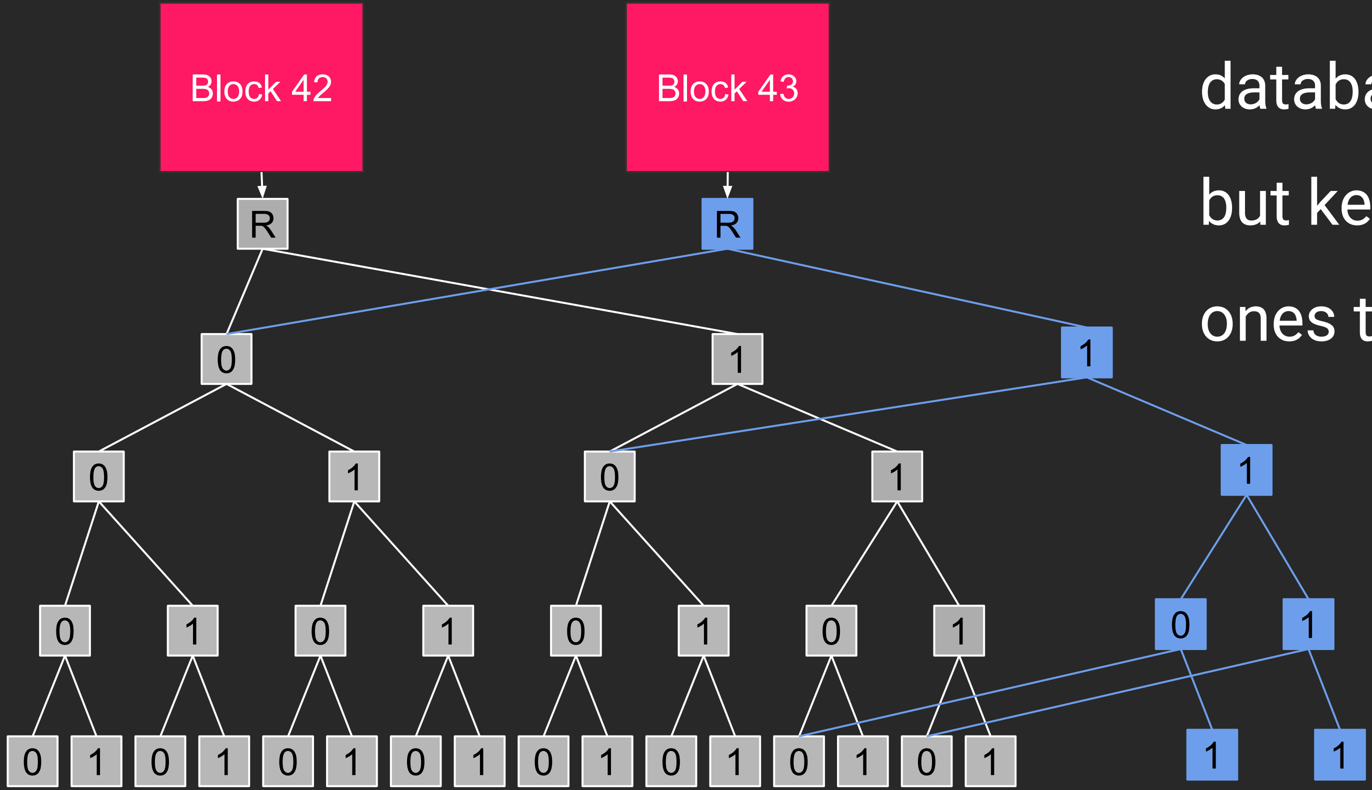
- Storage Value
  - `twox128(module) + twox128(storagename)`
- linked_map and map
  - `twox128(module) + twox128(storagename) + hasher(key)`
- linked_map head
  - `twox128(module) + twox128("HeadOf" + storagename)`
- double_map
  - `twox128(module) + twox128(storagename) + hasher(key1) + hasher(key2)`

# Pruning



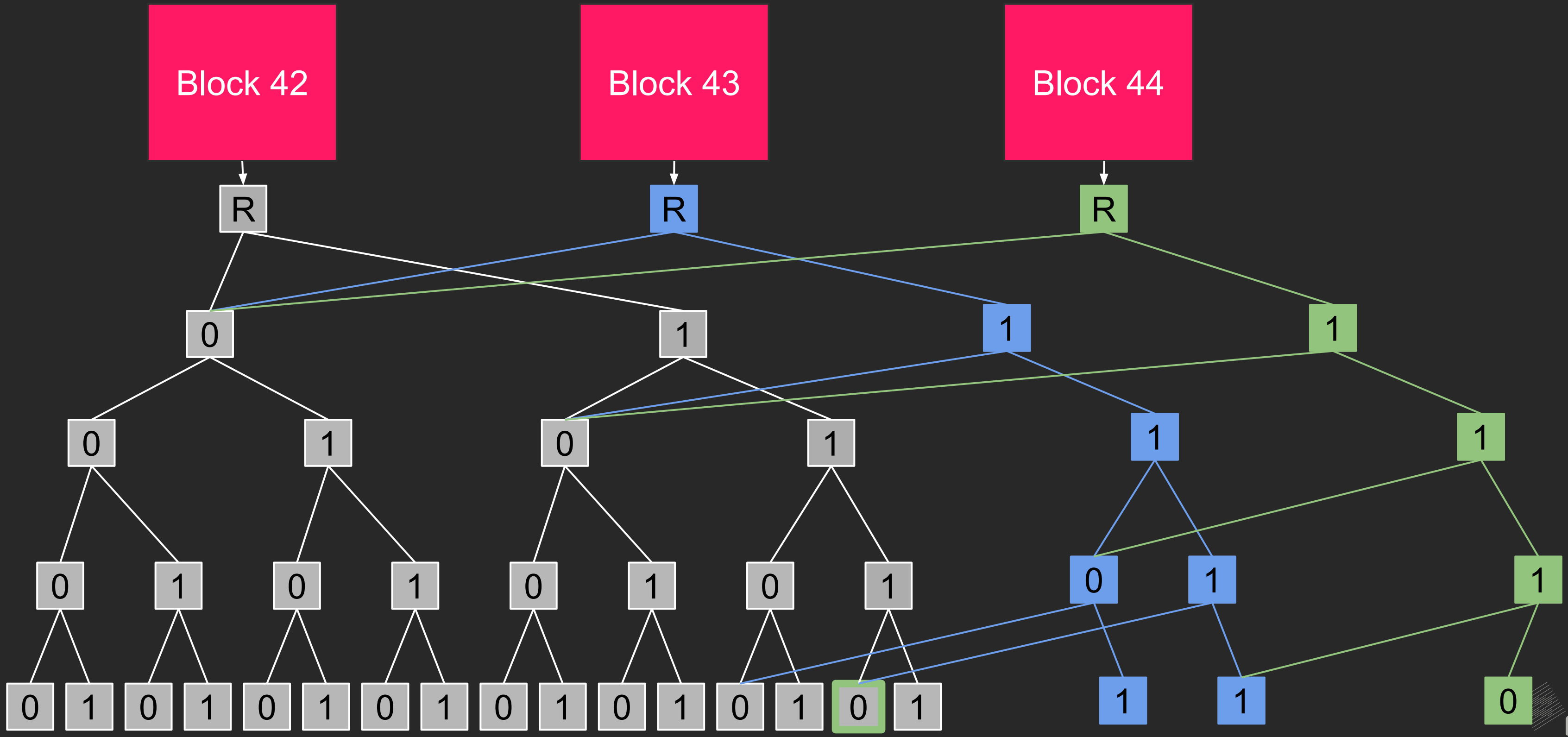
- For holding older block states, and cleaning it up.
- Let's update two values in this trie.

# Pruning



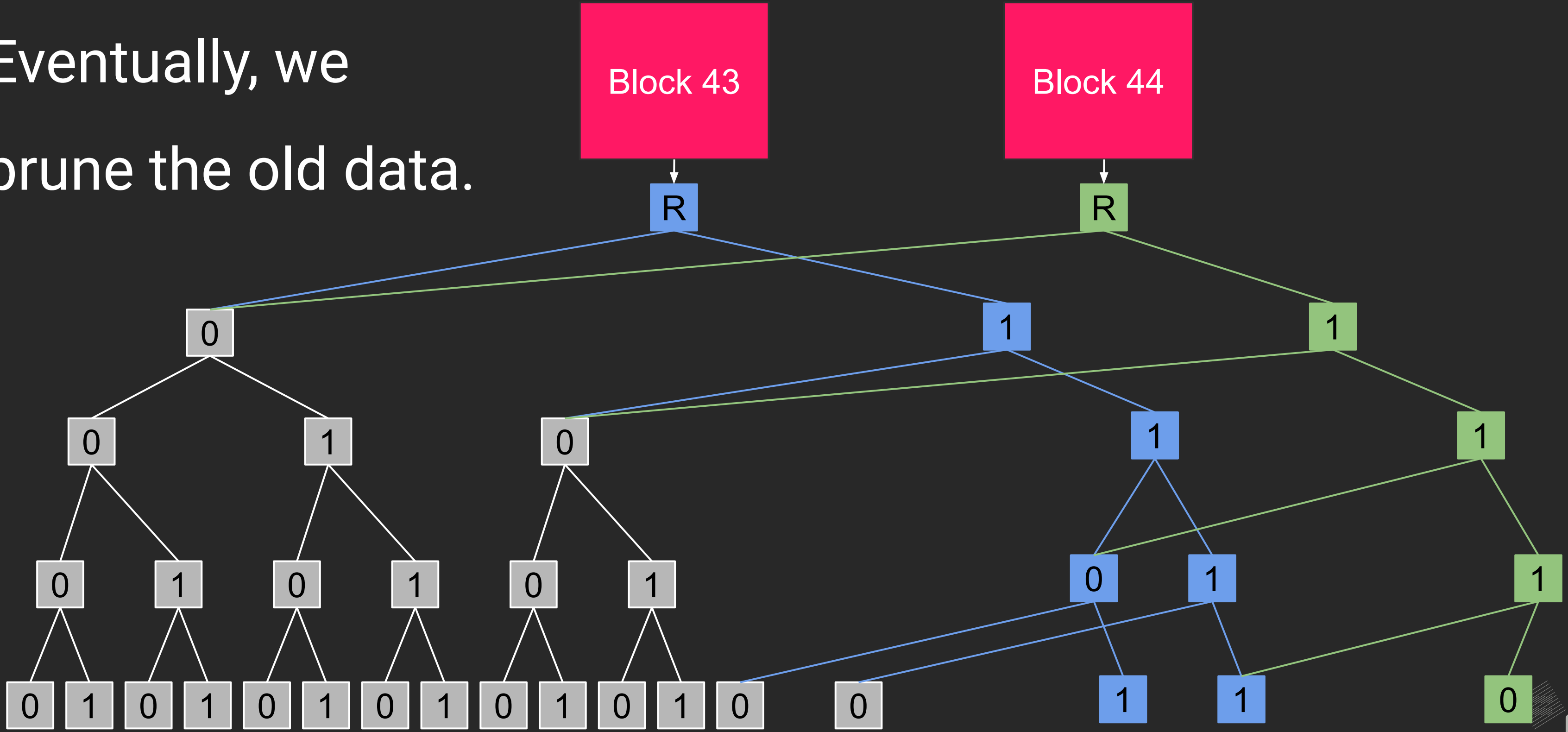
We create new database entries, but keep the old ones too!

# Pruning


Block 42
Block 43
Block 44
R
R
R
0
1
1
1
0
1
0
1
1
1
0
1
0
1
0
1
0
1
0
1
0
1
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
1
1
0

# Pruning

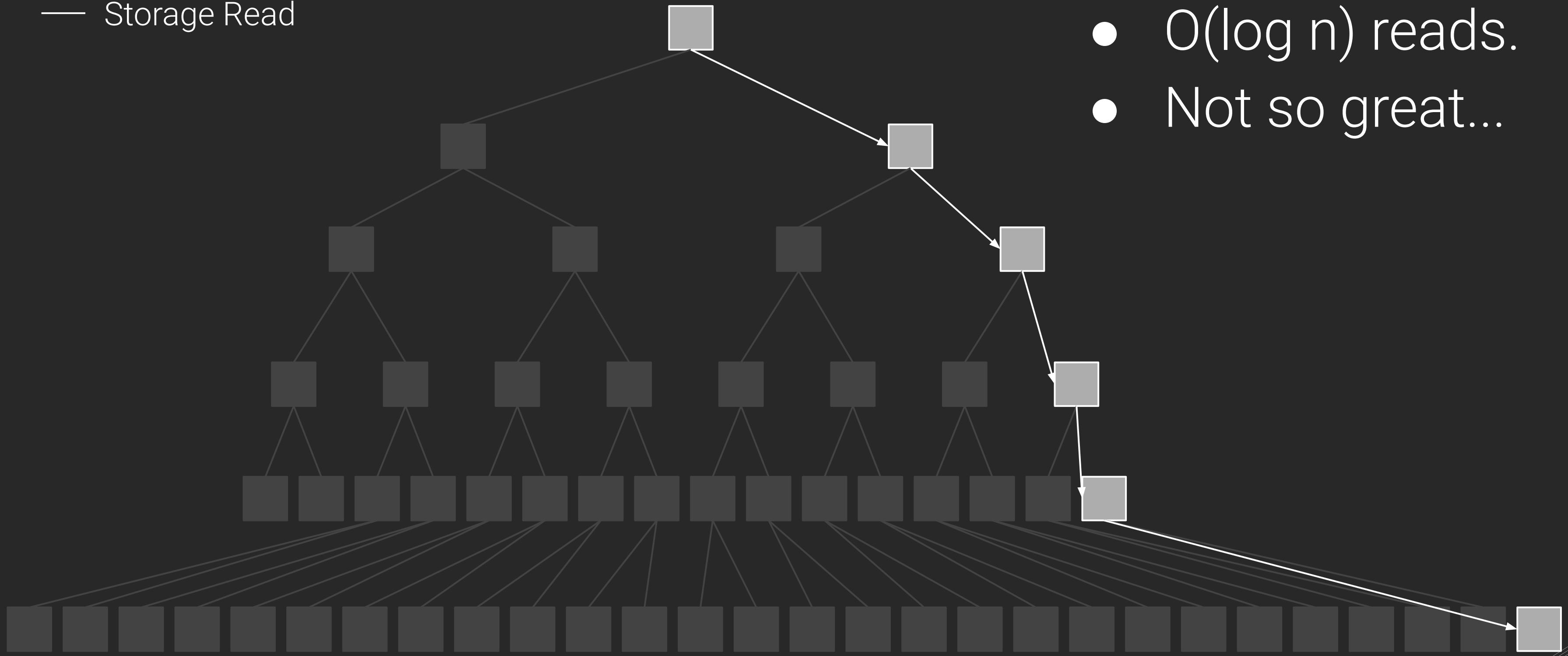
Eventually, we prune the old data.



# Merkle Trie Complexity

# Reading Data

— Storage Read



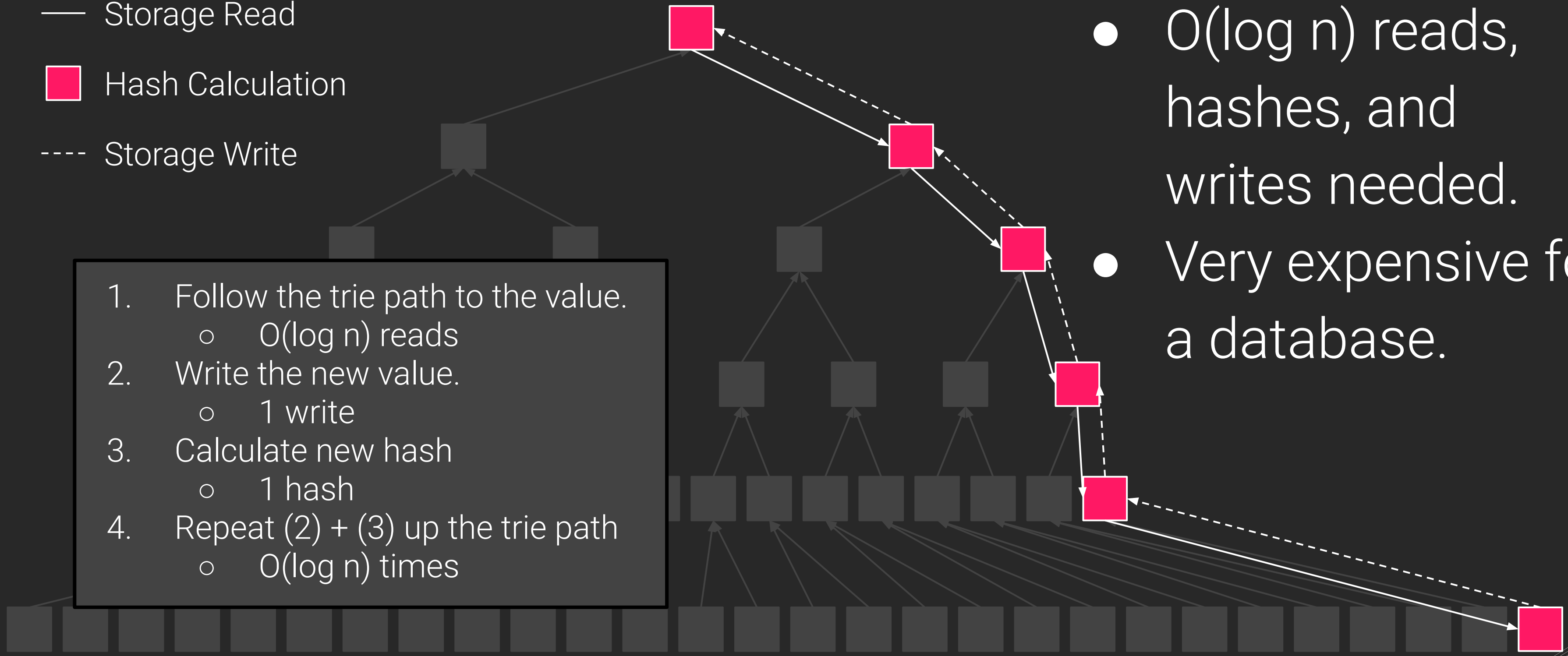
- O(log n) reads.
- Not so great...

# Writing Data

—— Storage Read

🟥 Hash Calculation

- - - - Storage Write



1. Follow the trie path to the value.
   - O(log n) reads
2. Write the new value.
   - 1 write
3. Calculate new hash
   - 1 hash
4. Repeat (2) + (3) up the trie path
   - O(log n) times

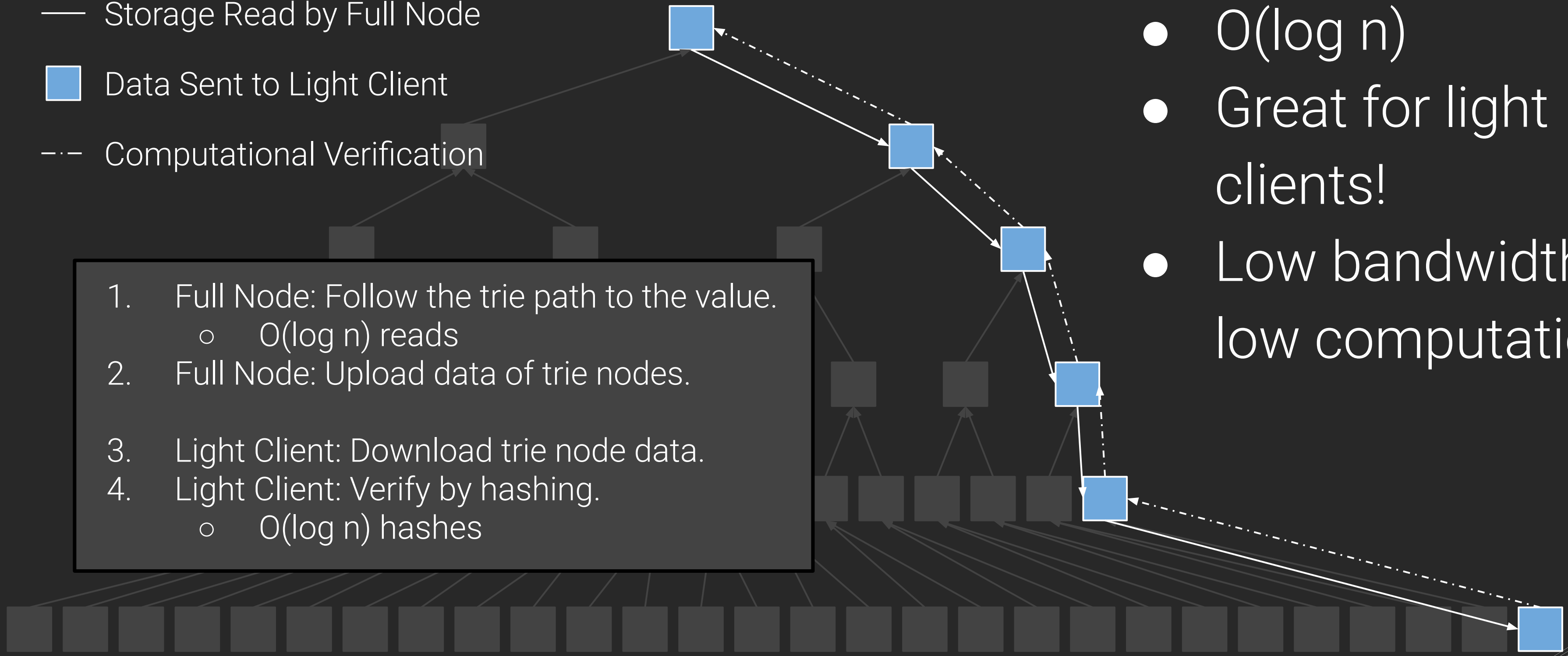
- O(log n) reads, hashes, and writes needed.
- Very expensive for a database.

# Merkle Proof

— Storage Read by Full Node

🟦 Data Sent to Light Client

–·– Computational Verification



1. Full Node: Follow the trie path to the value.
   - O(log n) reads
2. Full Node: Upload data of trie nodes.

3. Light Client: Download trie node data.
4. Light Client: Verify by hashing.
   - O(log n) hashes

- O(log n)
- Great for light clients!
- Low bandwidth, low computation

# Best Practices

In general...

Your fundamental goal is to minimize the amount of storage your runtime uses.

# You should only store consensus critical data in your runtime storage.

# Scenario: Decentralized Blog

- Runtime should be able to come to consensus about the content in a blog post…

★ Store the text on IPFS

★ Store the IPFS hash

➢ DO NOT store the text of the post in the storage!

# Struct or Multiple Values?

- Direct costs
  - O(log n) reads to get a value
  - O(log n) writes to update a value
- Indirect costs
  - Increase number of nodes (n)
  - Size of the value

## In general… store a struct:

★ Less reads/writes to update multiple values.

★ Less overall nodes in the trie.

★ Adding small items into large items accessed at the same time is essentially free!

➢ Less efficient for single value access.

➢ Upgrades requires storage migration.

# Define Your Storage Trie Path Generation

```
Foo: double_map hasher($hash1) u32, $hash2(u32) => u32
```

You can control the hashing algorithm used.
By default, these are configured to use Blake2 256.

**Final Trie Path:**

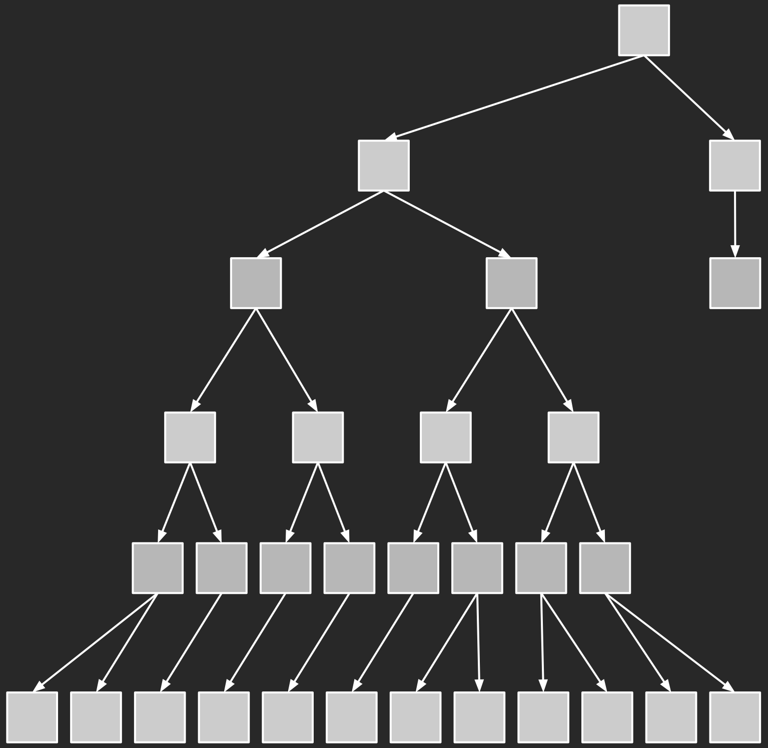
```
twox128(module) + twox128(storagename) + hasher(key1) + hasher(key2)
```

# XXHash vs Blake2

- What hashing algorithm should I use for trie path generation?

- Blake2
  - Cryptographic but slow…
  - Use when user can influence the input to the hash.
- XXHash (twox)
  - Non-cryptographic, but blazing fast…
  - When you (the runtime developer) controls this value, this is fine!

# Unbalanced Trie

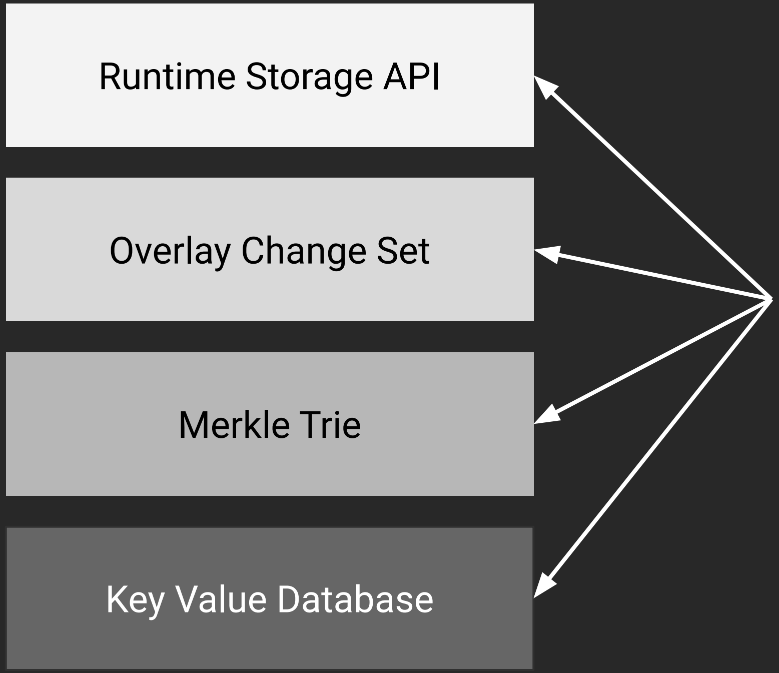


- Can happen if a user can influence the trie path.
- Operations are no longer O(log n)!

# Lists

- Vec: For storing a bounded number of values.
    - Good for when you need to change multiple values at a time (single read/write).
    - Enables iteration. Ex: The current validator set.
- Map: For storing an unbounded number of values.
    - Good for random access to data. Ex: User balances.
- Linked Map: For storing unbounded amount of data, but UI or an offchain worker needs to iterate on all the entries.
    - Ex: The list of nominators and their nominations.

# Abstractions of Substrate Storage



Runtime Storage API

Overlay Change Set

Merkle Trie

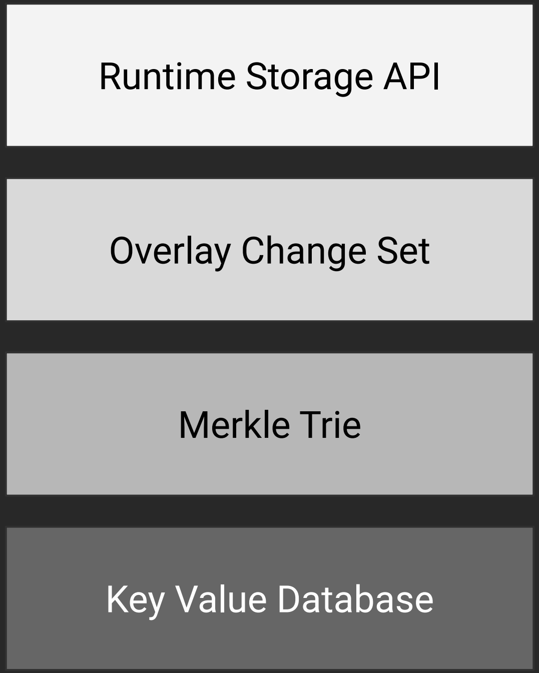
Key Value Database

Think about all the layers when you are writing to Substrate storage.

# Questions?

shawntabrizi@parity.io

@shawntabrizi

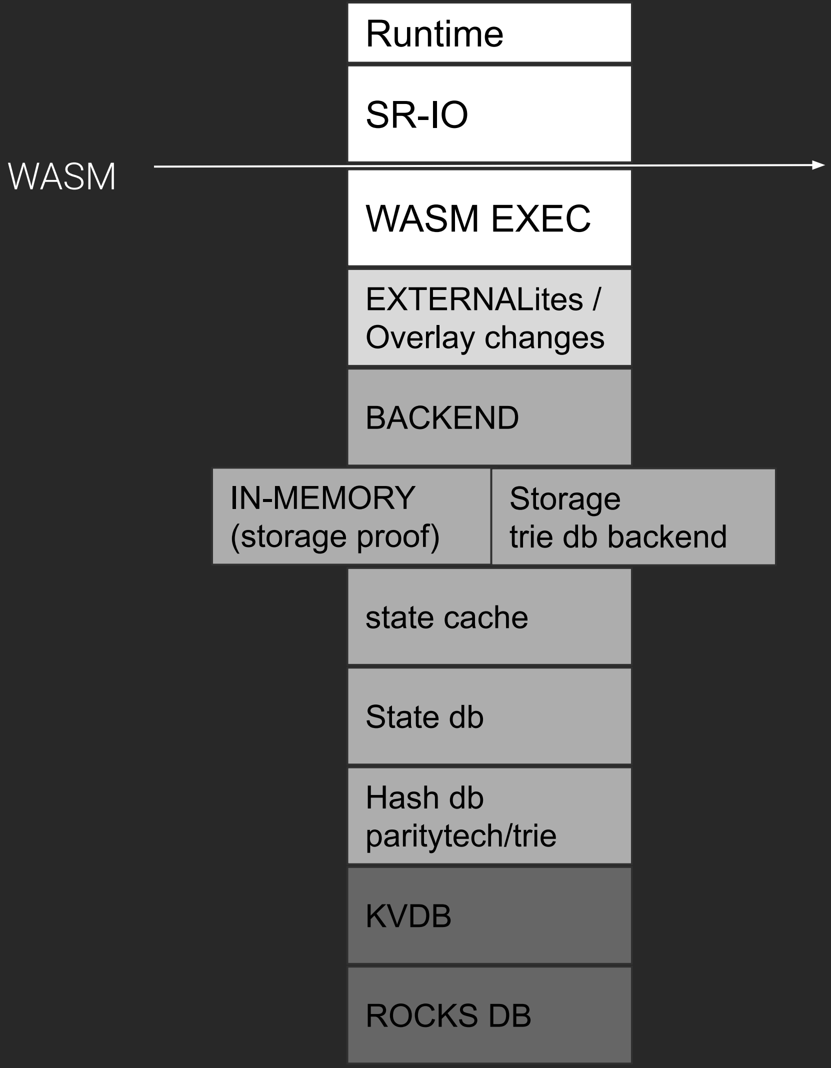


Runtime Storage API

Overlay Change Set

Merkle Trie

Key Value Database



Runtime

SR-IO

WASM

WASM EXEC

EXTERNALites / Overlay changes

BACKEND

IN-MEMORY (storage proof)

Storage trie db backend

state cache

State db

Hash db paritytech/trie

KVDB

ROCKS DB