

Introduction

What is Irys? Irys is the first programmable layer-1 datachain, purpose-built to make data actionable—unlocking value creation and network effects across AI, DePIN, and other data-centric ecosystems.

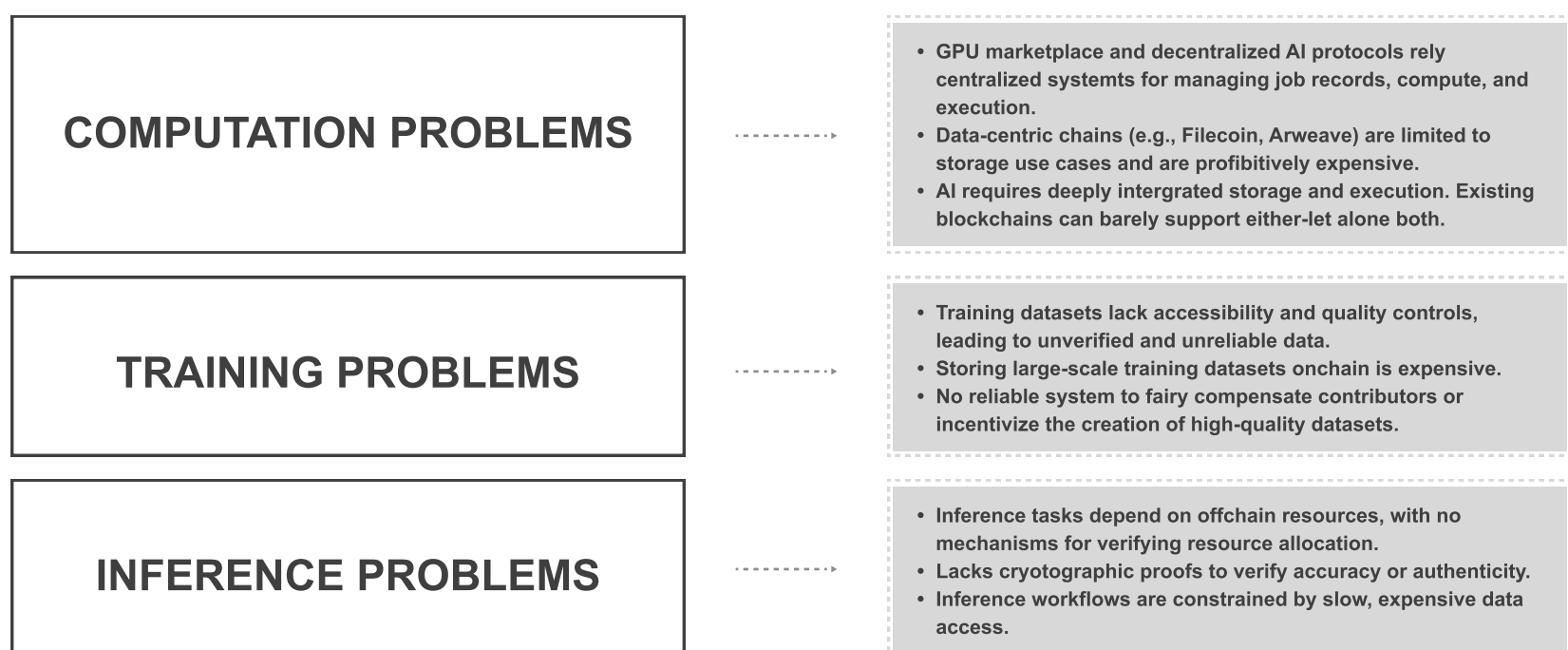
Irys is an integrated [datachain](#), including a high-performance storage layer and execution environment to enable [programmable data](#). Irys enables any amount of data to be stored onchain for any period of time (including permanent) for a fraction of the cost. Irys introduces several new primitives to make the data throughput and latency faster than previous players like Filecoin and Arweave.

Irys includes IrysVM, which enables data to be used within smart contracts, meaning a user could store 1EB of data for 1000x cheaper than other chains while using it within their onchain apps.

Irys achieves this by introducing several new primitives:



- PoW/S hybrid consensus for more reliable data and faster consensus
- Efficient sampling for lower user and miner costs for storing data
- Matrix packing and capacity partitions for higher data throughput and lower latency
- A multi-ledger system enabling users to store data for any period of time
- IrysVM—an EVM++ implementation—for programmable data

Irys is a layer-1 built with AI applications as first-class citizens. As Irys has integrated a high-performance storage layer, an integrated data availability layer, a native GPU supply, and a verifiable execution environment, developers can build AI applications for a fraction of the time and cost.



Comparison with other datachains

This comparison outlines several key differences in user features and protocol features.

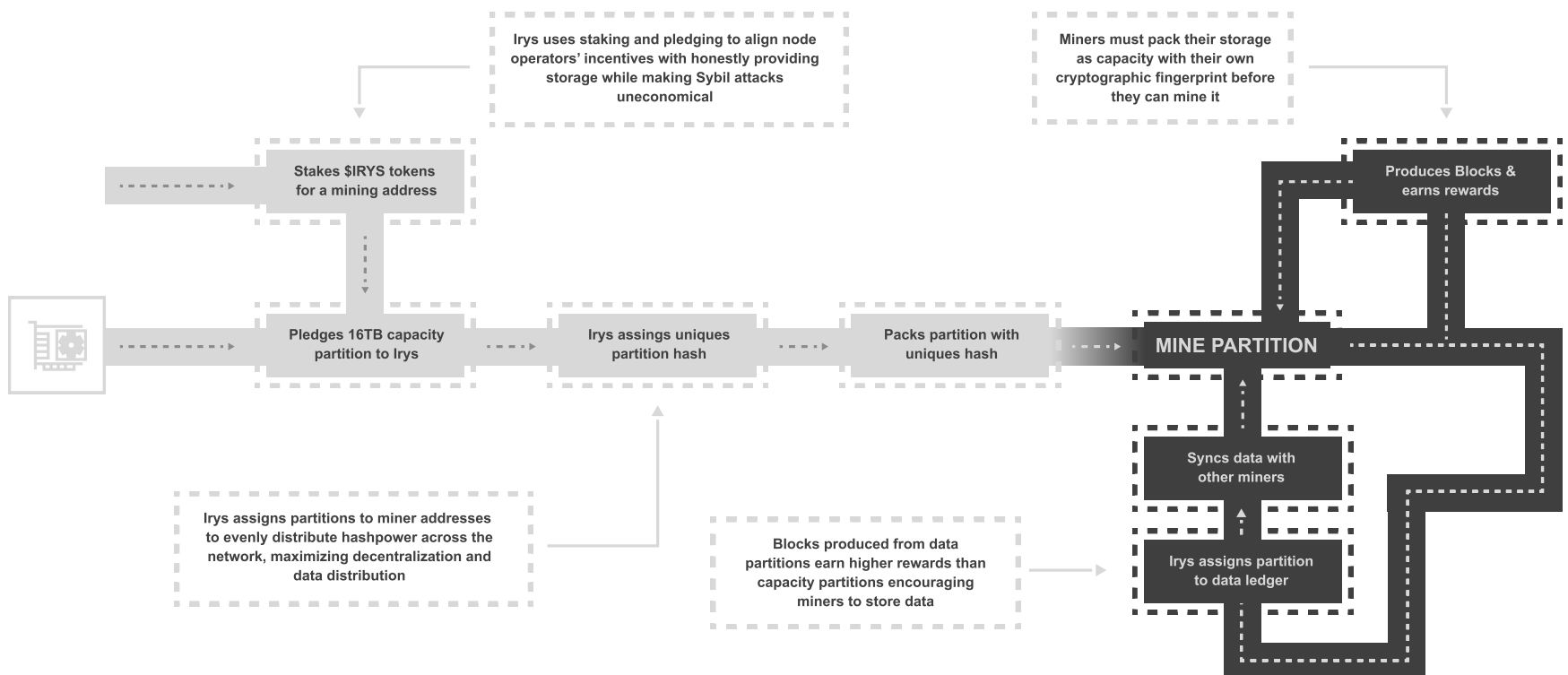
| |  |  |  |
|--------------------------------------|--|---|---|
| Execution | IrysVM allows you to develop onchain apps with programmable data. | FVM, which allows you to create deals programmatically. Limited utility as Filecoin deals are cold, and data can't be used in apps. | No native execution. They created AO, but it's a trusted rollup that isn't dependent on Arweave. |
| Type of storage | Hot access with cold pricing. Any duration. Uses a multi-ledger system to support any term and permanent data | Cold storage. Term based. The base primitive is "deal," which allows you to store data for 250 days. | Hot. Permanently only. |
| Storage costs | Permanent data: \$2.50/GB Shorter-term: \$0.0000753/GB/day | N/A | \$20/GB (and rising) |
| Latency | Instant. Partitions are pre-packed, meaning writing the data is limited by drive speed (limited by physics). | 6 hours to encode and decode data due to the use of ZK-friendly encoding. Insanely expensive at an exabyte scale. | Sub second per chunk but low parallelization meaning finalizing a transaction can take hours. |
| Throughput | Limited by hardware and physics. | Scales horizontally with number of miners (upper limit is [average miner bandwidth * number of miners]) | Limited to the throughput of a single node on the network as the whole network has to store the data. |
| Reliability (chance your data drops) | Very reliable with the use of economic security | Very reliable due to the use of ZK | Unreliable with thousands of dropped pieces of data |

High-level architecture

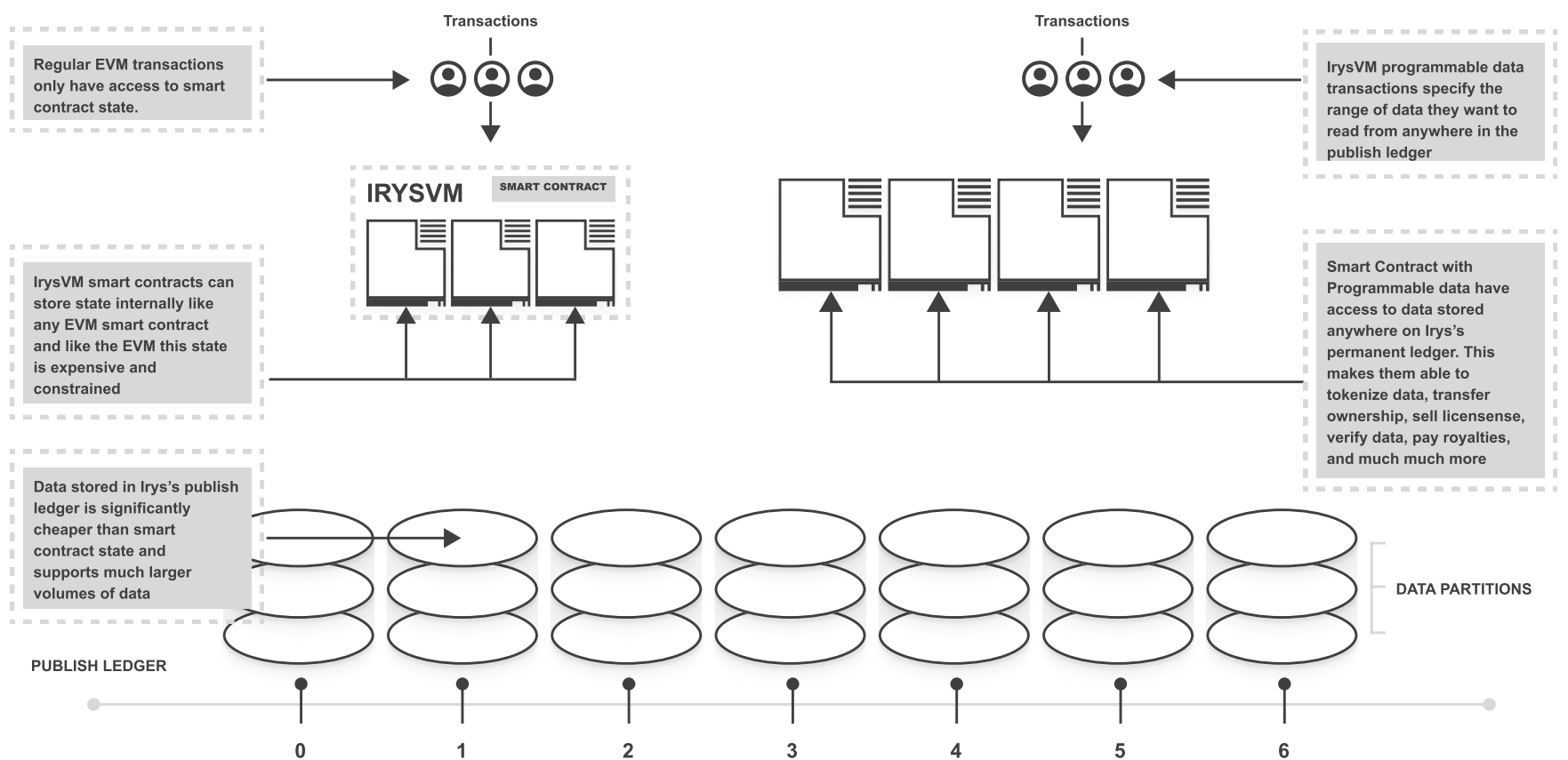
Irys is an integrated chain—integrating both storage and execution into a single chain enabling users to store data onchain for a fraction of the costs and building onchain application logic atop the data.

Our focus when designing Irys was vertical integration—the ability for developers to rely on only Irys for building their apps. We’ve observed developers need to use 100 different infrastructures to build their apps, and Irys aims to vertically integrate every part of the stack.

The below diagram outlines the processes within Irys that enable it to have its unique properties:



Irys combines high-performance data with robust verification, enabling a native execution environment for onchain data. Traditional storage protocols struggle to balance these two aspects, making it difficult to manage execution and storage in the same chain, meaning building onchain apps on datachains has been impossible to date. Irys overcomes this tradeoff by uniting Proof of Work (PoW) with staking and slashing mechanisms. This enables Irys to efficiently handle all forms of data alongside having robust security for fully onchain applications. We call this integration of secure storage and native execution in a single protocol Programmable Data.



IrysVM enables programmable data—the ability to tap into large amounts of data stored on Irys. It turns data from a static commodity to a useful asset that can be utilized and manipulated by smart contracts.

Blocks and transactions

There are two types of transactions:

- Data transaction: a transaction which is storing data on the network
- Execution transaction: a transaction that interacts with a smart contract through IrysVM. This includes programmable data transactions.

Each block contains two sets of transactions in separate block lanes. The block structure can be seen below:

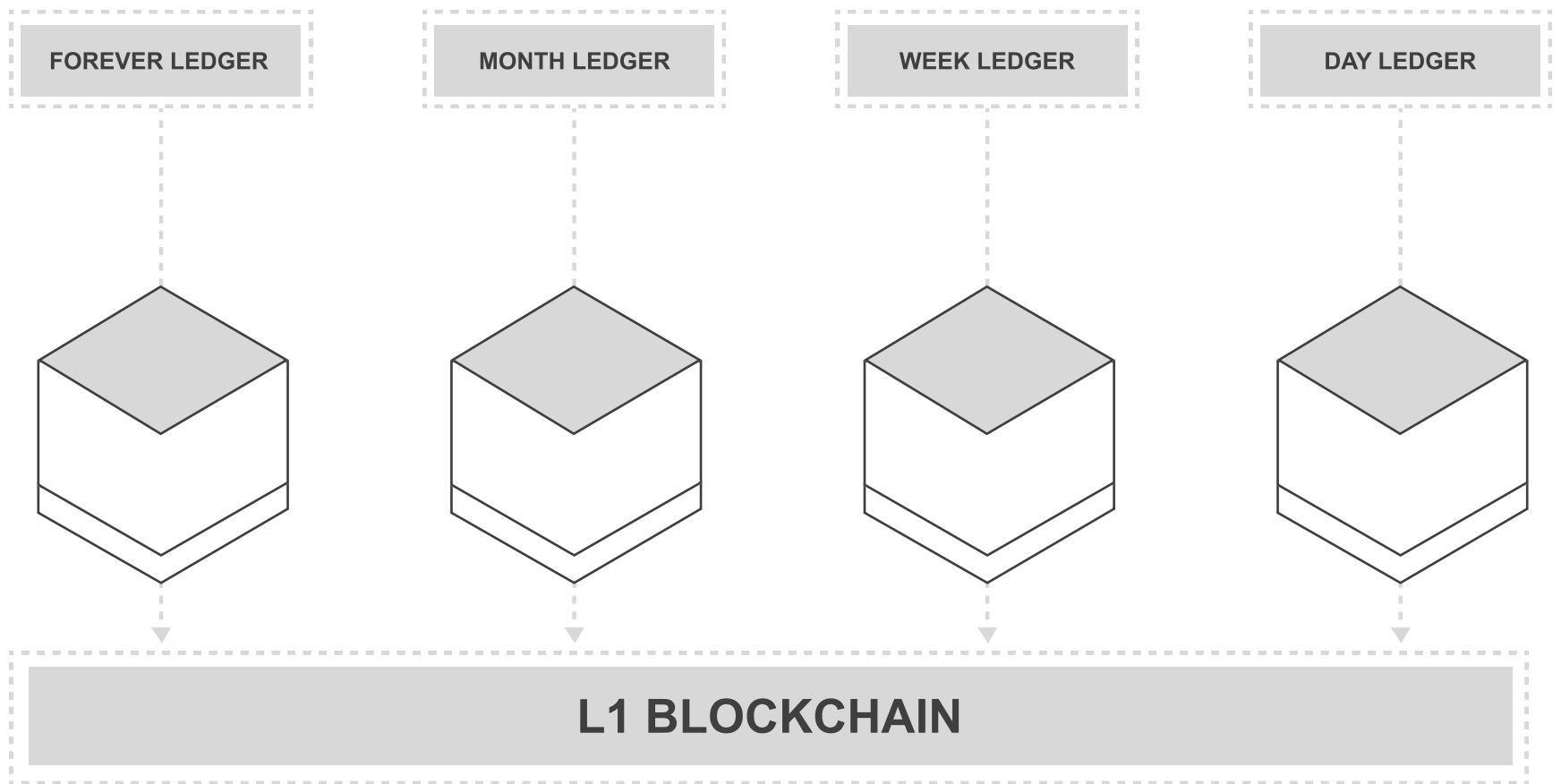
| Field Name | Description |
|-----------------|--|
| block_hash | The block identifier |
| height | The block height |
| diff | Difficulty threshold used to produce the current block |
| cumulative_diff | The sum of the average number of hashes computed by the network to produce the past blocks, including this one |

| | |
|--------------------------|--|
| last_diff_timestamp | Timestamp (in milliseconds) since UNIX_EPOCH of the last difficulty adjustment |
| solution_hash | The solution hash for the block |
| previous_solution_hash | The solution hash of the previous block in the chain |
| last_epoch_hash | The solution hash of the last epoch block |
| chunk_hash | SHA-256 hash of the PoA chunk (unencoded) bytes |
| previous_block_hash | Previous block identifier |
| previous_cumulative_diff | The previous block's cumulative difficulty |
| poa | The recall chunk-proof |
| reward_address | The address that the block reward should be sent to |
| miner_address | The address of the block producer - used to validate the block hash/ signature & the PoA chunk |
| signature | The block signature |
| timestamp | Timestamp (in milliseconds) since UNIX_EPOCH of when the block was discovered/produced |
| ledgers | A list of transaction ledgers, one for each active data ledger |
| evm_block_hash | The Ethereum Virtual Machine block hash |
| vdf_limiter_info | Information about the Verifiable Delay Function limiter |

Irys uses block lanes to ensure that users can always submit a data transaction, even when the execution lane is congested. This allows data transactions to have stable pricing.

Partition Lifecycle (Capacity & Data, Multi Ledger)

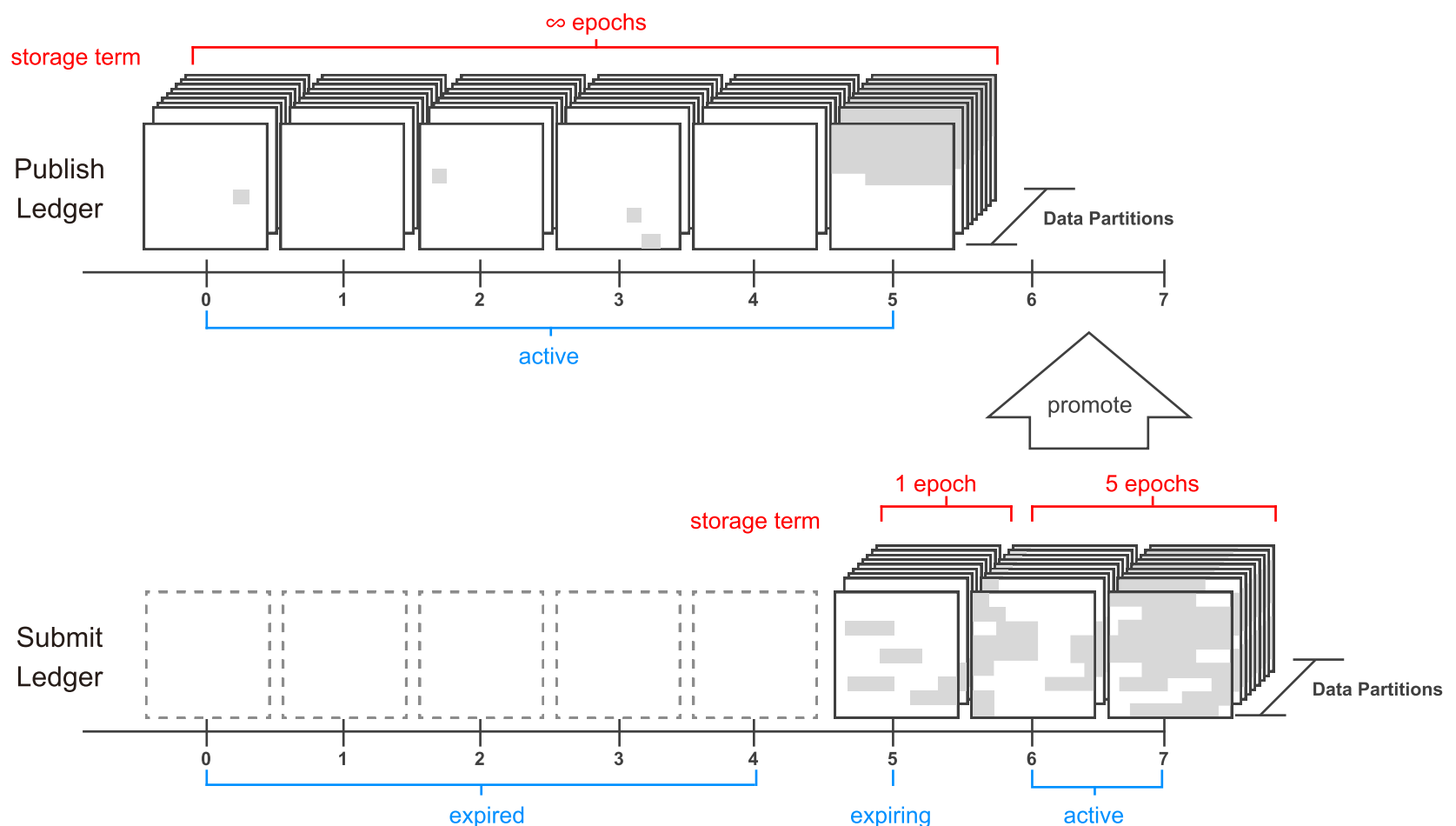
Irys introduces a multi-leader system where data can seamlessly transition between different terms. This effectively enables users to store data for any period of time (e.g., 1 day, 1 week, 1 year, or forever). There are also inbuilt mechanisms for “promoting” data to longer-term ledgers—e.g.,. when a user requests to store data on the permanent ledger, it enters the 5-day ledger and transitions to the permanent ledger once it’s been verified by the network.



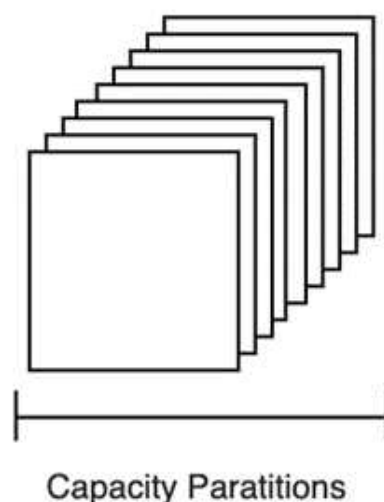
A ledger is a collection of data with a shared property (typically duration), i.e., all data in ledger 0 is stored for 5 days.

Partition Lifecycle

Ledgers on Irys are split up into 16TB partitions. This allows miners to affordably use HDDs and not be outcompeted by SSDs.



Irys measures the amount of data uploaded and then uses it to project the number of partitions needed on standby. These standby partitions do not contain any data (yet) and are referred to as Capacity Partitions.



Step 1: Partition Pledging

Miners post a pledge transaction to the network indicating their willingness to bring a new partition online. The protocol then randomly assigns unclaimed capacity partitions to the pledged miners; any miners who don't receive capacity partition assignments have their pledges refunded.

Step 2: Partition Packing

Once assigned a Capacity Partition, the miner packs it using Irys' matrix packing scheme. This process fully encodes the miner's fingerprint into the partition.

Step 3: Partition Mining

Once a capacity partition is packed, the protocol assigns a random 200MB sequential read to the miner every second. The miner takes these 200MB reads, splits them into 800 256KiB chunks, and proceeds to hash each chunk, looking for a mining solution. If the miner is lucky enough to find a mining solution, it wins the right to produce a new block and announce it to the network, thus earning rewards.

Step 4: Ledger Assignment

When it comes time to add more storage capacity to one of Irys' data ledgers, the protocol will randomly select a Capacity Partition that is actively being mined. The randomness is weighted towards partitions used to mine blocks, rewarding miners who have been participating longer and have been effective in mining.

Being assigned to a data ledger and mining it has higher rewards than mining empty capacity partitions, so miners are incentivized to demonstrate their capability by mining capacity efficiently.

Step 5: Partition Departure

There are two ways partitions leave the network: orderly departures and disorderly departures.

Orderly Departure

1. The miner posts a transaction to un-pledge their partition and recover the commitment they staked initially.
2. A timeout period begins when the protocol assigns another Capacity Partition to synchronize the data being taken offline.
3. Once the timeout has passed, the departing miner can recover their staked commitment and remove their partition.
4. If the miner goes offline before the timeout has passed, they risk losing their pledged bond.

Disorderly Departure

1. A miner engages in adversarial behavior toward the network (double signing blocks, for example).
2. The protocol takes their staked commitment.
3. The protocol assigns a new Capacity Partition to fill gaps left by the adversarial miner.

This is an undesirable departure as the network will have to assign new capacity partitions quickly. Because of this, Irys requires miners to make a significant upfront investment by staking tokens to their mining address in order to participate in the protocol and earn rewards.

Matrix packing

Packing describes the process of miners encoding data with their staking address. This proves they're storing a unique replica of the data, as opposed to mining off a single remote copy (a common attack vector on datachains).

Irys introduces Matrix Packing, which uses a VDF to encode the miner's address into each 256KiB chunk they store. This provides sufficient computational cost for each bit of data, making any adversarial mining pattern unprofitable. The process is outlined in detail in the below image:

Phase 1 - Sequential Hashing

mining_address, partition_hash, and chunk_offset are SHA-256 hashed into seed_hash.

seed_hash is SHA-256'd to form the initial **segment**.

The **segment** bytes are appended at the start of the empty chunk.

The **segment** is also used as input for the next SHA-256.

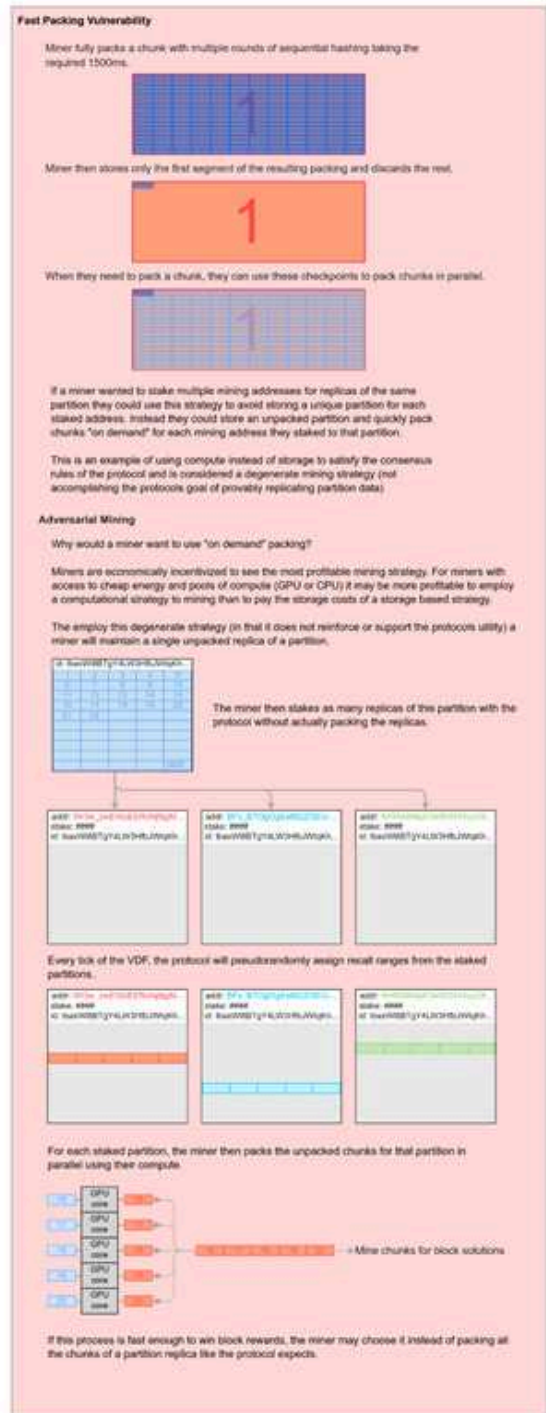
Resulting **segment** is appended following the previous segment's bytes.

Steps ② and ③ are repeated using the previous segment until the empty chunk is filled with segments.

The diagram illustrates the sequential hashing process for filling a 2096KB chunk. At the top, three inputs (mining_address, partition_hash, chunk_offset) are combined and hashed (SHA-256) to produce a seed_hash. The seed_hash is then hashed (SHA-256) to produce the first segment. This segment is appended to the beginning of a large orange rectangle representing a 2096KB chunk. The resulting combined data is then hashed (SHA-256) again to produce a second segment, which is appended to the end of the first segment within the chunk. This process repeats until the entire 2096KB chunk is filled with segments. A final step shows the completed chunk being hashed (SHA-256) to produce a new segment, labeled as 'repeat until segment fills the chunk'.

Sequential hashing ensures single-core chunk packing since each segment depends on the one before it, establishing a minimum packing time. This method allows mining of data-less "capacity" chunks that are verifiable by other miners deterministically (by recreating the packing locally and comparing it to the provided packed chunk).

With 32-byte segments and a 2096KB chunk, one pass of sequential hashing requires 8,192 hashes. The challenge is that a Ryzen 5900X can compute ~15 million SHA-256 hashes per second, making this amount of packing delay take a tiny fraction of a second. To deter adversarial miners from packing "on-demand" we'll have to increase the packing length, something on the order of ~1500ms.



Irys introduces a hybrid Proof-of-Work-Stake consensus algorithm. This was done to achieve the following set of requirements:

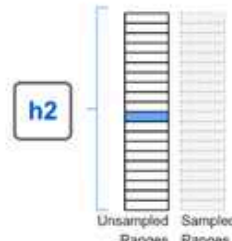
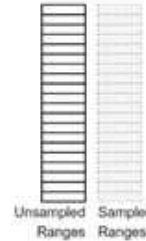
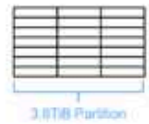
- A hybrid consensus enables us to adopt PoW economics with security mechanisms like slashing to provide maximally reliable onchain data with performant execution.

Consensus algorithm

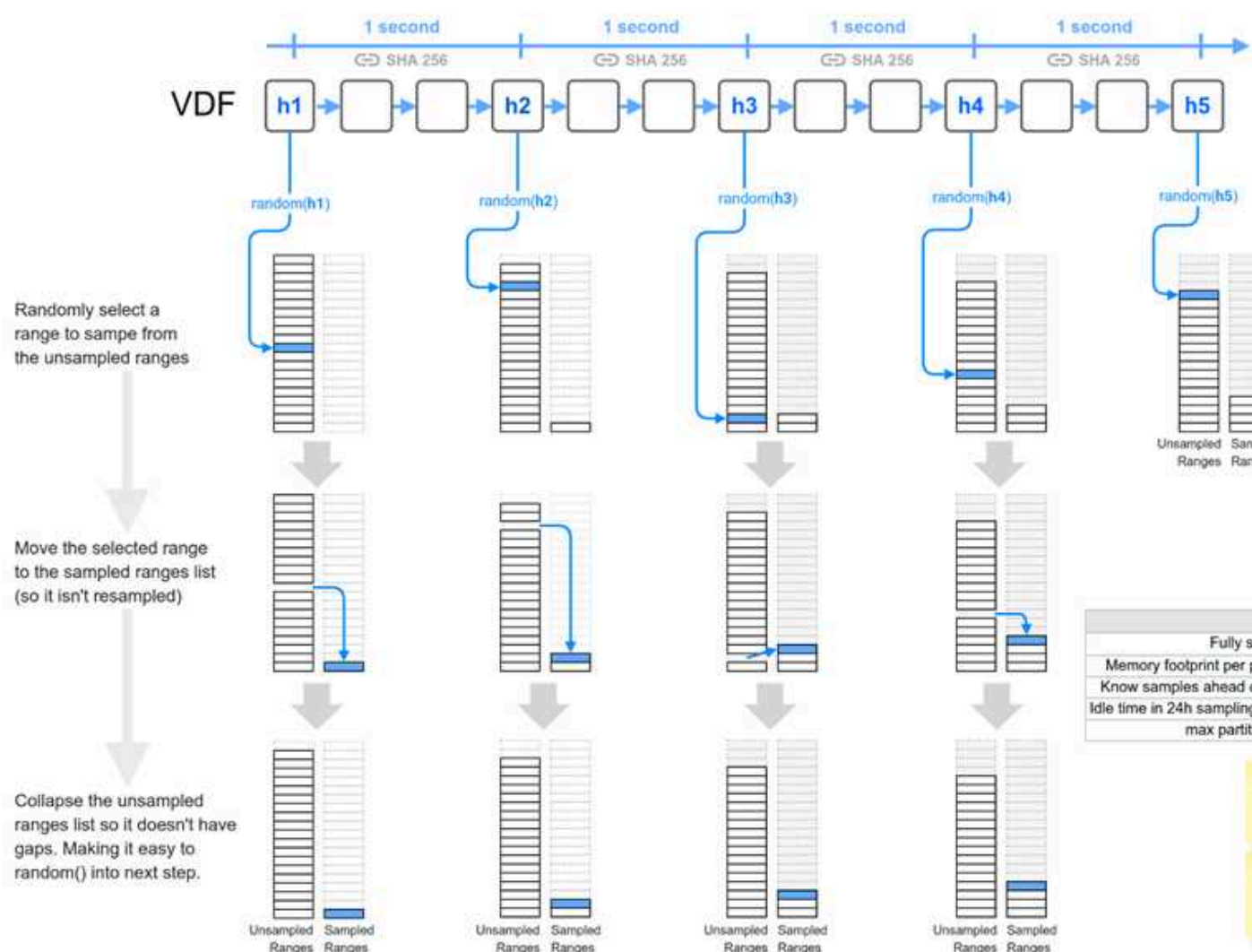
Irys uses a process known as efficient sampling, where the chain guarantees every partition is sampled entirely once per day.

Efficient Sampling

- 1 Each 3.6TiB Partition is divided into 200MiB recall ranges.
- 2 Range indexes are added to a "random state" of sampled and unsampled recall ranges.
- 3 VDF mining hashes are used to pseudorandom select a recall rand from the unsampled set.
- 4 Sampled ranges are moved from the unsampled list to the sampled list.
- 5 Repeat steps 3 and 4 using subsequent VDF mining hashes then start over at step 2.



Sampling Example



| | Stochastic | Efficient |
|----------------------------------|------------|-----------|
| Fully sampled | 27h | 4h |
| Memory footprint per partition | 0 | 37KiB |
| Know samples ahead of time? | no | no |
| Idle time in 24h sampling period | 0 | 20h |
| max partition size | 3.6TiB | 14.4TiB |

37KiB for a 3.6TiB partition.
Increases linearly with
partition size

efficient sampling with 14.4TiB
partition means less idle time
in a 24h sampling period.

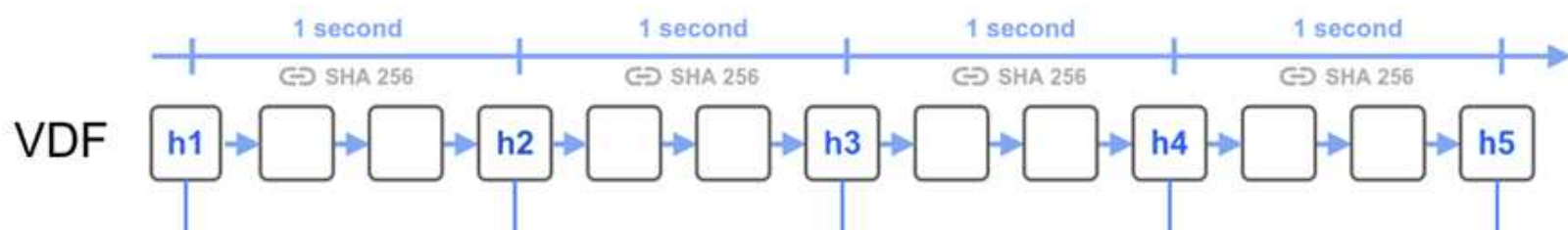
The algorithm can be summarized as so:

1. There is a VDF running "ticks" every 1 second, generating a seed
2. For every partition a miner stores, use the seed in a deterministic random function to generate the start position of a range
3. Read 200MiB worth of chunks (800)
4. For each chunk, calculate a solution
 - a. Convert into a number
5. If the solution is greater than the current network difficulty, then publish the block. If it's below the difficulty, then go back to step 1

The idea here being that miners are continuously sampling data to participate in consensus creating a strong guarantee around data being sampled. If miners lose data then they can be challenged and ultimately slashed if proven malicious.

VDF to synchronize read speed

One key requirement for Irys was to force people to use HDDs, or in other words, don't allow people to outcompete by using SSDs. A VDF is used to achieve this by only allowing 200MB to be read every second per partition a miner stores, essentially limiting the read speed of each partition at 200MB (approximately the max read speed of an HDD).



If the difficulty is greater than the network difficulty and the miner has sufficient stake, the block is accepted.

Data transaction lifecycle

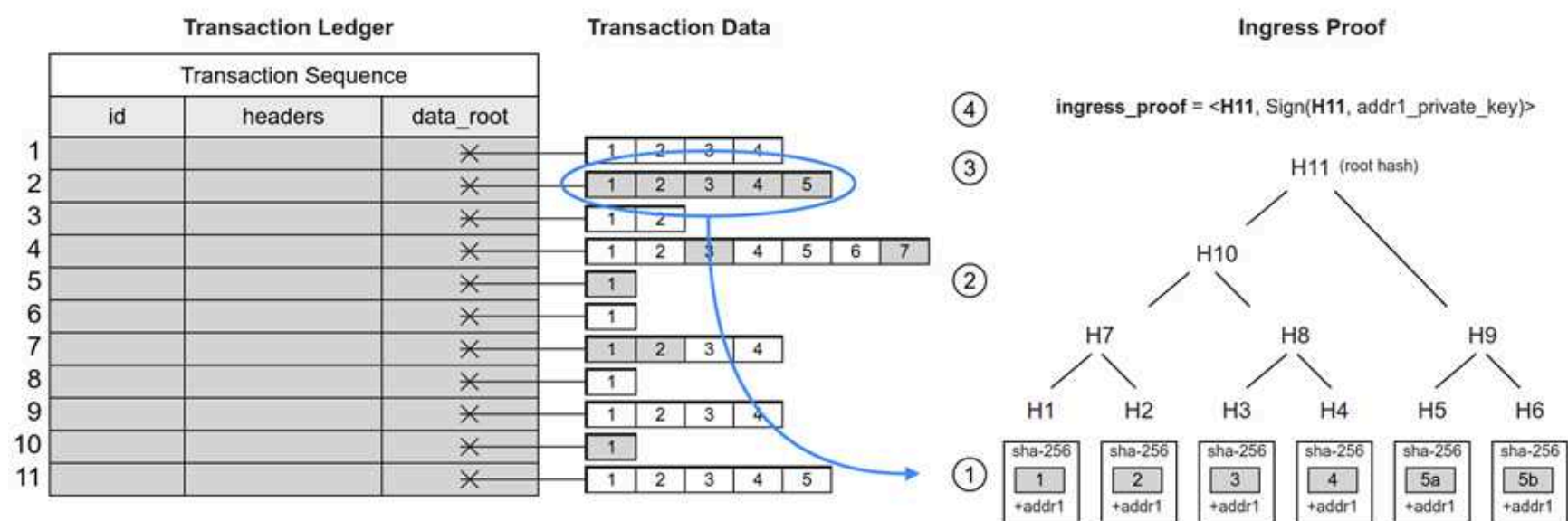
Each data transaction specifies a ledger the data should be stored in, and each ledger represents a duration the data should be stored for; for example, a transaction could represent "I want to store this 1 GB of data in ledger 0, which stores data for 2 weeks". Once a user has posted a transaction and all ingress proofs have been verified, the data is XOR'd into a partition where it can be used in consensus proofs.

The only special case is when the transaction specifies the permanent ledger. In this case, the transaction enters the submit ledger and then migrates to the permanent ledger once 10 miners have proved they're storing the data.

Ingress proofs

An ingress proof is a Merkle root that can only be generated by accessing the chunks of a transaction's data. Ingress proofs are used to prove a miner has some data. This is a required step before miners can store data, providing evidence that they store a unique replica of the data.

The data chunks are hashed together with the miner's address to create a proof unique to that mining address. This process makes the proofs easy to generate and validate by someone with the data. To prevent false claims, these proofs must also be signed by the miner's private key. Without this signature, any miner with access to the data could generate proofs for another mining address, falsely claiming that they had downloaded the data.



Identify recently added transactions in the submit ledger that are waiting to be promoted to the publish ledger. Obtain the data associated with these transactions, either directly from a user or through gossip with other miners.

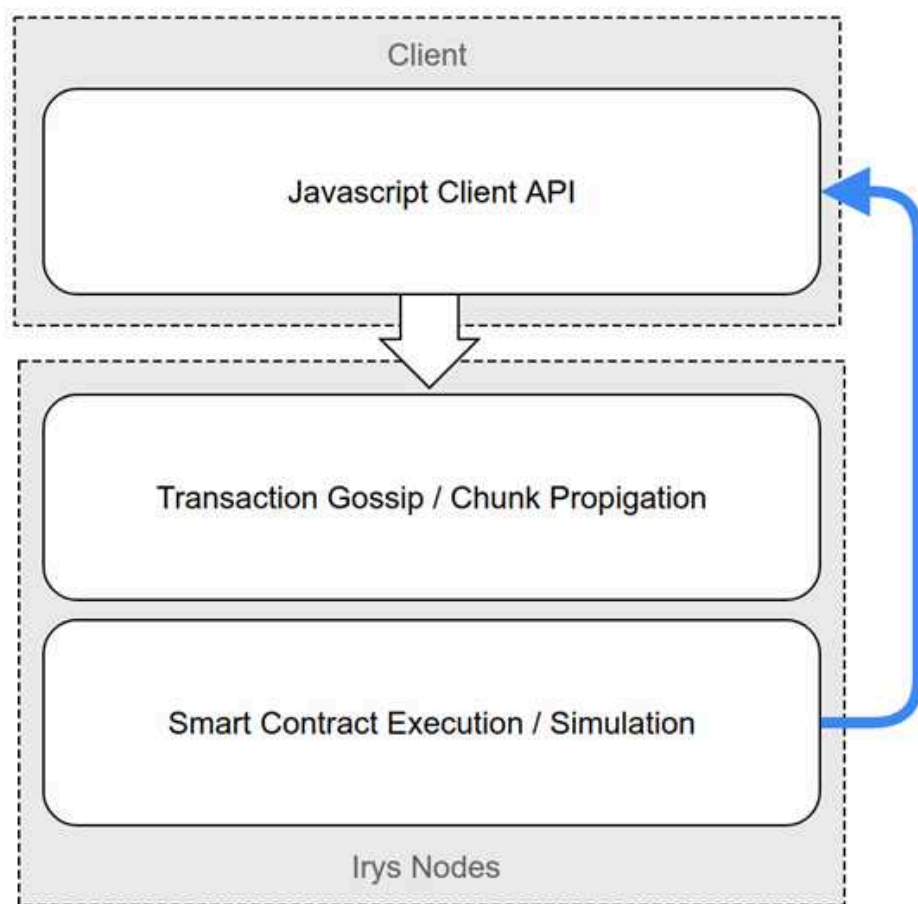
Once all the data chunks are collected, create an ingress proof as follows:

1. Split the data into chunks and hash each chunk together with the miner's address.
(note: chunk 5 is split into 5a and 5b to build a more balanced Merkle tree.)
2. Continue hashing these hashes to build the branches of a Merkle tree.
3. Compute the root hash of the Merkle tree.
4. Sign the root hash with the private key associated with the mining address used in step 1.

IrysVM and programmable data

Irys introduces IrysVM, an EVM++ implementation, which adds new opcodes. The main upgrade made is to enable programmable data — the use of stored data within smart contracts. Essentially this means you could store 1EB of data and use portions of the data within a secure environment.

To fully implement programmable data, we must integrate the feature across multiple layers in the tech stack. Starting with transactions that allow the caller to specify the range of chunks they wish to access with their PD SmartContract call. Next, gossip the transaction and request the range of chunks to verify their availability. Finally, include the transaction in a block and execute the state transition.



Posting Transactions

To post a PD transaction the transaction must specify the range of chunks it wishes to have available during the SmartContract invocation. To maintain compatibility with EVM toolchains this is done by including the range of chunks to reference as elements using [EIP-2930](#) access lists, where the address is the address of the PD precompile, and the data is one or more range specifications.

Range Specification

Chunk range specification follows a simple format.

```
<partition_index>:<offset>:<chunk_count>
```

partition_index: The partition index in the publish ledger of the partition containing the first chunk.

offset: The chunk offset in the partition to begin reading chunks

chunk_count: The number of sequential chunks to read after offset

partition_index: 26 bytes - 0 to a lot

offset: 4 bytes - 0 to ~80,000 (num chunks in a partition)

chunk_count: 2 bytes - 0 to 65,535 (PD is constrained to about 7,000 chunks per block, so this is safe by an order of magnitude)

Total Binary Bytes to represent a range: $26+4+2= 32$ bytes

Constraining the range to 32 bytes makes it easily storable in the EVM smart contract state.

Note: Ranges are always 32 bytes, and the values are unpacked by slicing the bytes at the correct offsets for each value.

Irys Client SDK

The default way of referencing data on Irys is by using the transaction ID, putting the burden on the developer to locate the partition and chunk offset of the data they are interested in for PD is high friction and requires a lot of knowledge of the internal data model of the chain.

To simplify the process of posting PD transactions, the client SDK implements some utility methods to abstract away the complexity of specifying chunk ranges.

```
const chunkRange = getTxChunkRange(txid);
```

The implementation of this function is provided by the gateway the client is using.

1. It looks up the bundle_txid if the txid is actually a DataItem ID
2. It verifies the txid is in the Publish Ledger
3. It looks up the chunk_offset in the Publish Ledger
4. It looks up the partition_index in the Publish ledger.
5. It uses 3 & 4 to compute the partition_offset.
6. It uses the Publish Ledger txid to look up the size of the transaction data.
7. It computes the number of chunks required to read the transaction.
8. It builds a Range Specification and returns it to the caller.

Estimating Transaction Price

The price of PD is determined by the same transaction simulation that estimates gas prices for posting transactions to the EVM. Because the chunk range is included in the call data of the transaction, when the simulation is run the simulation mechanism includes the pricing for the number of chunks needed to be retrieved along with the computational budget of the transaction.

To read more about how PD chunks are priced, see the [following document](#).

Gossip & Mempool

An important constraint on PD transactions is the maximum capacity for propagating PD chunks between a majority of nodes on the network between blocks.

Transmission

When a node receives a PD transaction, it broadcasts it to its peers, indicating whether it has the chunks. Receiving peers may request the chunks in their response.

The broadcasting node tracks which peers need the chunks and sends them after receiving them.

Receiving peers also broadcast the PD transaction, marking peers that have already received it and noting which peers have the chunks.

When a peer receives the chunks, it sends them to peers lacking them.

If a peer hasn't received the chunks after some time, it may retrieve them from assigned storage partitions by inspecting the ledger, locating partition owners, and requesting the chunks from a randomly chosen partition.

Validation

Transaction validation follows the same static validation as other transactions.

Chunk validation is a little more complicated. PD Transaction chunks can be retrieved in a number of ways.

1. The node has the cached unpacked chunks locally.
2. The node has the packed chunks in a partition they mine.
3. The node receives unpacked chunks from a peer.
4. A node requests packed/unpacked chunks from a peer.

Cached Unpacked Chunks: In this case, the node has already validated the chunks with their Merkle roots and can be confident the data in them is correct and ready to be exposed to the VM for PD execution.

Local Packed Chunks: The node happens to mine the partition that contains the PD chunks requested by the transaction, but they are packed.

The node:

1. Creates the entropy for the chunk range
2. Unpacks the chunks using the computed entropy
3. Builds a merkle-root out of the unpacked chunks
4. Looks up the transaction that posted the chunks from its block index
5. Compares the computed merkle-root with the one in the transaction.
6. If valid, the node posts the unpacked chunks to any peers marked as not having them.

Receives Unpacked Chunks: In this case the node is being sent unpacked chunks by one of their peers. While they do not have to unpack the chunks they do need to verify the chunks contain the correct data or risk proposing an invalid block.

The Node:

Follow steps 4-6 from the Local Packed Chunks path.

Requests Chunks: As a failsafe, if the node is not receiving any of the chunks within time D, where D is the propagation delay of the network (Assume D = 200ms for testnet).

The node:

1. Looks up the partitions responsible for storing the chunks.
2. Picks a partition at random to request the chunks.
 - a. If the partition provides packed chunks, unpack them.
3. Follow steps 4-6 from the Local Packed Chunks path.

Block Production

After a mining node receives a PD transaction and its chunks and validates them, it can include the transaction in a block. To minimize the chance of producing a block with a PD transaction that the majority haven't received the chunks for, the miner may wait for the propagation delay D before including it. This allows most of the network to retrieve the chunks and validate the PD transaction when it's included in a block.

Smart Contract Execution

Executing a PD smart contract interaction requires further exploration. There are a few possible approaches, but they will require exploration of the code to evaluate their feasibility.

Exposing Chunk Data

PD transactions include an instruction to a precompiled "system" contract which takes the Range specification as an input. This precompile will bring the chunk data into scope. There are at least two possible approaches.

Return Value: The foreign call to the precompiled "system" contract could return the chunks specified by the range as a buffer.

Global State: Once the precompiled "system" contract has been invoked the chunks become accessible via a global that is exposed to all subsequent instructions in the PD transaction.

Calculating Compute Units

Because the execution of a particular smart contract function may take one code path or another depending on the data read from the chunks, calculating compute units (CUs) can be problematic. There are a few possible approaches

Simulate With Chunks: The only way to deterministically simulate the CUs required to complete the execution of the instruction is to have the unpacked chunks available during the simulation. This would require the simulating node to retrieve the unpacked chunks during the simulation request.

Simulate Compute Upper Bound: In this case, the simulation would evaluate all code paths and return the cost of the most compute-intensive code path. This way the user always pays enough gas for any possible computational resources.

Programmable data roadmap

Blob Data - MVP

The first version of PD transactions will expose chunks as buffers or blob data to the contract and leave the interpretation of these bytes up to the caller. This will allow PD chunks to have any structure or format the caller can imagine.

Bundle Format v.1 - Bundles

Once the blob chunks are working, the next layer of functionality will be a DX upgrade that allows PD contracts to load a bundle and data items from the chunks. The IrysVM will parse the chunks in the range specification as a v1 bundle format.

Bundle Format v.2 - DataItems

Once the v2 (merkelized) bundle format exists, a DX upgrade will allow parsing of specific data items from a larger bundle or retrieve smaller nested bundles (or their data items) by loading only the chunks that store the specific data items the caller is interested in.

Programmable data L2s

In the future, we expect users will develop programmable data L2s that expand the compute capacity beyond a single-state machine. The end goal here is a shared dataset with the ability for anyone to spin up L2s to tap into data, compute, and liquidity resources.

Tokenomics

Token utility

The IRYS token is composed of four core components:

- **\$IRYS:** The \$IRYS token is the Irys platform native asset.
- **Fees:** Fees are charged on [all network operations](#), including payment for data storage and protocol execution. Unlike other datachains, both temporary and permanent data storage fees are pegged to a USD range and updated on a yearly basis.
- **Security:** Token rewards are used to incentivize node validators contributing to [Irys consensus](#) and to prevent spam and denial-of-service attacks.
- **Endowment:** \$IRYS is used to fund the endowment, which covers miners' future liabilities
- **Staking:** Miners must [lock \\$IRYS tokens as collateral](#), signaling their commitment to the network and creating clear economic consequences for failing to uphold their responsibilities. Users will also be able to delegate \$IRYS tokens in order to passively participate in contributing to the network's security model.

Burn mechanism

\$IRYS will have strong deflationary pressure early into its lifetime as the inflationary rewards decay. Irys has a couple of burn mechanisms:

- **Long-term storage:** fees paid towards “longer-term” ledgers (i.e.,>2 weeks) will be contributed to an endowment where the tokens will likely never be released, creating an effective burn/sink mechanism.
- **Fee market on execution transaction:** Irys will burn 50% of fees from execution transactions, so as programmable data demand increases, there’ll be greater burn occurring.

Incentivizing hardware

Irys adopts a traditional inflation decay curve which is distributed via block rewards to miners for storing data. The starting inflation rate is 8% and halves every 2 years.

[Insert graph to represent curve]

Fees

Minimum Fee Parameter

Irys implements a minimum fee to mitigate network spam and to ensure that tx fees can be easily denominated with the atomic units.

The minimum fee is \$0.001 (1/10th of a cent) as determined by Irys’ price approximation mechanism.

The same minimum fee is paid to the provider of ingress-proofs for publishing permanent data.

Term Fees

The pricing model for term storage determines the cost of providing storage for data, 10 replicas for n epochs

| Pricing Parameter | | Value | |
|--|--|----------------|-----------------------------------|
| Annualized Cost of operating 16TB HDD | | \$44 | |
| Number of Replicas | | 10 | |
| | | | |
| Calculation | | Value | |
| Daily Cost per TB | | \$0.0075 | |
| Daily Cost of 16TB HDD | | \$0.12 | |
| Total Fee Per Epoch Storage Price (TB) | | \$0.0753 | |
| Total Fee Per Epoch Storage Price (GB) | | \$0.00007358 | |
| | | | |
| | | | |
| Epoch Fee Calculator | | | |
| Number of Epochs | | Data Size (TB) | Total Fee Per Epoch Storage Price |
| 1 | | 1 | \$0.0753 |
| 5 | | 1 | \$0.3767 |

An additional 5% fee is added for inclusion in the block (scales with the size of transaction data)

1TB of Term Data in the Submit ledger (5 epochs)

$$\text{term_fee} = \text{term_cost} + 5\%$$

$$\text{term_fee} = \$0.3767 + 5\% = 0.3955 \rightarrow \$0.40$$

1GB of Term Data in the Submit ledger (5 epochs)

$$\text{term_fee} = \$0.00039$$

As \$0.00039 is below the network minimum fee of \$0.01 the term_fee becomes:

$$\text{term_fee} = \$0.01$$

Note: if repacking term partitions after they expire represents an ongoing expense to miners, this cost will be quantified and included in the term data pricing.

Perm Fees

The pricing model for permanent data has some additional factors to account for. Because the users are paying for centuries of storage upfront the model has to account for declines in the physical costs of storage (due to technological gains) over that time period. Irys chooses an extremely safe 1% annualized decline in the cost of storage as a factor for pricing permanent data. (Observed declines in storage costs over the last 50 years have been > 25% year on year.)

| Pricing Parameter | Values |
|---|------------|
| Annualized Cost of operating 16TB HDD | \$44 |
| Safe annual decline in cost of storage (decay rate) | 1.00% |
| Number of Replicas | 10 |
| Years of storage paid for | 200 |
| | |
| | |
| Cost Per TB | \$2,381.54 |
| Cost Per GB | \$2.33 |

Because permanent data must first pass through the submit ledger (term data) on its way to the publish ledger, the fee includes the cost of submit ledger storage as well.

Perm data requires 10 ingress-proofs, ingress-proofs are the same as the 5% immediate reward for including the transaction in a block. (scales with data size, shares the minimum \$0.01 fee floor).

1TB of Permanent Data

$$\begin{aligned}\text{perm_fee} &= \text{term_fee} + (\text{ingress_fee} * 10) + \text{perm_cost} \\ \text{perm_fee} &= \$0.40 + (\$0.018835 * 10) + \$2,381.56 \rightarrow \$2,382.14\end{aligned}$$

1GB of Permanent Data

$$\begin{aligned}\text{perm_fee} &= \text{term_fee} + (\text{ingress_fee} * 10) + \text{perm_cost} \\ \text{perm_fee} &= \$0.01 + (\$0.01 * 10) + \$2.33 \rightarrow \$2.44\end{aligned}$$

If a user fails to upload data during the submit ledger term duration or the network fails to achieve the required number of ingress-proofs, the user's ingress_fee's and perm_cost are refunded when the submit ledger transaction expires at the end of 5 epochs (the submit ledger term duration)

Consensus Pricing Mechanism

The process of promoting data from the submit ledger to the permanent ledger involves multiple phases, resulting in a staged payment model for permanent data. All transactions, whether intended for permanent (perm) or temporary (term) data, initially enter the submit ledger. The payment process for term data is consistent across all transactions, while permanent data incurs additional payments to incentivize the complete publishing process.

Term Data Payment Distribution

1. User posts a transaction, including the term_fee.
2. Block producer transaction inclusion:
 - Block producer includes the transaction in a block.
 - Block producer's balance increases by 5% of the term_fee.
 - Remaining 95% of term_fee is added to the treasury (tracked in block headers).
3. The user uploads data chunks associated with their transaction.
4. Miners assigned to store chunks gossip them amongst themselves.
5. Term ledger expiration payout:
 - When the transaction expires from the submit ledger (when the partitions containing its chunks are reset at an epoch boundary), each miner is paid their portion ($\text{term_fee} / 10$) for all assigned chunks expiring in their partition.
 - For a full 16TB partition, this payout is approximately \$0.60 per miner.
 - Miners continue to earn full inflation/block rewards from any blocks they produce while mining these partitions.

Additional Incentives

This payment structure creates additional incentives for miners to participate in term ledgers:

- Miners receive a payout when data expires from their partitions.
- Because miners must re-pack the partitions after expiration, this additional fee encourages ongoing participation and maintenance of the network.

Permanent Data Payment Distribution

Fee Structure

Users pay the following fees for permanent data storage:

- term_fee: Standard fee for term storage
- perm_fee: Fee for permanent storage
- 5% of term_fee for block inclusion
- 5% of term_fee for each ingress-proof

Fee Distribution

1. term_fee: Processed identically to regular term data transactions.
2. Block Inclusion Fee:
 - 5% of term_fee paid immediately to the block producer including the transaction.
3. Ingress-Proof Fees:
 - 5% of term_fee for each ingress-proof provided.
4. perm_fee:
 - Prepaid amount covering 200 years x 10 replicas with 1% annual decline in storage costs.
 - Added to the treasury.

Submit Ledger Expiry (Epoch Boundary) Processing

Refund Scenario

If transaction data was never uploaded:

- Ingress-proof fees and perm_fee are refunded to the uploader.

Promotion Scenario

If data was promoted to permanent storage:

- Protocol inspects all permanent transactions with ingress-proofs.
- Pays out the ingress-provers.

Epoch Boundary Payment Distribution Tasks

For each expiring submit ledger transaction:

1. Inspect the transaction to determine if it was intended for the publish ledger.
2. If intended for publish ledger, check if it arrived:
 - If Published: Reward ingress-proof submitters with their 5% rewards.

- If not Published: Refund perm_fee and ingress-proof fees to the address that posted the tx.
3. Tabulate the amount of data posted to the expiring partition.
 4. Pay each partition owner the term_fee for storing that amount of data.

Future work

Scaling programmable data

Programmable data at its core is the ability to have a shared dataset where any can permissionlessly access and build onchain applications atop. A key part of achieving this end vision is Programmable Data L2s.

These L2s would scale IrysVM and enable a trustless bridge to Irys's dataset. The end goal of this is to have a shared dataset of the public and private states, allowing anyone to compose on the data.

For the public state, anyone can use the data permissionlessly for their apps. Licensing can be used to monetize the usage of this data. Private state can utilize private compute primitives to enable apps to interact with the data.

Fast blocks and fast finality

Building applications on Irys will benefit from faster block times and programmable data L2s will need fast finality for composability.