bulk-crawling

1. Introduction

Le crawling de sites est un domaine souvent lié aux problématiques de type Search Engine Optimization. Le principe général est de parcourir de manière récursive et dynamique toutes les ressources composants un ou plusieurs sites web. La pertinence des informations remontées provient du filtrage des ressources associées: métadonnées, balises, images, styles, media…

L'exercice proposé n'est pas focalisé sur l'analyse fine de la structure de ces sites, mais sur la récupération en masse du contenu associé, via deux approches architecturales différentes, proposées sur Google Cloud Platform.

La première consiste à utiliser le système de lambdas GCP, les cloud functions, sortes de process éphémères entièrement stateless, faciles à développer, peu gourmandes et centrées sur la scalabilité.

La seconde demande d'instancier un cluster avec l'orchestrateur de containers Kubernetes, potentiellement plus complexe à mettre en place, mais promettant un contrôle plus fin des ressources.

2. Analyse / Diagramme des classes

Nous disposons d'un fichier contenant 146703 lignes d'URL.

Comme l'objectif de l'exercice est centré sur la récupération en masse du contenu de ces sites web, et non sur l'analyse du contenu de ceux-ci (probablement délégué à un autre micro-service intervenant plus loin dans le pipeline), nous nous contenterons de crawler les URL fournies de manière simple, sans analyse récursive de leurs sitemap, comme pourrait le faire un véritable robot.

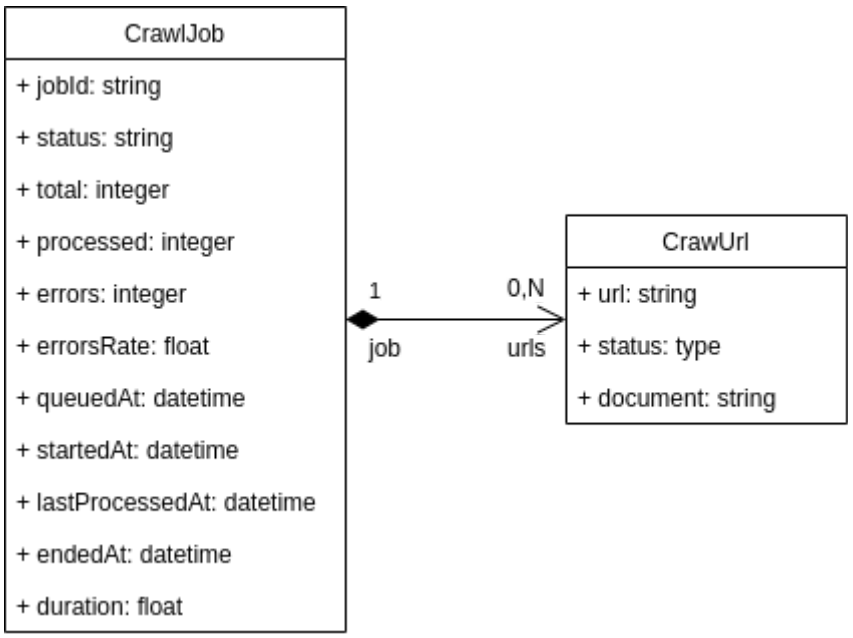
Certains sites ne seront pas accessibles, il faut donc gérer les erreurs associées, voire mettre en place des mécanismes de retry le cas échéant.

Etant donné le volume de données, la récupération doit se faire en mode asynchrone, l'utilisation de mécanismes de jobs / statuses semble toute indiquée.

La problématique fait émerger la structure de services génériques suivants:

- un service d'orchestration: chargé de recevoir la liste des URL à crawler, de les valider, et des les envoyer en arrière-plan aux services de crawling
- un bus de données: chargé de gérer la file des URLs à crawler par les services dédiés
- plusieurs services de crawling hautement scalables: chargés de traiter une ou de nombreuses URLs, puis d'envoyer les résultats au système de stockage des résultats

- un système de stockage: chargé de stocker le résultats des crawlings afin de les exposer à un service d'analyse plus loin dans le pipeline
- un stockage tampon de métriques concernant l'état du batch en cours

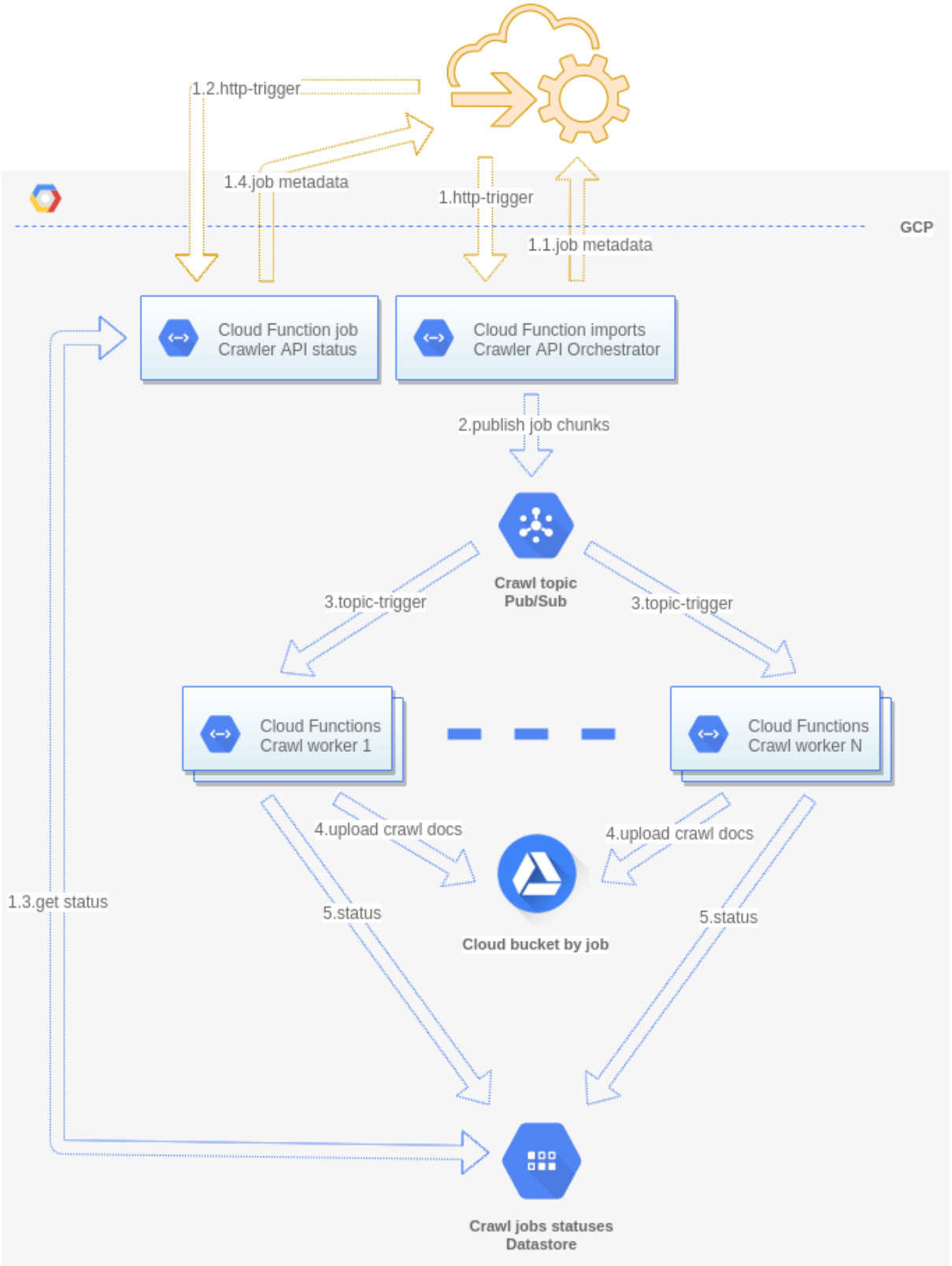


Description des entités:

- **CrawlJob**: l'envoi d'une payload de crawling nécessite une entité chargée de représenter l'état des processus en cours (status: pending, running, complete). De nombreuses métriques lui sont associées: nombre d'éléments à crawl, nombre de sites traités, erreurs + taux, timestamps de démarrage, de fin et durée. Il est associé à la liste des URLs à crawler
- **CrawlUrl**: l'entité la plus granulaire de ce projet, une URL à crawler, le status associé ainsi que le document-résultat

L'objectif est centré sur la performance de récupération, nous pouvons partir du principe que les sites inaccessibles ne doivent pas entraver le processus ou monopoliser des ressources trop longtemps dans la file.
Un paramètre intéressant pourrait être ajouté au niveau du service d'orchestration des jobs: la possibilité de découper le volume d'URLs à traiter en lots de batchs, et donc de laisser un même service de crawling gérer plusieurs sites à la fois. L'idée sous-jacente est de déterminer le seuil à partir duquel il est préférable d'augmenter le nombre de crawlings effectués en parallèle (et donc instancier de nouvelles machines, avec la latence induite) ou de ré-utiliser les ressources déjà allouées.

3. Architecture GCP Cloud Functions



Description (top-bottom):

- **Crawler API Orchestrator (imports)**: cloud function qui reçoit la payload d'URLs à crawler, la découpe en chunks envoyés dans le PubSub topic de crawling sous forme de messages contenant donc une ou plusieurs URLs à traiter (paramètre *"batchSize"*), et retourne une payload dee job associé
- **Crawler API status (job)**: cloud function qui permet de remonter les métriques liées au *"jobId"* passé en paramètre
- **Crawl PubSub (topic)**: gère la file des crawlings associés aux jobs qui vont provoquer l'instanciation des cloud functions de traitement
- **Crawl worker (crawl)**: background cloud function hautement scalable déclenchées par un événement de type message sur le topic de crawling. Elle gère le crawling des URLs contenues dans le message, la mise à jour des métriques associées au job (nombre d'éléments traités, erreur..), ainsi que l'envoi des documents sur le bucket de stockage
- **Cloud Bucket by job**: gère les documents générés par les services de crawling, organisés par jobs. Chaque document est nommé par suggification de l'URL associée
- **Cloud jobs statuses Datastore**: un store de données NoSQL qui gère les données liées aux jobs et leurs métriques

4. Architecture GCP Kubernetes

// todo

5. Tests et benchmarks

// todo

Afin de garantir la pertinence des benchmarks (cache, erreurs réseau…), ceux-ci sont calqués sur la moyenne de 3 exécutions successives du même scénarios.

6. Conclusion

// todo

J'avoue ne pas être un expert du SEO, ni du crawling en général. Au fil des missions / postes, j'ai eu l'occasion de mettre en place des crawlers de dashboards de statistiques, en utilisant des Frameworks comme Pupetter, PhantomJS, Selenium Web Driver.

Je n'avais jamais abordé les problématiques de scalabilité sur du crawling massif de sites. Cet exercice a été l'occasion pour moi de me replonger dans la plateforme GCP. Même si je salue la simplicité de mise en place des clouds functions, mon avis reste que les outils de développement en local avant soumission sur le cloud ne sont pas encore totalement aboutis (PubSub, topics, souscriptions), obligeant à patienter longuement pendant les deploy successifs.