Detailed Design of Tectonic

Learn the behavior and working of the Tectonic in detail.

We'll cover the following

We have already discussed the overview of the Tectonic architecture, and now it is time to learn about its components and how they interact with each other.

Client Library

The Tectonic client code needs to map its user's read/write requests to appropriate Tectonic API calls by interacting with the Metadata and Chunk Stores. As an example, when a user asks to read from a file, the Client Library utilizes the metadata service to find the corresponding data chunks on disks (by series of mapping from name layer to file layer to block layer to the chunks. We'll learn about this series of mappings a bit later in this lesson). Client code can cache some of this information and then goes to the specific storage node to read the data. All such intelligence makes the Client Library fairly complex.

Tectonic groups chunk to make blocks to amortize the cost of metadata keeping. Blocks in Tectonic are encoded using either Reed-Solomon (RS) encoding, which

divides the block into X data chunks and Y code/parity chunks, or are replicated, in which N identical chunks each hold a full copy of the block.

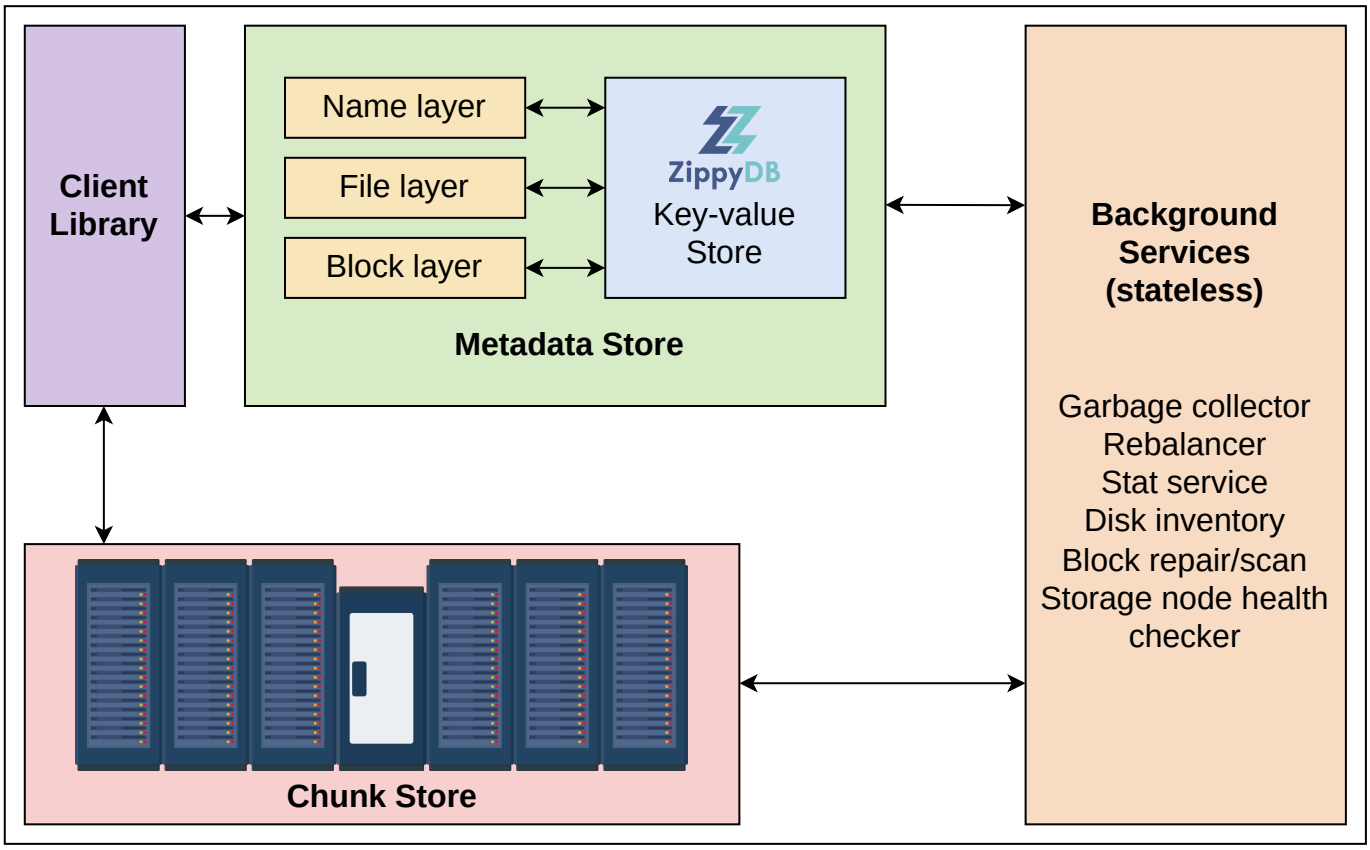
> **Note:** Reed Solomon codes are based on beautiful mathematics such as modular arithmetic and Lagrange interpolation. However, their detailed examination is beyond the scope of this lesson. Reed-Solomon codes are used in many places, such as QR codes, CDs, and other storage and communication media.

Compared to some other distributed file systems (such as GFS), Tectonic only allows a single writer to a file at a time. Tectonic uses locking on files to give exclusive writing permissions to one client. Doing so simplifies writing and replication by using hedged writes (meaning sending the request to a preferred node, if that one fails, we fall back to others). The writing operation is performed using the following steps:

1. When the file is opened for the append operation, the token is added to the file's metadata. This token will be required for mutations on the file.
2. The writer having the matching write token for that block with the file metadata can update that block.
3. When a new writer comes to update the same block, a new token will be generated and updated in the file's metadata (and the old one will be deleted). Since the token is updated, the previous writer will no longer have the right to update the block, and the new writer with the matching token can perform the write operation.

Metadata Store

The **Metadata Store** is a fine-grained partitioning system that uses hash-partitioning and ZippyDB servers for metadata storage to store shard replicas for scalability, load balancing, and operational simplicity.

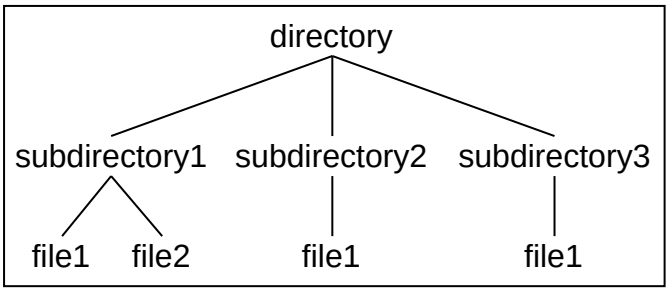


Architecture of Tectonic

Layers of the Metadata Store

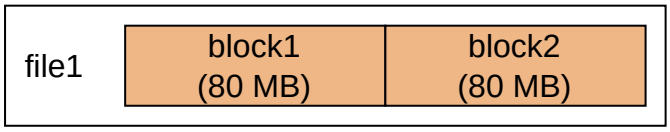
We break down the metadata information into three parts and store them in the respective layer, similar to the data mapping on the hard disk. The Metadata Store has the following logical layers, which are implemented as stateless microservices on top of ZippyDB:

- **Name layer:** This layer profiles the user-readable directory names to subdirectories and files, and file names to file objects.



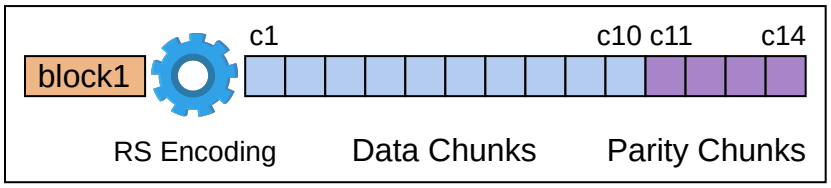
Namespace of directory

- **File layer:** This layer profiles the number of blocks of file objects.



Blocks of file

- **Block layer:** This layer profiles disk locations (list) for each block. It also stores the reverse-index of disk-to-block for maintenance purposes. Blocks are either RS encoded (for space efficiency) or replicated as they are.



RS encoding block

Layered Metadata Schema

Layer	Key	Value	Sharded By	
Name	(`directory_id` , `subdirec_name`) (`directory_id` , `filename`)	`subdirec_info` , `subdirec_id` `file_info` , `file_id`	`directory_id` `directory_id`	d
File	(`file_id` , `block_id`)	`block_info`	`file_id`	fil
Block	`block_id` (`disk_id` , `block_id`)	`list<disk_id>` `chunk_info`	`block_id` `block_id`	blo dis

> **Note:** We can scale the name, file, and block layers independently by separating them.

ZippyDB organizes keys into shards to guarantee that all key-value pairs with the same sharding ID are put into the same shard. This implies that Tectonic's Name layer, which is sharded by `directory_id`, can swiftly provide a list of all subdirectories and files within that directory from a single shard. Similarly, the File

layer, which is sharded by `file_id`, can provide the list of all blocks containing that file's data from a single shard. However, many of the usual recursive filesystem operations (all the operations with `r` flag, such as `ls`, `cp`, `rm`, and many more) cannot be performed since ZippyDB does not support any cross-shard transactions.

How to get the chunk?

The system uses the following steps to get the chunk:

1. Use the `filename` from the current directory in the Name layer.

```
file_info, file_id = get(directory_id, filename)
```

2. Use that `file_id` from the Name layer to access the blocks in the File layer.

```
block_id = get(file_id_)
```

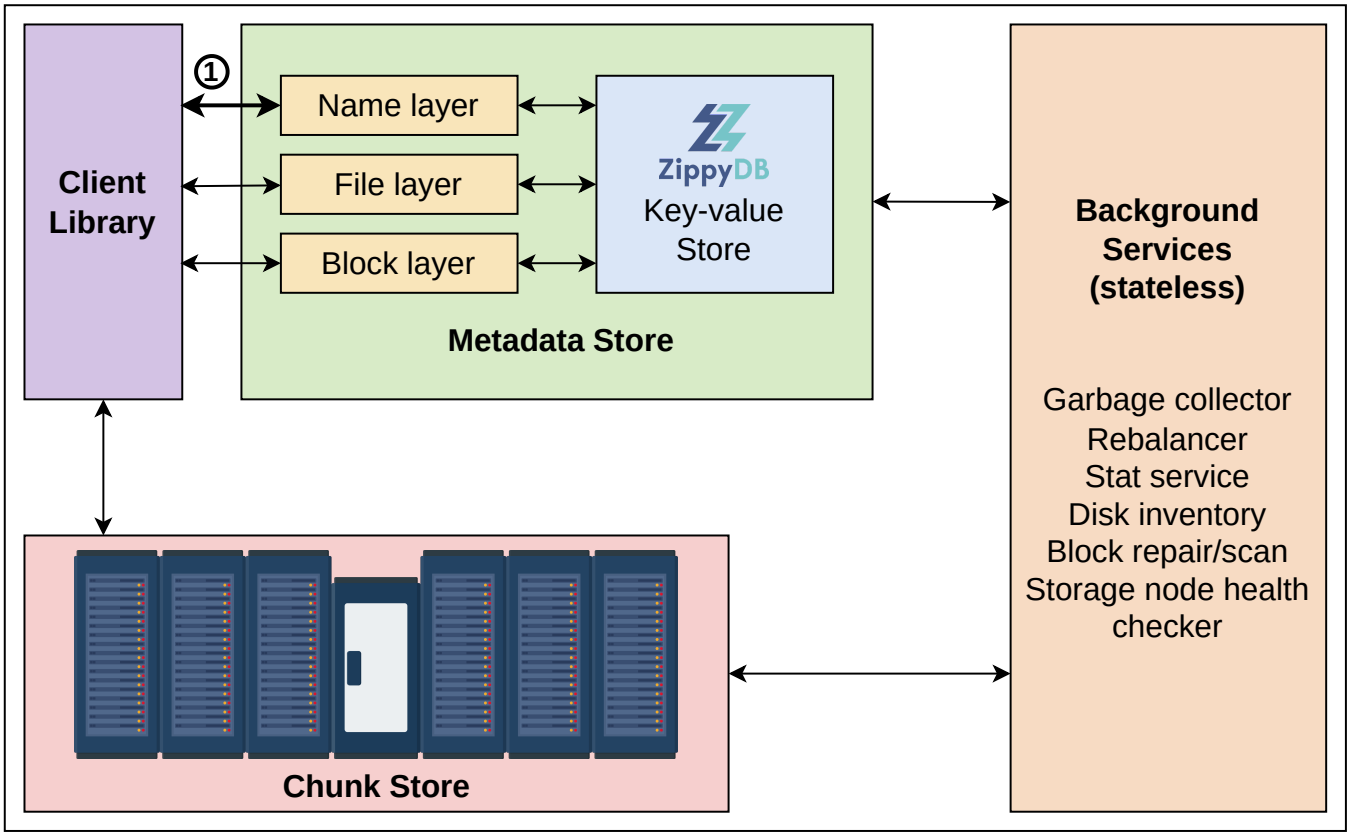
3. Use that `block_id` from the File layer to check which disks these blocks are located in the Block Layer.

```
disk_id = get(block_id)
```

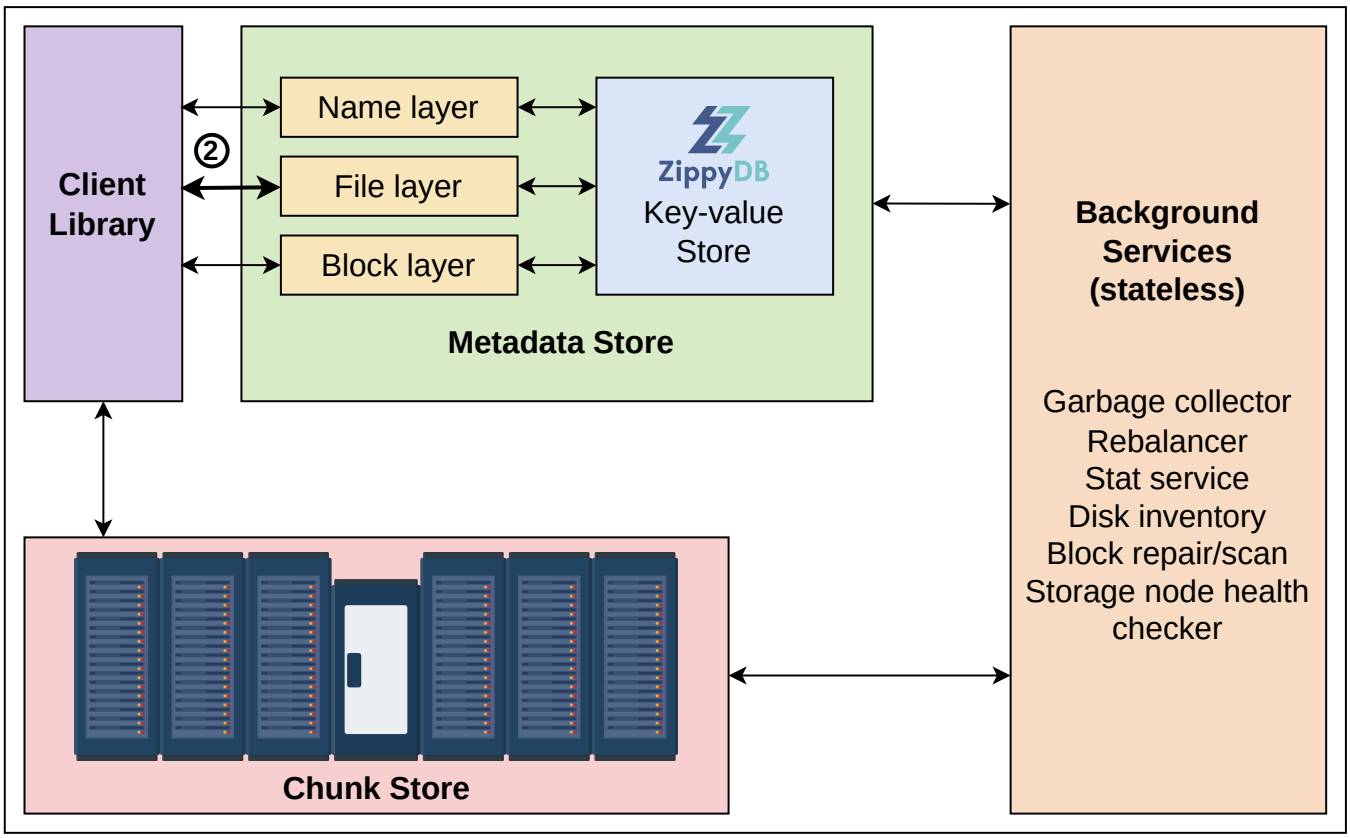
4. Use that `block_id` from the File layer and `disk_id` from the Block layer to access and get the chunk address that contains the data on the Chunk Store.

```
chunk_info = get(disk_id, block_id)
```

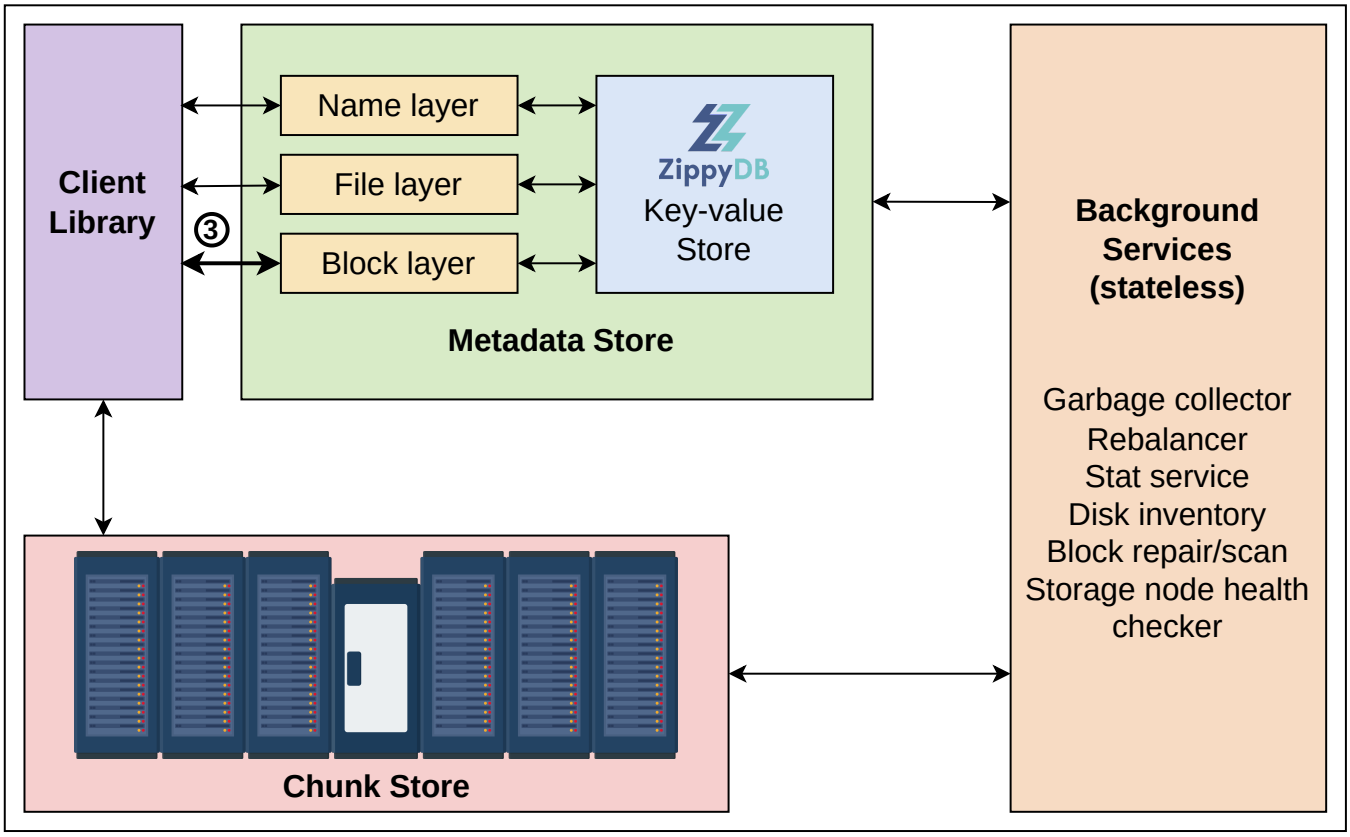
5. Go to the chunk server and perform the operation on the data.



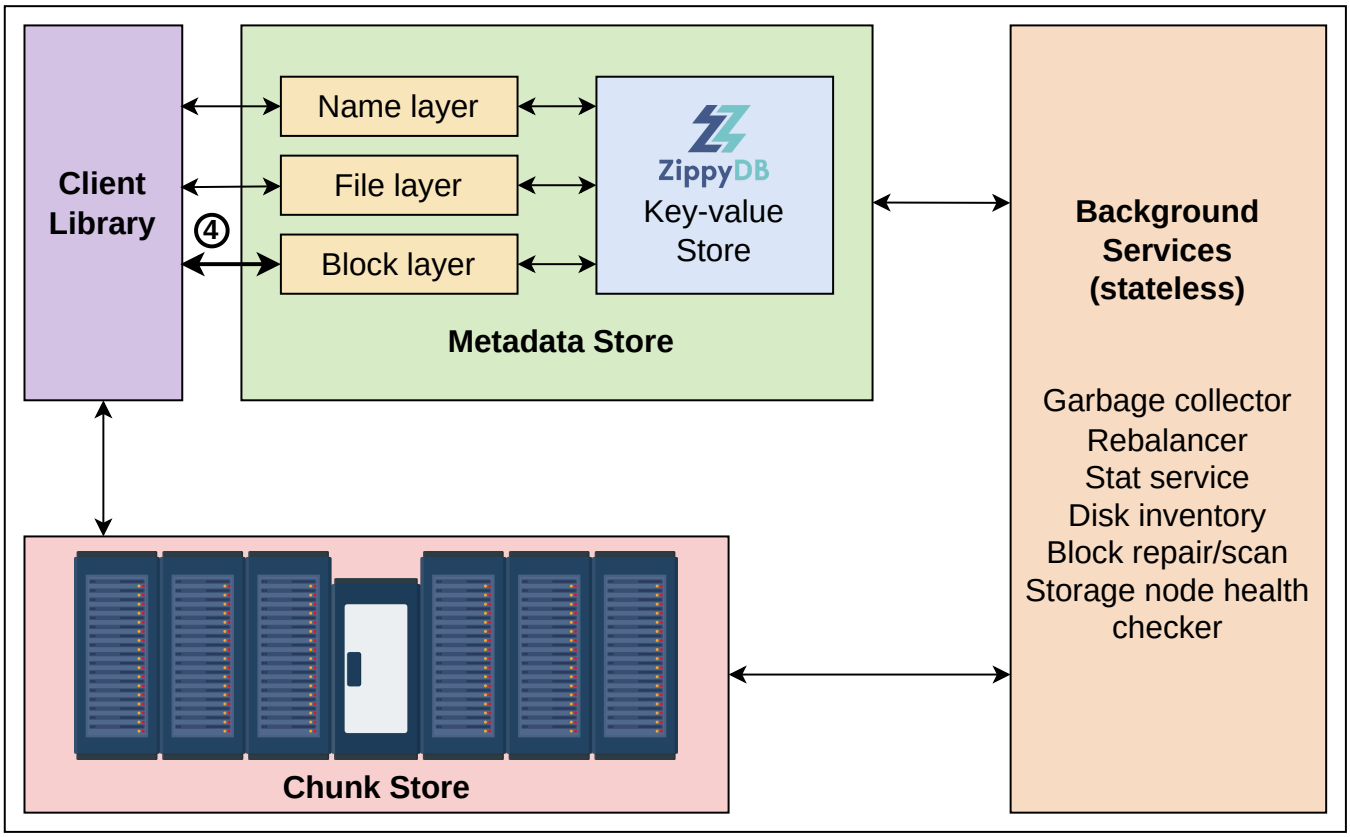
Fetching the file_id from the Name layer using filename



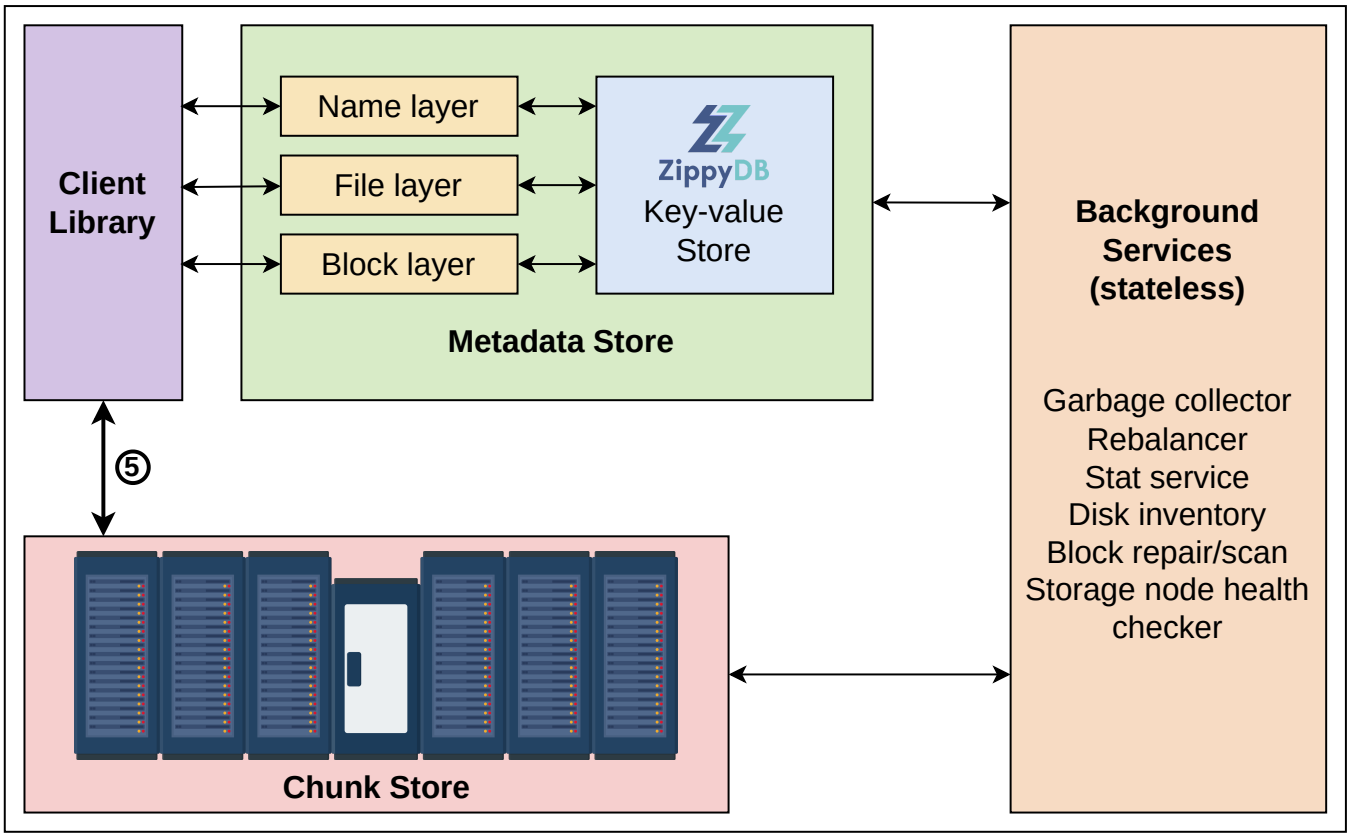
Fetching the block_id from the File layer using file_id



Fetching the disk_id from the Block layer using the block_id



Fetching the chunk_info from the Block layer using the disk_id



Performing the read or write operation

Caching sealed objects

Due to the limited throughput of the metadata shard, we have to minimize the load of reading requests on the meta-data shard. For that purpose, we make some data immutable by sealing them. We can seal directories, files, and blocks. Sealing directories doesn't lock their subdirectories but restrains adding objects to them. Now the client can cache sealed objects metadata for a longer time, reducing the load on the metadata shard.

Sealing is performed when new data as metadata is created on the Metadata Store under a directory. New files and folders cannot be created within the cluster until the Metadata Store doesn't unseal these objects. In such cases, the data can be sent to another cluster.

Consistent metadata operations

The system relies on atomic read-modify-write but guarantees consistent read-after-write for data and metadata operations in the same directory involving a single object. The directory from one shard moving to another directory on another shard is a two-phase process, and these phases are as follows:

1. A link from the new parent directory will be generated.
2. A link to the previous directory will be removed.

To detect the pending moves, the moved directory keeps the back pointer of its parent directory, ensuring that only one move operation can be performed on a directory at a time. In case of a file moving in the cross directory, we copy the files on the destination directory and delete them from the source directory.

> **Note:** Cross-shard transactions are usually not supported by key-value stores, which limit specific metadata operations in the filesystem. More

> sophisticated constructs (such as two-phase locking) might be required to achieve such functionality, which will come at the cost of some added latency.

Chunk Store

The chunks are saved in the Chunk Store. They increase at a rate proportional to the increase in the number of storage nodes. This linear relation between chunks and storage nodes makes the Chunk Store flat and enables it to store exabytes of data. The abstractions like files or blocks are managed by the Client Library and Metadata Store without the interference of the Chunk Store on the storage cluster, and it increases the performance for the diversity of tenants.

The cluster's storage nodes are used to store files in individual chunks and use core IO APIs for basic IO operations such as `get`, `put`, `append`, and `delete` on chunks. In addition, it also has the API for scanning and listing chunks. Local resource sharing is done relatively by the storage nodes among system tenants. Reed-Solomon ($RS(n, k)$; where k are the data units and n are the coded data units after applying the encoding) encoding or replications are done on the blocks for durability.

Background stateless services

There was a need for such services which maintain the consistency between different layers of metadata, manage to rebalance data across different storage nodes, repair lost data for durability, and handle the rack drains, and store system usage statistics. These services work on a single shard at a time.

Some of the services are as follows:

- **Garbage collection:** This service interacts with the metadata store and uses lazy object deletion to clean up metadata to keep the metadata to data mapping consistent.
- **Rebalancer:** This service interacts with both of the stores, Metadata and Chunk, and handles replica chunks' movement (relocation and deletion). This

movement is caused in case of hardware failures, increased capacity of the Chunk Store, and to manage rack drains.