Gawaboumga

Zeste de savoir

De Qlik à Power BI : rétrospective technique

mercredi 28 août 2024

Table des matières

Introduction

Il y a 6 mois, j'écrivais un billet sur la mise en place d'un data lake afin de répondre aux besoins de reporting et de data gouvernance. Il est temps de revenir une première fois sur la situation, sur ce qui a fonctionné, ce qui n'a pas réussi et les points qu'il reste à améliorer. Dans ce billet, on s'intéressera principalement aux aspects techniques de la problématique. Vous pouvez consulter les aspects métiers (business) dans cet autre billet .

1. Contracts

Suite à des discussions, j'ai eu plusieurs questions sur la notion de *contract* et sur le *Domain Specific Language (DSL)* qui a été mis en place. Afin d'éclaircir le propos, je vais revenir sur ces deux concepts et leurs principaux avantages :

- Le *contract* représente une description de la donnée (d'un *record*) au sein du data lake. Cela consiste principalement en l'énumération des champs et des types qui composent une "table" ainsi qu'un identifiant qui permet de déterminer le lieu de stockage de cette "table". Il contient parfois des informations additionnelles liées au bon fonctionnement de systèmes plus spécifiques tels qu'une description des indexes à créer, un *watermark* pour charger un différentiel dans les données (mécanisme d'*append*), la résolution automatique d'entités, ...
- Le *DSL* a été créé afin de répondre à 5 grandes problématiques :
  - La génération des dits *contracts* ;
  - La génération des requêtes SQL de récupération des données depuis une source ;
  - La génération des fichiers de transformation *dbt*, ce qui permet de facilement renommer une variable, de bénéficier de l'autocomplétion ou d'effectuer des changements techniques ;

— La génération de l'affiliation des données (*data lineage*), comme on décrit la transformation des données au sein du *DSL* (ce qui permet de fournir tant les contracts initiaux que ceux résultants), il est aisé de remonter l'arbre de dépendances pour recréer l'affiliation ou déterminer quelles sont les données qui ne sont pas exploitées. L'intégration avec les scripts de transformation (en python ou autres) est encore en réflexion ;
— La génération du glossaire afin de compléter notre catalogue de la donnée ;

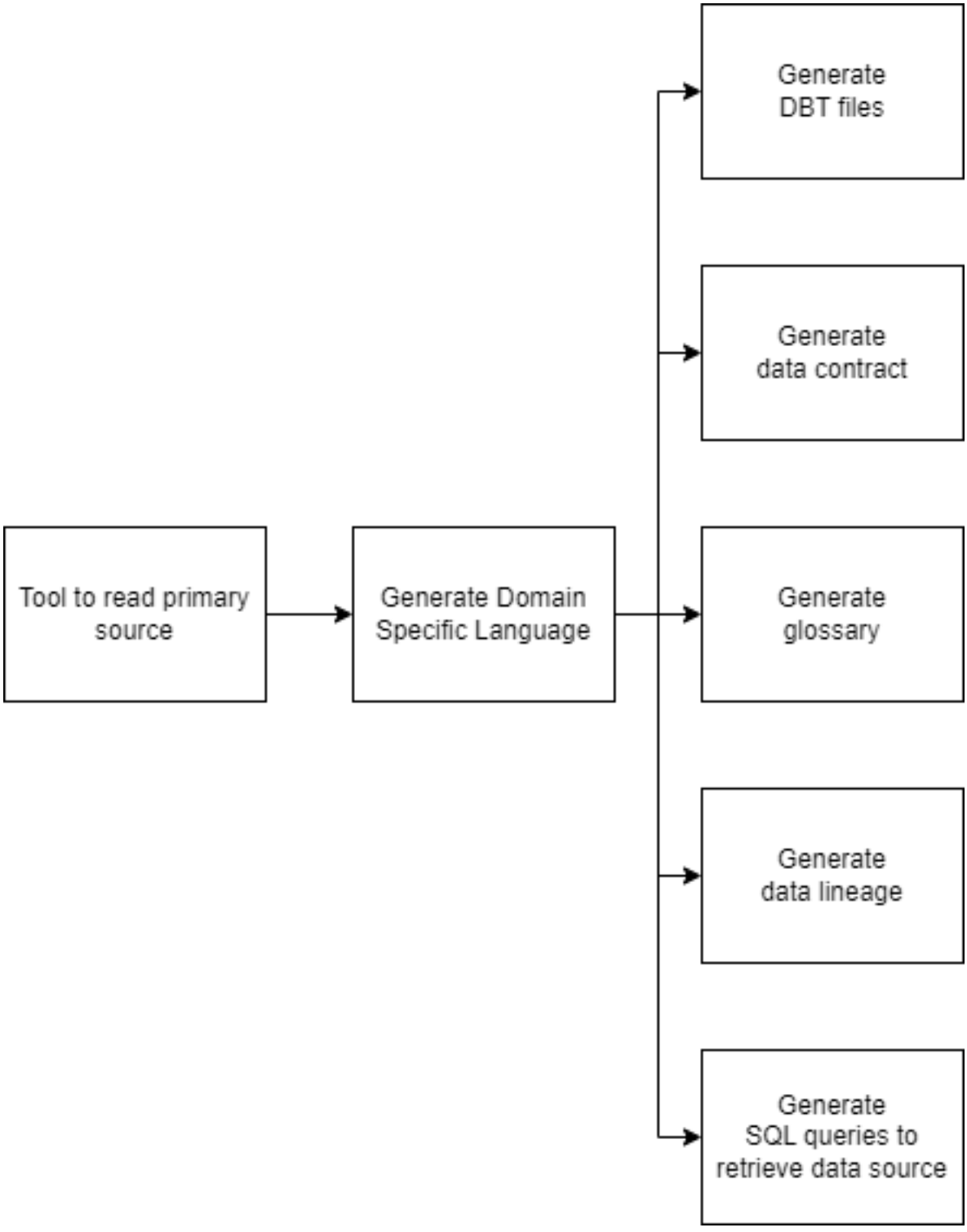

FIGURE 1.1. – Étapes liées au Domain Specific Language et ses capacités

En outre, un outil a été développé afin de lire une source primaire et créer un fichier correspondant à notre *DSL* et ainsi faciliter l'ajout de nouvelles sources.

2. Exemples

Sur un exemple fictif, on aurait quelque chose de ce type :

```
@stone('ma_source_de_données')
class MaTransformationEntreSourceEtIron:
    __origin__ = '[db].[schema].[table]'
    __source__ = 'nom_à_la_réception'
    __destination__ = 'ma_première_transformation'

    id = Integer('code', primary_key=True)
    postal_code = PostalCode('zip_code', nullable=False)
    fuel_french = String('carburant', nullable=True)
    fuel_dutch = String('brandstof', nullable=True)

i = From(MaTransformationEntreSourceEtIron)
@gold('mon_jeu_de_données')
class MaTransformationVersGold:
    __sources__ = [i]
    __destination__ = 'ma_destination_finale'

    id = i.id
    postal_code = i.postal_code
    fuel_category = UpperCase(Coalesce(i.fuel_french,
        i.fuel_dutch))
```

qui génèrerait (en autres) un *contract* de ce type :

```
class MaTransformationVersGold:
    __location__ = ('mon_jeu_de_données', 'ma_destination_finale')

    id = Integer(primary_key=True)
    postal_code = PostalCode(nullable=False)
    fuel_category = String(nullable=True)
```

Évidemment, ce n'est pas toujours aussi élégant, certains cas sont souvent complexes à décrire tels que les *Common Table Expressions (CTE)*, sous-requêtes, des unions, … mais il est possible d'offrir d'éventuels sucre syntaxiques (ex : `__where__ = NotExists(Table, lambda self: Equals(self.U, MaTableActuelle.T))`). Il peut être intéressant d'incorporées la gestion des colonnes optionnelles ou variadiques (même si DBT ne va pas aimer).

3. Ingestors

Les données d'une source de données (*data source*) sont ingérées grâce à la notion des *ingestors*. L'idée était d'unifier toutes les étapes liées à une ingestion (de la source originelle jusqu'au stockage des données exploitables) au sein d'un même concept afin de partager du code entre les différents éléments ou effectuer des traitements spécifiques à un flux. Les *ingestors* reçoivent en paramètre un contexte d'ingestion permettant de se souvenir quelles tables ont été ingérées, quand, avec un hash du *contract* (pour voir si on a apporté des changements aux données à récupérer) et avec éventuellement une information spécifique. L'ingestion se décompose en quatre composantes différentes dont l'ordre peut être surchargé en fonction du besoin spécifique :

— Collectors : Les collecteurs représentent des agents qui collectent d'éventuelles données depuis la source (classiquement un FTP ou un webservice) et les sauvegardent.
— Extractors : Les extracteurs prennent les données issues des collecteurs et fournissent un ensemble de flux de données (*datastreams*) - associé à un *contract*, avec un *parsing* plus ou moins avancé en fonction du besoin.
— Transformations : Les transformations s'appliquent tant sur les flux de données (*datastreams*) issues des extracteurs que sur la combinaison du résultat des différents extracteurs. C'est également à ce moment-là qu'on s'assure que les données correspondent bien au *contract* qui a été associé.
— Loaders : Les chargeurs s'occupent de stocker les données ainsi récupérées. On peut avoir plusieurs comportement en fonction de si l'on souhaite remplacer entièrement les données, les compléter ou les mettre à jour.

Plusieurs difficultés émergent dans une telle solution :

— Comment s'intégrer proprement avec Airflow ;
— Les *datastreams* sont (généralement) évalués de manière paresseuse (*lazy*), ce qui implique quelques complexités au niveau du code (par exemple, pour mettre à jour le contexte d'ingestion après que l'ingestion ait réussi) ;
— La gestion des flux de données de succès ou d'erreurs ; suite aux validations effectuées par rapport au *contract*, certaines données peuvent être considérées comme erronées mais doivent être enregistrées à des fins d'analyse. On se retrouve avec un *fork* du flux de données dans un contexte souvent lazy.
— Python gère difficilement le passage de *contract* créé dynamiquement entre différents processus (problème de *pickle*).

4. Architecture

De gros progrès ont été effectués afin de rendre l'infrastructure bien plus résiliente et flexible aux changements futures. Une partie de la problématique était liée à la gestion des droits d'accès pour accéder à des technologies externes telles que *sharepoint*, *alfresco* ou des *network fileshares (NFS)* ainsi que la duplication des environnements (développements locaux, test & production). On remercie très fortement l'équipe infrastructure qui a su nous épauler dans cette transition.

L'autre grande avancée a été sur l'architecture de la solution. Celle-ci a été bien mieux formalisée (entendez documentation bien plus complète) afin d'incorporer les nombreux aspects et problématiques dont nous cherchons à faire face et mieux communiquer à des parties-prenantes

du projet, qu'elles soient internes (IT) ou externes (métier ou business). Vous comprendrez que je ne partage ici qu'une vue très haut-niveau, simplifiée (et légèrement "fallacieuse") de notre solution actuelle.

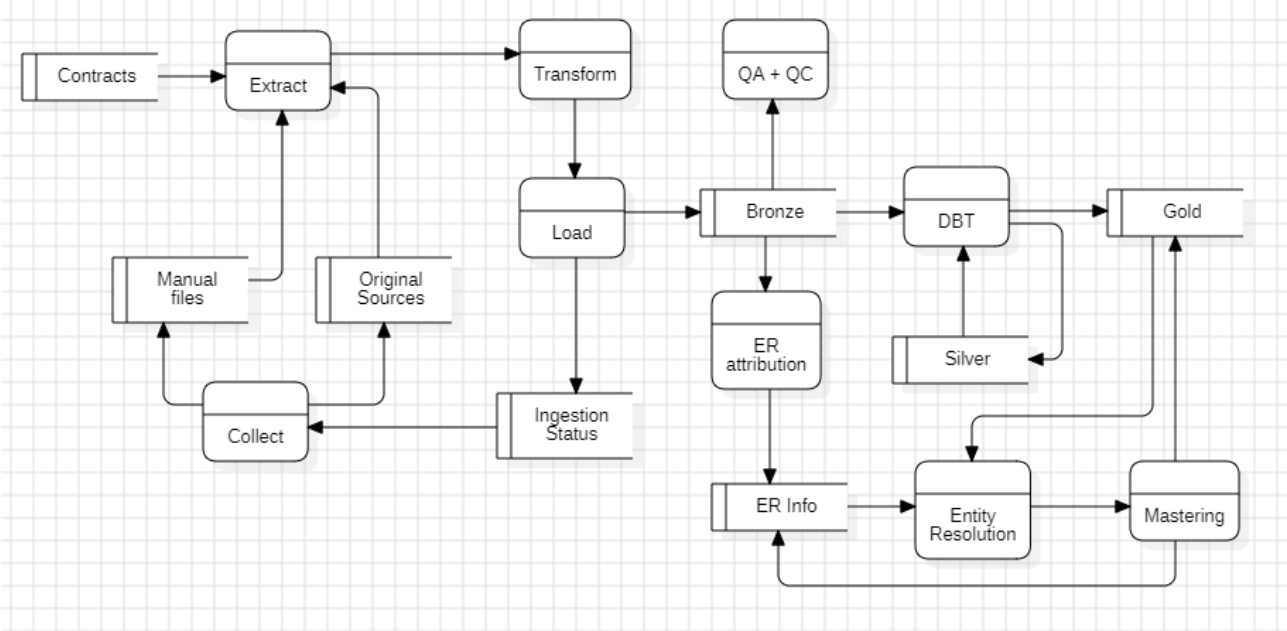

FIGURE 4.2. – Vue simplifiée de l'architecture donnée du projet

Plusieurs difficultés ont été rencontrées lors de la conception :

- Comment ordonnancer nos blocs d'opérations alors que certains sont optionnels et dépendent d'autres sous-blocs qui sont assez éloignés dans le code ?
- Comment proposer une résolution des entités simples ?
- Comment nommer nos différentes étapes ?
- Comment rendre cela le plus sobre numériquement ?
- Comment monitorer les différentes étapes de manière simple ? [1]
- Comment permettre de relancer le sous-arbre d'exécution ? [1]
- Comment permettre l'inclusion de codes *customs* de la part de nos utilisateurs ?
- Rôles et responsabilités de chaque élément ?
- Compromis entre "non aux données pourries" et "le métier ne peut pas faire autrement".

5. Airflow

Je ne vais pas passer par 4 chemins, mais Airflow me sort par les trous de nez.

- Les images *workers* sont des images Docker étendues de celles fournies par la fondation Apache afin d'inclure nos différentes dépendances logicielles ou infrastructures. En particulier, Azure, … Sauf que les *dags* sont lus par les autres images (*scheduler* ou *triggerer*) qui ne possèdent pas ces dépendances et donc, cela fonctionne non sans peine …

1. On reviendra sur ce point dans le prochain paragraphe avec Airflow.

— Airflow arrive avec sa version de SQLAlchemy (1.4) (de manière plus générale, son lot de dépendances) qui ne correspond pas à celle de notre projet (2.0). Oui, nous pouvons employer *PythonOperator* mais alors il faut spécifier nos *requirements* à chaque fois.
— Airflow ne gère pas les tâches qui emploient du multi-processing en interne. Il faut remplacer *multiprocessing* par *billiard* (et demander d'avoir un nouveau contexte à l'exécution des *tasks*) mais l'erreur est loin d'être évidente …
— Bon gré, mal gré, on ne peut pas faire de la dépendance sur d'autres dags. Nos sources sont mises-à-jour à différents intervales (et sont donc dans des dags). Mais nos datasets peuvent se comporter différemment, être executé sur un tout autre intervalle mais qui dépendent des mises-à-jour des sources préalablement.
— L'interface graphique d'Airflow est d'un autre âge. La mise-à-jour 2.8.0 a fait un grand bien.
— Développement local et tests ardus.
— Difficile de créer un flux dynamique d'exécution.
— Rotation native des logs ?

Oui, on peut passer outre les différentes difficultés rencontrées, mais devraient-elles exister à la base ?

Je réfléchis à passer à d'autres solutions, qui puissent opérer de manière transparentes avec DBT de préférence.

6. DBT

Quel est le réel bénéfice de DBT ? Sur le site web , ils mentionnent trois intérêts principaux :

— Version control & CI/CD ;
— Documentation ;
— Développement & gestion des dépendances ;

L'intérêt des deux premiers éléments deviennent quelques peu caduques avec l'usage de notre *DSL* qui est même plus puissant puisque nous avons notre affiliation (*lineage*) au niveau des champs et non au niveau des tables. Quant au développement & gestion des dépendances , il y a des aspects intéressants, mais qui sont souvent équivalents à notre solution :

— Matérialisation : Sur la manière dont une requête doit créer ou mettre à jour une relation existante => je vois un intérêt à DBT sur ce point, il est possible d'avoir le même comportement, mais c'est moins élégant ;
— Jinja : Cela permet de gérer des colonnes variadiques ou optionnelles par exemple => cela peut être géré au niveau de la génération des *contracts* ;
— Déterminer l'ordre d'exécution : Fourniture d'une résolution automatique des dépendances entre les scripts => on possède déjà le graphe de dépendance ;
— Test : Vérifier l'exécution des requêtes par des assertions => parent pauvre du QA ou QC, mais peut servir de briques à cette problématique ;
— Packages : Gestion de packages dans le cas de grandes organisations => on n'est pas assez grand pour avoir besoin de cela pour le moment ;
— Seed : Charger des fichiers brutes => Hors sujet ;
— Snapshot : Permet de reconstruire une vue historique => Hors sujet ;

Le principal intérêt que je vois dans cette technologie est qu'elle agit comme un orchestrateur de la donnée et dans une moindre mesure, un moteur d'exécution (*compute engine*) mais en moins cher que Spark puisque l'on peut réemployer le serveur SQL. Je vois un intérêt certain à la technologie, mais il faut bien comprendre le problème auquel DBT essaye de répondre. Je pense que cela joue un rôle particulièrement adapté dans des contextes moins dépendants de la donnée (entreprise à finalité autres qu'informatique ou équipe encore plus réduite).

La grande question que je me pose est :

Va-t-on jamais avoir un schisme à la *Elastic Search*, *Redis*, *CockroachDB*, *Vagrant*, *Terraform* (et plus généralement toute la suite *HashiCorp*), *Couchbase Server*, *Confluent* ou *MongoDB* au niveau de la licence ?

Globalement, cette composante nous est toujours nécessaire (et demanderait d'écrire pas mal de code pour la remplacer entièrement) mais est rendue à une portion plus que congruë en tant que détermination de l'ordre d'exécution (et exécution) et contrôle de la matérialisation.