What happens when files are shared on the decentralized web?

A step-by-step explanation of how and why files are added, organized, and shared using two of the leading peer-to-peer protocols (Dat and IPFS).

1. Initializing

Suppose Alice wants to start using the decentralized web.

She installs IFPS or Dat and creates a **working directory** (a file folder that connects with the Dat or IPFS network) on her computer. Now she's a peer (aka "node") on the peer-to-peer network and she can request files from other peers or share her own files.

Adding Files

Alice adds some of her files to the working directory to share them.

The set of files saved in the working directory can include a variety of file types and sizes.

For example...

File Name & Type	File Size
summary.txt	3.5 KB
img_254.png	249.1 KB
survey_2018.csv	510.0 KB

2. Chunking

When Alice adds the files to the working directory, she sets off some behind-the-scenes steps. First, the files are broken up into small chunks.

Chunking Files

The larger files are broken into chunks, so that there are no chunks bigger than the maximum chunk size.

In this example, we use a maximum chunk size of 256 KB. The only file larger than 256 KB is divided into 2 chunks — while the other files are too small to need chunking.

For example...

File Name & Type	File Size	Number of chunks	Size of each chunk
summary.txt	3.5 KB	1	3.5 KB
img_254.png	249.1 KB	1	249.1 KB
survey_2018.csv	510.0 KB	2	255.0 KB

3. Hashing

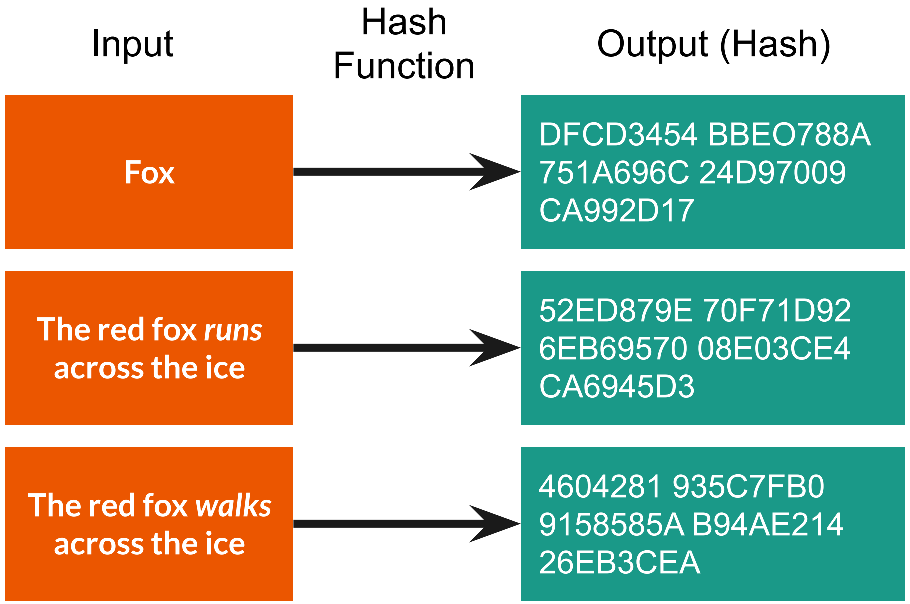
Each chunk is **hashed**.

Next slide: What is hashing?

What is hashing?

Hashing means using a *one-way* function to encrypt and compress data of any size into a text string with the following properties:

- **Fixed size.** Each of the outputs is the same number of characters — regardless of the input length.
- **Deterministic.** Running the hash function on "Fox" will always return the same output: "DFCD3454 BBEO788A 751A696C 24D97009 CA992D17".
- **Collision resistant.** Each output is unique to the input — even when just one word in the input is changed. It is nearly impossible to find two different inputs with the same hash.


Input
Hash Function
Output (Hash)
Fox
DFCD3454 BBEO788A 751A696C 24D97009 CA992D17
The red fox *runs* across the ice
52ED879E 70F71D92 6EB69570 08E03CE4 CA6945D3
The red fox *walks* across the ice
4604281 935C7FB0 9158585A B94AE214 26EB3CEA

Why is hashing useful?

File comparison and verification. Hashing can be used to check if there are any changes in the data — identical files will have the same hash, but a file with even a slight change may have a completely different hash. With hashing, users can quickly confirm that a copy of the data is an exact match.

Efficiency. With hashing, large amounts of data can be made identifiable with just fixed-size text strings.

For example...

Each of the four chunks is passed through a hash function.

File Name & Type	File Size	Chunk Size	Hash
summary.txt	3.5 KB	3.5 KB	f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
img_254.png	249.1 KB	249.1 KB	4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
survey_2018.csv	510.0 KB	255.0 KB	3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08
		255.0 KB	22ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94

4. Organizing

The hashes are organized into a **Merkle DAG**.

Next slide: What are Merkle DAGs?

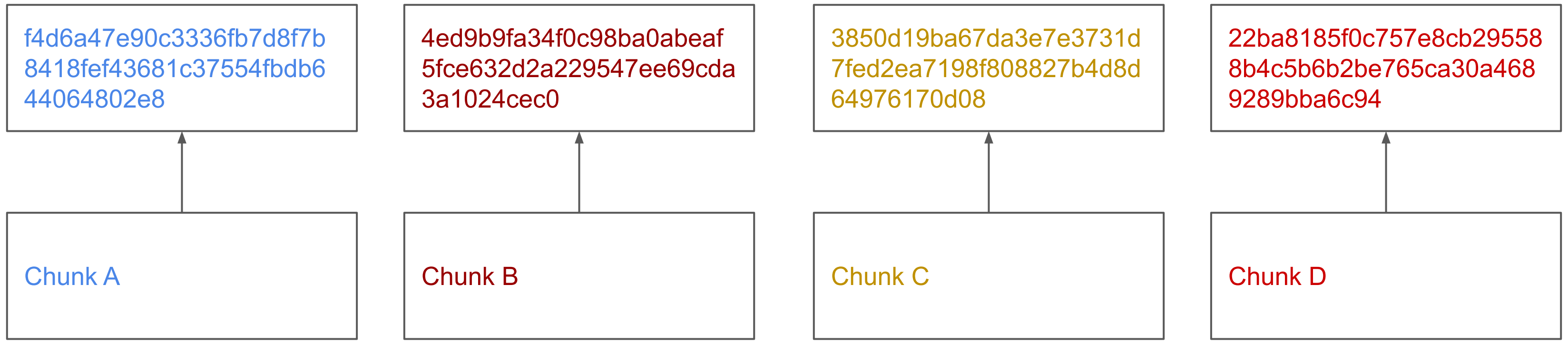
What are Merkle DAGs?

Merkle DAGs (directed acyclic graphs) are a type of hierarchical hash tree.

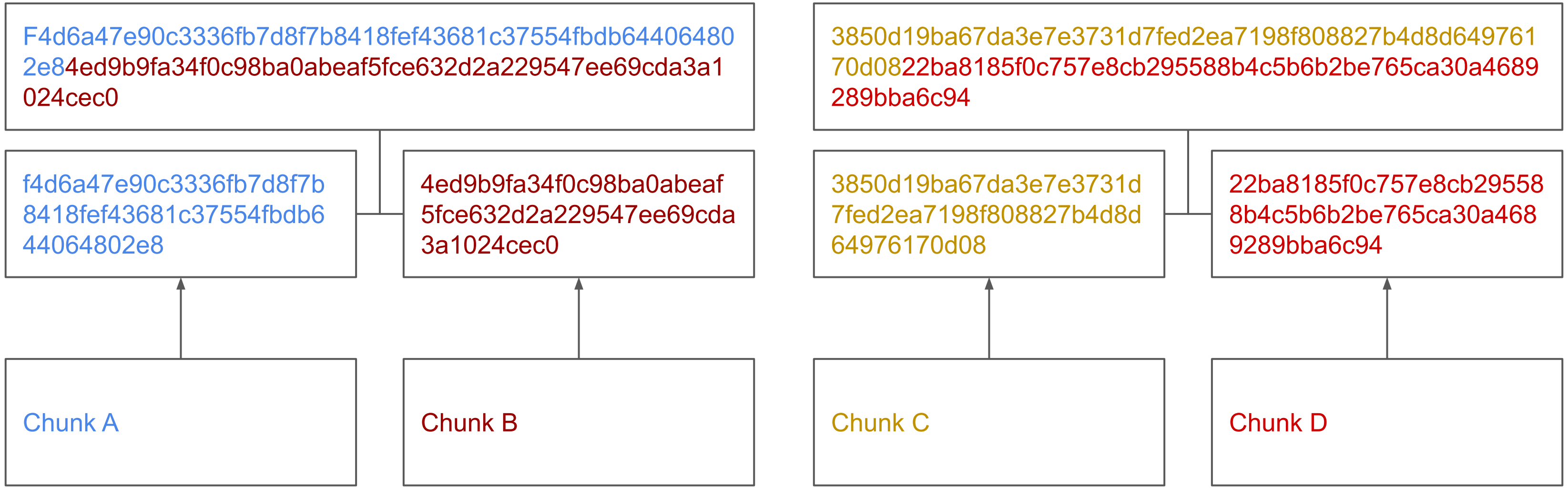
Let's see how they work using our example. Now that Alice's dataset is broken into four chunks, how can we efficiently organize their hashes?

Chunk A
Chunk B
Chunk C
Chunk D

1. Hash each data chunk.

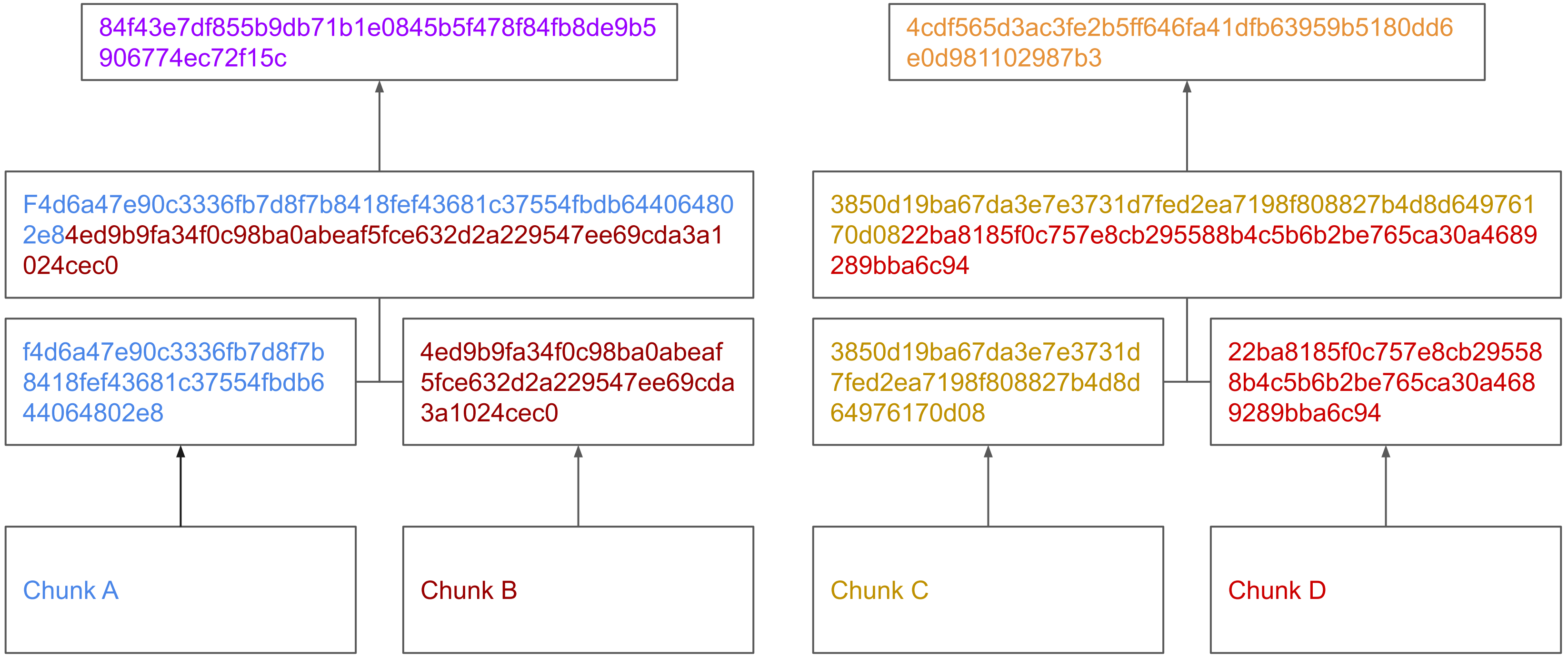

f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08
22ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
Chunk A
Chunk B
Chunk C
Chunk D

1. Hash each data chunk.
2. Concatenate adjacent hashes to form hash pairs.

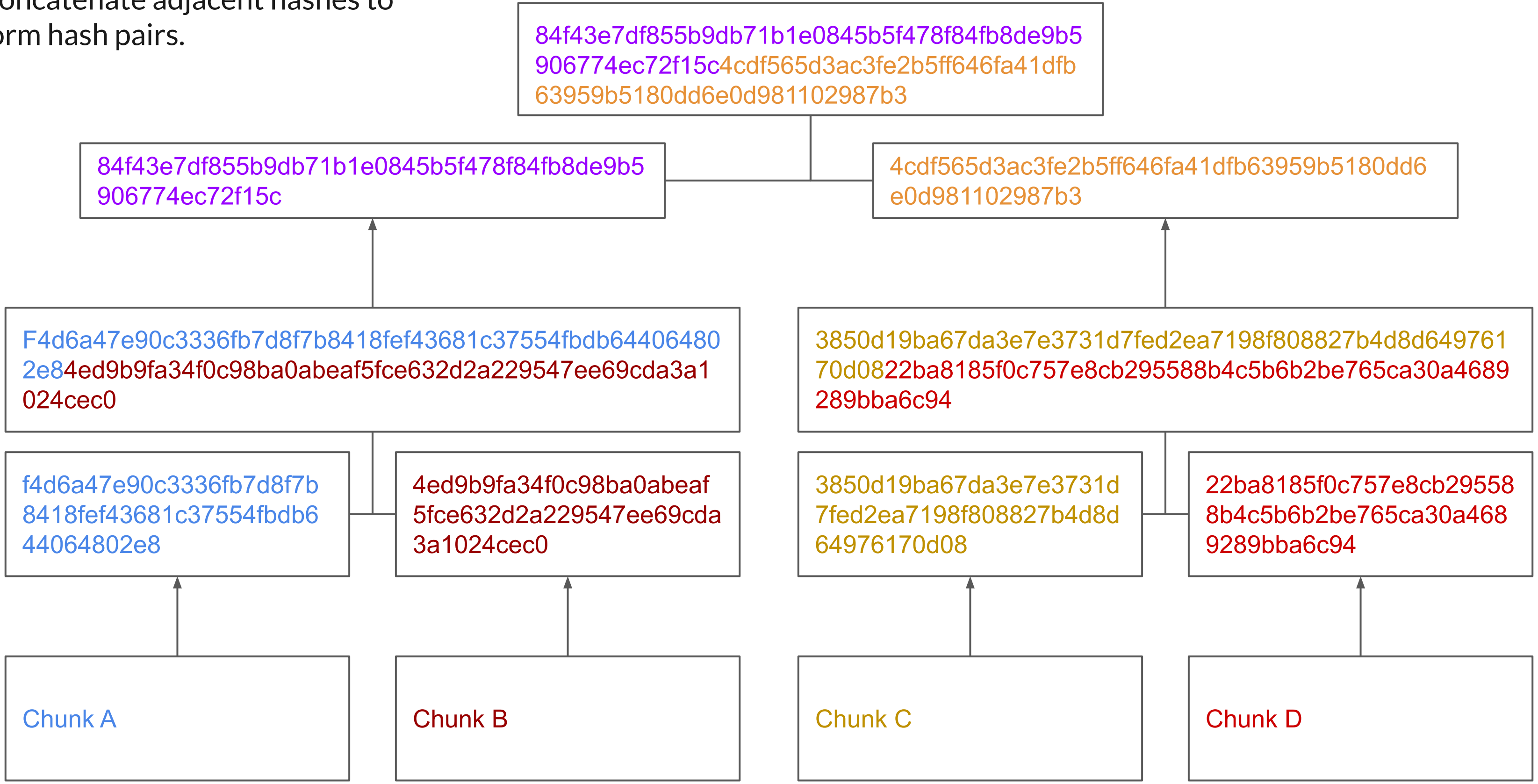


F4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb64406480 2e84ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1 024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d649761 70d0822ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689 289bba6c94
f4d6a47e90c3336fb7d8f7b 8418fef43681c37554fbdb6 44064802e8
4ed9b9fa34f0c98ba0abeaf 5fce632d2a229547ee69cda 3a1024cec0
3850d19ba67da3e7e3731d 7fed2ea7198f808827b4d8d 64976170d08
22ba8185f0c757e8cb29558 8b4c5b6b2be765ca30a468 9289bba6c94
Chunk A
Chunk B
Chunk C
Chunk D

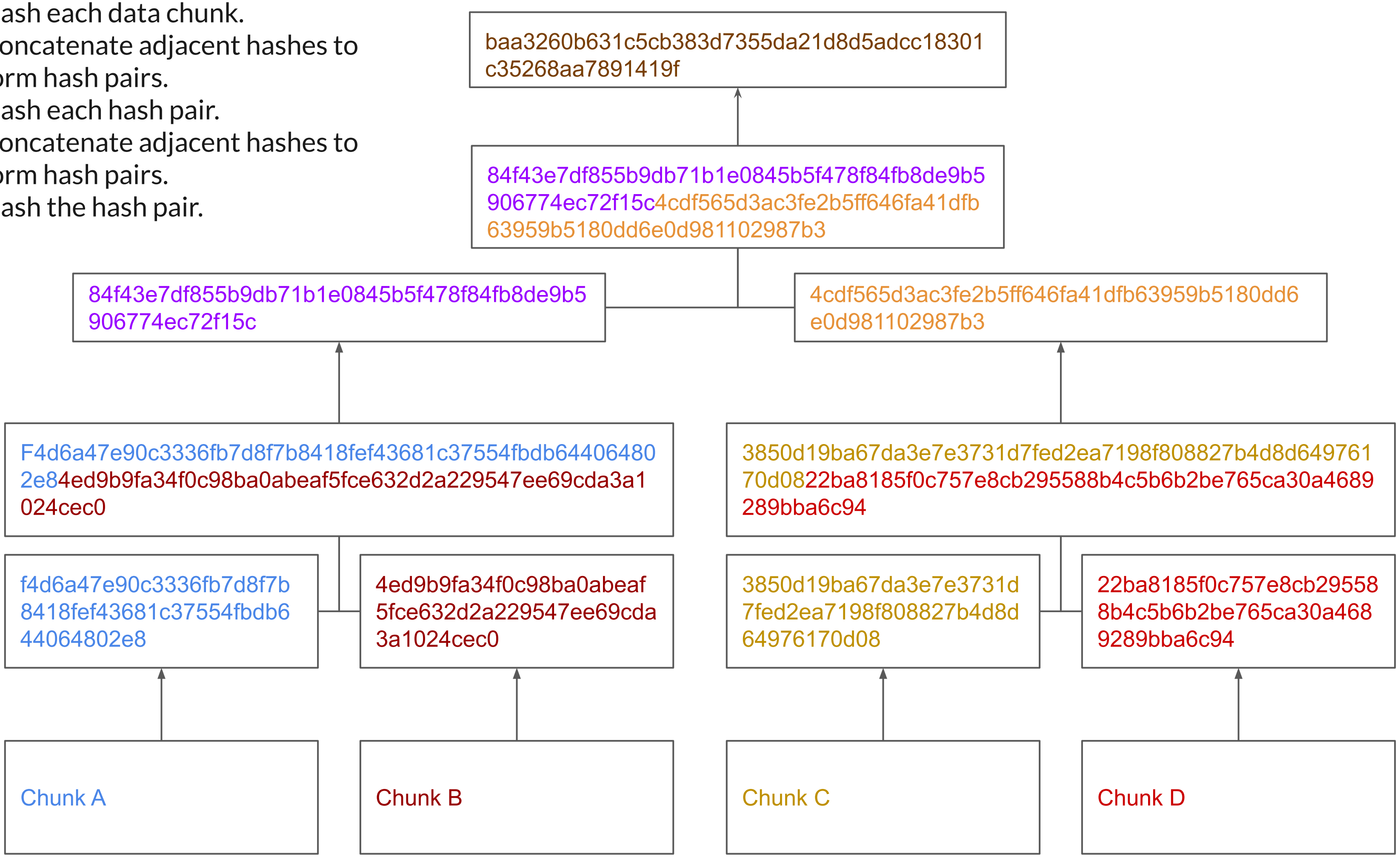
1. Hash each data chunk.
2. Concatenate adjacent hashes to form hash pairs.
3. Hash each hash pair.



1. Hash each data chunk.
2. Concatenate adjacent hashes to form hash pairs.
3. Hash each hash pair.
4. Concatenate adjacent hashes to form hash pairs.


84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c
4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
F4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e84ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d0822ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08
22ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
Chunk A
Chunk B
Chunk C
Chunk D

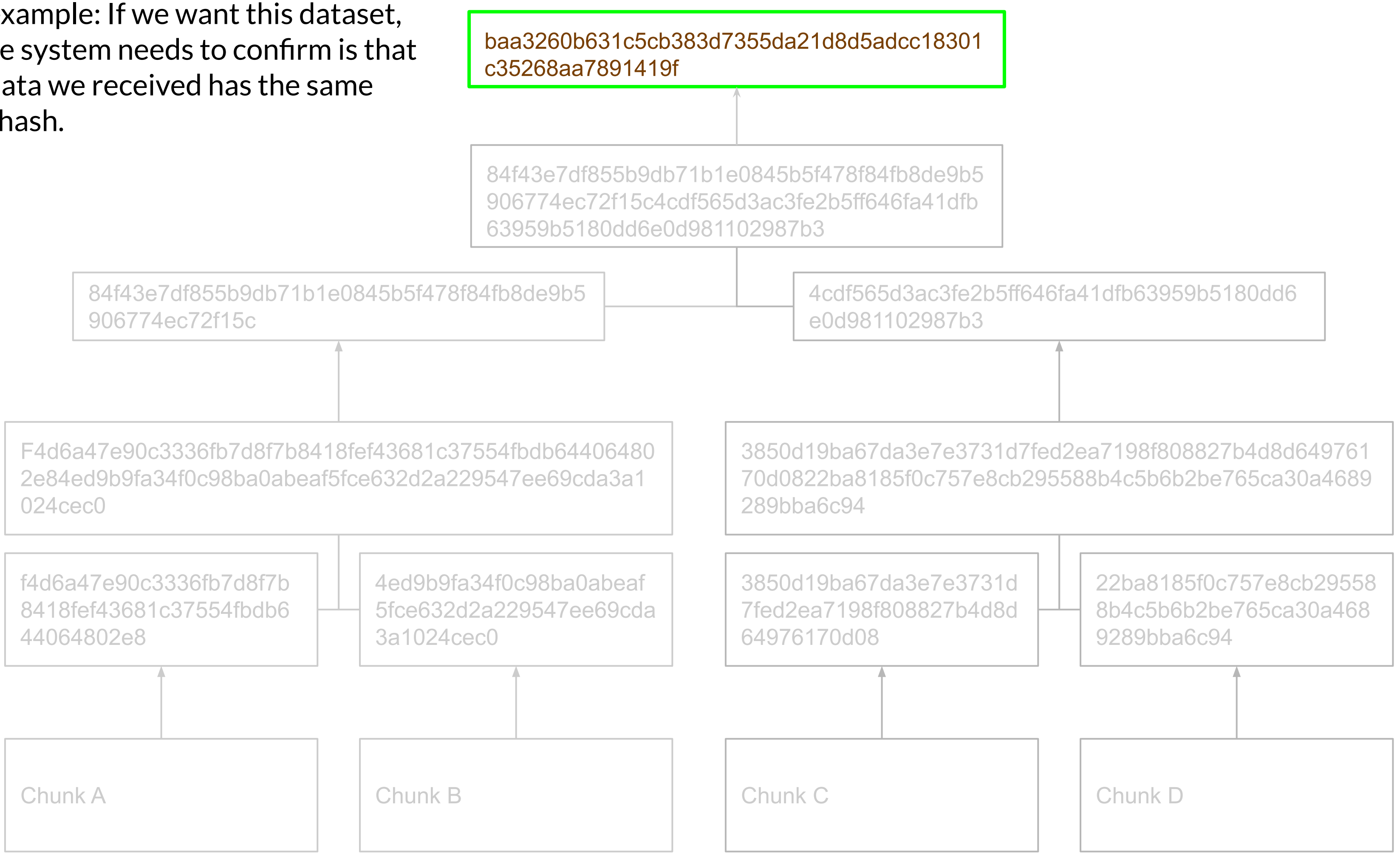
1. Hash each data chunk.
2. Concatenate adjacent hashes to form hash pairs.
3. Hash each hash pair.
4. Concatenate adjacent hashes to form hash pairs.
5. Hash the hash pair.


baa3260b631c5cb383d7355da21d8d5adcc18301c35268aa7891419f
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c
4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
F4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e84ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d0822ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08
22ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
Chunk A
Chunk B
Chunk C
Chunk D

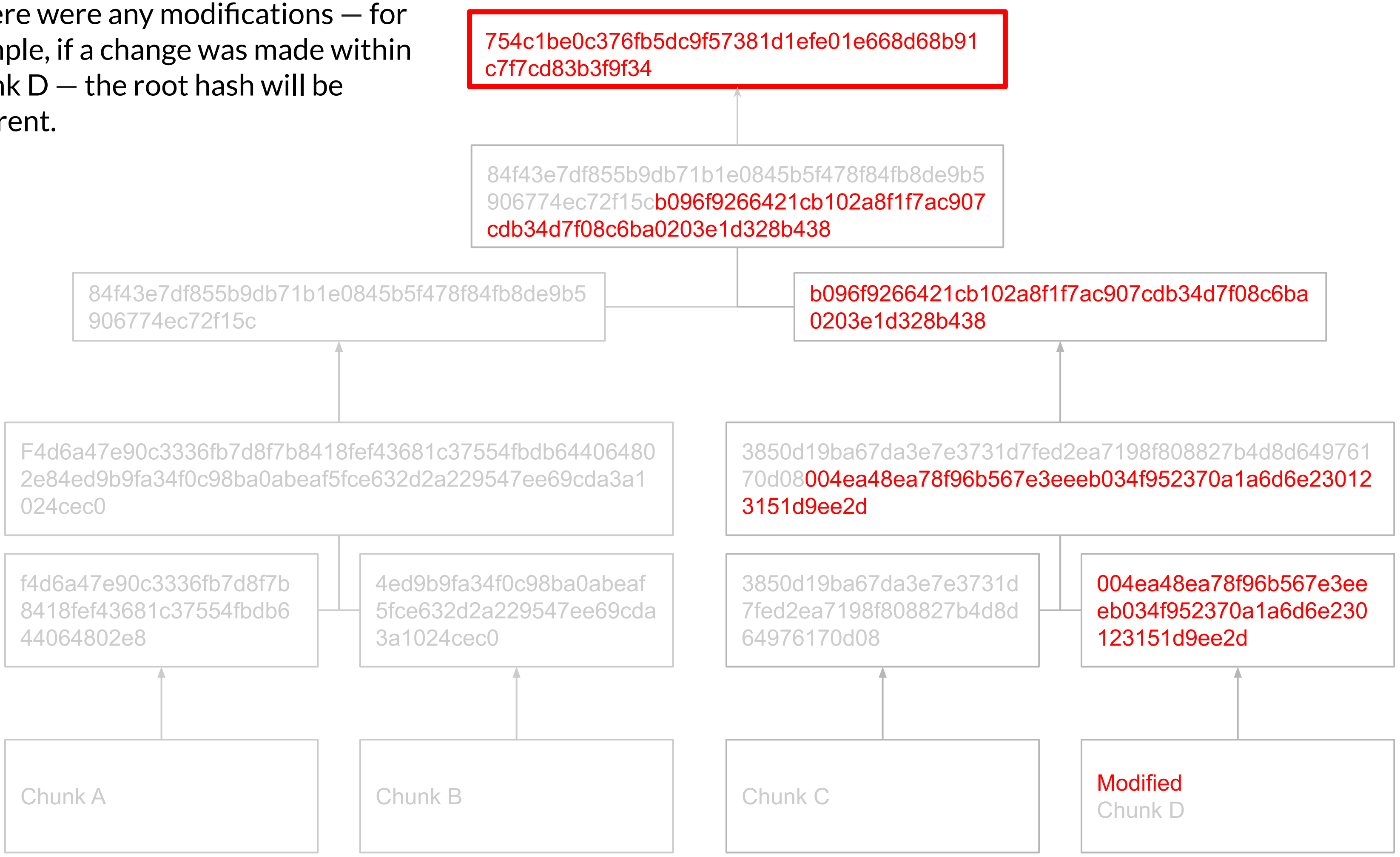
Why are Merkle DAGs useful? (1/3)

Validation. The entire dataset can be represented and verified by the final hash (the “root hash”). If the root hash of the data we want matches the root hash of the dataset we received, that means the contents are exactly the same — without any modified or missing information.

For example: If we want this dataset, all the system needs to confirm is that the data we received has the same root hash.


baa3260b631c5cb383d7355da21d8d5adcc18301c35268aa7891419f
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c
4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
F4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e84ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d0822ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08
22ba8185f0c757e8cb295588b4c5b6b2be765ca30a4689289bba6c94
Chunk A
Chunk B
Chunk C
Chunk D

If there were any modifications — for example, if a change was made within Chunk D — the root hash will be different.

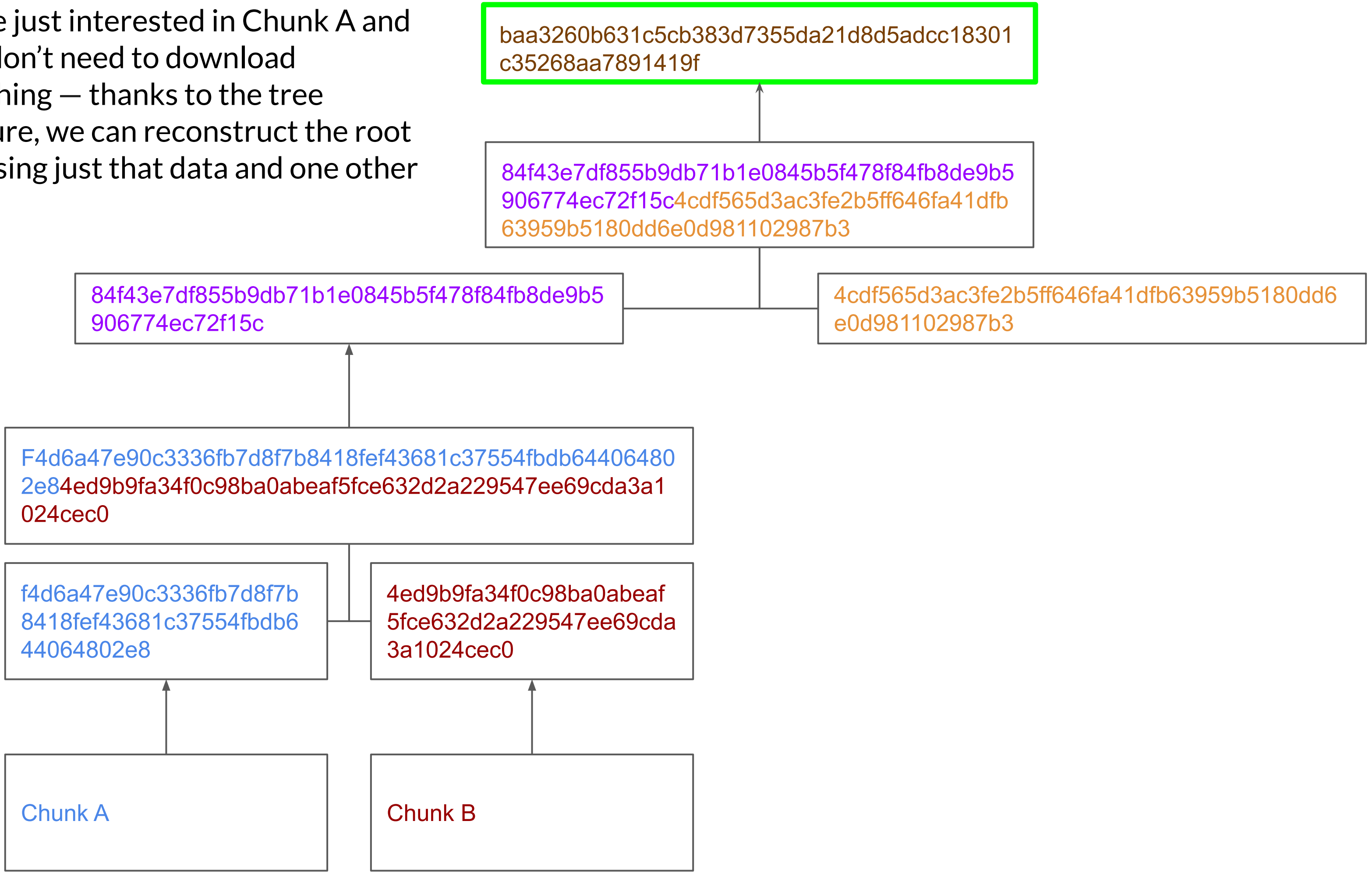

754c1be0c376fb5dc9f57381d1efe01e668d68b91c7f7cd83b3f9f34
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15cb096f9266421cb102a8f1f7ac907cdb34d7f08c6ba0203e1d328b438
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c
b096f9266421cb102a8f1f7ac907cdb34d7f08c6ba0203e1d328b438
F4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e84ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08004ea48ea78f96b567e3eeeb034f952370a1a6d6e230123151d9ee2d
f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
3850d19ba67da3e7e3731d7fed2ea7198f808827b4d8d64976170d08
004ea48ea78f96b567e3eeeb034f952370a1a6d6e230123151d9ee2d
Chunk A
Chunk B
Chunk C
Modified Chunk D

Why are Merkle DAGs useful? (2/3)

Partial verification. The Merkle DAG structure also makes it possible to verify the contents of a subsection of data without downloading all the files.

The root hash can be reconstructed using just the subsection of data and the top-level hashes corresponding to the other data — and then the root hash with the newly received data can be checked against the original root hash that came from a trusted source.

If we’re just interested in Chunk A and B, we don’t need to download everything — thanks to the tree structure, we can reconstruct the root hash using just that data and one other hash.


baa3260b631c5cb383d7355da21d8d5adcc18301c35268aa7891419f
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c
4cdf565d3ac3fe2b5ff646fa41dfb63959b5180dd6e0d981102987b3
F4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e84ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8
4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0
Chunk A
Chunk B

Why are Merkle DAGs useful? (3/3)

Efficiency and scalability. Because hash functions return fixed-length outputs, efficiency gains from hashing build upon each other when Merkle DAGs use hashes to identify other groups of hashes.

For example, through this process we can now use a single 57-character string to identify a set of two 57-character strings, which in turn identify a .txt file and a .png file. This efficiency is important when working with large datasets with many-layered Merkle DAGs.

Chunk A (the file “summary.txt”)	f4d6a47e90c3336fb7d8f7b8418fef43681c37554fbdb644064802e8	84f43e7df855b9db71b1e0845b5f478f84fb8de9b5906774ec72f15c
Chunk B (the file “img_254.png”)	4ed9b9fa34f0c98ba0abeaf5fce632d2a229547ee69cda3a1024cec0	

5. Accessing

Once the files are chunked, hashed, and organized into Merkle DAGs, other peers on the network can access the files.

Suppose Bob wants to access the set of files that Alice has shared. The way the system identifies data depends on which protocol Alice and Bob are using.

How is the data identified and accessed in IPFS?

To access the data in IPFS, Bob needs Alice or another trusted source to give him the **hash of the dataset's contents**. Then the system will scan the network to find peers with the data that matches the hash.

Note that if Alice modifies the data in any way, the hash will change. If she updates the spreadsheet by adding new results, Bob and other peers will need to use a new hash if they want to access the updated data. This makes IPFS a good choice for *static* data.

How is the data identified and accessed in Dat?

To access the data in Dat, Bob needs Alice or another trusted source to give him the **read key to Alice's working directory**. The read key is a unique, persistent identifier linked to Alice's working directory. The system will scan the network to find peers who have the current contents of Alice's directory.

Even if Alice modifies the data, Bob and other peers can still use the read key to fetch the data in her working directory. If she updates the spreadsheet by adding new results, Bob can use the same read key to access the updated data (as well as a log of the changes she's made in her directory). This makes Dat a good choice for *dynamic* data.

6. Retrieving

Once the system finds and retrieves the data on the network, Bob clones it onto his own computer.

Rehosting

Bob simultaneously receives the data and begins hosting it.

When someone else on the network searches for the same data, Alice doesn’t need to send all four chunks herself anymore — Bob can also send parts of the data.

The more people clone the data to their own computer, the more hosts there are — collectively sharing bandwidth and disk space while preserving the data through a decentralized, resilient network.

Sources

IPFS intro: https://docs.ipfs.io
Dat intro: https://docs.datproject.org/
IPFS summary: https://dev.to/jaybeekeeper/learning-about-dat-protocol-and-decentralization--1ghi
Chunking: https://medium.com/textileio/whats-really-happening-when-you-add-a-file-to-ipfs-ae3b8b5e4b0f
Hashing: https://komodoplatform.com/cryptographic-hash-function/
Merkle trees:
https://medium.com/byzantine-studio/blockchain-fundamentals-what-is-a-merkle-tree-d44c529391d7
Merkle trees: https://taravancil.com/blog/how-merkle-trees-enable-decentralized-web/
Dat structure: https://datprotocol.github.io/how-dat-works/
Dat diagrams: https://hackmd.io/-hJeNqjkS4WlMJ0P5PRw-A