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'URLs. Ceci représente 12mb de fichier décompressé, les Cloud Functions étant limitées à 10mb en terme de taille de requête entrante, nous allons tronquer le fichier à exactement 100500 lignes d'URLs. Comme amélioration, afin de ne pas limiter le nombre d'URLs, nous pourrions stocker le fichier sur un bucket, et laisser l'orchestrateur le récupérer. 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.

En prenant en compte 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:

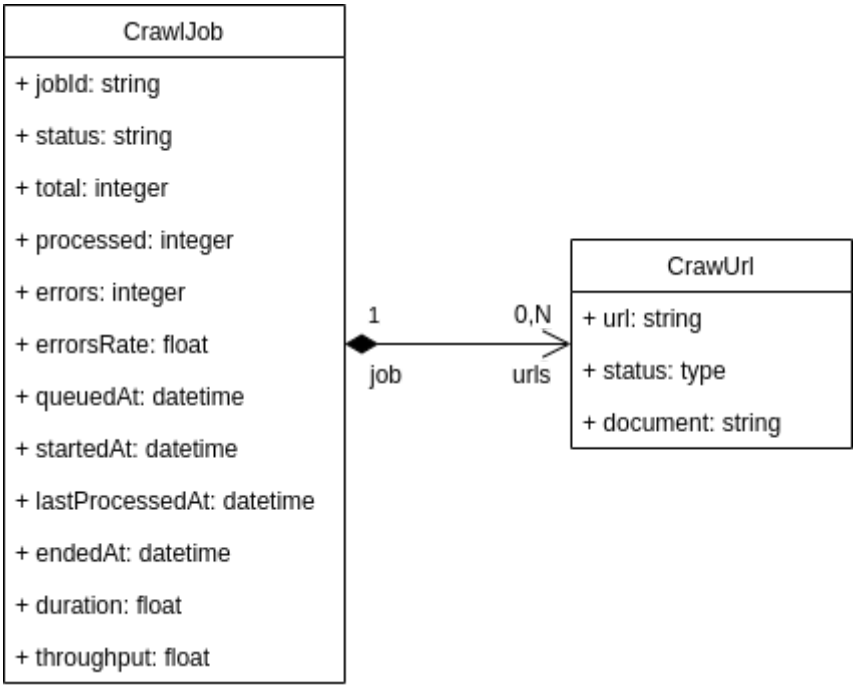
- un service d'orchestration: chargé de recevoir la liste des URL à crawler, de les valider, et des les envoyer en arrière-plan aux services de crawling

- 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



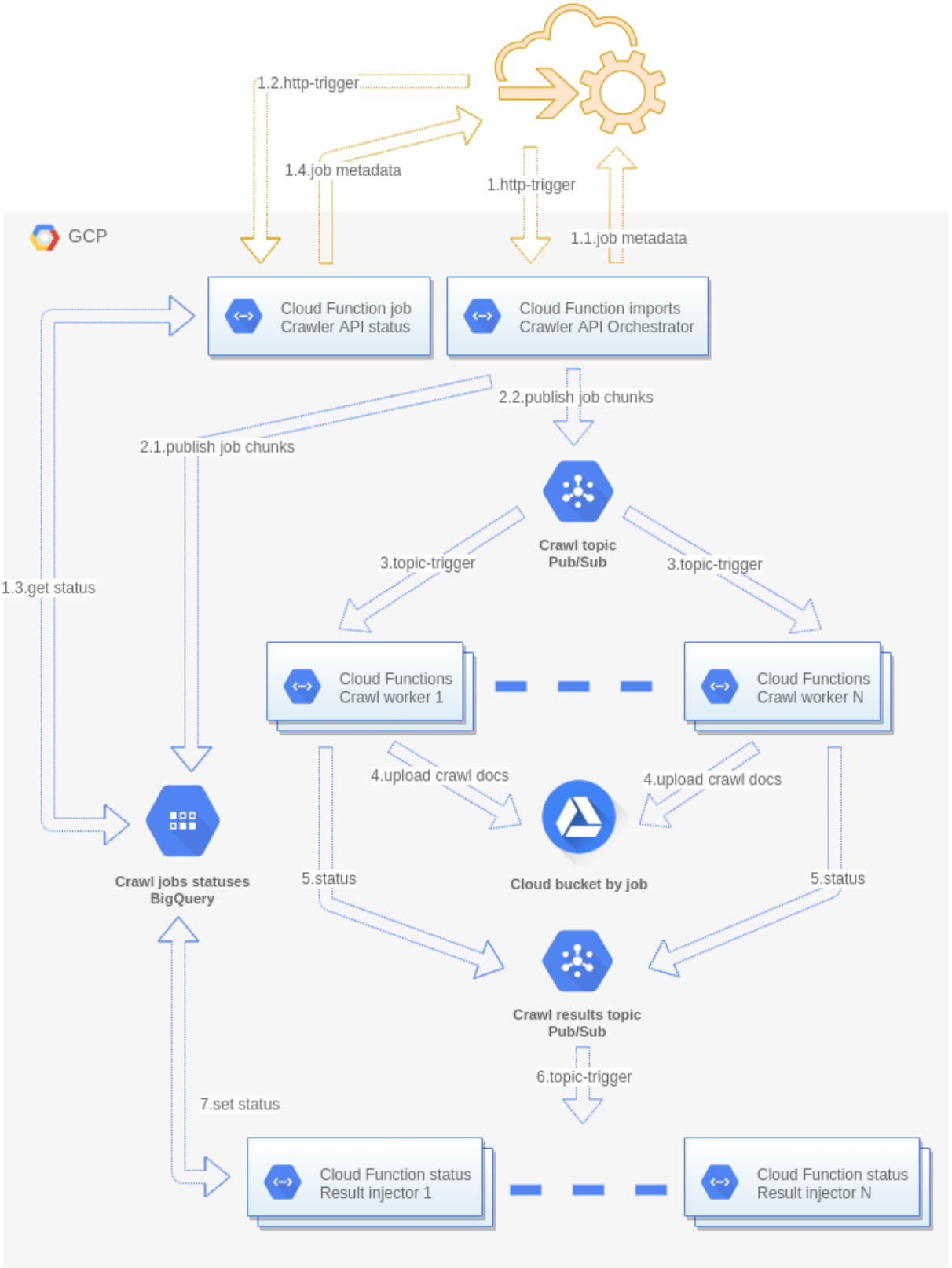
Description des entités:

- **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):

- **Cloud function Crawl 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. On peut agir sur le nombre de tentatives de retry par site en cas d'erreur de crawling avec le paramètre *"retries"*, ainsi que sur le timeout de communication avec les sites via le paramètre *"timeout"*

- **Cloud function Crawl 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 suivantes

- **Cloud function Crawl worker (crawl)**: background cloud function scalable déclenchées par un événement de type message sur le topic de crawling. Elle gère le crawling des URLs contenues dans le message, elle publie le résultat du batch de crawling (nombre d'éléments traités, erreur..), et envoie les documents sur le bucket de stockage

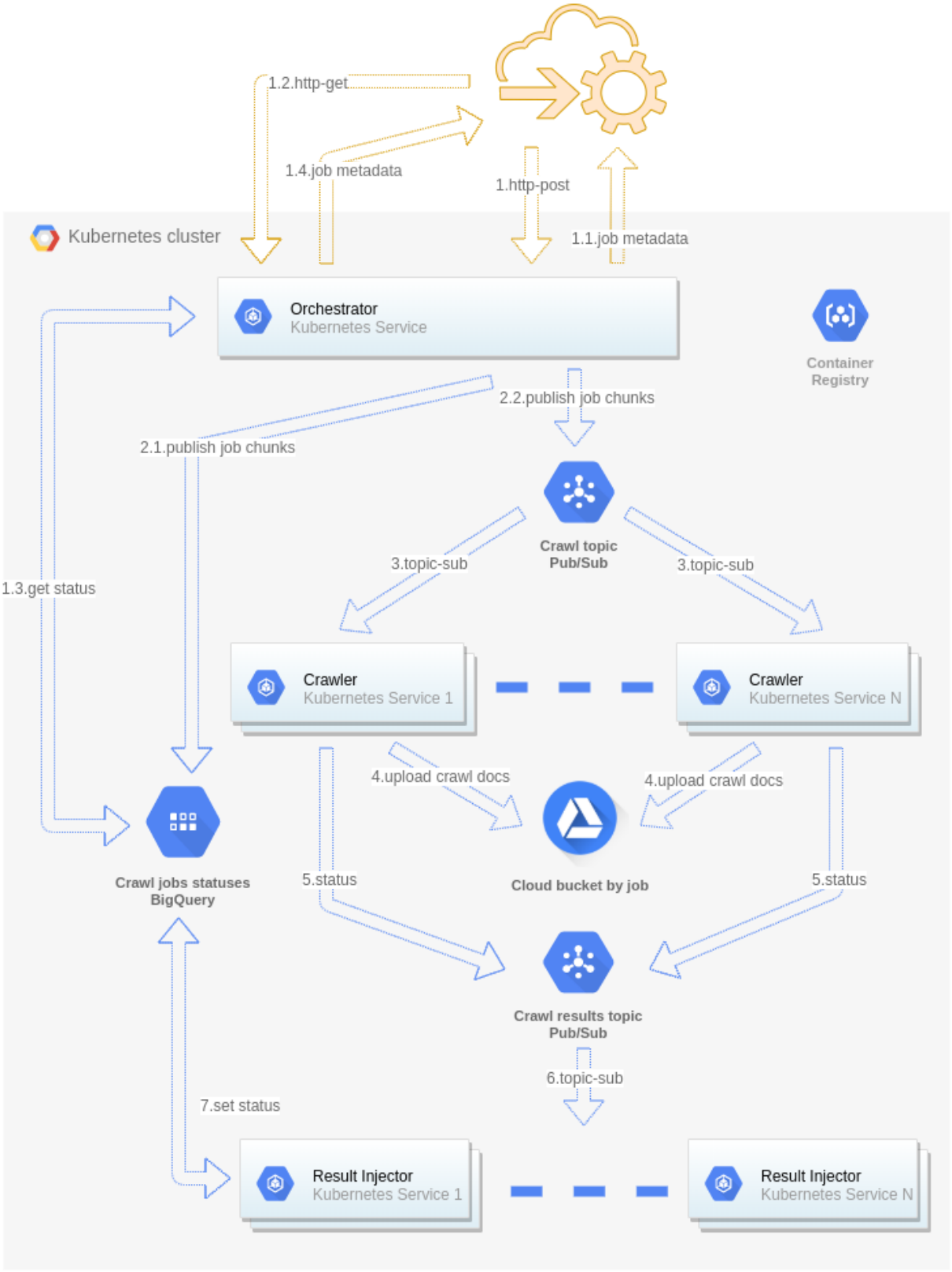
- **Cloud Bucket by job**: gère les documents générés par les services de crawling, organisés par jobs. Chaque document est nommé par *"sluggification"* de l'URL associée

- **Crawl results PubSub (topic)**: gère la file des résultats de batchs de crawling

- **Cloud function Result Injector (status)**: rapatrie les résultats de crawls et met à jour les métriques du job au sein du store

- **Cloud jobs statuses Datastore**: un store de données NoSQL Firestore qui gère les données liées aux jobs et leurs métriques

4. Architecture GCP Kubernetes



L'architecture générale est similaire à celle des CF. Les seuls différences notables sont que l'on instancie des services Kubernetes, images docker déployées à partir du container registry gcr.io.

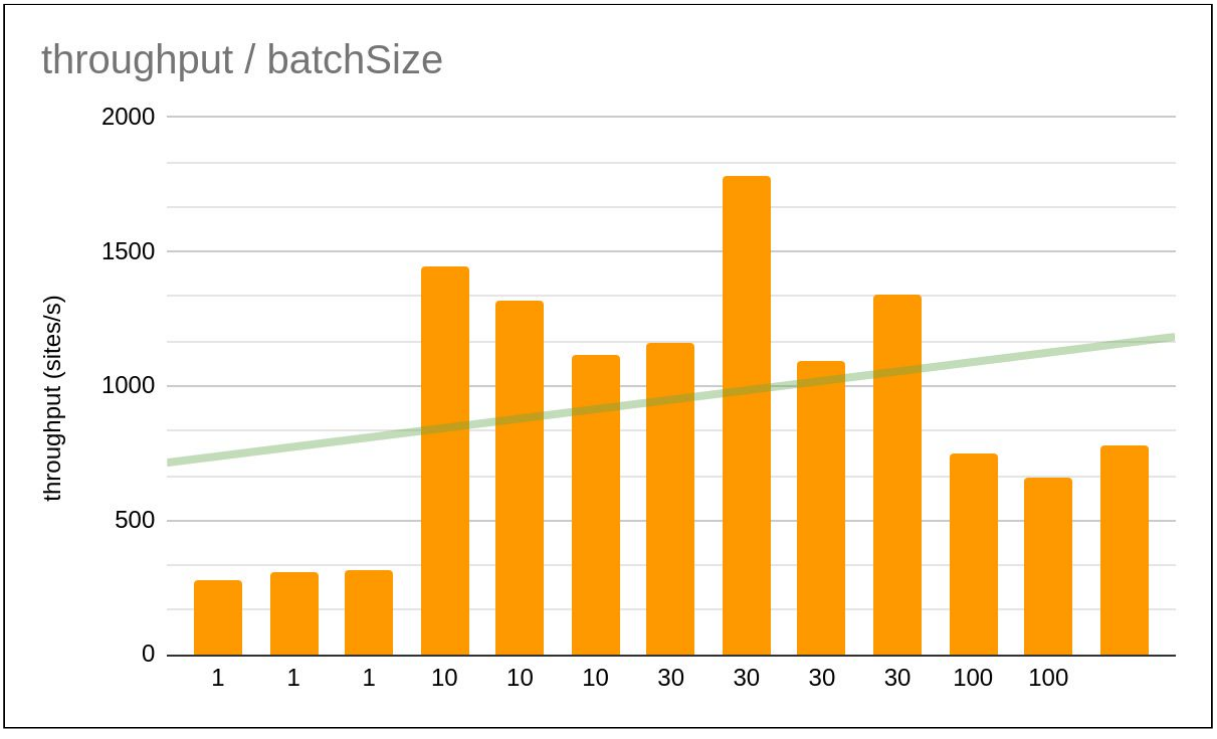
Il a été possible de fusionner les services d'import et de récupération des jobs au sein du même service, car on fait tourner un simple server http. On repose, pour des raisons de simplicité, sur le même système de bus de données (PubSub), ainsi que sur le même storage (BiqQuery et Bucket).

Les règles d'auto-scaling sont définies sur les services *"crawl"* et *"crawlResult"* pour un nombre de services compris entre 10 et 100.

5. Tests et benchmarks

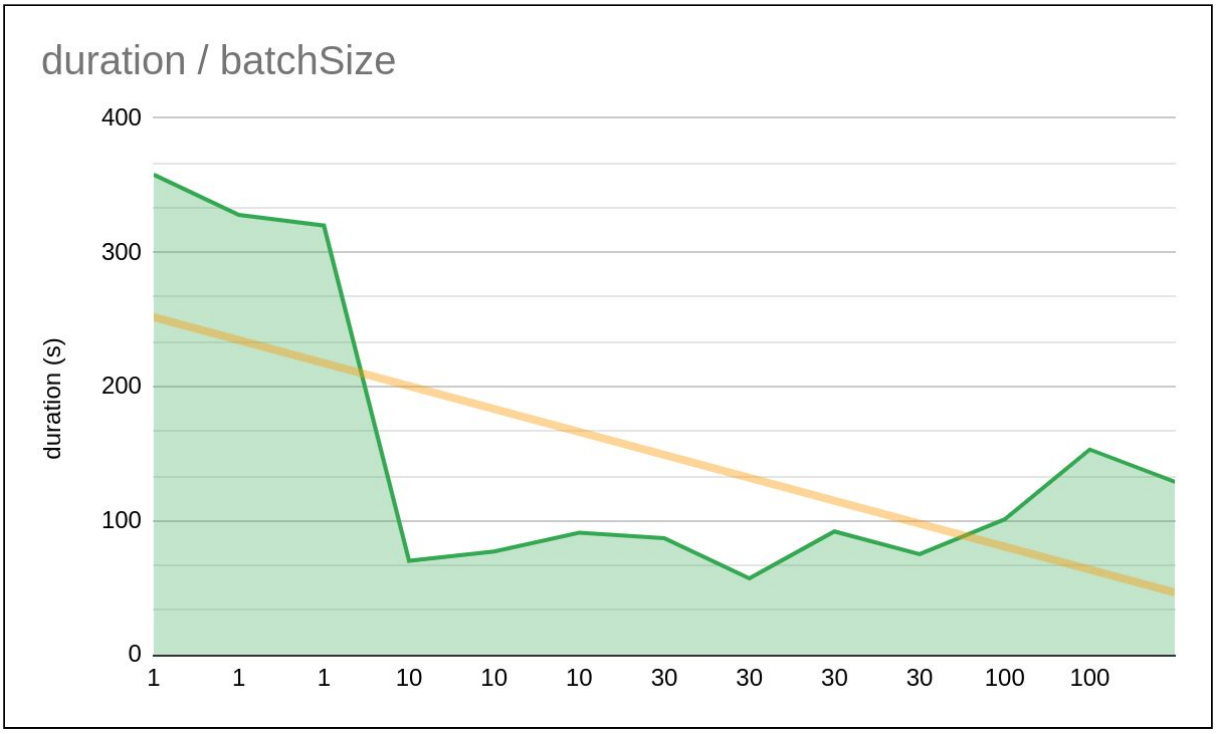
Afin de garantir la pertinence des benchmarks (cache, erreurs réseau…), ceux-ci sont alignés sur la moyenne d'au moins 3 exécutions successives du même scénario. Les Cloud Functions utilisent le paramètre "--memory 2048mb" pour avoir accès à la performance maximale de la plateforme. Le nombre maximum d'instances est fixé à 1000000, afin de pouvoir potentiellement associer une URL à une cloud function en et garantir le parallélisme maximum. Le nombre de tentatives "retries" en cas d'erreur de crawling est fixé à 1, le timeout de connexion à un site à 2000ms.

a. Cloud Functions



Ici, nous avons fait varier le nombre d'URLs traitées par chaque Cloud Function, de 1 (1 URL = 1 CF) à 100. On constate que le traitement d'une URL par une CF n'est pas très efficace, ceci doit être certainement dû à la latence forte d'instantiation des dans le pipeline de GCP. Le Load Balancer ne doit pas être en mesure de scaler plus efficacement le traitement des messages sur un plus grand

nombre de CF. Le throughput commence à être intéressant à partir de 10 URLs traitées par CF, et atteint son maximum à 30 (plus de 1700 sites/s). Les tentatives autour de 100 URLS pa CF se soldent trop souvent par des échecs et une baisse du throughput. Ce comportement serait-il dû aux limites en terme temps de calcul accordé aux CF ?



La durée du batch de crawling est fortement impactée par le batchSize. Ici, on constate que traiter un plus grand nombre de sites par CF permet de diviser le temps de traitement par 5, jusqu'à 30 URLs par CF.

b. Kubernetes

// todo

6. Conclusion

Concernant l'implémentation avec les CF :

- inconvénients :
  - limite en terme de taille de requêtes (10Mb)
  - contrôle du timeout: 60s par défaut
  - mémoire disponible maximale limitée: entre 128Mb et 2Gb
  - aucun contrôle sur le nombre minimal d'instances à conserver en ligne pour le démarrage à froid

- avantages :
  - simples à mettre en place
  - aucun besoin de paramétrer les règles de scaling
  - architecture serverless: pas d'images docker ni de registry à manager

Concernant l'implémentation avec le cluster Kubernetes:

- inconvénients :
  - learning-curve de Kubernetes
  - beaucoup de paramétrage à mettre en oeuvre: nodes, pods, deployments, ingress… par rapport à du serverless
  - tous les services doivent être développés sur du docker, peut représenter une perte de temps
  - scaling plus complexe à réaliser
  - coûts plus importants dûs au maintien à froid de certains services (LoadBalancer, minimum de replicas de services…) vs la politique tarifaire des CF sur la plateforme GCP

- avantages :
  - permet de gérer plus finement les règles de scaling
  - utilisation de docker: pas de limitation concernant les technologies utilisables
  - flexibilité sur le dimensionnement des machines
  - latence réduite: possibilité de conserver les pools de services pour une meilleure réponse aux sollicitations de crawl

7. Appendice

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.

La réalisation d'un prototype en NodeJS local m'a permis de tester l'architecture générale de la solution. Celle-ci a finalement été découpée plus finement en Cloud Functions indépendantes, alors que l'implémentation au sein de Kubernetes est finalement assez proche de celle d'origine.

Concernant les Cloud Functions, même si je salue la simplicité de leur mise en place, 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. On possède une maîtrise moins importante des versions des langages utilisés (NodeJS limité à la v10 en bêta, alors que nous en sommes à la 12.13 LTS), ce qui peut rester pénalisant pour l'utilisation de certaines fonctionnalités récentes (web worker pour la gestion des threads par exemple).

La plateforme Kubernetes possède l'avantage d'être liée aux images docker des services que l'on déploie. Il n'y a pas de limitation concernant les technologies. Le processus de déploiement par rolling-upgrade est plus long que les CF: il faut pousser les images mises à jour sur un registry, et ensuite demander une mise à jour des services correspondants. On peut gérer plus facilement le type de machines à utiliser sur les différents Nodes en terme de CPU, disque, RAM.

Au cours de cet exercice, j'ai utilisé plusieurs système de stockage avec plus au moins de succès: Datastore, Firestore, et finalement BigQuery. Firestore ne m'a pas permis de remonter de manière efficiente les métriques qui m'intéressaient (impossible d'effectuer des COUNT sur des jeux de résultats, il fallait tout réaliser en mémoire), je me suis donc rabattu sur un Dataset BigQuery qui permet d'exécuter des requêtes SQL assez puissantes sur de gros volumes de données.

Les sources de l'exercice sont disponibles sur le repository Github suivant :
⇒ https://github.com/Alahel/bulk-crawling