Perancangan System Crawler Dengan Menerapkan Arsitektur Distributed Task

Heri Santoso

Abstrak— **Kebutuhan akan data insight pada online marketplace sangat penting. Demikian juga cara dalam mendapatkan data yang cukup banyak tentunya membutuhkan otomatisasi seperti crawling data pada website marketplace. Karena data yang cukup banyak, crawler system sering tidak optimal dalam melakukan crawling data. Penerapan distributed task pada crawler system memberikan kemudahan dalam scaling server baik secara vertikal dan horizontal. Dengan demikian data yang banyak dan terus tumbuh dapat diatasi oleh crawler system. Perancangan aplikasi menggunakan bahasa python, dengan server aplikasi menggunakan Google Cloud Computing. Dalam arsitektur distributed task membutuhkan komponen berupa message broker. Message broker yang dipakai dalam perancangan system ini adalah RabbitMQ. Hasil dari perancangan crawler system akan digunakan oleh PDC Media Group untuk pengumpulan data produk marketplace. Data tersebut kemudian nantinya menjadi bahan penentuan pengambilan keputusan bisnis dari PDC Media Group sendiri. Pengujian crawler system menggunakan 3 skenario, yaitu dengan 1 worker, 2 worker, dan 3 worker. Hasil untuk skenario 1 worker adalah 19.3 request per second dan 332 ms untuk response time. Hasil untuk skenario 2 worker adalah 41.4 request per second dan 328 ms untuk response time. Sedangkan hasil untuk skenario 3 worker adalah 60 request per second dan 331 ms untuk response time.**

***Kata Kunci*: Distributed Task, Python, RabbitMQ, Web Crawler, skalabilitas.**

Abstract— **The need for data insight in the online marketplace is very important. Likewise, how to get quite a lot of data, of course, requires automation such as crawling data on the marketplace website. Due to the large amount of data, crawler systems are often not optimal in crawling data. The application of distributed tasks on the crawler system provides convenience in scaling the server both vertically and horizontally. Thus the large and growing data can be handled by the crawler system. The application design uses the python language, with the application server using Google Cloud Computing. In a distributed task architecture requires a component in the form of a message broker. The message broker used in designing this system is RabbitMQ. The results of the crawler system design will be used by PDC Media Group for data collection of marketplace products. The data will then be used as material for determining business decisions from PDC Media Group itself. Testing the crawler system uses 3 scenarios, namely with 1 worker, 2 worker, and 3 worker. The results for the 1 worker scenario are 19.3 requests per second and 332 ms for response time. The results for the 2 worker scenario are 41.4 requests per second and 328 ms for response time. While the results for the 3 worker scenario are 60 requests per second and 331 ms for response time.**

Keywords: Distributed Task, Python, RabbitMQ, Web Crawler, scalability.

[1] *Mahasiswa, Jurusan Tehnik Informatika Universitas Islam Balitar, Jalan Majapahit No 2-4, Sananwetan, Blitar 66131 (tlp: 021 5703303; fax: 021 5733125; e-mail:herisantoso1998@gmail.com)*

I. PENDAHULUAN

Web crawler atau yang dikenal juga dengan istilah *web spider* atau *web robot* adalah program yang bekerja dengan metode tertentu dan secara otomatis mengumpulkan semua informasi yang ada dalam suatu *website*. Web crawler mengunjungi setiap alamat website yang diberikan kepadanya, kemudian menyerap, dan menyimpan semua informasi yang terkandung di dalam website tersebut. Setiap kali web crawler mengunjungi sebuah website, maka crawler juga mendata semua link yang ada di halaman yang dikunjungi untuk kemudian dikunjungi lagi satu persatu [1].

PDC Media Group memiliki system crawler produk *marketplace*. system crawler ini bertugas untuk memgecek dan membuat metrik perkembangan penjualan dari setiap produk yang telah memiliki penjualan. pada suatu marketplace dilakukan crawling dan mencatat produk dan dimasukkan ke dalam suatu database. system crawler melakukan crawling pada produk yang sudah ada dalam setiap database setiap harinya. system crawler kemudian mencatat dan membuat metrik perubahan penjualan produk setiap hari

Pertumbuhan dan jumlah data menjadi masalah bagi system crawler. PDC Media Group sudah memiliki sekitar 4.500.000 data produk dari online marketplace dengan menyimpan data perubahan transaksi dan order selama 30 hari terakhir. Data produk yang dimiliki terus tumbuh sekitar 10.000 data baru per minggu seiring crawler menemukan produk produk baru. Hal tersebut membuat system crawler yang sudah ada tidak dapat mengecek semua perubahan transaksi dan order terhadap data produk setiap harinya.

Pada jurnal "Parallel computational workflows in Python" [2], dijelaskan infrastruktur system / workflow program yang memiliki skalabilitas lebih baik. infrastruktur yang digunakan adalah *distributed task* dengan menggunakan bahasa Python. Infrastruktur system secara umum dibagi menjadi 3 bagian, yaitu *unit publisher task* sebagai *generator task*, *message broker* sebagai

koordinasi dan pusat *endpoint* pertukaran data, dan *node worker* yang bertugas untuk menyelesaikan task yang sudah di-*generate*. *Worker node* dalam infrastruktur ini merupakan kumpulan server yang saling terkoneksi dengan message broker. infrastruktur system ini menawarkan skalabilitas yang lebih baik tidak hanya *vertical scaling* tapi juga dapat dilakukan *horizontal scaling*. Horizontal scaling pada infrastruktur ini dapat dilakukan secara tak terbatas dengan menambahkan worker node sebagai penyelesai task. Dengan demikian ketika data/task semakin banyak, waktu eksekusi dapat diatasi dengan cukup menambahkan worker node.

Pada journal "Supporting task parallelism in Python" [3] arsitektur sistem yang serupa juga dipakai dalam memproses *general task*. Untuk menyelesaikan perhitungan data matematis yang membutuhkan resource besar data dipecah menjadi beberapa task dan kemudian masing-masing diselesaikan oleh worker node yang ada.

Distributed task dipilih untuk solusi dari masalah karena proses crawling data bisa didistribusikan pada worker node sehingga memungkinkan waktu eksekusi yang singkat karena proses terjadi pararel di masing masing worker node. Skalabilitias *hardware* dari sistem yang menggunakan distributed task tidak bergantung pada vendor atau penyedia jasa layanan server. Sebagai contoh jika system membutuhkan sumber daya 4 vCPU (Virtual Central Processing Unit) dan 8 GB RAM, sementara penyedia layanan server memberikan layanan server paling tinggi dengan spesifikasi 2 vCPU dan 4 GB RAM. Sistem masih bisa scaling dengan 2 worker node 2 vCPU dan 4 GB RAM.

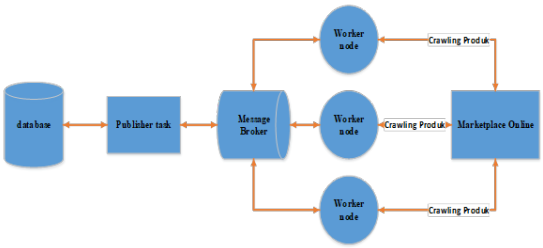
Peran Distributed Task pada system crawler agar system memiliki fleskibilitas performa yang dapat mengikuti jumlah data yang ada. Ketika data terus tumbuh, performa system crawler dapat terus juga ditingkatkan. Dibandingkan tanpa menggunakan distributed task, peningkatan performa hanya dapat dicapai dengan melakukan upgrade spesifikasi server. Sedangkan dengan menggunakan distributed task peningkatan performa dapat dicapai baik dari sisi menambah jumlah server atau melakukan upgrade spesifikasi server. Dimana hal tersebut lebih fleksibel karena system crawler bukan hanya scaling resource secara vertikal tapi juga horizontal.

II. METODE DAN ANALISIS

Dalam pembuatan sistem ini penulis menggunakan metode pengembangan sistem yaitu metode Waterfall menurut referensi Sommerville, yaitu metode yang menggambarkan proses software development dalam aliran berurutan. Model Waterfall yaitu suatu metodologi pengembangan perangkat lunak yang mengusulkan pendekatan kepada perangkat lunak. Sistematik dan sekuensial yang mulai pada tingkat kemajuan sistem pada seluruh analisis, desain, kode, pengujian dan pemeliharaan.

A. Analisis System

Desain system crawler dirancang dengan menggunakan distributed task. Proses tugas yang paling penting dalam system crawler yaitu pada proses crawling dan parsing data di distribusikan dan di eksekusi di node worker. System crawler yang dirancang menggunakan distributed task dibagi menjadi 3 komponen seperti blok diagram pada Gambar 1.



Gambar 1. Blok Diagram System Crawler

Publish task berfungsi sebagai unit komponen yang memproduksi task. Task sendiri merupa data object yang dibuat oleh publisher task berisi data url yang akan di crawling. Publisher task mengambil data url dari api database yang disediakan. Task tersebut kemudian di serialisasi kemudian dikirim ke message broker. Publisher task juga menunggu hasil task yang telah dieksekusi oleh worker node yang dikirim oleh message broker dari Worker Node untuk dimasukkan kembali ke database.

Message broker digunakan sebagai media komunikasi dan bertukar pesan antara worker node dan publisher task dalam crawler system. Message broker akan menerima task yang dikirim oleh publisher task untuk di distribusikan ke sejumlah worker node yang ada. Dalam perancangan ini peneliti menggunakan Rabbitmq sebagai message broker.

Worker Node adalah unit komponen yang melakukan eksekusi dari task yang telah diberikan oleh message broker. Jumlah Worker Node dalam suatu sistem bisa lebih dari satu. Worker Node terus menunggu task dari message broker. Ketika worker node mendapat task, worker node melakukan eksekusi dari task yang telah didapat. Hasil task yang telah selesai di eksekusi dikirim kembali ke publisher task melalui message broker untuk diolah kembali oleh publisher task.

B. Flowchart System

Flowchart system adalah *flowchart* yang menampilkan tahapan atau proses kerja yang sedang berlangsung di dalam sistem secara menyeluruh. Selain itu *flowchart* sistem juga menguraikan urutan dari setiap prosedur yang ada di dalam sistem. Proses *distributed task* dibagi menjadi 2 flowchart yaitu *flowchart publisher task* dan *flowchart worker node* yang menjelaskan proses dari kedua komponen yaitu komponen *publisher task* dan komponen *worker node*.

Gambar 2. Flowchart Publisher Task

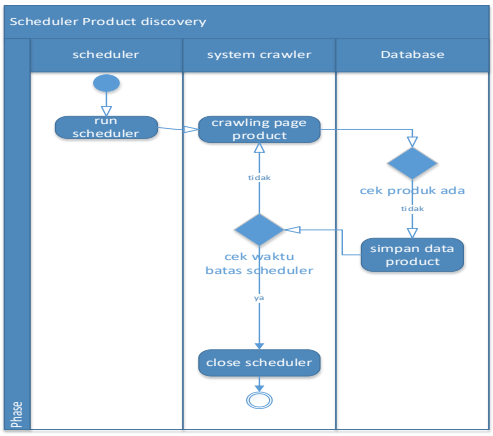
Pada gambar 2, proses *publisher task* dimulai dengan mengambil data *url* yang belum dilakukan *crawling*. Setelah data diambil kemudian *publisher task* membuat *task object* dari *url* yang didapat. Task yang telah dibuat kemudian dikirimkan ke *message broker* untuk dieksekusi oleh *worker node*. Publisher *task* akan menunggu hasil *task* yang sudah selesai dari *message broker*. Hasil dari *task* adalah merupakan data hasil *crawling* yang kemudian disimpan ke dalam *database*.

Gambar 3. Flowchart *worker node*

Gambar 3 adalah *flowchart worker node* yang bertugas untuk melakukan eksekusi *crawling task* yang sudah dikirim oleh *publisher task*. Proses dimulai dengan *worker node* melakukan koneksi ke *message broker*. Worker node akan mengecek dan menunggu untuk mendapatkan *task* yang dikirim dari *publisher task*. Ketika *worker node* mendapatkan *task*, *task* akan dieksekusi oleh *worker node* dan dilakukan *crawling* dari *url* yang diberikan *task*. Hasil *crawling* dari *task* yang sudah selesai kemudian dikirimkan kembali ke *message broker* dan akan diolah lebih lanjut oleh *publisher task*.

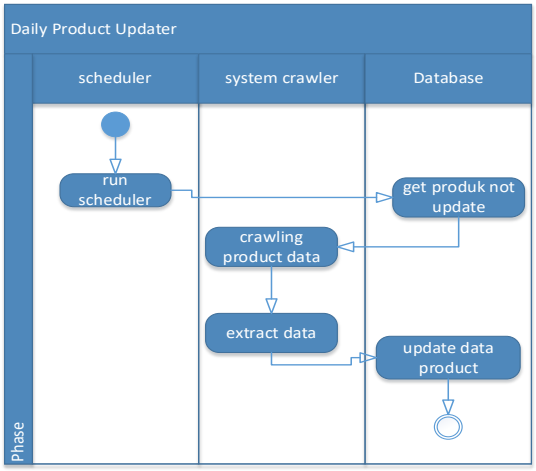
C. *Activity Diagram*

Activity Diagram merupakan rancangan aliran aktivitas atau aliran kerja dalam sebuah sistem yang akan dijalankan. *Activity Diagram* juga digunakan untuk mendefinisikan atau mengelompokan aluran tampilan dari sistem. Pertama adalah *activity diagram Product Discovery*. yang berfungsi untuk menemukan produk baru dalam *system crawler*.



Gambar 4. Activity *Product Discovery*

Pada gambar 4 adalah activity dari *product discovery*. *Product Discovery* dijalankan oleh *scheduler* seperti *cronjob*. Alur pertama *system* akan melakukan *crawling* produk. Kemudian *system* melakukan cek produk di *database*. Jika produk sudah ada maka produk akan diabaikan. Sebaliknya jika produk tidak ada maka data produk akan ditambahkan ke dalam *database*. Selain itu sistem juga akan melakukan cek batas waktu *scheduler*. Jika *system* berjalan melewati batas waktu *scheduler* maka *product discovery* akan dimatikan.

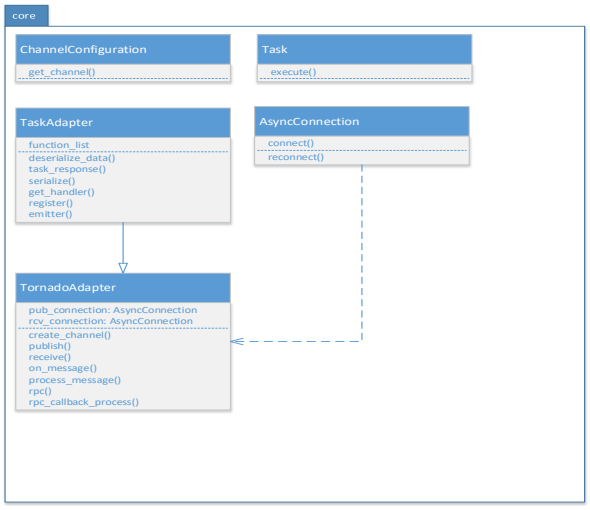


Gambar 5. Activity Daily Product Updater

Pada gambar 5 adalah *activity diagram* dari *daily product updater* sebagai komponen yang melakukan pencatatan perubahan statistic produk setiap harinya. Daily product updater dijalankan oleh *scheduler* seperti *cronjob*. System akan mengambil semua data produk dari *database*. Data produk dilakukan *crawling* kembali oleh *system*. Data baru dari setiap produk hasil dari *crawling* disimpan kembali di dalam *database*. Selain itu *system* juga akan melakukan cek batas waktu *scheduler*. Jika *system* berjalan melewati batas waktu *scheduler* maka *daily product updater* akan dimatikan.

D. Class Diagram

Dalam *system crawler* terdapat beberapa kelas penting yang mengatur dan mengelola *task* yang dieksekusi secara distribusi.

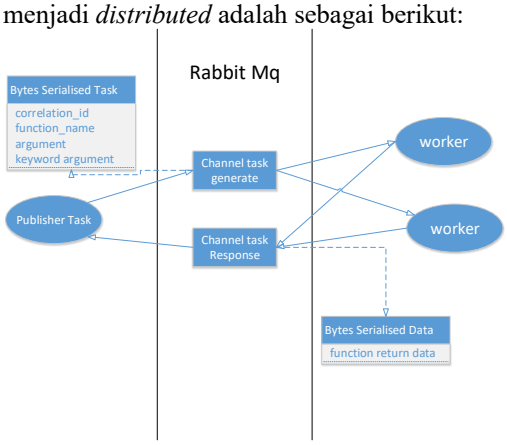


Gambar 6. Class diagram *distributed task*

Kelas *channel configuration* berfungsi sebagai tempat konfigurasi *channel* yang akan digunakan dalam berkomunikasi dengan *RabbitMQ*. Kelas *task adapter* berfungsi pengatur pertukaran data *task*, kebutuhan serialisasi data, deserialisasi data, pengirim dan *penerima task* dari *RabbitMQ*, dan juga sebagai *register* fungsi yang bisa dieksekusi secara *distributed*. Tornado Adapter merupakan *inheritance* dari *Task Adapter* yang berfungsi agar fungsi yang telah *diregister* di dalam *distributed task* dapat juga dijalankan secara *asynchronous*. Task merupakan kelas untuk *object* yang berisi fungsi *distributed* yang telah *diregister* melalui *task adapter*. Task ini yang akan diserialisasi atau deserialisasi dan dieksekusi di *worker*. Kelas *Async Connection* berfungsi untuk membuat koneksi *asynchronous* ke *RabbitMQ*

E. Desain Eksekusi Task Distributed

Pada *system crawler* lama eksekusi *crawling* hanya dapat dilakukan di local. Sedangan pada *system crawler* baru *task* bisa dieksekusi local maupun *distributed* ke *worker node* yang ada. Desain pola komunikasi agar bisa suatu *task* menjadi *distributed* adalah sebagai berikut:



Gambar 7. Pola Komunikasi Distributed

Gambar 7 adalah pola komunikasi *distributed*. Pertama eksekusi *task crawling* dilakukan di *worker node*. Di *crawler system* yang baru, argumen fungsi akan di-*generate* menjadi *task* dan di serialisasi menjadi byte. Task yang sudah di serialisasi dikirim ke *RabbitMQ* yang kemudian dibagikan ke *worker node*. Task dalam bentuk Bytes yang telah dikirim ke *worker node* di deserialisasi kembali agar dapat dibaca *worker*. kemudian *worker* akan melakukan eksekusi dari *task* telah di deserialisasi. Hasil eksekusi *worker node* dari *task* adalah berupa function return data. Function return data ini di serialisasi kembali dan dikirimkan ke *RabbitMQ* yang selanjutnya akan diterima oleh komponen yang melakukan *generate task* / *publisher task*.

III. IMPLEMENTASI

A. Implementasi Core Distributed System dalam system crawler

Core Distributed System dibuat menjadi *package* python terpisah agar mudah dalam implementasi lanjut di berbagai pecahan file kode. *Core Distributed System* meng *export* kelas yang berguna sebagai *decorator* fungsi. *Decorator* adalah mengubah suatu fungsi/object di python waktu

runtime. Fungsi yang didekorasi oleh kelas *decorator Core Distributed System* memiliki 2 *behaviour* yaitu dapat dieksekusi secara lokal biasa atau di eksukusi di dalam *worker* di beda mesin.

Terdapat dua kelas penting yang digunakan yaitu *TaskAdapter* dan *TornadoTaskAdapter*. Kelas *TaskAdapter* digunakan ketika program ingin menjalankan *distributed system* di dalam program nya. Sedangkan kelas *TornadoTaskAdapter* merupakan *inheritance* dari kelas *TaskAdapter* yang dapat di *wrapper* ke dalam *http service.*

```
# inisiasi Task adapter untuk koneksi ke rabbit mq
RABBIT_URI = "amqp://localhost:5672/"
adapter = TaskAdapter(rabbitmq_url=RABBIT_URI, configuration=configuration, io_loop=io_loop)


# adapter mendekorasi fungsi crawl_homepage
@adapter.register("fib_server_q", publish_exchange="test_rpc")
def crawl_homepage():
    req = requests.get('https://tokopedia.com')
    return req

# fungsi dieksekusi biasa
crawl_homepage()

# akan dieksekusi di worker node
crawl_homepage.execute()
```

Gambar 8. Implementasi Task Adapter

Gambar 8 adalah contoh implementasi *task adapter* agar suatu fungsi dapat dieksekusi secara *distributed task*. *Variable adapter* merupakan *object* inisiasi dari kelas bernama *TaskAdapter*. *Variable adapter* ini selanjutnya dapat digunakan bebas dalam mengubah / *decorator* fungsi biasa agar dapat dieksekusi di *worker node* atau *distributed system.* Cara me-*decorator* suatu fungsi adalah dengan menambahkan notasi @ di atas deklarasi fungsi. Seperti pada fungsi *crawl_homepage,* fungsi tersebut telah didekorasi oleh *variable* adapter sehingga Fungsi *crawl_homepage* dapat meiliki 2 *behaviour*. Ketika ingin mengeksekusi fungsi secara local / biasa, cukup dengan memanggil fungsi *crawl_homepage()*. Sedangkan ketika ingin mengeksekusi secara *distributed* maka bisa dijalankan *crawl_homepage.execute()*.

B. Implementasi Kode Crawler.

Implementasi kode *crawler* hampir sama dengan *core distributed.* Kode crawler ditaruh di package tersendiri agar mudah dimanipulasi lagi. *Kode crawler* berupa sekumpulan fungsi yang berinteraksi dengan API Tokopedia. Fungsi tersebut diantaranya generator *request* seperti *HTTP header, query grapqhql* tokopedia, fungsi parsing data dari data mentah tokopedia, dan fungsi *HTTP request* yang di pakai di worker dan publisher untuk dieksekusi secara *distributed*. Pada *package crawler* terdapat 3 komponen penting yaitu toko, *product*, dan discover. Komponen toko menyediakan fungsi untuk mengambil info toko dari API Tokopedia. Berikut adalah contoh ketika komponen toko digunakan untuk mengambil informasi dari suatu toko yang ada di Tokopedia.

```
F:\python3\stattest>python -m src.crawler.toko
{'city_name': 'Kota Bandung',
 'district_name': 'Sumur bandung',
 'jum_product': 316,
 'kep_bad': 0,
 'kep_good': 20,
 'kep_neutral': 0,
 'last_updated': datetime.datetime(2021, 8, 11, 12, 39, 3, 284772),
 'location': 'Kota Bandung',
 'percent_tx': 99.051,
 'rating': 4.9,
 'review': 434,
 'score': 1389,
 'score_map': 8,
 'shopid': 4152404,
 'sold': 664,
 'tx': 626,
 'url': 'https://tokopedia.com/diskonlaptop',
 'username': 'diskonlaptop'}
```

Gambar 9. Hasil Komponen Toko

Gambar 9 merupakan contoh hasil data *crawling* toko oleh komponen toko dari *crawler system.* Data dari toko yang diambil adalah sebagai pelengkap data produk. Data toko yang diambil adalah informasi dasar toko seperti lokasi, jumlah produk, *username*, *id*, *url,* dan statistik dasar yang sudah disediakan Tokopedia diantaranya *rating, review,* tansaksi atau penjualan.

Komponen kedua yang ada di *crawler* adalah *product.* Komponen *product* menyediakan fungsi untuk mengambil informasi produk yang ada di tokopedia. Komponen *product* ini akan digunakan di *worker* untuk melakukan *crawling.* Komponen *product* inilah yang akan mengambil data penting dari produk melalui Tokopedia.

```
F:\python3\stattest>python -m src.crawler.product
{'cat_id_1': '57',
 'cat_id_2': '3133',
 'cat_id_3': '3160',
 'description': 'barang bagus.\ntersedia warna \nnatural dan hitam ',
 'diskusi': 4,
 'itemid': 1623875975,
 'last_updated': datetime.datetime(2021, 8, 11, 13, 42, 7, 215013),
 'name': 'Gitar Akustik Yamaha apx 500ii termurah',
 'pictures': ['https://ecs7.tokopedia.net/img/cache/300/VqbcmM/2021/3/15/db7cc118-7a14-47a0-b7d4-b009465dfa07.jpg',
              'https://ecs7.tokopedia.net/img/cache/300/VqbcmM/2021/3/15/e5a651e8-0978-4477-aa39-82397980ed11.jpg',
              'https://ecs7.tokopedia.net/img/cache/300/VqbcmM/2021/3/15/7f3c9a40-5fdc-4e0b-ba59-2e57b631d108.jpg',
              'https://ecs7.tokopedia.net/img/cache/300/VqbcmM/2021/3/15/896bac0b-06fd-4daa-b240-1d5f7d233102.jpg'],
 'price': 305000,
 'rating': 0,
 'rating_count': 0,
 'review': 10,
 'shopid': 5249814,
 'sold': 10,
 'stock': 8,
 'tx': 18,
 'url': 'https://www.tokopedia.com/leluto/gitar-akustik-yamaha-apx-500ii-termurah',
 'view': 12085,
 'weight': 7000}

F:\python3\stattest>
```

Gambar 10. Hasil Komponen Product

Gambar 10 adalah contoh hasil dari pengambilan data *yang* dilakukan oleh komponen *product.* Data yang diambil dari komponen *product* adalah data produk seperti info kategori, deskripsi, *id* produk, jumlah diskusi, info transaksi dan penjualan, stok, *url*, view, gambar, dan berat.

Komponen yang ketiga adalah *discover.* Komponen ini *berfungsi* mencari produk yang ada di tokopedia. Komponen ini mencari produk berdasarkan halaman kategori dan komponen *discover* ini mengunjungi setiap *page* dari web dan mengambil info item produk.

Gambar 11. Hasil Komponen Discover

Gambar 11 menunjukkan hasil dari komponen *Discover* dalam menemukan data produk baru. Data yang didapat adalah berupa *id* produk, *url* produk, dan info toko. Data item produk yang diperoleh dari komponen *discover* masih parsial. Sehingga data item produk ini akan di *request* kembali oleh komponen *product* agar mendapat data yang lengkap.

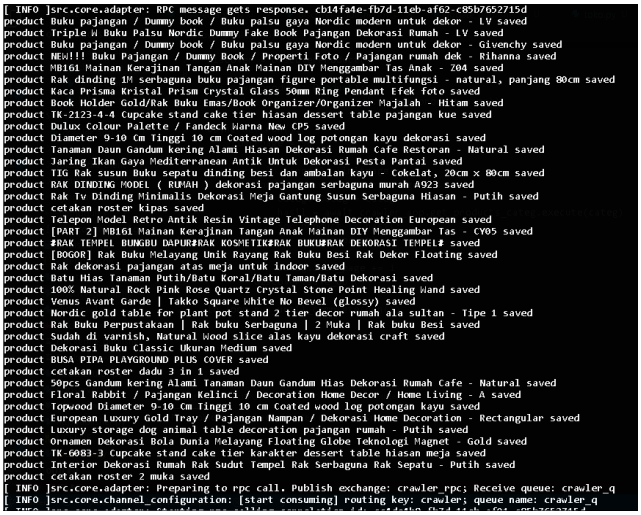
C. *Implementasi Publisher*

Script publisher mengimport fungsi *crawler* dan mengubahnya menjadi distributed. Ada 2 tipe publisher yang ada. Pertama *daily product updater* yang berfungsi mengupdate data produk setiap harinya.

Gambar 12. Tampilan Daily Product Updater

Gambar 12 adalah merupakan tampilan *daily product updater. Daily product updater* ini bertugas melakukan *update* produk yang telah ada di dalam *database. Daily product updater* pada *crawler system* akan dieksekusi dan dijadwal harian untuk mencatat perubahan informasi dari suatu produk.

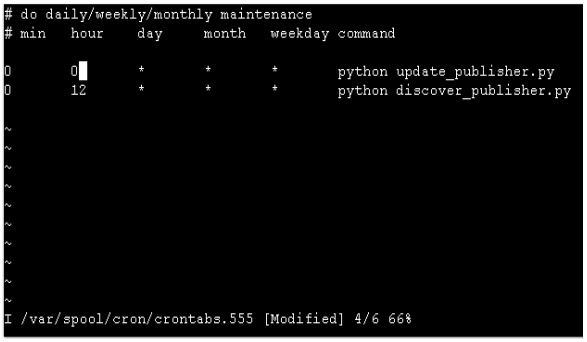
Publisher yang kedua adalah *product_discovery. product_discovery* pada *crawler system* berfokus pada mencari produk baru. *Product_discovery* melakukan *crawling* setiap *url* dan mengecek dari setiap url produk yang ditemukan merupakan produk baru. Jika produk baru ditemukan, data produk akan dimasukkan ke dalam *database* yang kemudian akan dicek kembali oleh *Daily Product Updater.*



Gambar 13. Tampilan Log Discover Publisher

Gambar 13 adalah contoh *running execution* dari *product discovery.* Ketika *product discovery menemukan* produk baru, maka aka nada *log info* dengan nama produk yang telah disimpan ke dalam database.

Di server, *publisher* dijalankan setiap hari. *Cronjob* diperlukan untuk menjalankan otomatis tugas *publisher.* Untuk server yang menggunakan *os Ubuntu* menginstall *cronjob* dapat menggunakan *apt* dengan perintah *sudo apt-get install cron.* Untuk membuat konfigurasi *cron* dapat dilakukan dengan perintah *crontab –e.*



Gambar 14. Tampilan Konfigurasi cronjob

Pada Gambar 14 terdapat 2 baris konfigurasi untuk *publisher* yang ada di *cron.* Konfigurasi *pertama* adalah *cron* untuk menjalankan *daily product updater.* Notasi *0 0 * * *** berarti bahwa perintah *python daily product*

updater.py dijalankan pada menit 0 jam 0 setiap harinya. Konfigurasi kedua adalah *cron* untuk menjalankan *product_discovery*. Notasi *0 12 * * *** berarti bahwa perintah *python product_discovery.py* dijalankan pada menit 0 jam 12 setiap harinya.

D. Implementasi Worker

Worker meng *import* semua fungsi *crawler* dan mengubahnya menjadi *distributed*. Worker hanya melakukan listening ke *RabbitMQ* dan mengeksekusi *task* yang telah dikirimkan. Pada *production environtment, worker* dapat berjalan lebih dari satu sesuai dengan kebutuhan *workload.*

Gambar 15. Tampilan Log Worker

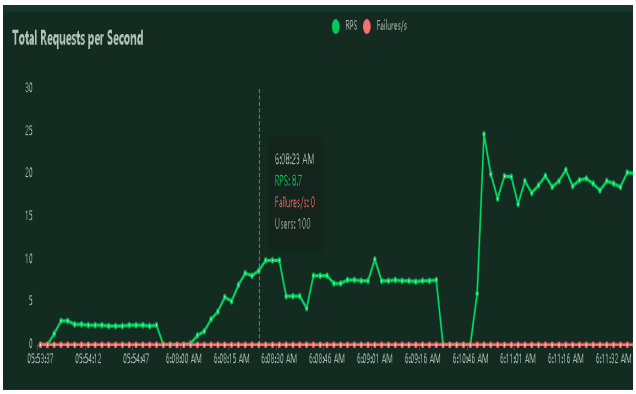
Gambar 15 adalah tampilan *log worker* yang sudah *running. Worker* sudah melakukan *listening* ke *RabbitMQ* dengan ditandai *log creating connection to RabbitMQ* dan *start consuming to routing key.* Untuk *log worker server* ketika sudah selesai dalam melakukan eksekusi *task* adalah *message has been processed successfully.*

IV. HASIL DAN PEMBAHASAN

Hasil dan pembahasan meliputi *load testing crawler system* yang telah dibuat sesuai dengan desain dan juga testing *crawler system* di *production server*. Hasil *load testing* adalah berdasarkan pada 3 skenario dengan perbedaan jumlah *worker node.* 3 skenario tersebut adalah testing *system crawler* dengan 1 *worker*. 2 *worker* dan 3 *worker*. Pengujian menggunakan *docker* sehingga resource server sudah terisolasi dengan baik dan juga untuk pengukuran *RPS* dan *Response Time* menggunakan framework testing yaitu *locust*.

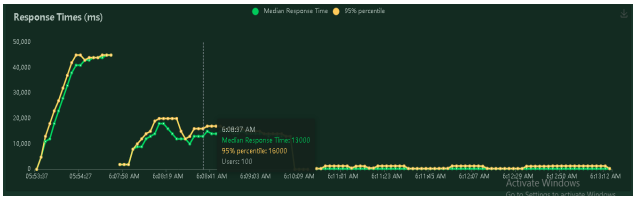
A. Skenario Testing dengan 1 Worker

Untuk testing pertama, *worker* yang dijalankan berjumlah 1 *worker*. *resource server* yang digunakan per *worker* adalah 1 *cpu core* dan dan 512 mb memori.



Gambar 17. Hasil RPS dengan 1 *worker*

Gambar 17 adalah hasil RPS dari *task crawling* yang berhasil dieksekusi oleh system *crawler* dengan 1 *worker*. *Task* yang berhasil dieksekusi oleh *system crawler* berada di kisaran 5 – 25 request per detik.



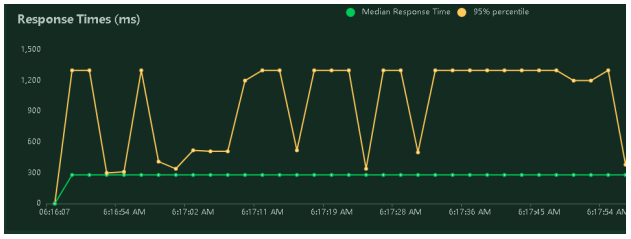
Gambar 18. Hasil Response Time dengan 1 *worker*

Gambar 18 adalah hasil *Response Time*. *Response time* adalah waktu tunggu yang diperlukan oleh *crawler* untuk menyelesaikan suatu *task*.

B. Skenario Testing dengan 2 Worker.

Testing kedua adalah mencoba mengukur *system crawler* dengan 2 *worker*. Karena *environtment testing* menggunakan *docker*, untuk melipat gandakan *worker* dapat dilakukan dengan perintah “docker compose scale *worker*=2”. Jadi sekarang *system crawler* memiliki 2 *worker* dengan masing masing 1 cpu dan 512 mb memori.

Gambar 19 menunjukkan report RPS pada locust setelah *scaling worker* menjadi 2 menjadi naik. RPS pada *crawler* berkisar di 40 *request per* second.

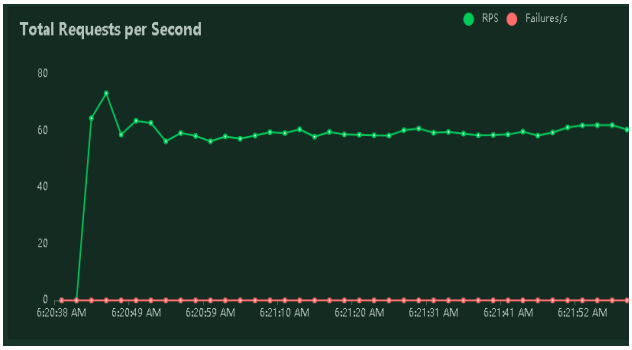


Gambar 19. Hasil Response Time dengan 1 *worker*

Pada gambar tampak hasil *Response Time*, waktu yang dimiliki *crawler system* berkisar diantara 300 sampai 1.200 *milisecond*.

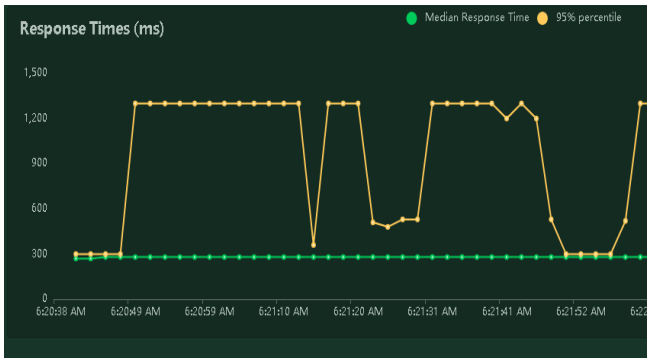
C. *Skenario Testing dengan 3 Worker.*

Testing *system crawler* yang ketiga adalah dengan mencobanya dengan 3 *worker*. Untuk *scaling worker* dengan perintah yang sama dengan yang dilakukan sebelumnya yaitu "docker compose scale *worker*=3". Sekarang *system crawler* memiliki 2 *worker* dengan masing masing 1 cpu dan 512 mb memori.



Gambar 20. Hasil RPS dengan 2 *worker*

Gambar 20 menunjukkan hasil RPS pada locust setelah *scaling worker* menjadi 3 menjadi naik lebih tinggi dari yang sebelumnya 2 *worker*. RPS pada *system crawler* berkisar di 60 *request per second*.



Gambar 21. Hasil Response Time dengan 3 *worker*

Gambar 21 menunjukkan hasil *Response Time*, waktu yang dimiliki *system crawler* berkisar diantara 300 sampai 1.200 ms hampir sama dengan 2 *worker*.

D. *Hasil Testing Dari Ketiga Skenario.*

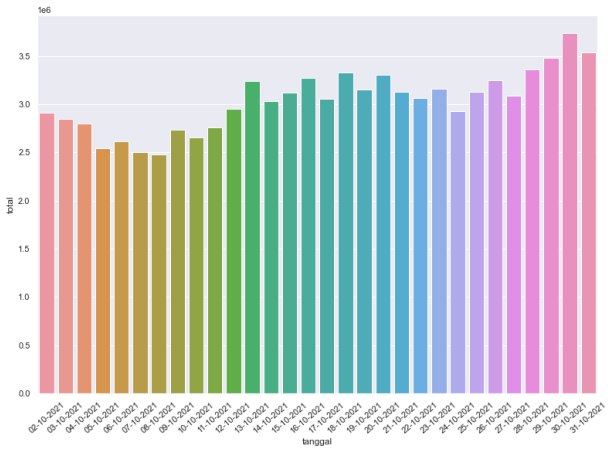
Dari hasil testing *system crawler* dengan ketiga skenario didapat data sebagai tabel berikut:

Skenario	RPS (Request Per Second)	Response Time
1 *worker*	19.3 request per second	332 ms
2 *worker*	41.4 request per second	328 ms
3 *worker*	60 request per second	331 ms

Pada tabel hasil testing dapat diperoleh beberapa informasi dari percobaan ketiga skenario. Setiap penambahan *worker* pada *system crawler*, RPS (Request Per Second) pada *system crawler* juga mengalami peningkatan kurang lebih 20 RPS. *Response Time* pada setiap skenario tidak mengalami perubahan yang signifikan.

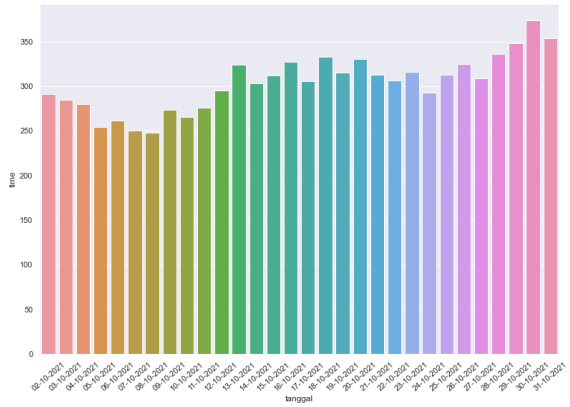
E. *Hasil Pengujian Penggunaan Crawler pada Production Server*

System crawler juga diuji di *production server*. Pengujian pengguna berfungsi untuk melihat hasil kinerja dari *system crawler* yang sebenarnya di server asli dengan melihat seberapa lama *system* melakukan *crawling* dengan data yang sudah ada di PDC Media Group. Spesifikasi dari *production server* adalah 1 vcpu dan 1 gb untuk masing masing *worker* dan *tasker* dengan jumlah *worker* yang ada yaitu 10 *worker*.



Gambar 22. Grafik jumlah produk 30 hari terakhir

Gambar 22 merupakan jumlah produk yang berhasil di *crawling* menggunakan *system*. Rentang data yang ada berkisar diantara 2.5 juta sampai 4 juta. Data tersebut mengalami penambahan atau pengurangan dalam setiap hari karena ada beberapa data produk yang dihapus dari web atau produk baru yang terupload di web.



Gambar 23. Grafik waktu Crawling

Gambar 23 adalah statistik waktu yang diperlukan unntuk melakukan *crawling data* yang telah dijelaskan pada gambar 4.30. Dengan jumlah dan pertumbuhan data pada gambar 4.30, penulis juga mencatat waktu yang diperlukan *system crawling* untuk melakukan *crawling* update data yang sudah ada. Waktu yang diperlukan *system crawling* adalah berkisar diantara 250 menit sampai 400 menit. Waktu yang dibutuhkan *system* untuk *crawling* berbeda karena dipengaruhi oleh jumlah data yang ada pada gambar grafik sebelumnya.

V. KESIMPULAN

Penerapan distributed task pada system crawler dapat dicapai dengan membuat system crawler menjadi 3 komponen yaitu publisher, message broker dan worker. Penyesuaian performa dan seberapa banyak system crawler dapat melakukan crawling per detik terhadap jumlah yang ada juga bisa melakukan penambahan worker atau upgrade spesifikasi worker. Worker yang ada tidak harus ada dalam satu mesin yang sama dengan spesifikasi besar. Tetapi juga dapat dipisah ke mesin server yg lebih kecil tapi banyak. System Crawler diuji dengan menggunakan framework testing yaitu locust.

Pengujian dilakukan dengan 3 skenario dengan membedakan jumlah worker yang ada di system. Dari pengujian system yang dilakukan jumlah request per second pada system meningkat seiring dengan penambahan worker. Untuk skenario 1 worker, hasil yang didapat adalah 19.3 request per second dan 332 ms untuk response time yang dibutuhkan. Untuk skenario 2 worker, hasil yang didapat adalah 41.4 request per second dan 328 ms untuk response time yang dibutuhkan. Sedangkan untuk skenario 3 worker, hasil yang didapat adalah 60 request per second dan 331 ms untuk response time yang dibutuhkan. Sehingga system crawler dapat melakukan scaling horizontal dengan lebih baik.

DAFTAR PUSTAKA

[1] V. Wjaya, “Pentingnya web crawling Sebagai Cara Pengumpulan Data Di era Big Data,” Pentingnya Web Crawling sebagai Cara Pengumpulan Data di Era Big Data, 07-Mar-2020. [Online]. Available: https://www.teknologi-bigdata.com/2016/07/web-crawling-di-era-big-data.html. [Accessed: 23-Jan-2022].

[2] Enric Tejedor, " Parallel computational workflows in Python," The International Journal of HighPerformance Computing Applications, pp. 1-17, 2015.

[3] A. P. E. Hadjidoukas∗, "Supporting task parallelism in Python," IBM Research, Zurich, Switzerland, pp. 1-6, 2019.