

---

# Hypercore

See the full API docs at [docs.pears.com](https://docs.pears.com)

Hypercore is a secure, distributed append-only log.

Built for sharing large datasets and streams of real time data

## Features

- **Sparse replication.** Only download the data you are interested in.
- **Realtime.** Get the latest updates to the log fast and securely.
- **Performant.** Uses a simple flat file structure to maximize I/O performance.
- **Secure.** Uses signed merkle trees to verify log integrity in real time.
- **Modular.** Hypercore aims to do one thing and one thing well - distributing a stream of data.

Note that the latest release is Hypercore 10, which adds support for truncate and many other things. Version 10 is not compatible with earlier versions (9 and earlier), but is considered LTS, meaning the storage format and wire protocol is forward compatible with future versions.

## Install

```
1 npm install hypercore
```

## API

**const core = new Hypercore(storage, [key], [options])** Make a new Hypercore instance.

**storage** should be set to a directory where you want to store the data and core metadata.

```
1 const core = new Hypercore('./directory') // store data in ./directory
```

Alternatively you can pass a function instead that is called with every filename Hypercore needs to function and return your own abstract-random-access instance that is used to store the data.

```
1 const RAM = require('random-access-memory')
2 const core = new Hypercore((filename) => {
3   // filename will be one of: data, bitfield, tree, signatures, key,
4   // the data file will contain all your data concatenated.
5   secret_key
6 })
```

---

```
5
6 // just store all files in ram by returning a random-access-memory
  instance
7 return new RAM()
8 }
```

Per default Hypercore uses random-access-file. This is also useful if you want to store specific files in other directories.

Hypercore will produce the following files:

- **oplog** - The internal truncating journal/oplog that tracks mutations, the public key and other metadata.
- **tree** - The Merkle Tree file.
- **bitfield** - The bitfield of which data blocks this core has.
- **data** - The raw data of each block.

Note that **tree**, **data**, and **bitfield** are normally heavily sparse files.

**key** can be set to a Hypercore public key. If you do not set this the public key will be loaded from storage. If no key exists a new key pair will be generated.

**options** include:

```
1 {
2   createIfMissing: true, // create a new Hypercore key pair if none was
    present in storage
3   overwrite: false, // overwrite any old Hypercore that might already
    exist
4   sparse: true, // enable sparse mode, counting unavailable blocks
    towards core.length and core.byteLength
5   valueEncoding: 'json' | 'utf-8' | 'binary', // defaults to binary
6   encodeBatch: batch => { ... }, // optionally apply an encoding to
    complete batches
7   keyPair: kp, // optionally pass the public key and secret key as a
    key pair
8   encryptionKey: k, // optionally pass an encryption key to enable
    block encryption
9   onwait: () => {}, // hook that is called if gets are waiting for
    download
10  timeout: 0, // wait at max some milliseconds (0 means no timeout)
11  writable: true, // disable appends and truncates
12  inflightRange: null // Advanced option. Set to [minInflight,
    maxInflight] to change the min and max inflight blocks per peer
    when downloading.
13 }
```

You can also set **valueEncoding** to any abstract-encoding or compact-encoding instance.

---

valueEncodings will be applied to individual blocks, even if you append batches. If you want to control encoding at the batch-level, you can use the `encodeBatch` option, which is a function that takes a batch and returns a binary-encoded batch. If you provide a custom valueEncoding, it will not be applied prior to `encodeBatch`.

**const { length, byteLength } = await core.append(block)** Append a block of data (or an array of blocks) to the core. Returns the new length and byte length of the core.

```
1 // simple call append with a new block of data
2 await core.append(Buffer.from('I am a block of data'))
3
4 // pass an array to append multiple blocks as a batch
5 await core.append([Buffer.from('batch block 1'), Buffer.from('batch
  block 2')])
```

**const block = await core.get(index, [options])** Get a block of data. If the data is not available locally this method will prioritize and wait for the data to be downloaded.

```
1 // get block #42
2 const block = await core.get(42)
3
4 // get block #43, but only wait 5s
5 const blockIfFast = await core.get(43, { timeout: 5000 })
6
7 // get block #44, but only if we have it locally
8 const blockLocal = await core.get(44, { wait: false })
```

options include:

```
1 {
2   wait: true, // wait for block to be downloaded
3   onwait: () => {}, // hook that is called if the get is waiting for
  download
4   timeout: 0, // wait at max some milliseconds (0 means no timeout)
5   valueEncoding: 'json' | 'utf-8' | 'binary', // defaults to the core's
  valueEncoding
6   decrypt: true // automatically decrypts the block if encrypted
7 }
```

**const has = await core.has(start, [end])** Check if the core has all blocks between `start` and `end`.

---

**const updated = await core.update([options])** Waits for initial proof of the new core length until all `findingPeers` calls has finished.

```
1 const updated = await core.update()
2
3 console.log('core was updated?', updated, 'length is', core.length)
```

`options` include:

```
1 {
2   wait: false
3 }
```

Use `core.findingPeers()` or `{ wait: true }` to make `await core.update()` blocking.

**const [index, relativeOffset] = await core.seek(byteOffset, [options])**  
Seek to a byte offset.

Returns `[index, relativeOffset]`, where `index` is the data block the `byteOffset` is contained in and `relativeOffset` is the relative byte offset in the data block.

```
1 await core.append([Buffer.from('abc'), Buffer.from('d'), Buffer.from('efg')])
2
3 const first = await core.seek(1) // returns [0, 1]
4 const second = await core.seek(3) // returns [1, 0]
5 const third = await core.seek(5) // returns [2, 1]
```

```
1 {
2   wait: true, // wait for data to be downloaded
3   timeout: 0 // wait at max some milliseconds (0 means no timeout)
4 }
```

**const stream = core.createReadStream([options])** Make a read stream to read a range of data out at once.

```
1 // read the full core
2 const fullStream = core.createReadStream()
3
4 // read from block 10-15
5 const partialStream = core.createReadStream({ start: 10, end: 15 })
6
7 // pipe the stream somewhere using the .pipe method on Node.js or
  // consume it as
8 // an async iterator
```

---

```
9
10 for await (const data of fullStream) {
11   console.log('data:', data)
12 }
```

options include:

```
1 {
2   start: 0,
3   end: core.length,
4   live: false,
5   snapshot: true // auto set end to core.length on open or update it on
                  // every read
6 }
```

**const bs = core.createByteStream([options])** Make a byte stream to read a range of bytes.

```
1 // Read the full core
2 const fullStream = core.createByteStream()
3
4 // Read from byte 3, and from there read 50 bytes
5 const partialStream = core.createByteStream({ byteOffset: 3, byteLength
  : 50 })
6
7 // Consume it as an async iterator
8 for await (const data of fullStream) {
9   console.log('data:', data)
10 }
11
12 // Or pipe it somewhere like any stream:
13 partialStream.pipe(process.stdout)
```

options include:

```
1 {
2   byteOffset: 0,
3   byteLength: core.byteLength - options.byteOffset,
4   prefetch: 32
5 }
```

**const cleared = await core.clear(start, [end], [options])** Clear stored blocks between `start` and `end`, reclaiming storage when possible.

```
1 await core.clear(4) // clear block 4 from your local cache
2 await core.clear(0, 10) // clear block 0-10 from your local cache
```

---

The core will also gossip to peers it is connected to, that is no longer has these blocks.

`options` include:

```
1 {
2   diff: false // Returned `cleared` bytes object is null unless you
                enable this
3 }
```

**`await core.truncate(newLength, [forkId])`** Truncate the core to a smaller length.

Per default this will update the fork id of the core to + 1, but you can set the fork id you prefer with the option. Note that the fork id should be monotonely incrementing.

**`await core.purge()`** Purge the hypercore from your storage, completely removing all data.

**`const hash = await core.treeHash([length])`** Get the Merkle Tree hash of the core at a given length, defaulting to the current length of the core.

**`const range = core.download([range])`** Download a range of data.

You can await when the range has been fully downloaded by doing:

```
1 await range.done()
```

A range can have the following properties:

```
1 {
2   start: startIndex,
3   end: nonInclusiveEndIndex,
4   blocks: [index1, index2, ...],
5   linear: false // download range linearly and not randomly
6 }
```

To download the full core continuously (often referred to as non sparse mode) do

```
1 // Note that this will never be considered downloaded as the range
2 // will keep waiting for new blocks to be appended.
3 core.download({ start: 0, end: -1 })
```

To download a discrete range of blocks pass a list of indices.

```
1 core.download({ blocks: [4, 9, 7] })
```

To cancel downloading a range simply destroy the range instance.

---

```
1 // will stop downloading now
2 range.destroy()
```

**const session = await core.session([options])** Creates a new Hypercore instance that shares the same underlying core.

You must close any session you make.

Options are inherited from the parent instance, unless they are re-set.

`options` are the same as in the constructor.

**const info = await core.info([options])** Get information about this core, such as its total size in bytes.

The object will look like this:

```
1 Info {
2   key: Buffer(...),
3   discoveryKey: Buffer(...),
4   length: 18,
5   contiguousLength: 16,
6   byteLength: 742,
7   fork: 0,
8   padding: 8,
9   storage: {
10    oplog: 8192,
11    tree: 4096,
12    blocks: 4096,
13    bitfield: 4096
14  }
15 }
```

`options` include:

```
1 {
2   storage: false // get storage estimates in bytes, disabled by default
3 }
```

**await core.close()** Fully close this core.

**core.on('close')** Emitted when the core has been fully closed.

---

**await core.ready()** Wait for the core to fully open.

After this has called `core.length` and other properties have been set.

In general you do NOT need to wait for `ready`, unless checking a synchronous property, as all internals await this themselves.

**core.on('ready')** Emitted after the core has initially opened all its internal state.

**core.writable** Can we append to this core?

Populated after `ready` has been emitted. Will be `false` before the event.

**core.readable** Can we read from this core? After closing the core this will be false.

Populated after `ready` has been emitted. Will be `false` before the event.

**core.id** String containing the id (z-base-32 of the public key) identifying this core.

Populated after `ready` has been emitted. Will be `null` before the event.

**core.key** Buffer containing the public key identifying this core.

Populated after `ready` has been emitted. Will be `null` before the event.

**core.keyPair** Object containing buffers of the core's public and secret key

Populated after `ready` has been emitted. Will be `null` before the event.

**core.discoveryKey** Buffer containing a key derived from the core's public key. In contrast to `core.key` this key does not allow you to verify the data but can be used to announce or look for peers that are sharing the same core, without leaking the core key.

Populated after `ready` has been emitted. Will be `null` before the event.

**core.encryptionKey** Buffer containing the optional block encryption key of this core. Will be `null` unless block encryption is enabled.



---

**core.length** How many blocks of data are available on this core? If `sparse: false`, this will equal `core.contiguousLength`.

Populated after `ready` has been emitted. Will be 0 before the event.

**core.contiguousLength** How many blocks are contiguously available starting from the first block of this core?

Populated after `ready` has been emitted. Will be 0 before the event.

**core.fork** What is the current fork id of this core?

Populated after `ready` has been emitted. Will be 0 before the event.

**core.padding** How much padding is applied to each block of this core? Will be 0 unless block encryption is enabled.

**const stream = core.replicate(isInitiatorOrReplicationStream)** Create a replication stream. You should pipe this to another Hypercore instance.

The `isInitiator` argument is a boolean indicating whether you are the initiator of the connection (ie the client) or if you are the passive part (ie the server).

If you are using a P2P swarm like Hyperswarm you can know this by checking if the swarm connection is a client socket or server socket. In Hyperswarm you can check that using the `client` property on the peer details object

If you want to multiplex the replication over an existing Hypercore replication stream you can pass another stream instance instead of the `isInitiator` boolean.

```
1 // assuming we have two cores, localCore + remoteCore, sharing the same
  key
2 // on a server
3 const net = require('net')
4 const server = net.createServer(function (socket) {
5   socket.pipe(remoteCore.replicate(false)).pipe(socket)
6 })
7
8 // on a client
9 const socket = net.connect(...)
10 socket.pipe(localCore.replicate(true)).pipe(socket)
```

---

**const done = core.findingPeers()** Create a hook that tells Hypercore you are finding peers for this core in the background. Call `done` when your current discovery iteration is done. If you're using Hyperswarm, you'd normally call this after a `swarm.flush()` finishes.

This allows `core.update` to wait for either the `findingPeers` hook to finish or one peer to appear before deciding whether it should wait for a merkle tree update before returning.

**core.on('append')** Emitted when the core has been appended to (i.e. has a new length / byteLength), either locally or remotely.

**core.on('truncate', ancestors, forkId)** Emitted when the core has been truncated, either locally or remotely.

**core.on('peer-add')** Emitted when a new connection has been established with a peer.

**core.on('peer-remove')** Emitted when a peer's connection has been closed.

**core.on('upload', index, byteLength, peer)** Emitted when a block is uploaded to a peer.

**core.on('download', index, byteLength, peer)** Emitted when a block is downloaded from a peer.