Assignment4: Improving performance
(after presentation)

Group Member: Kejian Tong, Chuan-Hsin Chen, Chien-Chia Chiu, Chieh-Lin Lee

Tuning Kafka Producer parameters:

There are lots of configurable parameters in Kafka producer, among which ***batch.size*** and ***linger.ms*** are the most straightforward ones that would influence producer's performance. Larger *batch.size* and *linger.ms* would reduce the number of network round trips from producer to Kafka, but would also increase the processing time of each single batch, which is a tradeoff. So we tested with different values to find the best set of parameters for our application.

Producer config			**Performance on Client Side**			
batch.size (byte)	**linger.ms**	max.in.flight.requests.per.connection	**POST Throughput (req/s)**	**Mean response time (ms)**	Min response time(ms)	Max response time(ms)
16384	5	5	2581	77	31	816
163840	50		1152	171	54	2141
1638	0		2435	81	16	1560
7000	3		2750	72	28	1513
4500	1		2915	67	24	1362
3000	**1**		**3018**	**66**	21	1396

Result of sending 500K requests

According to the above results, we consider batch.size = 3000 byte (i.e. around 10 swipe messages), linger.ms = 1ms as the best parameters and would use them in all of the following experiments.

Tuning Kafka partitions:

Using more partitions in Kafka would provide a greater extent of parallel processing, both when writing messages received from the producer and when delivering messages to the consumer. However, more partitions would also require more open file handles (Rao, 2015). Therefore, we did some research and experiments trying to find the optimal number of partitions for our application.

We found this formula in this Confluent blog post (Rao, 2015) suggesting that the theoretical best number of Kafka partitions should be *max(t/p, t/c)* , where *p* and *c* are the producer and consumer throughput you can achieve on a single partition, and *t* is your target throughput of Kafka. According to the benchmark testing conducted by Confluent, (*Apache Kafka® Performance, Latency, Throughout, and Test Results*, n.d.), the upper bound of Kafka throughput is confined by the maximum disk throughput and network bandwidth of the machine that it's deployed on. So we looked into the specs of EC2 t2.micro instances. The network bandwidth of t2.micro is around 60-70 Mbit/s(7~8MB/s). As for the maximum disk throughput, we conducted a write test on the two disks and got a disk throughput around 68MB/s. Therefore, the best possible throughput(i.e. the target throughput *t* in the formula) for our Kafka cluster is confined by network bandwidth 7~8MB/s. Considering that the size of our "swipe message body" is around 320 bytes, the target throughput *t* is 23000 req/s.

We then conducted a test to measure our producer and consumer throughput on a single partition. On the producer side, we measured the wall time of processing 500K requests including request body validation, sending the message to Kafka and waiting for the acknowledgement (i.e. the offset) synchronously, which takes 3670s and yields a producer throughput *p* of 136.2 req/s. As for the consumer side, we measured the wall time of polling messages from Kafka and batch updating to MongoDB for both Matches Consumer and Stats COnsumer, which results in an average consumer throughput *c* of 98.1 req/s.

Based on the above data and formula *max(t/p, t/c)*, the theoretical best number of partitions for our case is 200 partitions. Considering the commonly adopted value for most applications (which is 12 partitions) and the overhead of having too many partitions, we conducted a few more tests with 10, 50, and 100 partitions and with a fixed number of consumer threads (10 threads). It turns out that 100 partitions results in the best practical throughput on the producer side.

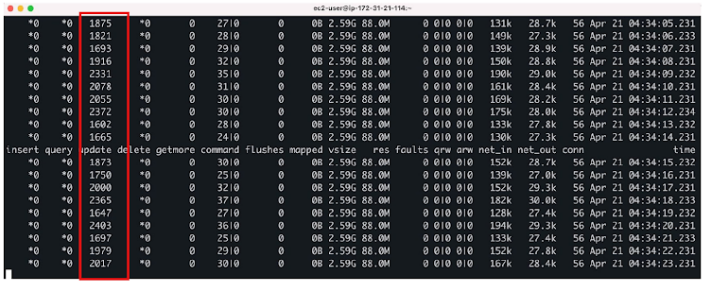
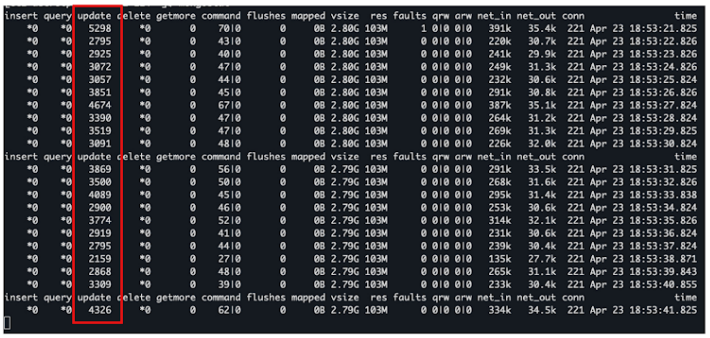
Number of Kafka Partitions	Producer Throughput(req/s)
10	1362
50	1836
100	**2423**
200	1556

Result of sending 500K requests

Tuning Kafka Consumer threads:

As is explained in our presentation, our multithreaded Kafka consumer adopted the "1 connection, multiple worker threads" model, so the number of worker threads is a main factor

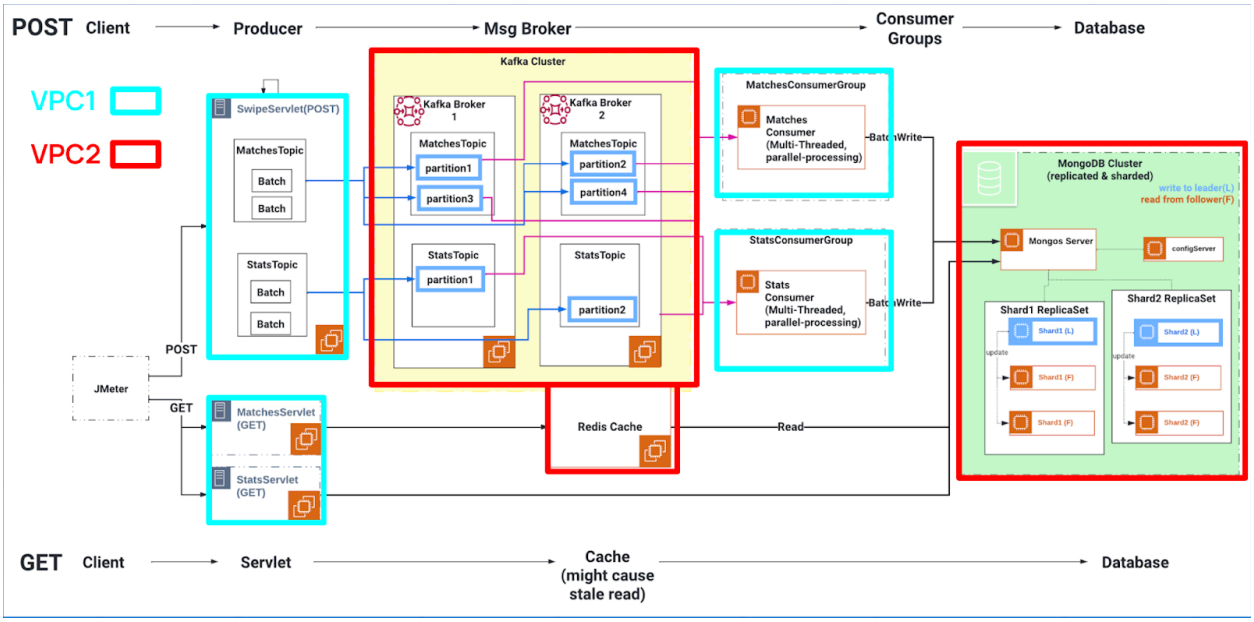
that would affect the consumer throughput . We compared the update rate of MongoDB and the consumer throughput of using 10 v.s. 100 consumer threads. The results are shown as follows:

# of consumer threads	MongoDB update rate (writes/sec)	Consumer throughput
10		1180 offset/sec
100		1303 offset/sec

With more worker threads in consumer, there is more parallel update into the MongoDB which results in a higher update rate and consumer throughput. However, the throughput on the producer side and the client side did not change much, which makes much sense according to this post (Kreps, 2014) suggesting that “Kafka always persists O(1) performance with respect to unconsumed data volume”, corresponding to the mechanism of Kafka that decoupled producer and consumer.

Test within same VPC & scaled out structure:

Reviewing our test environment for the above experiments, we found that there is much “cross-vpc” traffic involved due to our deployment model. Because of the limitation of AWS student accounts, there can be at most 9 EC2 instances running simultaneously, so we had to deploy Kafka cluster, MongoDB cluster and Redis in one account/VPC, and Producers/Consumers in another.



Cross-VPC deployment

We wanted to see if moving every component into the same VPC would result in a better overall performance, so we opened a personal AWS account which allows at most 32 EC2 instances running at the same time.

Sending 500K requests	Client Throughput(req/s)	Producer Throughput(req/s)	Consumer Throughput(req/s)
Cross VPC	2436	2423	871
Same VPC	2190	**2799**	**1210**

The effect of reducing cross-vpc traffic is quite obvious on the producer and consumer throughput. However, the client throughput did not increase as we expected. We reckoned that the client throughput might have been affected by the fluctuating internet speed of traveling from local client to AWS VPC, so we further moved our Client to a AWS EC2 within the same VPC. The result of such a “everything in the same VPC(including client)” structure took another leap on both the producer side and the client side.

Sending 500K requests	Client Throughput(req/s)	Producer Throughput(req/s)	Consumer Throughput(req/s)
EC2 Client, Same VPC	**3015**	**3048**	1180

In all the above tests, the client only sends “Post” requests. To evaluate the performance of GET requests, we used local JMeter to send Post and Get requests at the same time. Compared to the result we showed in our presentation, the average response time of both GET requests

have reduced by 30%, because we have moved the two GET servlets to two separate EC2 instances to provide dedicated capacity for each servlet.

Get Servlets share 1 EC2 instance

POST

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	16000	0	0.00%	34.66	23	297	34.00	39.00	41.00	52.00	863.28	189.49	228.51
POST Request	16000	0	0.00%	34.66	23	297	34.00	39.00	41.00	52.00	863.28	189.49	228.51

GET

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	64.11	16	683	54.00	78.00	83.00	93.00	1549.22	1028.01	226.60
GET Mathces Request	64000	0	0.00%	64.11	16	683	54.00	78.00	83.00	93.00	1549.22	1028.01	226.60

GET

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	65.16	17	1097	53.00	78.00	84.00	95.00	1520.98	326.34	216.53
GET Stats Request	64000	0	0.00%	65.16	17	1097	53.00	78.00	84.00	95.00	1520.98	326.34	216.53

Get Servlets on separated EC2 instances(scaled out)

POST

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent K
POST R...	16000	43	42	53	57	79	19	210	0.00%	759.0/sec	166.60	200
TOTAL	16000	43	42	53	57	79	19	210	0.00%	759.0/sec	166.60	200

GET

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent K
GET Mat...	64000	40	40	52	55	71	19	222	0.00%	2109.5/...	1814.14	310
TOTAL	64000	40	40	52	55	71	19	222	0.00%	2109.5/...	1814.14	310

GET

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent K
GET Sta...	64000	40	40	52	56	71	17	222	0.00%	2108.6/...	456.54	300
TOTAL	64000	40	40	52	56	71	17	222	0.00%	2108.6/...	456.54	300

Incorporate Load Balancer

After transitioning our client to EC2 instances, we observed a significant increase in both client and producer throughput. Consequently, we suspected that the producer/servlet may now be the bottleneck in our system. To investigate this further, we decided to implement a load balancer to optimize performance.

Upon incorporating the load balancer, the results were striking: both client and producer throughput improved considerably. This confirms our hypothesis that the producer/servlet was indeed the limiting factor, and it demonstrates the effectiveness of the load balancer in addressing this issue.

Sending 500K requests,100 Kafka partitions,10 Consumer threads, local client

	Client Throughput(req/s)	Producer Throughput(req/s)	Consumer Throughput(req/s)
With Load Balancer	4909	5000	926
Without Load Balancer	3139	3205	/

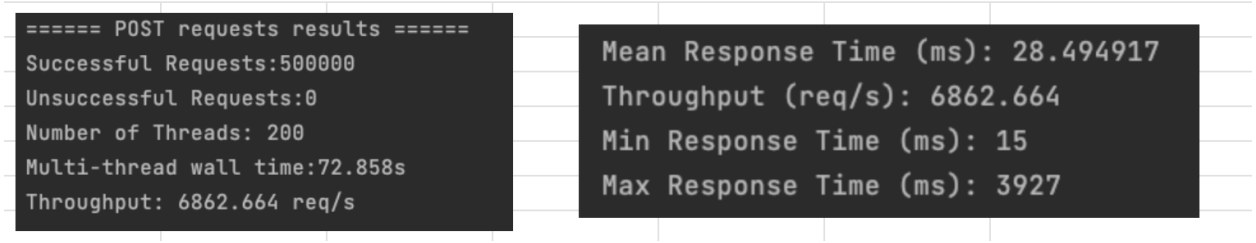
Producer: Sync -> Async write

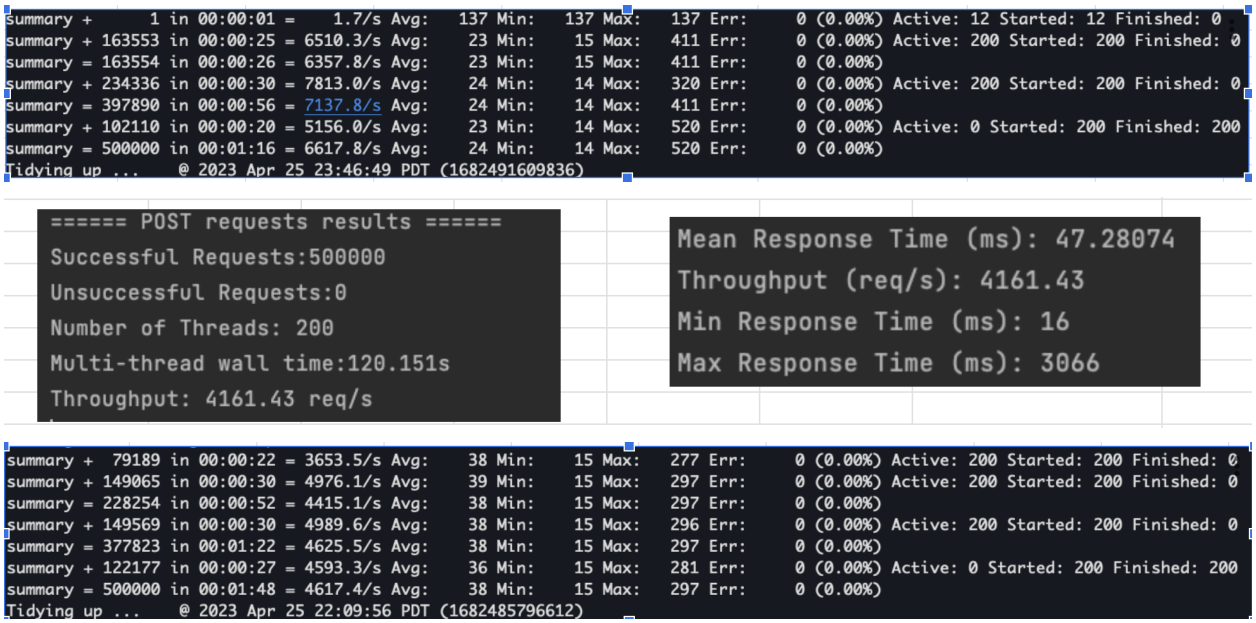
We also recognized that the producer's communication with the Kafka broker was synchronous, and we were interested in exploring the potential benefits of switching to asynchronous messaging. Although this approach could potentially introduce some data safety concerns, our primary goal was to determine if it could enhance throughput.

The results demonstrated that after implementing asynchronous messaging for the producer, we observed a substantial increase in throughput compared to the synchronous approach. This experiment further validated our efforts to improve the overall performance of our system.

Sending 500K requests,100 Kafka partitions,10 Consumer threads, local client

	Throughput (req/s) -Client	Throughput(req/s) -JMeter	Min response time(ms)	Max response time(ms)	Mean response time(ms)
With Load Balancer	**6862**	**6617**	115	3927	**28**
Without Load Balancer	4161	4617	16	3066	47





Summary

In conclusion, we have successfully optimized our system through various approaches, including adjusting Kafka producer parameters, increasing the number of Kafka partitions, modifying Kafka consumer threads, and testing both within the same and across VPCs. Additionally, we scaled out key components by adding extra servlets, Mongos instances, and dedicated EC2 instances for each member of the MongoDB sharded cluster replica set. We also incorporated a load balancer and implemented asynchronous messaging for the producer's communication with the broker. After conducting numerous experiments and tests, we are proud to report that our efforts have led to a substantial increase in throughput and significantly reduced latency compared to our initial design.

References

Apache Kafka® Performance, Latency, Throughout, and Test Results. (n.d.). Confluent Developer. Retrieved April 26, 2023, from https://developer.confluent.io/learn/kafka-performance/

Kreps, J. (2014, April 27). *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)*. LinkedIn Engineering. Retrieved April 26, 2023, from https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

Rao, J. (2015, March 12). *How to Choose the Number of Topics/Partitions in a Kafka Cluster?* Confluent. Retrieved April 26, 2023, from https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/

Assignment 4

Team Name: Call 911

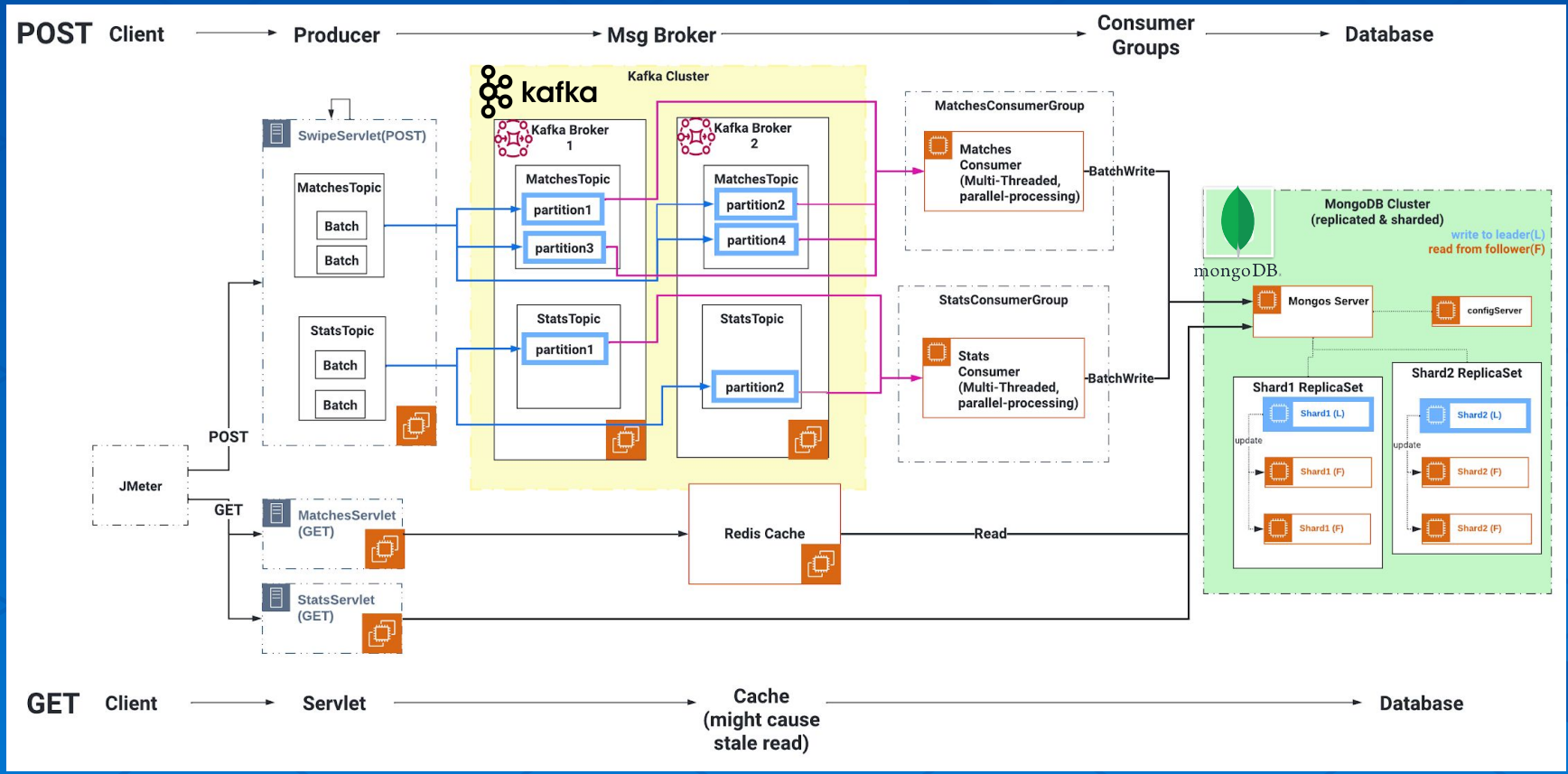
Members:
Kejian Tong, Chuan-Hsin Chen,
Chien-Chia Chiu, Chieh-Lin Lee

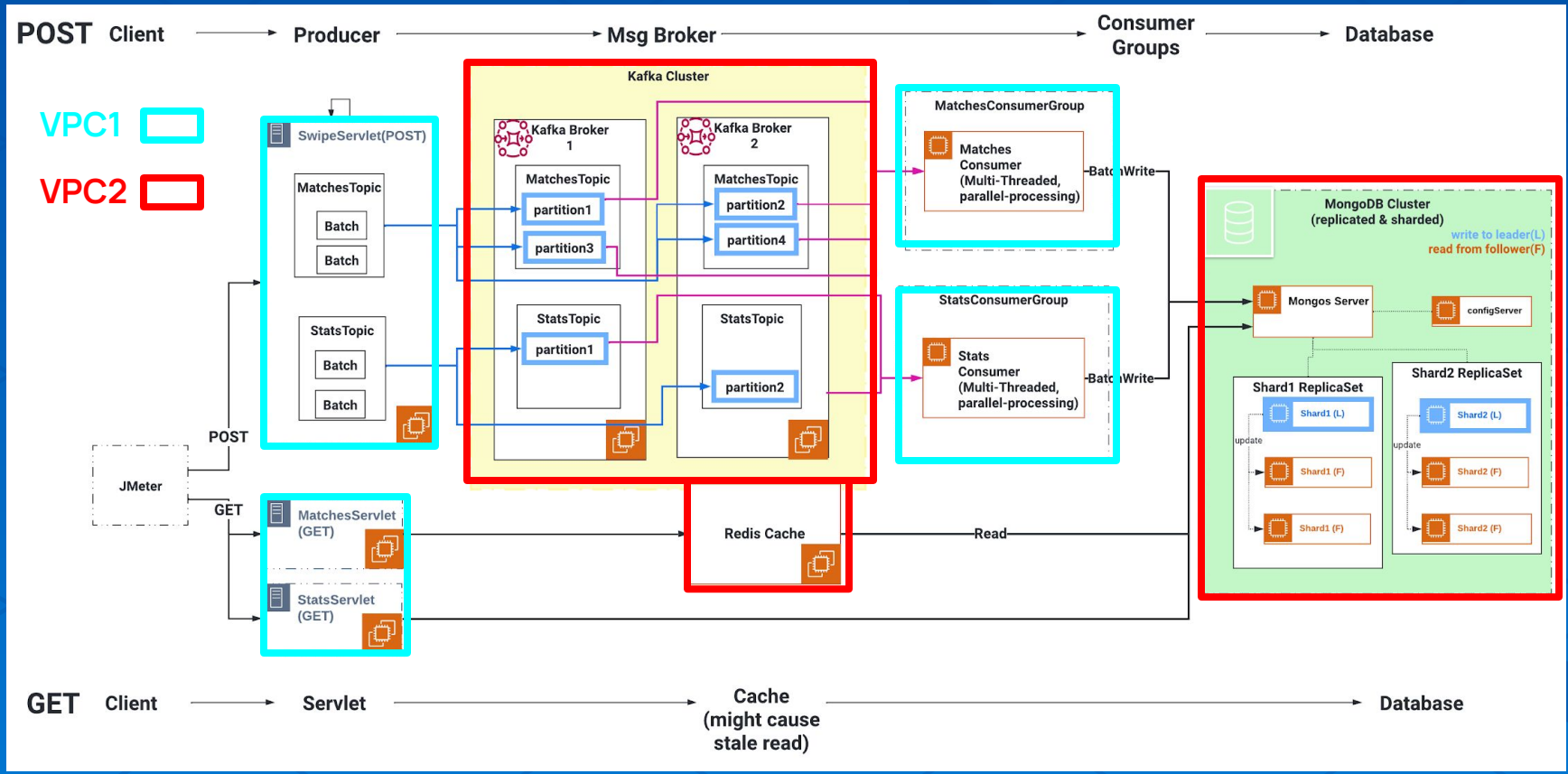
Table of contents

01

Architecture

Architecture




POST
Client
Producer
Msg Broker
Consumer Groups
Database
VPC1
VPC2
SwipeServlet(POST)
MatchesTopic
Batch
Batch
StatsTopic
Batch
Batch
Kafka Cluster
Kafka Broker 1
MatchesTopic
partition1
partition3
StatsTopic
partition1
Kafka Broker 2
MatchesTopic
partition2
partition4
StatsTopic
partition2
MatchesConsumerGroup
Matches Consumer (Multi-Threaded, parallel-processing)
StatsConsumerGroup
Stats Consumer (Multi-Threaded, parallel-processing)
MongoDB Cluster (replicated & sharded)
write to leader(L)
read from follower(F)
Mongos Server
configServer
Shard1 ReplicaSet
Shard1 (L)
update
Shard1 (F)
Shard1 (F)
Shard2 ReplicaSet
Shard2 (L)
update
Shard2 (F)
Shard2 (F)
JMeter
POST
GET
MatchesServlet (GET)
StatsServlet (GET)
Redis Cache
Read
GET
Client
Servlet
Cache (might cause stale read)
Database

Data Model (NoSQL)

twinder.Stats

5 DOCUMENTS

Documents | Aggregations | Schema | Explain Plan | Indexes | Validation

Validation Action ERROR Validation Level STRICT

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      '_id',
      'likes',
      'dislikes'
    ],
    properties: {
      _id: {
        bsonType: 'int',
        minimum: 1,
        maximum: 50000,
        description: 'swiperId'
      },
      likes: {
        bsonType: 'int',
        minimum: 0,
        description: 'the number of users liked by this swiper'
      },
      dislikes: {
        bsonType: 'int',
        minimum: 0,
        description: 'the number of users disliked by this swiper'
      }
    }
  }
}
```

twinder.Matches

5 DOCUMENTS

Documents | Aggregations | Schema | Explain Plan | Indexes | Validation

Validation Action ERROR Validation Level STRICT

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      '_id',
      'matches'
    ],
    properties: {
      _id: {
        bsonType: 'int',
        minimum: 1,
        maximum: 50000,
        description: 'swiperId'
      },
      matches: {
        bsonType: 'array',
        items: {
          bsonType: 'int',
          minimum: 1,
          maximum: 50000
        },
        uniqueItems: true,
        description: 'must be an array of swipeeId, which ranges from 1 to 50000. '
      }
    }
  }
}
```

_id (i.e. the swiperId)	matches(i.e. an array of swipeeId's)
552	[10, 1043, 72]
7	[233]
...	...

02

Highlights (i.e. Bonus Points!!)

Highlights

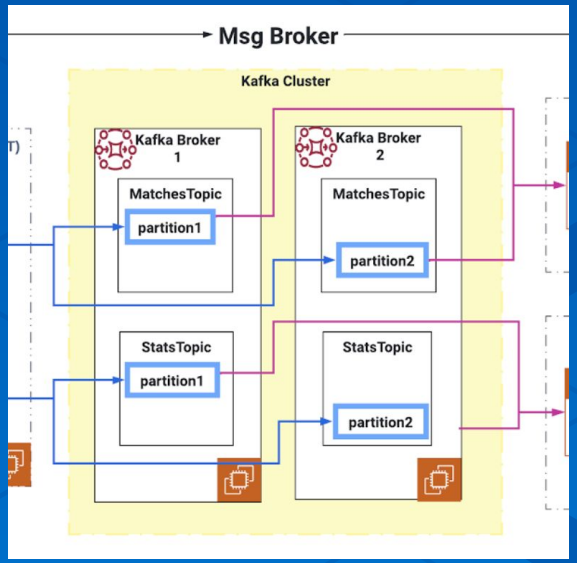
High Performance

Scalability

Availability

Data Safety

Performance



Kafka > RMQ

Sequential read /write from disk

batch processing

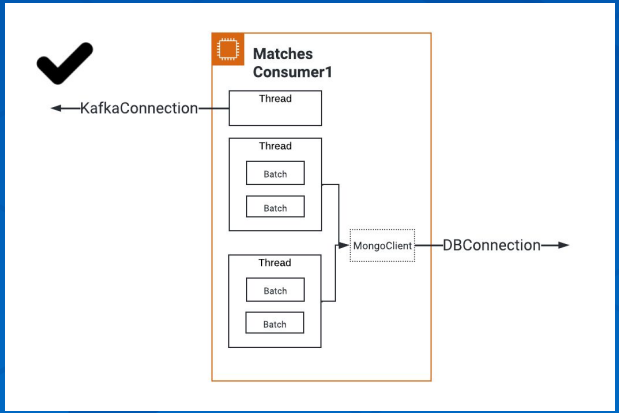
Kafka Partitions

Separate write loads evenly to multiple Kafka brokers.

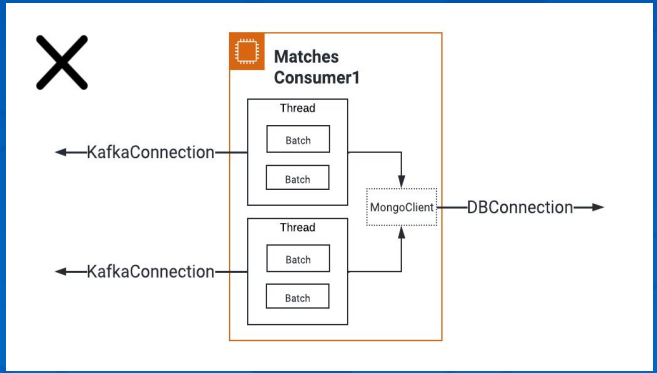
Partitions also provide prerequisite for parallel delivery in Consumers

Performance

Efficient Multi-threaded Consumer



1 Connection model,
saves Kafka Connection resource



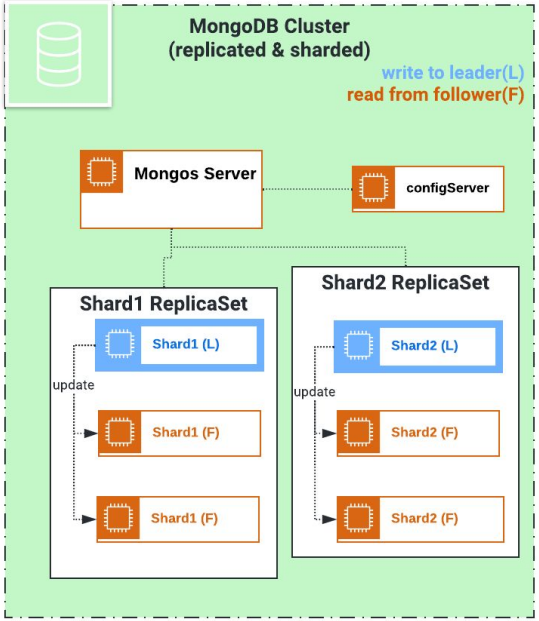
Connection-per-thread model

Batch update

Consumer to mongoDB

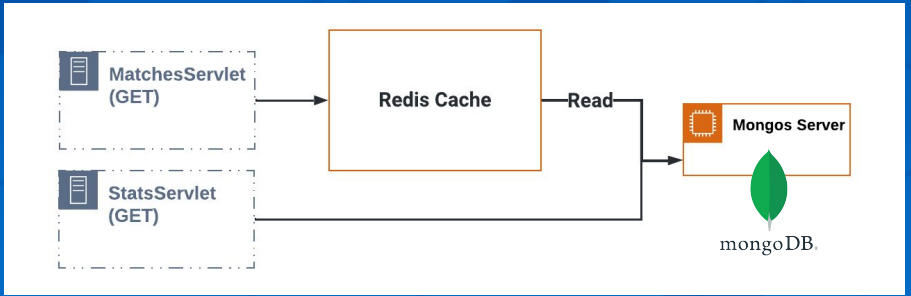
MongoDB

- CQRS: **Read replica** improve read efficiency
- **Sharding: Distribute query loads** to multiple nodes.

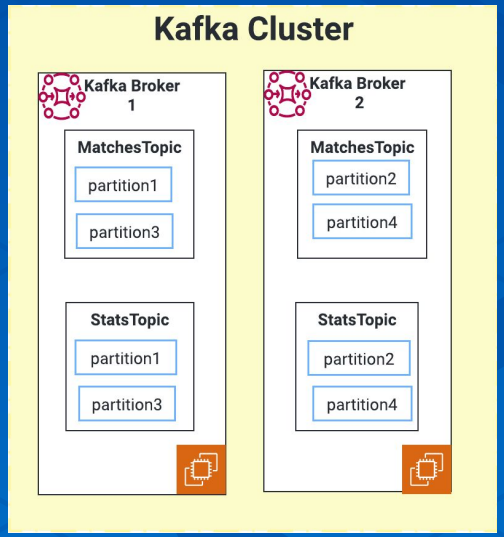


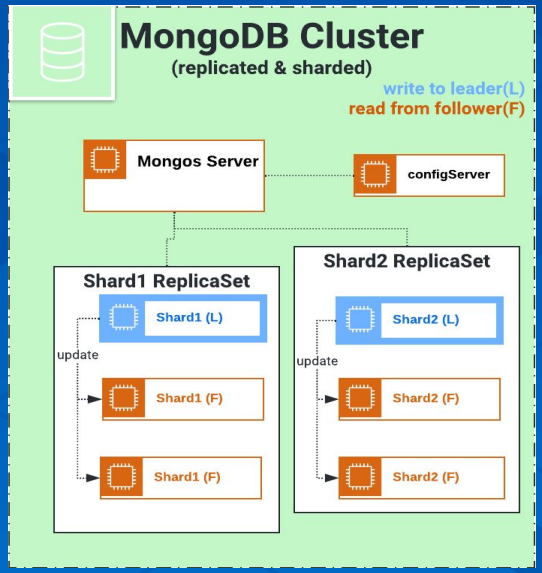
Cache

Use Redis as cache for Get requests, to improve read efficiency



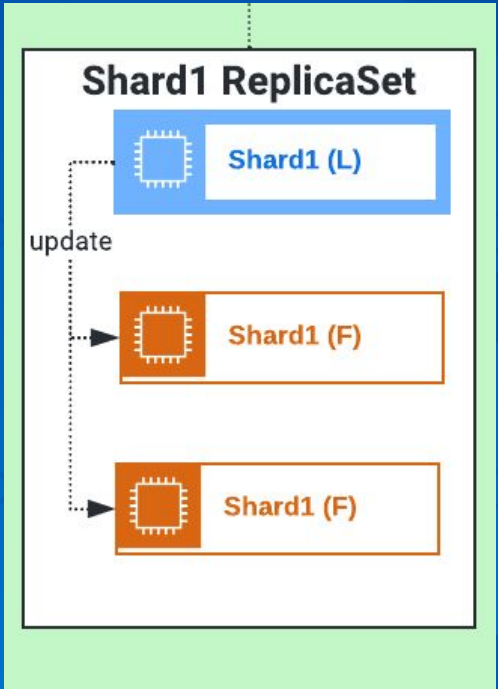
Scalability





kafka & mongoDB. can be easily scaled by adding more nodes (& partitions) into the clusters

Availability



Replica Set

In MongoDB, when leader is dead, fails over to followers

Data Safety

Kafka Broker

Acknowledges Producer, only **after the msg is written to the brokers.**

```
props.put(ProducerConfig.ACKS_CONFIG, "all");
```

Kafka Consumer

Kafka consumer **manually commit**, to prevent data loss.

Kafka consumer **ensures processing order** within each partition. (guaranteed by Multi-threaded model)

03

JMeter Result

Result (Run at the same time)

Throughput: POST 863, GET 1550

POST

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	16000	0	0.00%	34.66	23	297	34.00	39.00	41.00	52.00	863.28	189.49	228.51
POST Request	16000	0	0.00%	34.66	23	297	34.00	39.00	41.00	52.00	863.28	189.49	228.51

GET *

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	64.11	16	683	54.00	78.00	83.00	93.00	1549.22	1028.01	226.60
GET Mathces Request	64000	0	0.00%	64.11	16	683	54.00	78.00	83.00	93.00	1549.22	1028.01	226.60

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	65.16	17	1097	53.00	78.00	84.00	95.00	1520.98	326.34	216.53
GET Stats Request	64000	0	0.00%	65.16	17	1097	53.00	78.00	84.00	95.00	1520.98	326.34	216.53

* Matches & Stats were run at the same instance due to the limit of EC2 instances

Result (Run separately)

Discover 1: Throughput of GET is faster when running separately. (1550 → 2749)
→ Less connection to Redis and MongoDB

GET match
(run separately)

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	27.75	12	273	25.00	32.00	35.00	42.00	2749.14	1877.55	402.11
GET Mathces Request	64000	0	0.00%	27.75	12	273	25.00	32.00	35.00	42.00	2749.14	1877.55	402.11

GET match
(run together)

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	64.11	16	683	54.00	78.00	83.00	93.00	1549.22	1028.01	226.60
GET Mathces Request	64000	0	0.00%	64.11	16	683	54.00	78.00	83.00	93.00	1549.22	1028.01	226.60

POST: (Ignored since no significant change)

Result (Run separately)

Discover 2: Redis Cache improves GET response time

GET Match

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	27.75	12	273	25.00	32.00	35.00	42.00	2749.14	1877.55	402.11
GET Mathces Request	64000	0	0.00%	27.75	12	273	25.00	32.00	35.00	42.00	2749.14	1877.55	402.11

With Redis cache

GET Stats

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	64000	0	0.00%	35.49	14	236	28.00	43.00	49.00	59.00	2367.56	507.99	337.05
GET Stats Request	64000	0	0.00%	35.49	14	236	28.00	43.00	49.00	59.00	2367.56	507.99	337.05

Without Redis cache

Redis Cache Hit rate: 92%

```
[ec2-user@ip-172-31-31-79 ~]$ redis-cli info | grep -E 'keyspace_hits|keyspace_misses'
keyspace_hits:460737
keyspace_misses:51263
```

Result (POST 500K with 200 threads)

In HW4, throughput is 25% up using Kafka, compared with HW3 (using RabbitMQ).

HW4
(Kafka)

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	500000	0	0.00%	51.91	17	1104	36.00	45.00	48.00	56.00	3544.26	777.77	938.18
POST Request	500000	0	0.00%	51.91	17	1104	36.00	45.00	48.00	56.00	3544.26	777.77	938.18

HW3
(RabbitMQ)

MainPart2

```
====== POST requests results ======
Successful Requests:600000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:210.519s
Throughput: 2850.099 req/s


Mean Response Time (ms): 69.15736
Throughput (req/s): 2850.099
Min Response Time (ms): 17
Max Response Time (ms): 2034
```

04

Improvements

Availability

- **Avoid SPoF**

All Servlets/Consumers/Kafka Cluster/ Redis can also have replicas.

- **Avoid Leader/follower split**

Odd number of nodes in all Leader-Follower pattern (MongoDB & Kafka cluster & Redis) ;

To ensure **when leader fails**, rest of the instances can successfully **elect a new leader**.

Performance/Scalability

- **Consumer Group**

Add more instances in one Consumer Group,
-> parallel polling from Kafka, increase consume rate

- **Get Servlets**

Separately run two GET Servlets on dedicated EC2 instances.

Thanks!