Consistency:

We use 3 replicas to ensure consistency with 2 simultaneous failures. For consistency level, we choose ALL for write and ONE for read to ensure quorum is achieved. We are doing the same amount of work compared to using a fixed quorum for both read and writes, and there are some performance trade offs when there are more writes than reads. However, this also makes our consistency analysis easier:

For (i), the query node will block until read and write consistency levels are reached, and we have merged which sort of functions as read repair so even if some writes failed to go through. Coupled with re-replication during failures, replicas will eventually all contain the most up to date file, guaranteeing eventual consistency.

In our specific case, since we are writing to ALL, all files will contain the most up to date version no matter what. So this requirement becomes trivial. Although we implement consistency levels in a way that's changeable so if we do decide to switch to a different consistency level, the eventual consistency is still guaranteed.

For (ii), we have a sequence number local to each node, which increments with every write, and can be used to determine ordering of writes initiated by a fixed node. Also since we are writing to ALL, all files should contain the correct sequence of appends, in the order that they were sent.

For (iii), again, since we are writing to ALL, we will get the most up to date version by reading at any node, which should contain all appends made by the query node. We implement read consistency level anyways so any other quorum policy should still work. We also read from LRU write through cache (so appends invalidate the cache) and memory (local file table) whenever possible to improve efficiency.

Merge:

For merge, each file chunk contains metadata about the node that appends it and a sequence number local to that node that increments every time with a write. This ensures that we have unique identifiers for each chunk that also maintains the order that they were append/create for a node.

Algorithm:

Query node is the coordinator, who asks all replicas to provide a list of metadata, one metadata for each chunk of the file. Coordinator then sort all metadata based on client id and sequence number (in our case timestamp is also used although it is not used to ensure correctness). The sorted list of metadata should reflect the most up to date version of the file, and we sent it back to all replicas who will compute the correct version of the file based on the new list, fetching any missing chunks from other replicas when needed. We are sending data in tcp so we shouldn't have to look at the contents inside each data chunk.

Re-replication:

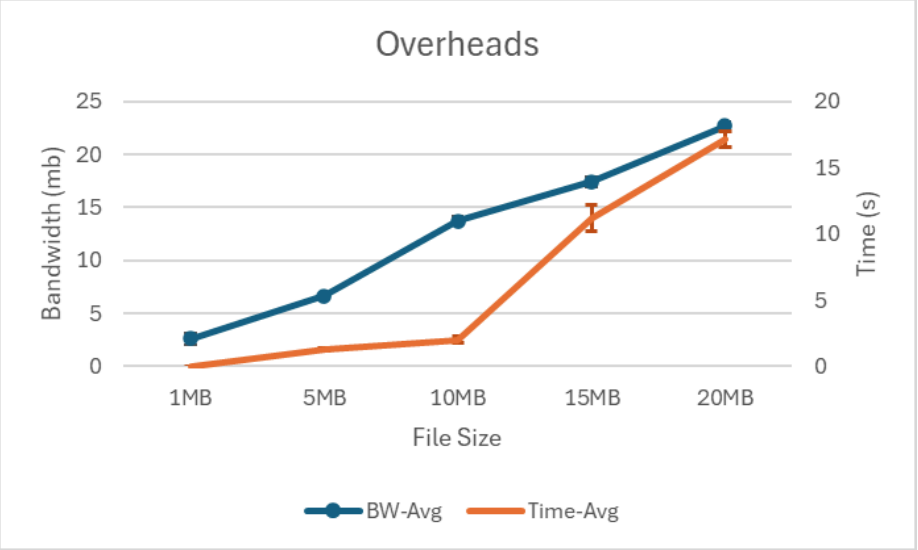
We set a callback function in the failure detector so that when a node fails, all nodes will start the replication process where they look at each file in their own table. If they are the 1st replica (determined locally through hashing), they send the file to other replicas who currently don't have the file. When a node rejoins, we add an additional logic to check if node is the new node's successor, and whether it contains any file that should be moved to the new node. We also run a cleaning function that removes excess files from any node periodically, where the decision to remove is made locally for efficiency.

Past MP Use:

In MP3, we used the membership protocol from MP2 to manage cluster membership. It is a local instance in our HDFS running on a separate port. We used MP1 to make sure that get() commands results are consistent across servers.

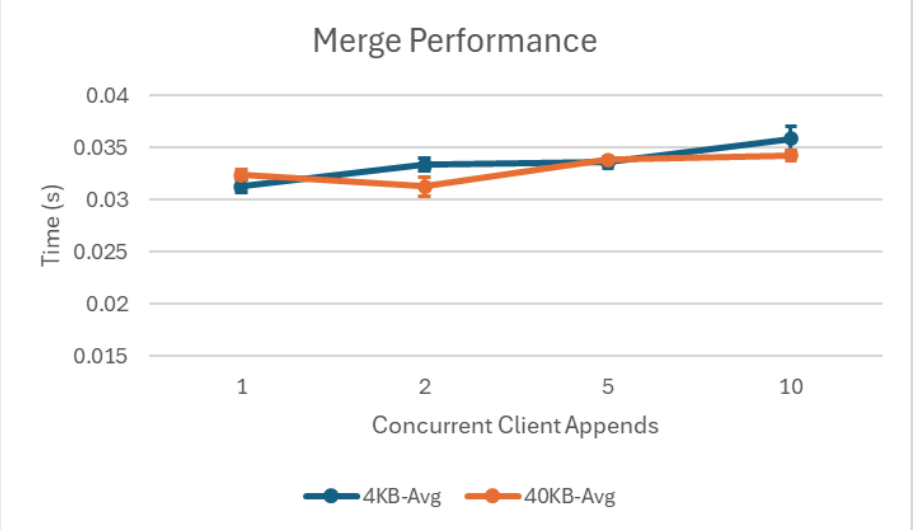
Measurements:

(i) **Overheads** Re-replication time and bandwidth upon a failure (you can measure for different filesizes ranging up to 100 MB size, 5 different sizes).



Comment: We are replicating the entire file so bandwidth using one node so bandwidth is going to reflect that. On the other hand it is unclear why time is increased drastically after 10MB, with increasing variance as well.

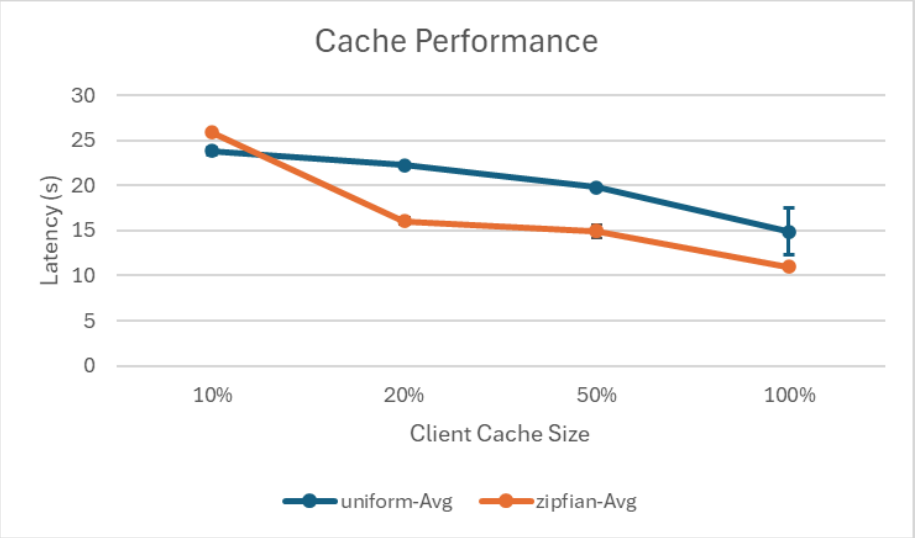
(ii) **Merge performance:**



Comment: In our implementation, we append to ALL so that all nodes will have the up to date version after append success. Therefore merge is just sending metadata to coordinator -> coordinator sort metadata and send back -> sort local file chunks using the

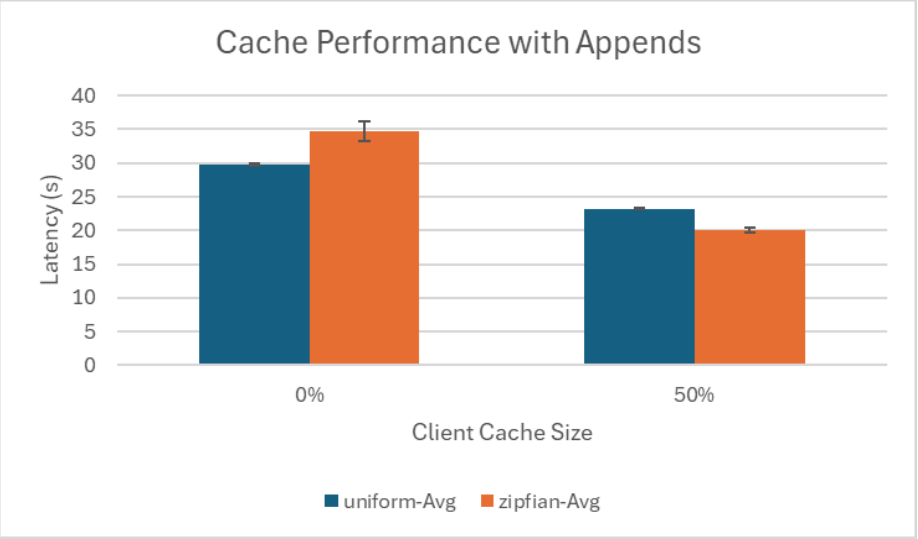
metadata. None of these steps scale with file size or number of nodes doing concurrent appends, so the performance does not change at all.

(iii) **Cache Performance**:



Comment: As the client cache size increases, the latency decreases for both uniform and zipfian access patterns, but zipfian distribution shows a much sharper latency reduction due to its access pattern resulting in more cache hits.

(iv) **Cache Performance with Appends**: Same experiment as above but with the workload consisting of 10% appends and 90% gets.



Comment: With client caching, both uniform and zipfian access patterns have reduced latency. For zipfian, where certain files are accessed more frequently, a 50% cache size reduces latency because the cache can keep the most frequently accessed files, which causes more cache hits and improved performance. Without cache zipfian does not have an advantage over uniform reads.