Distributed Job Scheduler

Job Scheduler, where you define a job to be scheduled at a specific time and this job can be one-off or recurring.

Functional Requirements:

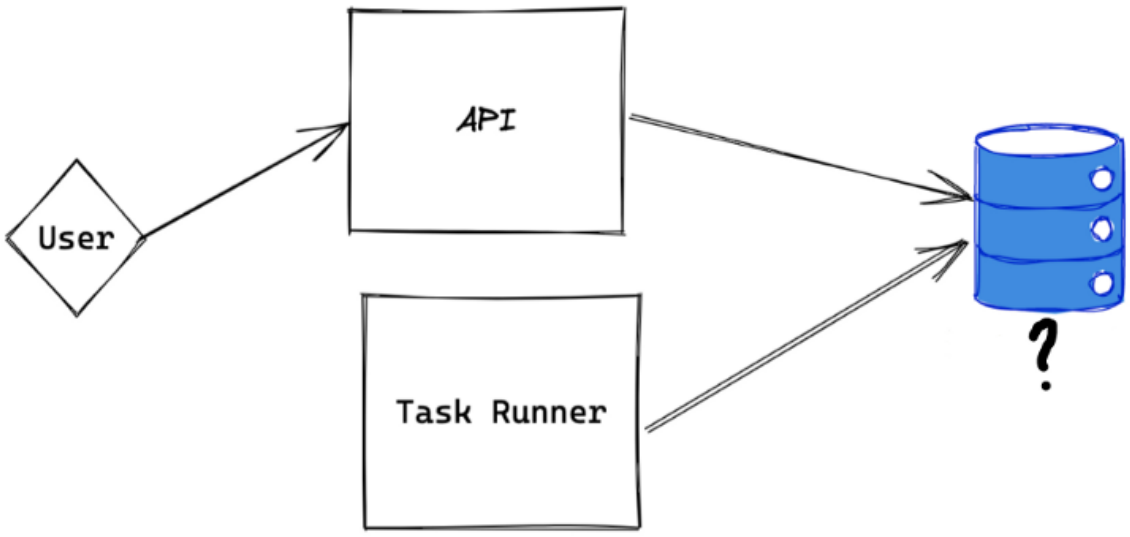
- A job can be scheduled for one time or multiple executions (cron job) by other services/microservices.
- Query all jobs owned by an application/user.
- Query the status of a job (running, failed, finished, etc).
- Query the execution history of a job.
- Retry support for failed tasks.
- On-time execution (when a job is scheduled to run at 1 PM, it should be triggered around 1 PM)

Non-Functional Requirements:

- **Scalability**: Thousands or even millions of jobs can be scheduled and run per day.
- **Durability**: Jobs must not get lost -> we need to persist in jobs
- **Reliability**: Jobs must not be executed much later than expected or dropped -> we need a fault-tolerant system
- **Availability**: It should always be possible to schedule and execute jobs -> (dynamical) horizontal scaling
- Jobs must not be executed multiple times (or such occurrences should be kept to a minimum).

Initial design thoughts

It's always a good idea to go from a single machine or most straightforward design first, then try to scale or optimize the solution. Here is my initial idea:



Here we have three components:

API server: A HTTP web server where customers can hit directly to add jobs.

Database: This is the database where we will store the job-related information. We will discuss more on this which database to use.

Task Runner: It is a daemon that will run in the background. Every minute it will run a query on the DB to find out the due jobs and run them concurrently as much as possible.

For simplicity, we will not use any DB, we will spawn a daemon thread as a task runner.

Let's say our class exposes an API called `add_action()` *that'll take a parameter* `action`*, which will get executed after user-specified seconds. Anyone calling this API should be able to specify after how many seconds should our class invoke the passed-in action. To simulate this our class would be:*

```
class DeferredCallbackExecutor():
    def __init__(self):
        self.actions = list()
        self.cond = Condition()
        self.sleep = 0

    def add_action(self, action):
        pass

    def start(self):
        while True:
            # execution logic comes here
            pass
```

We will define a simple `DeferredAction` class, an object of which will be passed into the `add_action` method. This method will be adding the action to our min-heap which is represented by variable `actions`. Additionally, we'll be using the `heapq` module to heapify and perform other heap related operations on the `actions` list variable. We also define the `__lt__()` method in the `DeferredAction` class which helps compare two actions based on when they will get executed.

```
# class representing an action
class DeferredAction(object):
    def __init__(self, exec_secs_after, name, action):
        self.exec_secs_after = exec_secs_after
        self.action = action
        self.name = name
```

```
    def __lt__(self, other):
        return self.execute_at < other.execute_at
```

The `add_action` complete implementation:

```
class DeferredCallbackExecutor():
    def __init__(self):
        self.actions = list()
        self.cond = Condition()
        self.sleep = 0

    def add_action(self, action):
        # add exec_at time for the action
        action.execute_at = time.time() + action.exec_secs_after

        self.cond.acquire()
        heapq.heappush(self.actions, action)
        self.cond.notify()
        self.cond.release()
```

The start() method is the thread daemon which will act as our task runner.

```
def start(self):

        while True:
            self.cond.acquire()

            while len(self.actions) is 0:
                self.cond.wait()

            while len(self.actions) is not 0:

                # calculate sleep duration
                next_action = self.actions[0]
                sleep_for = next_action.execute_at -
math.floor(time.time())
                if sleep_for <= 0:
                    # time to execute action
                    break

            self.cond.wait(timeout=sleep_for)

            action_to_execute_now = heapq.heappop(self.actions)
            action_to_execute_now.action(*(action_to_execute_now,))
```

```
            self.cond.release()

# The Driver function to call the above.

def say_hi(action):
    pass

if __name__ == "__main__":

    action1 = DeferredAction(3, ("A",), say_hi)

    action2 = DeferredAction(2, ("B",), say_hi)

    action3 = DeferredAction(1, ("C",), say_hi)

    action4 = DeferredAction(7, ("D",), say_hi)

executor = DeferredCallbackExecutor()

t = Thread(target=executor.start, daemon=True)

t.start()

executor.add_action(action1)

executor.add_action(action2)

executor.add_action(action3)

executor.add_action(action4)

# wait for all actions to execute

time.sleep(15)
```

Issues with this design

There are multiple single points of failure.

*The most crucial component is the **task runner** which needs to find out the due jobs and run the tasks. One single machine will not be able to run 1000 tasks concurrently. We need more machines to handle this task run load.*

Another issue is we need to run a query on the DB table to get the due jobs every single minute. We definitely need indexing to reduce the query load.

A single DB can't hold all the data, So we need more sharded nodes.

Traffic Estimation

Is this for internal use or it is a client-facing scheduler?

Client-facing schedulers can be bombarded with tons of traffic (such as schedulers in AWS). In this article, we assume the traffic is large for better generalization:

In this article, we assume the traffic is large for better generalization:

- Assuming that we are ingesting 1 million jobs each of size 200KB per day.
- Assuming jobs for 5 years, total number of jobs = 10M*365*5 = 19 Billion.
- Total size of data in MySQL for 5 years = 19billion*200bits = ~570GB.

Based on the estimations above, we can make the following conclusions:

- The backend database should be horizontally scalable, as the execution history grows quickly (~ 100 million rows of data created every day).
- The system is read-heavy, as a job is created once and read many times by the scheduler. The same holds true for execution history, an entry is created and can be queried many times by the user.
- We need a distributed group of workers to run jobs concurrently with varying capabilities.
- All critical services should be replicated to handle the large traffic.

Which Database to use?

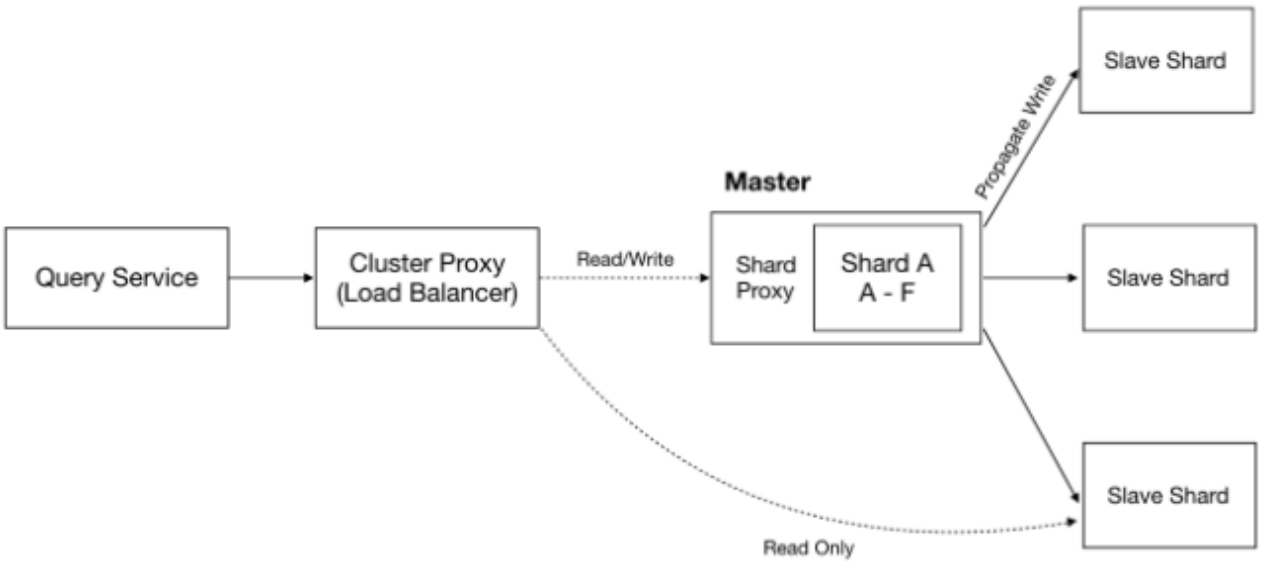
After all the above discussion we can use **RDBMS** because our system is **read-heavy**. MySQL is used because it has ACID properties which allow transactions and row-based locking.

This is required so that multiple worker processes reading the pending jobs and taking them up for execution, will not see inconsistent view for a job i.e. a process while updating status of a job from PENDING to PROCESSING, another process is trying to read it will read the status as PENDING, thus 2 processes will be taking up the same job.

Availability SQL Databases:

We use Active-Passive/Master-Slave Replication for each Partition in a semi-synchronous fashion. One Slave/Follower will follow synchronously while the others will receive the Replication Stream asynchronously. That way, we can be sure that at least one Slave holds up-to-date data in case the Master fails (due to network partitions, server outage, etc.) and that Slave will be promoted to be the new Leader.

Write will always performed on master and read can be done from any slave , so the read will be faster.



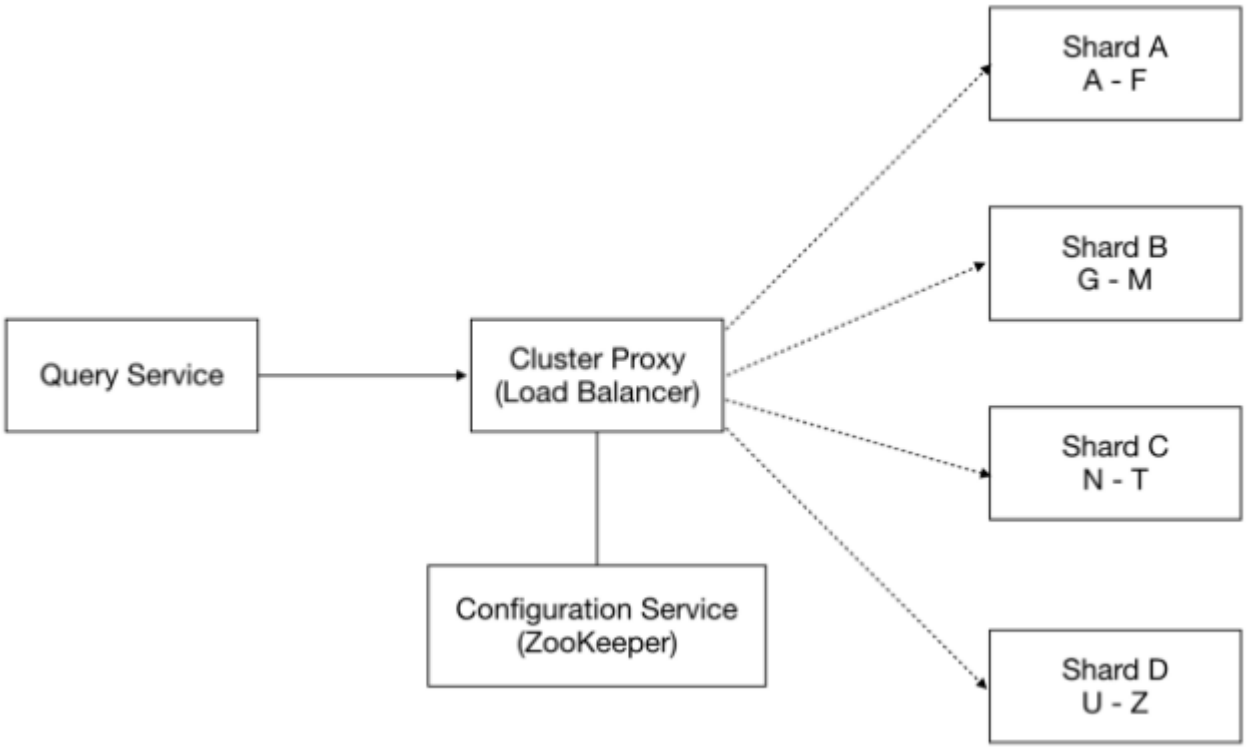
Replication to Slave Shards

Scaling SQL Databases

The database is sharded into an adequate number of shards to distribute the load and data.

Assuming that we've already applied the sharding logic to our SQL database, you might be asking, how would a query service know which shard to communicate with to retrieve data?

If you're thinking about a middleman, you're on the right track. A solution is to place a proxy, like a load balancer, that sits in front of the query service and the database. Underneath the hood, the load balancer relies on a Configuration Service such as ZooKeeper to keep track of the shards and their indices. By processing the query, the load balancer will know exactly which shard to direct the request to.

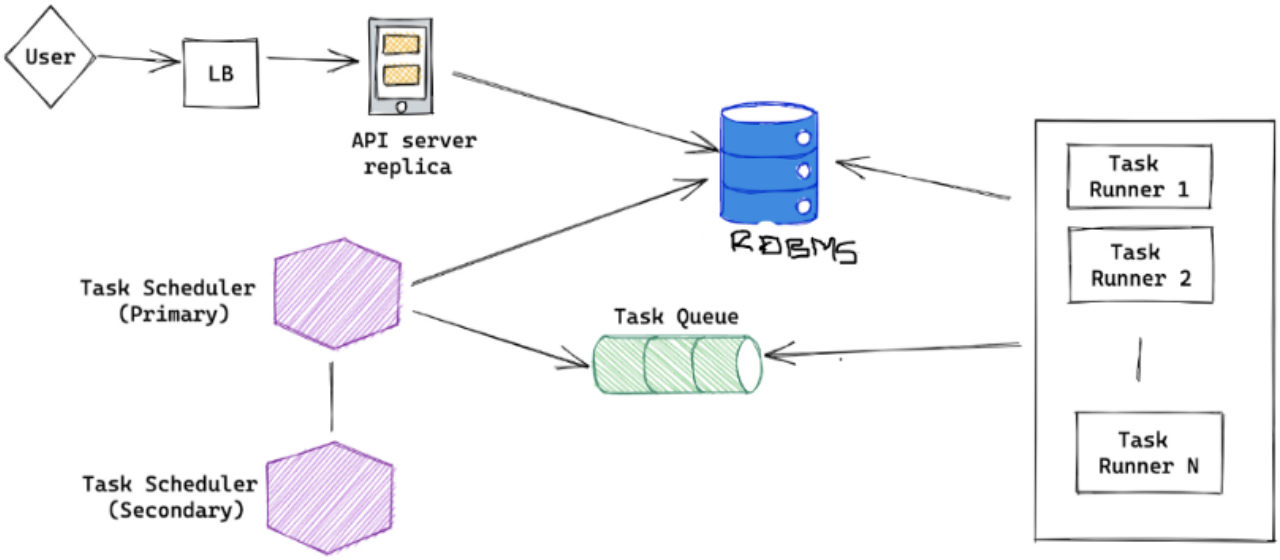


Consistency in SQL Database

SQL databases were not designed with scalability in mind but with ACID properties (Atomicity, Consistency, Isolation, and Durability).

Going back to the master-slave architecture, it will take some time for the data to be propagated from the master to its slaves. Therefore, it exists a window of time that the master and its slaves can have different states. In scaling a SQL database, we sacrifice consistency for **eventual consistency.**

High-Level Design



Database Model:

Table: Job

- Given userId, add job
- Given userId, retrieve all jobIds

This DB will satisfy the above requirements.

Attribute	Type
user_id (sharding key)	uuid
job_id	uuid
actual_job_execution_time	date
job_status	string
job_type	string
job_interval	int
current_retries	int
max_retries	int
scheduled_job_execution_time	date

job_status can be one of the following: NOT_STARTED, CLAIMED, PROCESSING, SUCCESS, RETRIABLE_FAILURE, FATAL_FAILURE

Table: Execution_history Table

This table is used to store the execution details of a job. This can be used to fetch the result of the scheduled job.

```
+ — — — — — — — — — — —  + — — — + — — — — +
| Attribute              | Type  |
+ — — — — — — — — — — —  + — — — + — — — — +
| actual_job_execution_time |date|
| job_id (sharding key)| uuid |
| Status               | string|
| retry count          | int|
+ — — — — — — — — — — — — — — — — + — — — — +
```

Table: Schedule

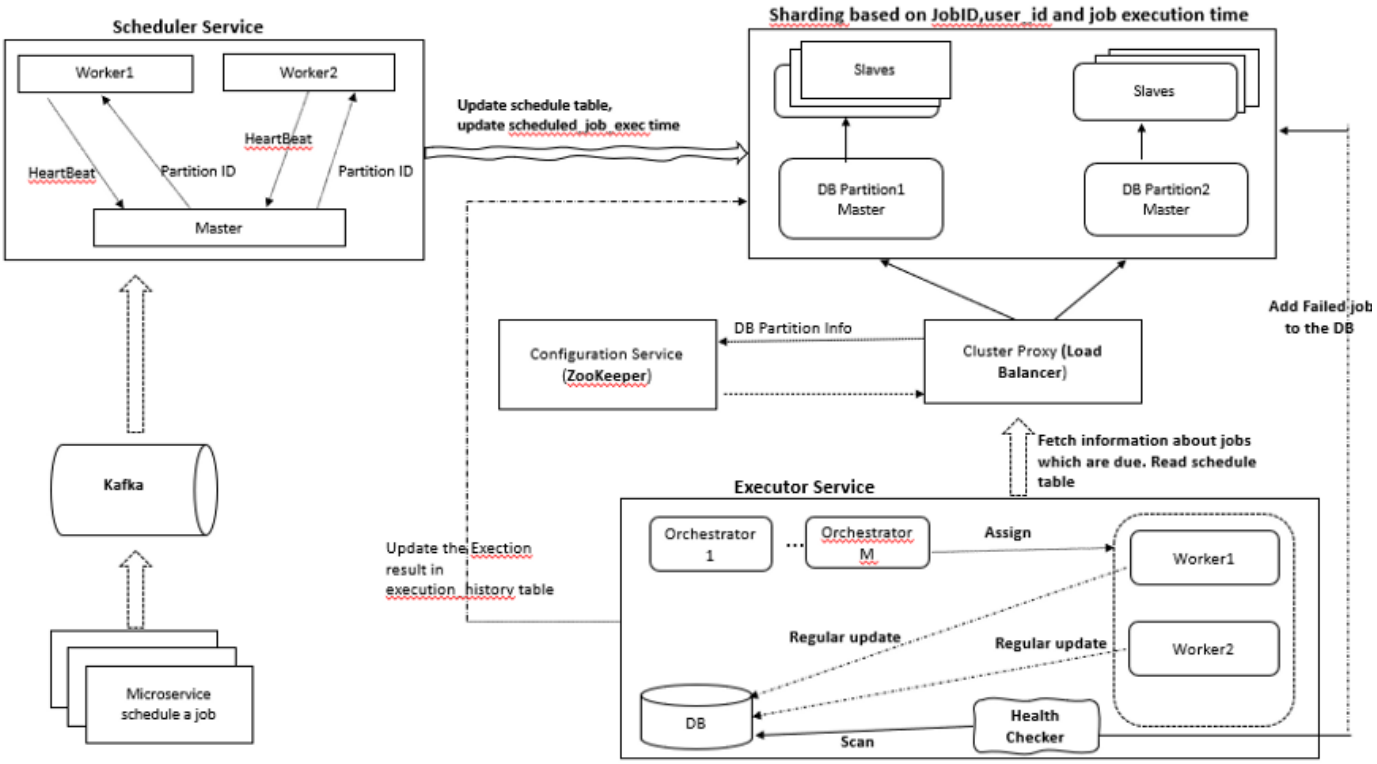
The core feature of any job scheduler is, of course, running jobs on time. Therefore, we need a data model that makes it easy to filter jobs by execution time. Below is a simple design that works well for a small amount of data:

```
+ — — — — — — — — — — —  + — — — + — — — — +
| Attribute              | Type  |
+ — — — — — — — — — — —  + — — — + — — — — +
| scheduled_job_execution_time (Sharding key) |date|
| job_id | uuid |
+ — — — — — — — — — — — — — — — — + — — — — +
```

execution time is converted to UNIX timestamp with **minute-level granularity**. This means that jobs that are scheduled to run in the same minute will share the same partition key or sharding key. To get jobs that are scheduled to run right now, simply find the right partition with the current UNIX timestamp:

```
SELECT * FROM ScheduleTable WHERE NextExecution > "1641082500" AND
NextExecution < "1641082580"
```

Deep Dive:



How does Scheduler Service Work?

Will consume the Messages from Kafka (requesting a Job enqueueing). They will generate a unique job Id using e.g. the **Snowflake ID Generation** concept. Based on that ID (e.g. by hashing it) they decide into which Database Partition the Job will go. They create a Job and Trigger record according to the Message in the corresponding Database Partition. Will update zookeeper about the partition.

When the number of jobs running concurrently is small, the query yields a reasonable amount of data. However, if, let's say, 100K jobs are scheduled to run in this minute, we certainly need more workers to handle the ingress data from the query as well as push messages to the queue.

To make sure a complete work assignment, we can borrow some ideas from **MapReduce**, where a master is used to assign and monitor workers. If a worker dies, the master will resend its work to some other nodes.

How does a job executor work ?

On Startup it will fetch the Database Partitioning info from ZooKeeper. It will send/store the Partition Assignment to ZooKeeper.

Job executor service has multiple consumers that pull data from the DB. Consumer machine also has master and worker processes. Both master and worker processes operate on a Pull based model. Master will poll jobs from the DB and worker process will continuously poll the job.

It pulls information from the Database Partition and fights with other Executor instances assigned to the same Database Partition for Jobs that are due to execute. The fighting works by using Row Locks.

Job Executor also maintains a local DB to monitor health of worker and to retry the failed tasks.

- When a job is picked up from the DB, consumer's master updates JOB db attribute job_status=CLAIMED.
- When worker process picks up the work, it updates job_status=PROCESSING and continuously send health check to local DB.
- Upon completion of a job, worker process will push the result to **execution_history DB** update JOB db job_*status=COMPLETED or FATAL_FAILED*, and local db with the status.
- Both worker processes and master will update the health check inside the local database.

Role of Health Checker?

1. When a job is assigned, an entry is created in the local database with the latest update time.
2. All workers are required to refresh the update time ~10s.
3. The health checker will periodically scan the table, looking for jobs with stale update time (e.g. older than 30 seconds).
4. Jobs that meet the above criteria are sent back to the queue for re-execution.

Role of Message Queues, in general:

We use Message Queues (here Kafka which is more of an (append) log than a Queue in the conventional sense, you can find great docs on the official website) in order to:

- Be able to scale the consumer and producer nodes independently (in Kafka we have Topics which can be horizontally partitioned and scale by that)
- We decouple the consumer and producer from each other
- Lower latency for the producer (doesn't have to wait for a response)
- Durability and Reliability: When a Consumer Node crashes another Node can process the Message which otherwise would be lost (see Offset in Kafka). The Messages are persisted.
- We can throttle/limit the number of messages the consumers process (see Backpressure)
- Kafka offers message ordering

Useful Resources

- https://leetcode.com/discuss/general-discussion/1082786/System-Design%3A-Designing-a-distributed-Job-Scheduler-or-Many-interesting-concepts-to-learn
- https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox
- https://sre.google/sre-book/distributed-periodic-scheduling/
- https://towardsdatascience.com/ace-the-system-design-interview-job-scheduling-system-b25693817950

If you enjoyed the article, clap and con