Scalable WEB Scraper

Problem -

1. Scraper implemented in Task 1 is a single node synchronous architecture along with simplest scraping library.
2. But scraper is only suitable for small projects.
3. It will perform ETL for each website(url) in a sequential manner i.e one after another .
4. But Case study involves extracting from large number of websites.
5. This will take scraper long time to finish execution hence not an efficient solution
6. It possess both high time and space complexity thus not a suitable scenario.

Potential Architecture Solutions -

1. Naive

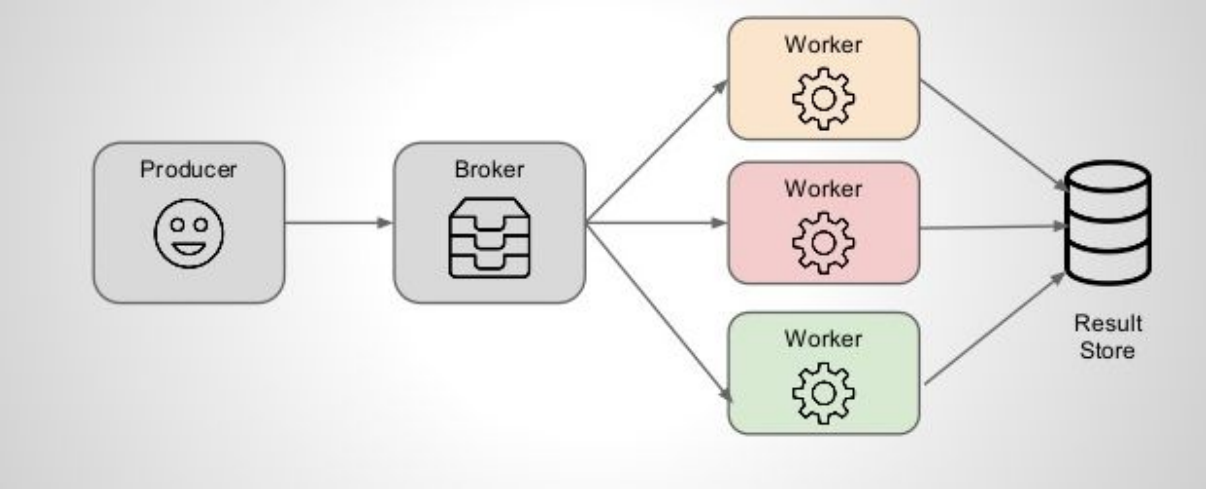
a. **Python multithreading** - Assign each thread with seperate ETL.
   i. Improves Time Complexity
   ii. But huge increase in Code complexity.
   iii. Space Complexity still an issue , there will be a huge load on single node.

b. **Seperate Machines** - Each ETL assigned a separate machine (lets say docker containers).
   i. Improves Time Complexity
   ii. Code Maintenance still a challenge as there will be huge number of docker containers.
   iii. Not Scalable

2. Async

a. **Single node async** - Each ETL written in an async manner.
   i. Python >= 3.5 supports async await . So no reinventing wheel
   ii. Improves Time Complexity
   iii. But still challenge remains to manage code complexity

b. **Pub Sub Architecture** - Async architecture built with background Task management



i. **Components**

1. Celery - Task Management
2. Redis - Task Queuing (Broker)
3. Worker Nodes - Executable Unit
4. Flower - Worker Management
5. MySql DataBase
6. RedShift - if data volume is getting huge we can shift
7. Scheduler - There are two options
    a. Treat each ETL as CRON Job
    b. Celery has a built in scheduler

ii. **WorkFlow**

1. Celery publisher creates tasks (in our case it will be ETL for each website) and enque to Redis queue
2. Celery consumers(Workers) will deque the tasks run the ETL pipeline and finally save the results back to Db

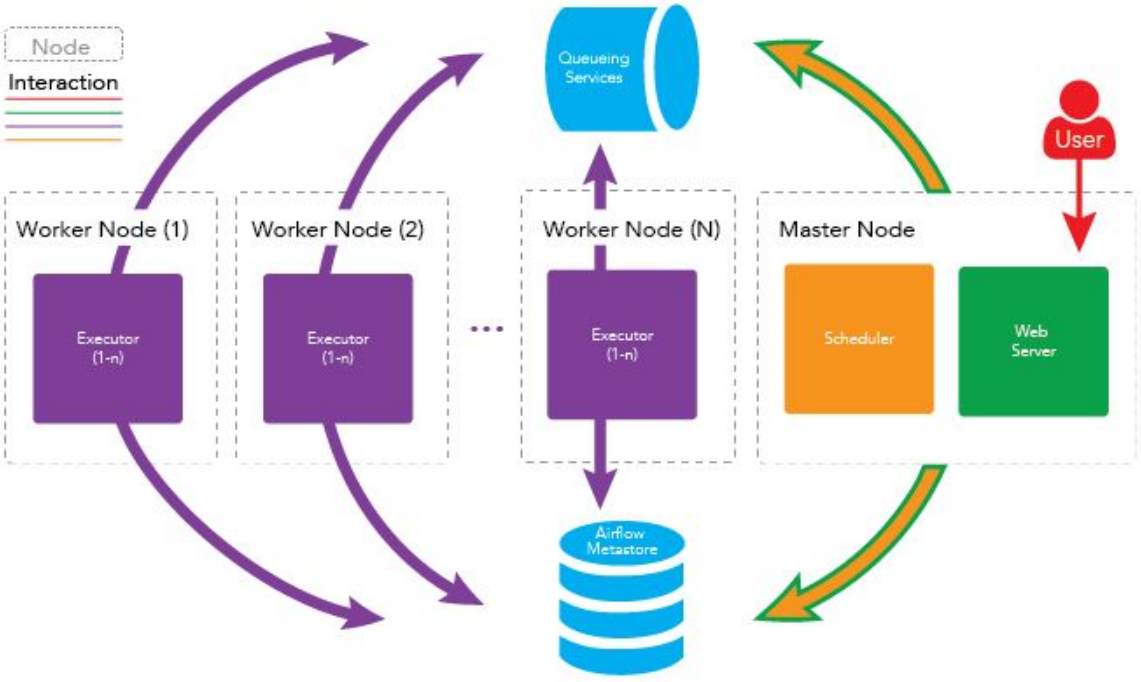
iii. **Advantages**

1. Development tool availability -. Celery and Redis(can be any message broker) both are now very mature tools in python ecosystem.
2. Easy Deployments- Worker Node and Celery Node easily dockerizable thus easy to deploy
3. Scalability - Both single node multi workers(thread) scrapper or multi node multi worker(thread) architecture possible

iv. **Disadvantage**

1. Close Coupling of scheduler and task manager

3. **Large Scale Distributed Scraper** - Separating the concern between scheduler , workers and task management



a. Solved pub sub tight coupling.
b. **Components of architecture**
   i. Master Node - Airflow Scheduler + Web Server + Celery Executor
   ii. Worker Nodes - Executable Unit Cluster

iii. Task Queuing - Redis

c. **Workflow** -
   i. Airflow creates ETL DAGs for each website.
   ii. Airflow fires Celery and it submit DAGs to Redis
   iii. Workers pick individual DAG and execute separately.
   iv. Airflow monitors as well can schedule these DAGs
d. **Advantages** -
   i. **Distributed Processing** -
      a. Separation of concerns (Master and Worker Nodes)
      b. Lowered the Code Complexity
      c. Improved Time and Space Complexity
   ii. **Higher Availability** - Any Worker Node Failure won't effect the cluster
   iii. **Scaling Workers**
      1. **Horizontally** - add or remove more executor nodes to the worker cluster without any downtime.
      2. **Vertically** - Increase the number of celeryd daemons running on each node. Simple configuration change in airflow config file
e. **Disadvantages**
   i. Steep Learning Curve
   ii. Airflow is still young