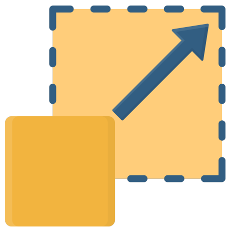
Evaluation of Tectonic

Understand how our design decision fulfills the requirements of Tectonic.

We'll cover the following

Let’s examine how well the Tectonic system meets its non-functional requirements, as discussed in the first lesson.

Scalability



We provide storage and IOPS scalability. Each storage device (and storage node that will house tens of such devices) provides us with the maximum number of available IOPS per device (or per storage node). Due to the horizontal scalability of our system, we can either add more storage devices to a storage node or can add more storage nodes in the cluster. Doing so enables us to meet growing needs in terms of IOPS.

Before Tectonic, large-scale storage systems were unable to achieve scalability up to exabytes within a cluster. The systems before Tectonic were able to store multiple petabytes of data within a cluster which was enough in the early stages, but with the passage of time, that storage was not sufficient to meet the new era's requirement.

Performance isolation

We use many measures to ensure that each tenant (and their applications) get their requested resources, but they don't impact others. Each tenant's storage and IOPS needs profile is maintained using quotas initially. Tenants can go beyond their quota if free IOPS is available. Request throttling and fair queuing keep each tenant to its allocated resources.

Because of the exclusive storage allocation, tenants are physically isolated in terms of storage. We prioritize storage traffic using TrafficGroups and TrafficClasses. Doing so helps us provide low latency service to the applications that need them.

Availability

Primarily, our system consists of metadata and data components. Metadata is managed by a separate, highly-available service ZippyDB. We have ensured availability in the following threecomponents:

1. In Metadata Store, we used snapshot reads so that data is available for readers if a write operation is in progress to increase the availability of the data. In addition, we used file-listing in the data warehouse and used hash-partitioning to distribute the queries workload on different layers to avoid hotspots in the Metadata Store.
2. In the Chunk Store, the data is on many storage nodes, where data is either encoded with error-correcting codes or data is fully replicated. Each of these schemes helps to recover lost or corrupt data.
3. On the cluster level, we can share the spare ephemeral resources (IOPS) with other tenants using different TrafficGroups and TrafficClasses. Because failures are

common in a large system, our system can throttle or gracefully degrade the service, if necessary, based on TrafficCasses. By doing so, we can manage the availability of each cluster.

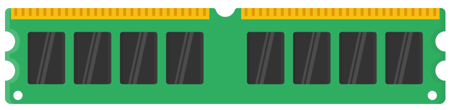
Durability

Data durability is critical for a file system. Once the data is accepted by our system, it should persist permanently (until a user explicitly deletes it). We have ensured durability in the following three components:

1. In background services, our system uses techniques such as repairing lost or damaged data that increase durability.
2. In the Chunk Store, we provided per-block durability by applying replication or RS-encoding on blocks.
3. In the Metadata Store, the data is synchronously replicated between storage nodes, which all happens within a shard. The write operations are also logged once the operation is performed on all the storage nodes. Upon completion, the writer gets the confirmation of the write operation, providing data durability.

Data integrity

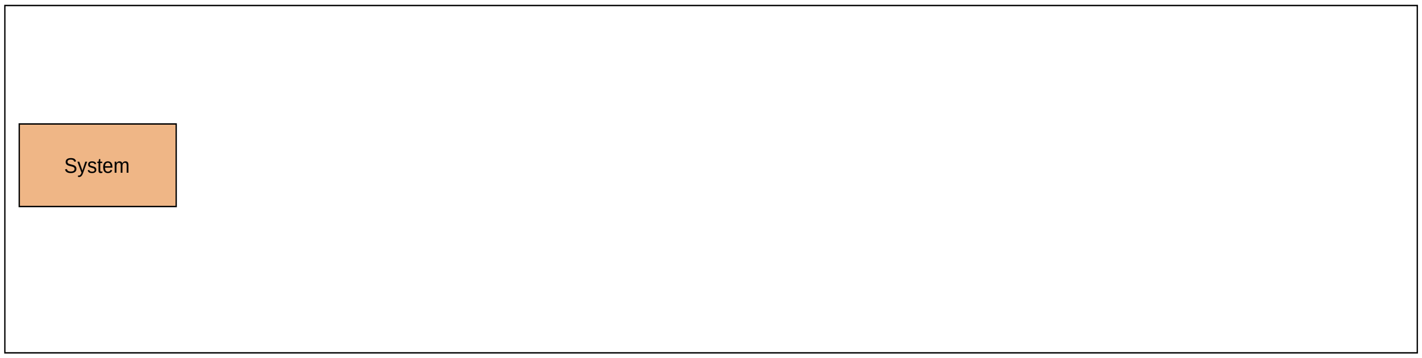
Keeping the data integrity intact has been very crucial in large-scale systems. Since these systems face lots of data corruption, users want to ensure that it is not corrupted data they are accessing. In-memory data corruption is very common in such large-scale systems. To avoid this, we have used checksums to ensure data integrity.



Performing operations (read and write) on a large amount of data on thousands of machines, the corruption of data is probable for in-memory and on disk data. This issue can be solved by using checksums after transformation (within the process) as well as between the storage node (process boundaries) and the Client Library, which is passed by the client with the data when writing to the Chunk Store. By ensuring the transformation integrity, we have avoided the corruption generated by the reconstructed chunks due to the failure of the storage node.

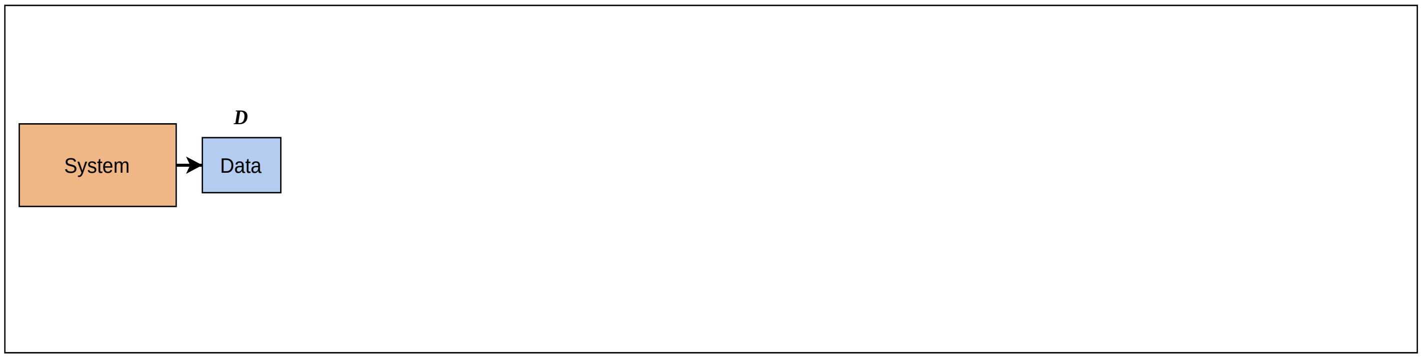
The system ensures the integrity of the data by using checksums. For the verification of data in the memory, we use the following steps:

1. We take the data (D) and generate a checksum (CS_D).
2. We'll perform an in-memory transformation (T) such that $D' = T(D)$.
3. We'll convert transformed data (D') back to original data (D) by using the inverse function (I) of transformation (T) to verify the data such that $D = I(D')$.
4. We'll compare the checksum of inverted transformed data ($CS_{I(D')}$) with the checksum of original data (CS_D). If both are equal, data is not corrupted. Otherwise, data is corrupted.



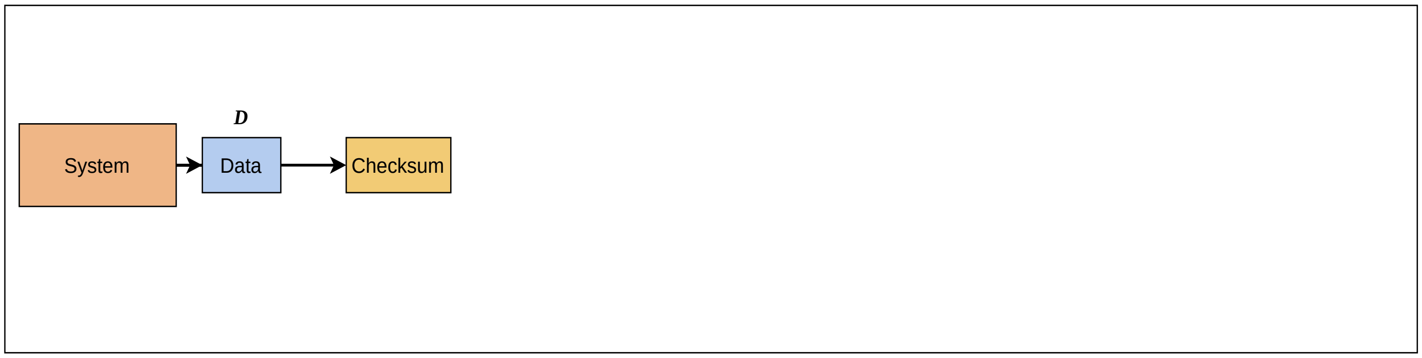
The initial state of the system

1 of 11



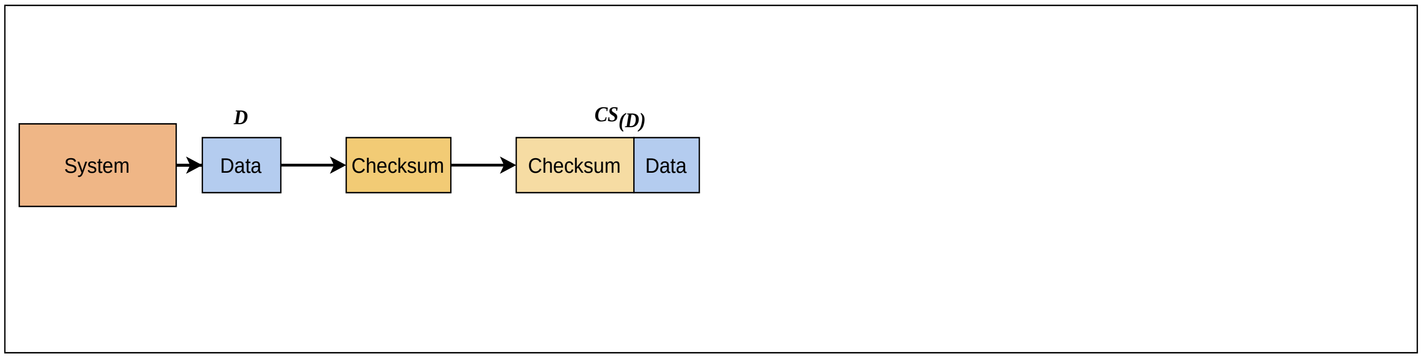
Fetching the data

2 of 11



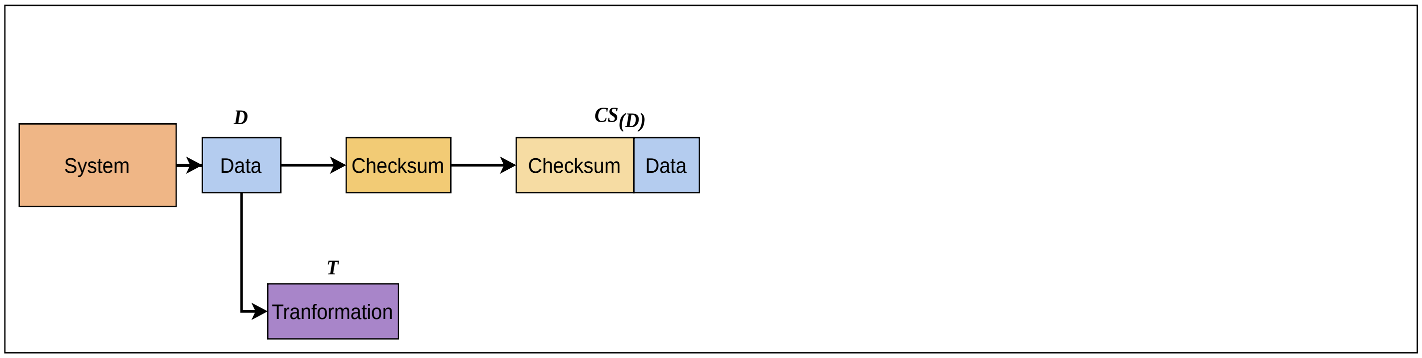
Applying the checksum on the data

3 of 11



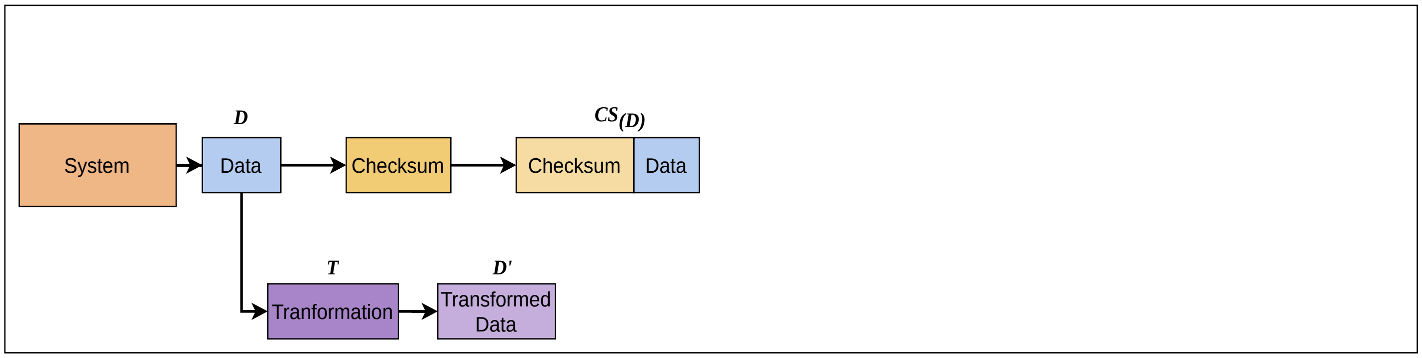
Retrieving data with checksum

4 of 11



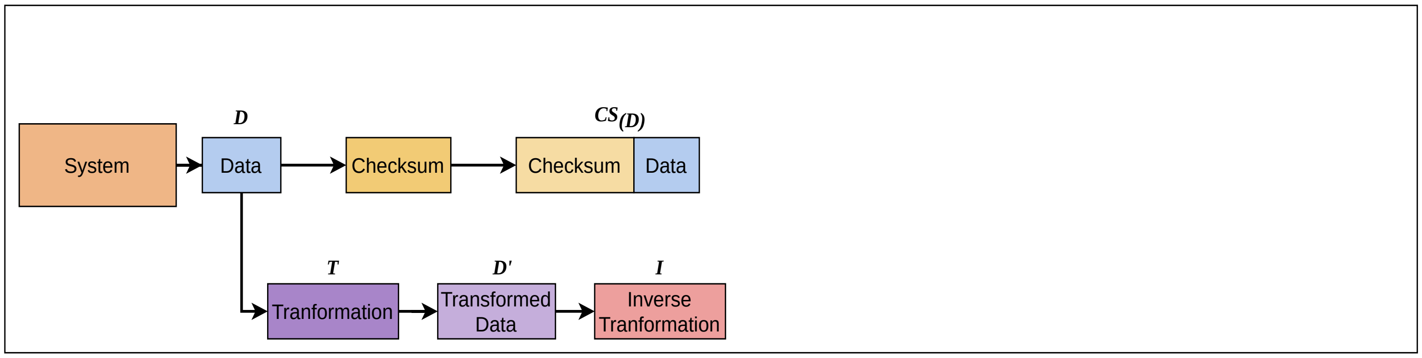
Applying transformation on the data

5 of 11



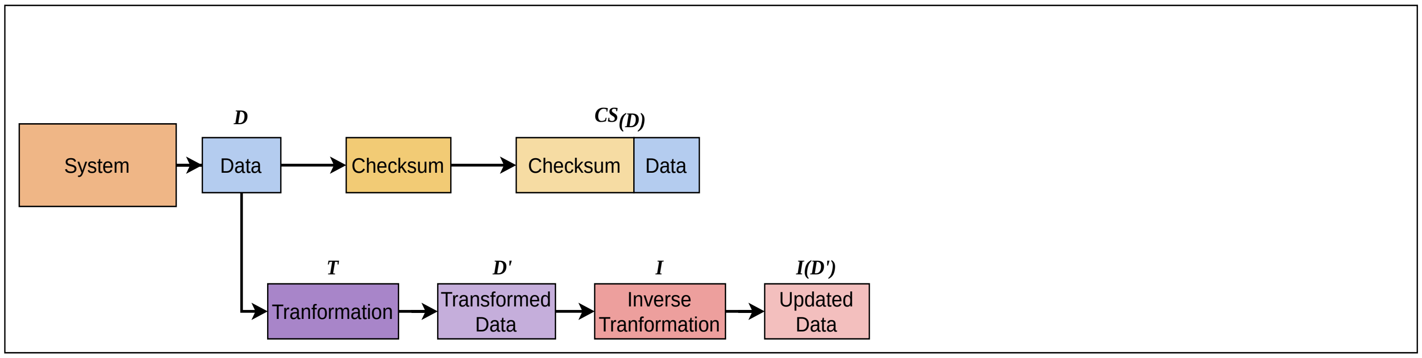
Retrieving transformed data

6 of 11



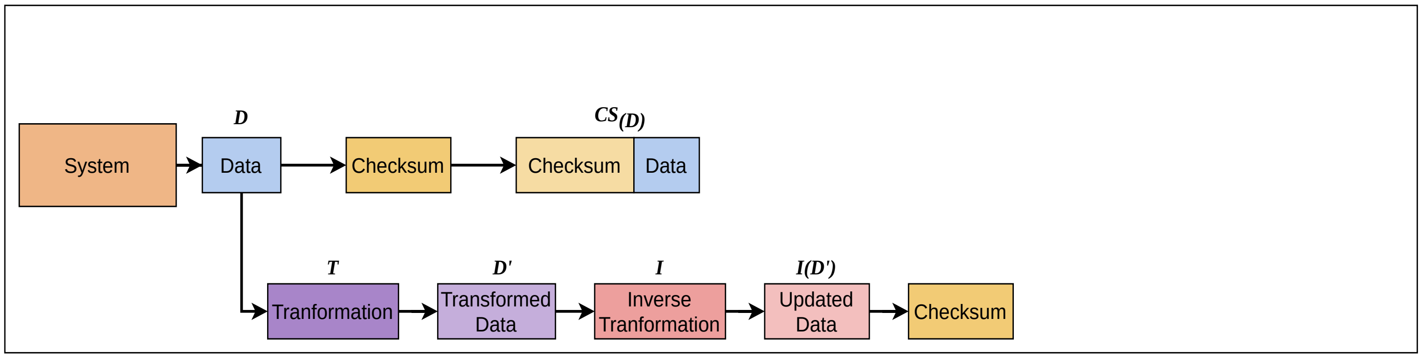
Applying the inverse function to the transformed data

7 of 11



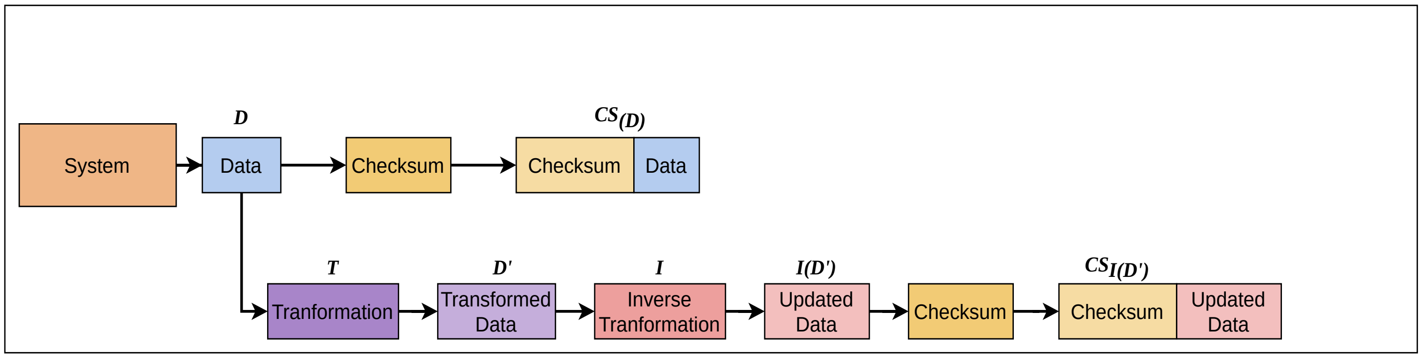
Retrieved updated data after inverse transformation

8 of 11



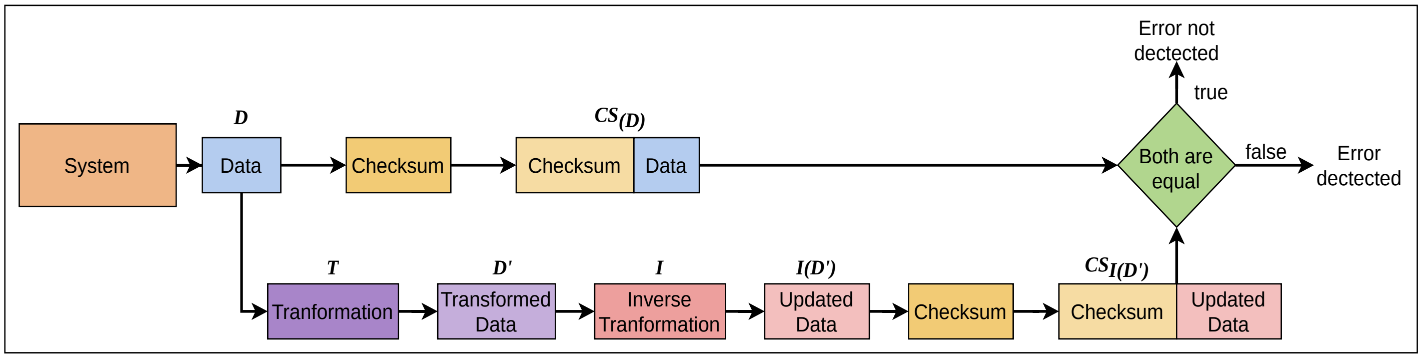
Applying checksum on the updated data

9 of 11



The updated data with the check has been retrieved

10 of 11



Checking whether the checksum of original data and updated data is same or not

11 of 11

The checksum (to compute RS-encoded data) is computationally expensive. Though this tradeoff to save storage space is viable for our system. Additionally, special-purpose processors (ASICs) can be employed to speed up RS encoding and decoding.

Conclusion

The Tectonic file system disaggregates metadata and data components of the system and delegates management of the metadata to a building block—ZippyDB. Additionally, user-level file naming, system-level file objects, and associated blocks and chunks are managed by different sub-systems so that each sub-system can scale independently. We can provide exabytes-level storage where disk space and IOPS are efficiently and fairly shared among applications of many tenants. Applications can choose from the Reed-Solomon encoding or full replication to meet their storage use versus latency tradeoffs.

System design wisdom in Tectonic

- Tectonic is another example of carefully demarcating different system parts, where some parts can be delegated to a different service (we can think of them as building blocks). We used a well-managed, highly available key-value store