Database Caching and Sharding Strategies in Chained MR Jobs

Andrew Wong awong52@jhu.edu
Ryan Demo rdemo@jhu.edu
Ben Hoertnagl-Pereira bhoertn1@jhu.edu
Daniel Sohn dsohn3@jhu.edu

Abstract

Caching and replication improve data availability in distributed systems via pull and push data models, respectively. In practice, caching has become the standard approach, especially for static hosting in web applications [3]. In this paper, we investigate the marginal benefits of both working in conjunction for optimal performance, specifically for workloads that compute on large amounts of uniformly sampled data queried from a database.

1 Introduction

Cloud providers that offer multi-tenant services must handle compute hotspots reliably in order to optimize resource usage. Usually the main issue involves servers encountering hardware bottlenecks to execute all the query requests, or lacking sufficient network bandwidth to respond with the results to the requester. Even if these are hardware limitations and not necessarily database dependent, response latency can be mitigated by a combination of sharding, replication and caching strategies. By routing the queries to different shards or load balancing over multiple copies of the database, parallelizing job tasks has the potential to reduce overall job time. One approach to handle dynamic compute-heavy localities relying on the same data is via distributed database sharding. For example, compute-heavy jobs requiring access to the same data shards may be bottlenecked by database reads, where network congestion (bandwidth) or CPU can be limiting factors of how fast requests are processed [4].

In context of the CAP theorem, there may be multiple optimal replication strategies based on the priorities of the distributed database. For example, given different loads such as read-heavy, mixed read/write, write-heavy operations, caching might be better in read-heavy situations where there are many cache hits, while sharding might be better when the data is updated frequently.

2 Related Work

The trade-offs between caching and replication have been studied in a wide array of contexts including distributed cloud storage systems, web hosting, and search applications. Analysis shows that the selection of best mechanism is heavily dependant on the data workload and requires careful analysis of the application characteristics [3].

ShardFS is a server replicated directory lookup state, employing a novel file system specific hybrid optimistic and pessimistic concurrency control favoring single object transactions over distributed transactions. Experimentation suggests that if directory lookup state mutation is a fixed fraction of operations (strong scaling for metadata), server replication does not scale as well as client caching, but if directory lookup state mutation is proportional to the number of jobs, not the number of processes per job, (weak scaling for metadata), then server replication can scale more linearly than client caching and provide lower 70 percentile response times as well [7].

In addition to fixed replica strategies that shard keys upfront, dynamic resharding could provide improved data availability. For example, by taking advantage of the spatiotemporal locality and correlation of user access, replication strategy for spatiotemporal data (RSSD) mines high popularity and associated files from historical user access information, and

then generates replicas and selects appropriate cache node for placement. Experimental results show that the RSSD algorithm is simple and efficient, and succeeds in significantly reducing user access delay [6].

Atlas is a cloud-based MongoDB service that provides automatic horizontal sharding on clusters MongoDB currently does not expose an open-source implementation of their heuristics for dynamic resharding [2].

3 Systems

Our main goal is to have a controlled framework for comparing relative metrics between various caching/sharding configurations, with less focus on optimizing the per-packet routing latency of data between nodes. We use a Virtual Private Cloud (VPC) on the Google Cloud Platform (GCP) in order to manage our compute and database cluster.

3.1 Network Architecture

We split functionality across different nodes: MapReduce, mongos (MongoDB Shard) app router/config server, and 4 shards. The mongos app router is an interface to two eponymous databases: single and cloud, where single holds one collection of data on the app router node, and cloud holds the sharded collection of data on the shard nodes. The config server exists to hold metadata and shard lookup information for the app router during queries. All nodes have aliased access to one another.

3.2 Database

We decided to use MongoDB as our database since it allows for finer grained control over sharding but also has built-in libraries and default configurations to allow easy adding and removing of shard servers.

3.2.1 Data Ingress

We decided to structure our data in two main ways: one single node that contained an entire replica set, and a set of sharded nodes that evenly distributed the data over the sharded nodes. Each data entry

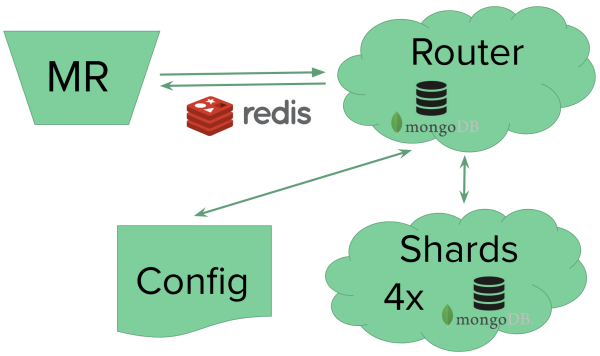

Figure 1: Architecture

was created by randomly generating an 100 character message using a Python script, exporting batches of messages to JSON. To ingress the data, we used the mongoimport command line utility to insert the JSON data into MongoDB. Full data ingress of our 32 GB database proved to be a time bottleneck (8 hours on average to ingress all of the data) even with multithreading over 8 vCPUs and incorporating best practices. Using 8 vCPUs, data ingress took 40 seconds on average per 500,000 documents, increasing linearly in time as shard size increased. Additional challenges on data ingress included data corruption and shard failures, which significantly blocked our ability to run benchmark tests. As a result, we decided to make the design decision to use a smaller database size of 8 GB in order to be able to reconfigure the cluster more often and run more test variations.

3.2.2 Sharding

Each shard was initialized as a replica set with two replicas per shard. To maintain the shards we also needed to initialize a router node which acted as an interface for the shards. It handled the data distribution (load balancing using the MongoDB balancer running on the primary config server) and all queries were sent through this node. Furthermore, the router instance was built on top of a config instance which handled all the metadata of the shards and shard lookup.

It is important to note that the balancer used range sharding to allocate documents to nodes sequentially by document id. In the experiment results, the 2-node sharded cluster containing shard 5 and shard 6 contained 44% and 56% of the data respectively. The 4-node sharded cluster containing shard 3, 4, 5, and 6 contained 14%, 14%, 39%, and 33% of the data respectively. The slight imbalance in node load distribution might have been due to convergence rates of the balancer, which moves chunks from larger shards to smaller shards.

A limitation on the number of shards was the amount of virtual machines we could instantiate in a single zone on Google Cloud Platform. Having some shards in different zones would affect the data transfer speeds and our experimental data. Because we could only have eight virtual machines, and two were used for the MapReduce interface and MongoDB interface, we were limited to a maximum of six shards. We eventually chose to run experiments on $1, 2$, and 4 shards.

3.2.3 Caching

We decided to use Redis for caching, as it is the gold standarad in practice. Redis is an in-memory data structure project implementing a distributed, in-memory key-value database with optional durability. The redis server runs on our mapreduce node as a layber between the mongos app router.

4 Workload

Given the extensive related work in caching vs replication for web applications, we decide to focus on a more compute heavy workloads. We consider chained MapReduce, in which intermediate results from a previous reduce phase are used as the input to the next map phase. Each intermediary phase will require fetching new documents from the database.

4.1 Random, Uniform Samples

Our experimental workload approximates mapping a randomly, uniformly sampled set of database entries into another set of randomly, uniformly sampled set of database entries that become input to the next MR phase in the chain. To implement this approximation, one iteration of the workload starts with a randomly generated set of keys and fetching the associated documents, mapping the entire document to its SHA-256 hash modulo the database entries, and then reducing to the unique keys. Given the pre-image and collision resistance properties of cryptographic hash functions that output pseudorandom values, this workload essentially randomly samples new keys uniformly based on the current keys' values [8].

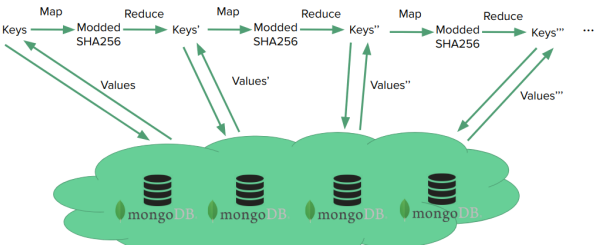

Figure 2: Chained MapReduce with Inter-MR Phase Database Reads

4.2 Interarrival Times

Since database read queries all happen at the end of a reduce phase, the traffic going through Redis to the database cluster is inherently bursty. Requests occur in bursts before every map phase and do not occur while a map or reduce phase is computing on a worker node.

4.3 Applications

One real world task that this workload approximates is a traversing a hash-pointer derived graph, nested hash table or other hash-pointer based data structure. When ids are not indexed in such data structures, traversal is dependent on hashing some read values to generate the pointer to the next nodes. Further, if metadata for a node is referenced by its hash, it requires another read and an MR phase to reduce to the desired data.

Computing or verifying a Merkle tree, a hash-based binary tree, from a set of leaf nodes takes $2^{(\log n)+1}$ hashes. The data of nodes at each depth is dependent on the subtree beneath it such that the two immediate children n_1, n_2 of a parent node n_p determines the value of n_p: $n_p = H(n_1||n_2)$. With a map phase of hashing a node and a reduce phase of hashing the concatenated hashes, this chained job eventually reduces to the Merkle root with $\log n$ jobs.

Chained MapReduce jobs are useful in other graph processing tasks, like calculating the lowest or highest degree vertex in a graph: a two-phase MR job. Since the entire graph must be processed, there is a uniform amount of read from the graph structure. Job 1 calculates the degree for every vertex. Job 2 reduces the vertex, degree pairs to the result of the max/min/other operation.

5 Experiments

5.1 Design

We ran experiments on a range of parameters to observe trends related to caching, replication and sharding.

For all runs, we experimented on an 8GB db instance, with the following permutations:

- Replicas per Shard: 2
- Shards: 1, 2, 4
- Cache Size (GB): 0, 1, 2, 8, 16
- MR Chain Length: 1-8

We measured the following dependent variables on a MapReduce worker node with 8 vCPUs and 38 GB memory hosting a Redis memcache layer:

- Job Completion Time (nanoseconds)
- Average DB CPU % Usage
- Max DB CPU % Usage
- Average DB Memory Usage (KB)
- Max DB Memory Usage (KB)

We also measured baseline performance on a 32GB instance.

In order to ensure fair resource utilization across configurations, we restart the MongoDB daemon and flush the Redis cache between every execution. MongoDB uses memory incrementally across data reads by building indices, which could provide better performance for later run configurations. Similarly, the Redis cache must be emptied so that later runs do not have a performance bias.

From all permutations, we aimed to collect results to determine best practices for caching and replication depending on mainly the chain length for a fixed workload.

5.2 Implementation

Our implementation consists of two main components - fetch and computation - both of which interface with STDIN and STDOUT. To fetch data, we use the PyMongo API to implement clients that interface with MongoDB for the single and sharded collections cases; redis is used for caching results. To compute on this data, we implement a mapper and reducer that transform one set of keys to another. We test various increasing length chains in 2^i for $i \in \{0, 1, 2, 3\}$, and monitor latency within the chained mapreduce script and MongoDB CPU/RAM utilization via bash scripts. All of our code can be found on our GitHub repository.

5.3 Results

Our baseline results on a 8GB database with 1 shard and 1 replica per shard (i.e. one central database node) showed that max % CPU usage, average memory allocated and max memory allocated for the database node were all essentially the same, subject to fluctuations expected of vCPUs and slightly different inputs. Average % CPU usage marginally went down as the cache size increased, and the length of the chain had no effect.

Job time decreased overall with the presence of a cache. With a chain length of 1, there is no clear relationship between job time and cache presence, but in chains of length 2 or higher the job time is less

than that of no cache. Interestingly, the smaller cache sizes slightly outperformed larger cache sizes (i.e. job time of 1GB cache $<$ 2GB $<$ 8GB $<$ 16GB). This is likely due to the overhead of maintaining a bigger cache where entries are more fragmented in RAM. Since our workload queried an essentially random id through the hash function, the chance of a query colliding is $1/n$ where n is the number of database entries. We see in the Redis logs that the 8GB cache had 106 hits and 159,875 misses with 31365306 output bytes; the 16GB cache had 149 hits and 159,807 misses with 19241251 output bytes. With essentially the same hit/miss ratio, the size of the Redis cache is constant even with more space but is only able to serve a small number of requests. We conclude the Redis cache is not entirely useful with this workload since the requests query $O(20\text{k})$ keys with a keyspace of $O(275\text{M})$. Actual Redis memory usage is capped at $O(100\text{MB})$. Generalizing, the cache is only useful as the chance of a query colliding $1/n \to 1$, i.e. in a database with fewer, bigger documents.

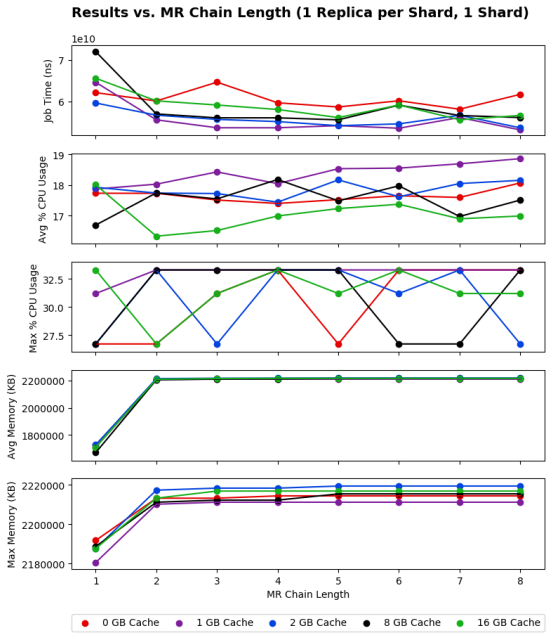

Figure 3: Baseline Results

Beyond the baseline, we looked at how number of shards and number of replicas per shard affected an MR chain of length 8 in:

- Job completion time for the entire 8-long chain
- Average average DB shard node CPU % usage
- Average max DB CPU % Usage
- Average average DB allocated memory (KB)
- Average max DB allocated memory (KB)

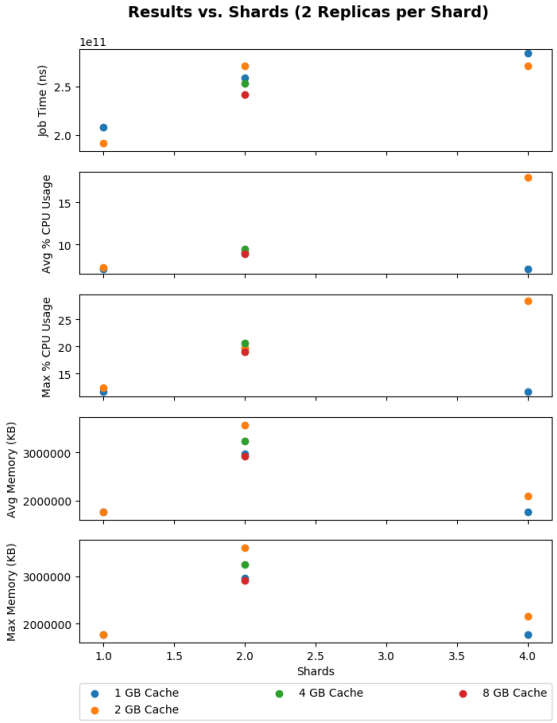

Figure 4: Sharding and Replication Results

We observed that the average and max % CPU usage went up per node as there were fewer shards. Job time fell with the number of shards, since the request was parallelized across multiple nodes. Average and max allocated memory per node increased with the number of shards, since an instance of mongod or mongos was running per node. When compared side

by side to other shard configurations, the single shard had the lowest memory footprint.

Multi-shard configurations were only marginally less latent and resource heavy with caching enabled. Within that marginality, the Redis memcache was most significant in decreasing average and max memory allocation on the database nodes combined. We observed that the largest cache (8GB) translated to the lowest amount of allocated memory in the database nodes, but there was no clear correlation behind the size of the cache as long as it could hold at least the many queried small documents.

6 Future Work

6.1 Network Improvements

One limitation of GCP is that it only allows up to eight instances per region. Given the added latency of cross-region communication (i.e. us-east to us-central), we required that all interacting machines are hosted in the same region to avoid this potential bottleneck. This was not an issue for the case of a single database; however, for the sharded case, we were limited to using at most 6 shard nodes after provisioning the router and MapReduce nodes.

We would like to better consider the placements of replicas as a way to improve fault-tolerance. While the replicas of each shard were placed within the same server to help with performance, we found that it did not help resolve some crashes. Nodes crashed more commonly than single MongoDB instances which affected all replicas of a given shard. Finding a way to spread replicas while still maintaining locality would be an improvement to the network.

6.2 Database Improvements

Our original motivation for choosing MongoDB was its "ease of use" for adding/removing shards dynamically; however, configuration sharding became the main blocking factor in preparing data for compute.

Given more financial resources to try proprietary service-based solutions, a better strategy would have been to explore MongoDB Atlas as a baseline for sharding. While Atlas does not expose shard parameter tuning that we require, it would provide some insight into performance.

Another interesting approach to consider is how using a SQL database like Postgres. While it does not support direct control over sharding natively, 3rd party services such as Citrus provide this feature [**?**].

6.3 Implementation Improvements

Currently, we use simple streaming implementation of MapReduce in Python - while this does affect relative experimental results, it is sub-optimal data transfer method. Using Hadoop's Streaming API could improve performance by parallelizing mappers and data ingress, although it would require interacting with HDFS, which does not allow clear control over shrading. Since the size of our starting key set is quite small compared to the database size, $O(10e3)$ vs $O(100e6)$, the mapping phase does not contribute nearly as much to end-to-end latency as data fetching, so using Hadoop will likely not improve the end-to-end latency.

In addition, Hadoop's Java API provides a JobControl interface that allows for more fine grained execution of map and reduce phases. This could help get more accurate data on the memory and CPU usage of each process by incurring less overhead.

7 Conclusion

For workloads that use data sampled randomly from a uniform distribution, we observed that the best strategy is using sharding with a small Redis cache on the query router. Depending on the number of documents in the database and their size, the cache will become more useful if there is a higher chance of hits. With our small document based workload, this turned out not to be the case. Replicating was marginally more useful when parallelism in CPUs was leveraged, since more servers could respond to more requests. In multi-chain MR experiments, after the first chain, cache hits were only slightly more likely, and overall did not contribute meaningfully to

latency improvement but were nonetheless quicker. As per our original hypothesis, we determined that cluster configuration should be provisioned based on expected workload.

References

[1] Ben Hundley. Optimizing Elasticsearch: How Many Shards per Index. QBox Technical Blog. 2015.

[2] Mat Keep. Scaling your MongoDB Atlas Deployment, Delivering Continuous Application Availability. MongoDB Best Practices. 2016.

[3] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of Caching and Replication Strategies for Web Applications IEEE Internet Computing, Volume 11: Issue 1, pages 60–66. 2007.

[4] Flvio R. C. Sousa, Leonardo O. Moreira, Jos S. Costa Filho, and Javam C. Machado. Predictive elastic replication for multitenant databases in the cloud. Concurrency and Computation: Practice and Experience - Volume 30, Issue 16. 2018.

[5] Uras Tos. Data replication in large-scale data management systems. Universit Paul Sabatier - Toulouse III Archives. 2017.

[6] Lian Xiong, Liu Yang, Yang Tao, Juan Xu, and Lun Zhao. Replication Strategy for Spatiotemporal Data Based on Distributed Caching System. MDPI Sensors Journal, pages 222–236. 2018.

[7] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems. SoCC '15 Proceedings of the Sixth ACM Symposium on Cloud Computing, pages 236–249. 2015.

[8] Ronald L. Rivest. Reference implementation code for pseudo-random sampler for election audits or other purposes. Rivest Sampler source code. 2011.

[9] Mouhamadou Diaw. Sharding with PostgreSQL. DBI Services. 2016.