Distributed & Replicated p2p Data Storage Engine with Tune-able Consistency

Saqib Ali Khan
Technical University of Munich
Munich, Germany
saqib.khan@tum.de

Syed Saad bin Shameem
Technical University of Munich
Munich, Germany
saad.shameem@tum.de

Mahdi Mohebali
Polytechnic University of Madrid
Madrid, Spain
m.mohebali@upm.es

ABSTRACT

In the emerging world of data, millions of devices interact with each other in one way or the other in order to exchange information, which requires building scale-able systems that are also fault-tolerant and reliable. In this paper, we attempt to present a coherent system that aims to fulfill the aforementioned guarantees and provide a distributed storage service that can perform the CRUD (create, read, update and delete) operations efficiently at scale. Moreover, our system aims to create an elastic scale-able storage service, extended by means of replication. Furthermore, we extend the support for mission critical systems by providing a tune-able consistency model at the cost of extra query processing time. Most importantly, our proposed system is peer-to-peer and thus does not have a single point failure. In order to evaluate our system, we provide a client system for interacting with the server.

ACM Reference Format:
Saqib Ali Khan, Syed Saad bin Shameem, and Mahdi Mohebali. 2022. Distributed & Replicated p2p Data Storage Engine with Tune-able Consistency. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the modern world, the scale of data has increased tremendously due to which more and more systems are built to cope up with the needs of today's data intensive applications. Moreover, the data landscape has evolved and traditional relational databases are no longer the answer because they are quite restrictive when scaling out. In order to efficiently process data at scale, researchers developed document-based (NoSQL) storage engines that are more efficient when it comes to scaling but have limited support for some queries e.g., relational joins. Although some real-world NoSQL storage engines such as MongoDB [2] have extended their support for join-like queries, it came at the cost of latency. However, NoSQL databases are distinguished based on their availability, reliability, scale-factor, latency and throughput, and designing a system that incorporates all the aforementioned properties and provides the relational query support is not a trivial task. For this reason, all current systems specialize in a subset of properties and the choice of the system highly depends on the application and business domain.

In this paper, we propose a storage engine that can be scaled efficiently and offers high reliability and availability. To start out, we create simple key-value servers that are able to perform CRUD operations, and on top of that are able to serve as replicas of other nodes in the system. We use clustered B+ trees to store data on the disk which is highly beneficial during repartitioning and re-replication, as discussed in the following sections of the paper. Next, we enable distributed data storage using an External Configuration Service (ECS) that manages the metadata information using consistent hashing by forming a CHORD-like circular ring. Furthermore, we implement two consistency models: i) Eventual consistency that balances the replicas after some timing threshold is reached; and 2) Stronger consistency model that uses FIFO consistency (Refer to Section 3.5) to reach consensus between replicas. However, using the stronger consistency causes considerable spike in latency at the benefit of increasing the overall reliability of the system. Finally, we remove the bottleneck of ECS being a single point of failure by implementing a leader election protocol which essentially re-initializes the ECS in a purely decentralized fashion.

We summarize our contributions and guarantees in the following points:

(1) **Scale-ability:** Our system is highly distributed and replicated.
(2) **Fault-Tolerance:** We offer fault-tolerance using a peer-to-peer leader election algorithm that automatically detects faults and reach consensus on the re-initialization.
(3) **Reliability:** We offer stronger and weaker consistency models that trade offs latency but can be tuned according to the type of application.
(4) **Durability:** We use Write-Ahead-Logging (WAL) that persists the request to disk before it executes it locally and on the replicas. WAL uses efficient memory-mapped IO which does not cause significant drop in performance.

2 SYSTEM ARCHITECTURE

Our proposed system architecture consists of a number of multiple key-value storage servers spread around in a ring topology using consistent hashing (See Figure 1). Each position in the ring corresponds to a particular value from the range of a hash function, with values wrapping around at the end of the range. Each server is responsible for maintaining all the data that lies in it's hash-range in a clockwise direction which is assigned by the ECS. Moreover, the KV servers are also responsible for maintaining the data that lies in the range of it's previous two predecessors for the purpose of replication. Every server persists data to the disk with the fraction

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

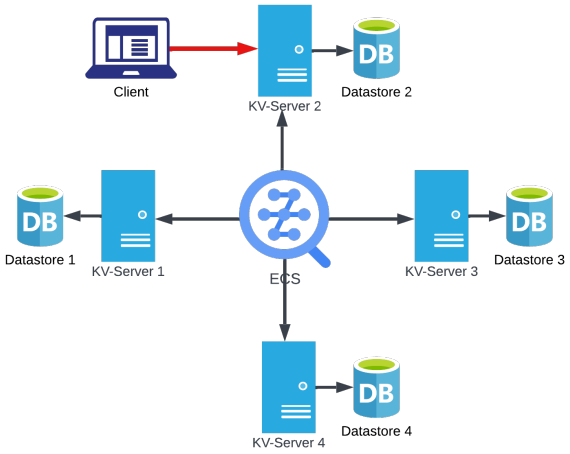

Figure 1: Architecture diagram depicting various components of the system and how they are glued together.

of data also stored in the main memory cache for immediate access. Each KV server contains information about its predecessor and successor nodes which helps in the replication and distribution of data across the cluster. We also offer a standalone client which communicates with the KV server for CRUD operations. Finally, the ECS is at the heart of the cluster which manages all the configuration services like metadata information, churn of nodes, repartitioning and replication and runs in a standalone mode. The complete architecture containing the different components is demonstrated in Figure 1.

2.1 Client

The client serves as the main interface for interaction with the server. It supports the following operations:

- **get**: get key value pair from the store.
- **put**: insert key value pair in the store.
- **delete**: delete key value pair from the store.
- **connect**: initiate connection to a specific server.
- **disconnect**: disconnect a connection.
- **keyrange**: retrieve the key ranges and addresses of all servers in the system.
- **help**: displays a list of supported commands and their usage.
- **quit**: shuts down the client.

The put and get commands also check for whether the server returns *server-not-responsible* message, in which case they initiate a *keyrange* command to receive the updated metadata of all servers and their ranges, and then reissue the original command to the responsible server.

2.2 External Configuration Service (ECS)

The ECS is a standalone service that monitors all key value servers. It is also responsible for adding and removing nodes from the cluster, which also involves repartitioning the data and it's associated metadata among the servers after a node joins or leaves the ring. Moreover, it is also responsible for initiating replication when three nodes join the cluster. It is worth mentioning here, that the ECS itself does not participate in the process of moving the data around from one server to the other. It only signals the participating servers to initiate repartitioning or replication and the servers themselves are responsible for sending the data to the responsible nodes. The ECS, however, does maintain the metadata of the servers and emits heartbeats at regular intervals that enables the KV servers to know the most recent state of the ring.

2.3 Key-Value Servers

The key-value servers are responsible for storing the data by persisting it to disk while also holding some data in the cache for faster access. The caching strategy for each server can be set by the system administrator. On the server, the data is stored in B+ trees on disk. The main use-case of B+ trees is in file-systems, where their high fan-out reduces the number of I/O operations required to find an element in the tree. This makes the use of this data structure for storing key-value pairs on disk very attractive in data storage engines. They also provide O(log n) complexity for insertion and retrieval making them very efficient. The key size is set to 20 bytes, although this can be changed in the configurations of the system.

For caching mechanisms, we have three options: First-In-First-Out (FIFO), Least-Frequently-Used (LFU), and Least-Recently-Used (LRU). These eviction strategies can be triggered while starting the K-V servers. Whenever a "put" command is received, the data is stored into the cache along with being stored in the underlying disk storage. Similarly, when a "get" command is received, the system first checks in the cache for the data, and only in case of a cache-miss does it search in the disk storage. If data is found in the disk storage, it is also put in the cache along with being returned to the client.

The "put" commands are only serviced while there isn't a write lock on the server, in which case the server returns the "server-write-lock" error and the client tries again after some time. After this, the server checks if the key belongs in the hash range of said server, returning "server-not-responsible" error in case it's not. These conditions are also checked when deleting a key. When the the server receives a "get" statement, the server first checks if the hash of the key is in it's range and returns the "server-not-responsible" error in case this check fails. If replication is turned on, this check also includes checking for the key's hash in the range of the servers that this node is a replica for.

Every server also has a Write-Ahead-Log (WAL) associated with it. For every "put" command, said command is also put in the WAL along with being processed on the server. This WAL is implemented using memory-mapped I/O and maintains a file on the system that holds all the commands that haven't been processed at the replicas. Once replication has been turned on, a separate thread runs every 100 ms (tune-able) that aims to synchronize the replicas. This is done by reading the WAL and running all commands in that log on the node's replicas. After running those commands, the pointer is moved forward marking those commands as already processed. This pointer also makes sure that we don't process the same command in the WAL twice.

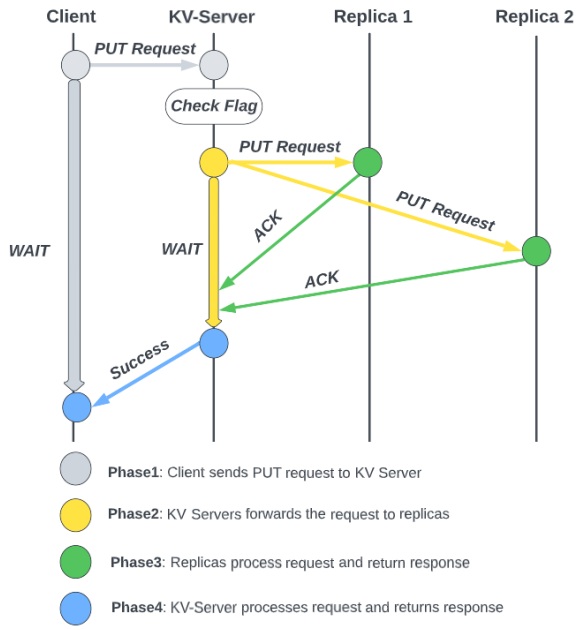

Figure 2: The sequence of operations that happens across the cluster for a PUT request under the FIFO consistency model.

2.4 (Clustered) B+ Trees as Storage Data Structure

For the underlying data storage mechanism on the disk, we chose B+ Trees. They are an extension of B Trees which allow efficient insertion, deletion and search operations. Records can only be stored on the leaf nodes while internal nodes can only store the key values. The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient. B+ Trees are used to store the large amount of data which can not be stored in the main memory. Listed below are some of the advantages of using B+ trees:

- We can access the data stored in a B+ tree sequentially as well as randomly.
- Keys can be used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.
- Because B+ trees don't have data associated with interior nodes, more keys can fit on a page of memory. Therefore, it will require fewer cache misses in order to access data that is on a leaf node.

Clustered B+ trees is term used when we want to efficiently update and retrieve ranges of data such as transferring the whole block of data from one node to the other during the churn of KV servers. In this approach, the leaves of the B+ trees contain the pointers to the actual block of data on the disk.

2.5 Stronger data Consistency

For stronger data consistency, we use the FIFO consistency model. In this model, writes by a single process are seen by all other processes in the order in which they were issued. However, writes from different processes may be seen in a different order by different processes (See Figure 2).

A stronger consistency model (FIFO data consistency) can be toggled for the servers. Under this model, when a client sends a put request to a KV-Server, it first checks for the consistency flag. If that flag is on, the server forwards the request to its replicas. This is done by writing the query in the server's Write-Ahead-Log (WAL), which works as a FIFO queue for operations. The replicas then process the request in FIFO order, and return their response to the KV-Server. Then, the responsible server processes the request and returns the response to the client. This ensures that after the client receives response from the server, all processes see writes from each other in the order in which they were issued; writes from different processes need not be seen in that order. This is a stricter consistency model than the weaker consistency where shared data can only be counted on to be consistent after synchronizing. The following is the pseudo-code running on the KV-Server for this behaviour.

```
case: put
    if (!write_lock){
        String key_hash = get_md5_hash(key);
        if (keyhash in range){
            if (this.SDConsistency){
                wal_process(command);
                sleep(300);
                cache.put(key_hash, value);
                Store.put(key_hash, value);
                return message;
            }
        }
    }
```

One potential drawback of using the FIFO consistency model is the degradation of efficiency as now the system has to wait for the data to be processed on all three nodes before it can acknowledge the process. We will look at the effects of turning on stronger data consistency later on. The status of these commands can be seen from the server logs. The algorithm for our application of stronger consistency is provided above.

2.6 Leader Election

In our proposed system architecture, ECS is a single point of failure as evident from the architecture diagram in Figure 1. If the ECS crashes the whole system will be partitioned into individual KV servers which will break the ring and thus the data is no longer consistent and available. In order to overcome this problem, we introduce a leader election protocol which allows every KV server in the system to detect the ECS failure using a fail-stop mechanism which uses a monitor to detect a possible failure using a timing threshold on metadata updates. Our proposed protocol is explained

in the following paragraph and its process diagram can be visualized in Figure 3.

Algorithm 1 Leader Election Protocol on a single Node

Require: *metadata*
  $m \leftarrow broadcast\_msg$
  if $m = ECS\_DOWN$ **then**
    $T \leftarrow check\_ECS()$ ▷ if ECS failure is also detected locally
    if $T = True$ **then**
      $x \leftarrow send\_ack(m.addr)$
      $pid \leftarrow get\_PID()$
      if $pid \geq metadata.get\_highest\_PID()$ **then**
        $start\_ECS\_locally()$
        $send\_metadata\_snapshot(metadata)$
      else if $pid < metadata.get\_highest\_PID()$ **then**
        $terminate\_leader\_election()$
        $wait\_for\_heartbeat()$
      end if
    else if $T \neq True$ **then**
      $x \leftarrow send\_nack(m.addr)$ ▷ The sender after receiving the NACK resets the ECS monitor and tries again.
    end if
  end if

Our proposed leader election protocol can be initiated by any node in the system as soon as it detects the ECS failure which is regarded as the first phase of the protocol. After detection, the responsible node sends a broadcast message to all the other online nodes in the system using its knowledge from the most recent metadata update. Upon receiving the broadcast message in the second phase, the nodes check with their local ECS monitor and acknowledge the message as either a positive or negative signal. As soon as the responsible node has received a message from every other node, the system moves towards determining the leader that will be responsible for starting the ECS. In the third phase of the system, every node locally determines its PID and compares it with PIDs of other nodes in the system, if it has the highest PID it sends a metadata snapshot to the ECS otherwise it steps out of the leader election protocol and waits for the leader to start the ECS. In the final phase of this protocol, the elected leader will start the ECS instance on its local node and will send a metadata snapshot in order to resume the ECS from its previous state. As soon as the ECS is started, it will send a metadata update (heartbeat) to all the nodes in the system which will be used as an indicator that the ECS is back online. The proposed algorithm as shown in algorithm 1 is a simpler adaptation of the original Chang & Roberts algorithm for ring-based coordinator leader election [1].

3 EXPERIMENTS AND BENCHMARKS

For benchmarking the system, we have used the Enron Email dataset [1] that contains data from about 150 users, mostly senior management of Enron, organized into folders for a total of about 0.5 Million messages. However, for consistency and ease of use on our machines, we used 1500 key-value pairs for each of our experiments.

[1]https://www.cs.cmu.edu/ enron/

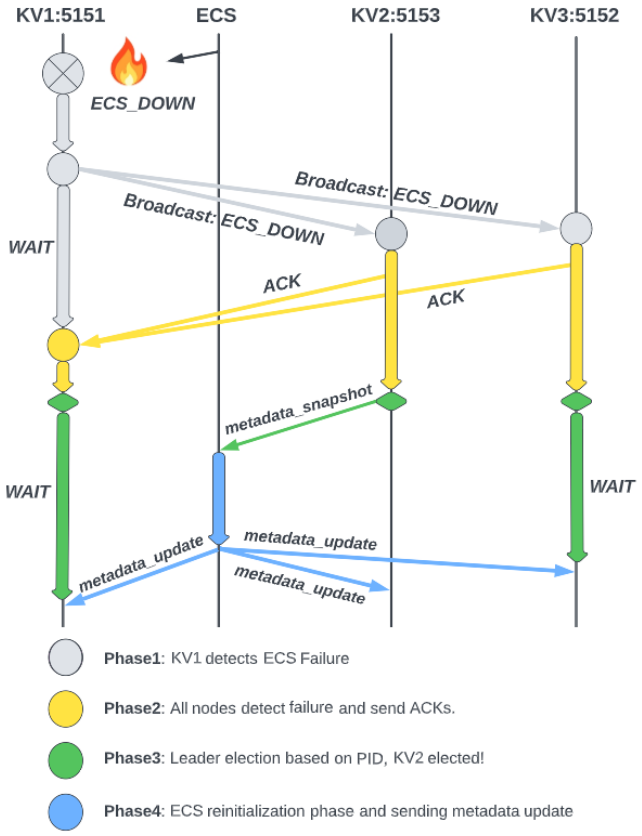

Figure 3: Leader Election Protocol: The process timeline is divided into four phases of the leader election colored identified through their assigned colors.

The client connects to the server and continuously sends put and get requests and times the response. We then average the responses of the 1500 function calls for our throughput benchmark. The time taken to call keyrange in case of a *server-not-responsible* and then connecting and running the command on the correct server is also considered as part of the timing. For benchmarking, we use the following different configurations:

- Command: Two possibilities, get or put. Sent by the client.
- Stronger Consistency: Stronger consistency on or off. Set through config arguments of the kv-server.
- Caching strategy: Strategy used for caching. Can be one of FIFO, LFU, LRU. Set through config arguments of the key-value server.
- Number of Servers: The number of servers in the ring. We run multiple servers on different ports.

Moreover, the system specifications for testing are as follows:

- Intel Core i5 7200
- 8 GB RAM
- 512 GB SSD

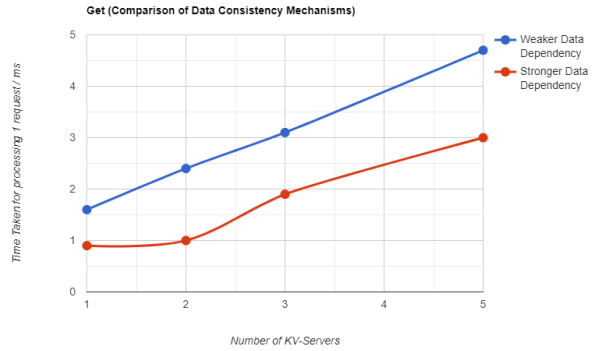

Figure 4: Time taken for get requests with and without Stronger data Consistency

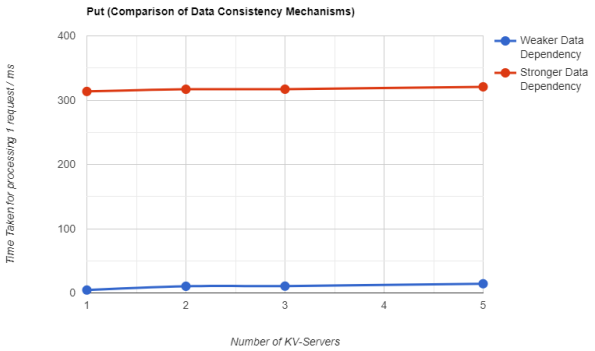

Figure 5: Time taken for put requests with and without Stronger data Consistency

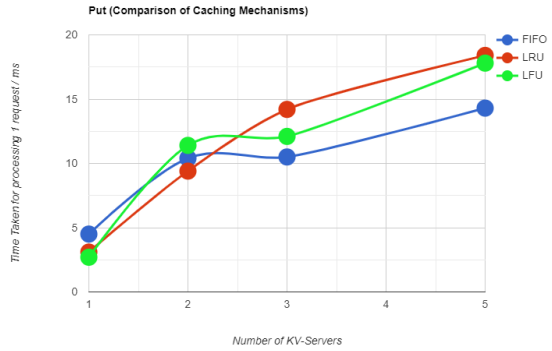

Figure 6: Time taken for put requests with different caching strategies

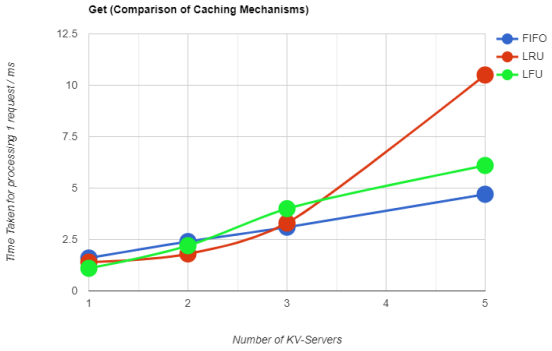

Figure 7: Time taken for get requests with different caching strategies

3.1 Effect of Consistency on PUT

For our first experiment, we send multiple put commands to the KV servers in the ring. We ran this process 4 times, each time with an increasing number of servers in the ring. The caching strategy was set to the default (FIFO), and the Consistency configuration was also set to the default value (No Strict Consistency). We then noted the time taken for these commands and averaged them to calculate the throughput. Next, we ran the same experiment except this time, we set the *Stronger Data Consistency* on, and then we measured the results. The results can be seen in Figure 5. Evidently, stronger consistency has an inverse effect on the latency with each request now taking more time to process since the replicas now have to synchronize the data before a response can be sent to the client.

3.2 Effect of Consistency on GET

For our second experiment, we send the multiple get commands to the KV servers in the ring. We also ran this process 4 times, each time with an increasing number of servers in the ring. The caching strategy was set to the default (FIFO), and the Consistency configuration was also set to the default value (No Strict Consistency). We then noted the time taken for these commands and averaged them to calculate the throughput. Next, we ran the same experiment except this time, we set the Stronger Data Consistency, and then we measured the results. The results can be seen in Figure 4.

As can be expected, stronger consistency has no effect on the performance of get requests. The small performance difference seen in the figure have more to do with fluctuations in the background processes being run on the machine. Otherwise, there is no discernible effect of consistency on the performance of get requests.

3.3 Effect of Caching strategies on PUT

For our third experiment, we run the get commands on a server in the ring. We ran this process 4 times, each time with an increasing number of servers in the ring. The consistency configuration was set to the default (Weak Consistency), and the caching strategy was also set to the default value (FIFO). We then noted the time taken for these commands and averaged them to calculate the throughput.

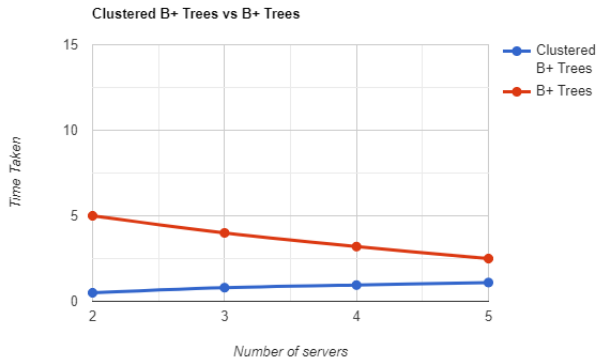

Figure 8: Comparison between B+ Trees vs Clustered B+ Trees during repartitioning of the system.

Next, we ran the same experiment except this time, we changed the caching strategy to LRU, and then we measured the results. The same was done again, this time with the LFU caching strategy. The results can be visualized in Figure 6.

We can see from the results that for lower number of nodes, LRU caching provides the best results. However, as the number of nodes increases, the performance of LRU starts to drop significantly in favour of FIFO. LFU strategy seems to be the middle of the road strategy here.

3.4 Effect of Caching strategies on GET

For our fourth experiment, we run the get commands on a server in the ring. We ran this process 4 times, each time with an increasing number of servers in the ring. The consistency configuration was set to the default (Weak Consistency), and the caching strategy was also set to the default value (FIFO). We then noted the time taken for these commands and averaged them to calculate out throughput. Next, we ran the same experiment except this time, we changed the caching strategy to LRU, and then we measured the results. The same was done again, this time with the LFU caching strategy. The results can be visualized in Figure 7.

We can see from the results that for lower number of nodes, there isn't a clear best caching strategy as they all behave very similarly. However, as the number of nodes increases, the performance of LRU drops significantly, as compared to FIFO and LFU. Out of those two, FIFO generally shows better performance compared to LFU.

3.5 Effect of Clustered B+ Trees

In the normal settings, when the data is repartitioned between the nodes, it takes on average approximately 4.6 ms per key-value pair to move the data from one node to the other. This is because of the fact that in practice random access takes more time than sequential access, which is obvious from the application of B+ trees and clustered B+ trees (which stores the data sequentially at disk level) in the following paragraph.

In order to evaluate the performance of using clustered B+ trees for sequential access, we first setup the cluster using 5 KV servers and then start removing them one after the other until there is only one server in the system. Figure 8 shows the final benchmark results based on the time it takes to transfer the block of data between the nodes. It is obvious that when there are more servers in the system, the ring would be congested and the block that is to be transferred would be small (this also depends on the hash function that is used) and hence the difference between the sequential and random access would not be significant. However, as the number of servers in the system decrease the KV servers becomes responsible for more and more key-value pairs, and hence during churn, there will be more data transfer between the servers which will induce a significant drop in performance.

4 CONCLUSION

In this paper, we have described our system architecture for implementing a distributed & replicated data storage engine with fault-tolerance and reliability guarantees. We have demonstrated different components that offer these guarantees and benchmarked our system using different configurations such as the number of servers, caching strategies and consistency models to demonstrate the impact and trade-offs that occur by using these components. Our proposed data storage engine is peer-to-peer. in the sense that in cases of ECS going down, our system is able to detect and re-initialize the ECS using message passing between the online nodes and thus it is not a single point of failure. Furthermore, we offer stronger consistency model (FIFO) for more mission-critical systems that prefer reliability over latency. In summary, our system can be used in a wide variety of applications with sufficient guarantees.

REFERENCES

[1] Ernest Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (may 1979), 281–283. https://doi.org/10.1145/359104.359108

[2] Hema Krishnan, M.Sudheep Elayidom, and T. Santhanakrishnan. 2016. MongoDB – a comparison with NoSQL databases. *International Journal of Scientific and Engineering Research* 7 (05 2016), 1035–1037.