Etat de l'art

Nous avons réalisé un état de l'art sur la génération d'un graphe aléatoire. Nous n'avons constaté que les travaux liés au "*workload modeling*" conviennent mieux à ce qu'on voulait faire. Dans le contexte de "*workload modeling*" les nœuds d'un graphe représentent des tâches à exécuter et les arcs sont des contraintes d'antériorité. Donc, un graphe est construit niveau par niveau (i.e. un graphe à multi-niveau) et les nœuds situés dans un niveau donné sont connectés aux nœuds situés dans les niveaux précédents.
Comme un graphe de dérivation est construit par le même principe, cela nous a permis de répondre aux questions suivantes:

- Comment un graphe de "*workload modeling*" est construit?
- Avec quels paramètres pouvons-nous construire un graphe?

Nous avons également cherché les outils de "*workload modeling*" sur internet comme un point de départ. Parmi les outils trouvés, nous avons choisi DAGGEN qui est accessible par **https://github.com/frs69wq/daggen** pour que ce soit la base de notre générateur de graphe aléatoire.

Mise en oeuvre d'un générateur de graphe aléatoire

Nous n'avons pas utilisé directement le code de DAGGEN comme un point de départ, car il fallait aussi migrer/charger les données du graphe généré vers une base de données de HANA une fois que le graphe est généré, et cette étape aurait donc pris du temps de développement à part les étapes de la génération de graphe. En plus, quand la taille d'un graphe généré est élevée, cette étape serait un coup supplémentaire en fonction du temps d'exécution.

En conséquence, nous avons utilisé le langage SQL Script et nous avons directement créé un générateur de graphe aléatoire à l'intérieur de la base de données de HANA en utilisant des procédures stockées. Ainsi, nous pouvons directement migrer/charger les données d'un graphe aléatoire vers les tables de Data Lineage. Noter que les tables de Data Lineage sont particulièrement 2 tables: La première est utilisée pour stocker les informations d'arcs et la deuxième est utilisée pour stocker les informations de nœuds avec leurs propriétés. Dans notre générateur de graphe, nous avons inspiré de quelques idées de DAGGEN; mais nous avons quand même crée notre propre générateur de graphe aléatoire selon nos besoins.

Dans la *figure 1*, nous voyons un exemple d'un graphe à multi-niveau. Un carré représente un nœud de données qui peut être une table de base de données ou une vue Sql, et un rond représente un nœud de *design object*. On peut imaginer qu'un nœud de *design object* représente une procédure stockée qui prend **a** tables ou vues en entrée et produit **b** vues en sortie.

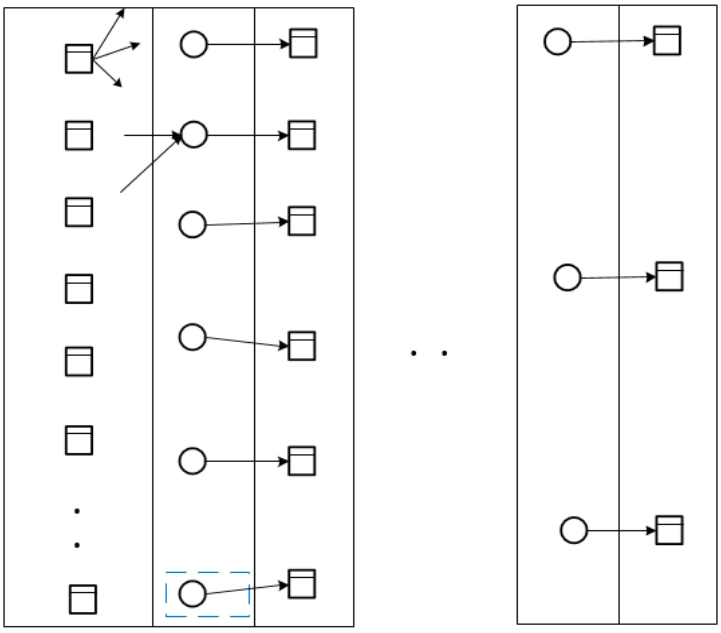

Figure 1: Un exemple de graphe orienté acyclique à milti-niveau (avec des tables/vues et des design object)

Dans la génération d'un graphe aléatoire, il faut définir le contenu de chaque *design object*. Un exemple de contenu d'un *design object* pourrait être comme illustré dans la *figure 2*. Les ronds en noir représentent des attributs, les ronds simples en blanc représentent des opérations de données comme l'union, les fonctions d'agrégation, la concaténation de plusieurs attributs, etc., et les doubles ronds en blanc représentent les opérations ensemblistes comme la jointure, le filtrage ou les fonctions de fenêtre.

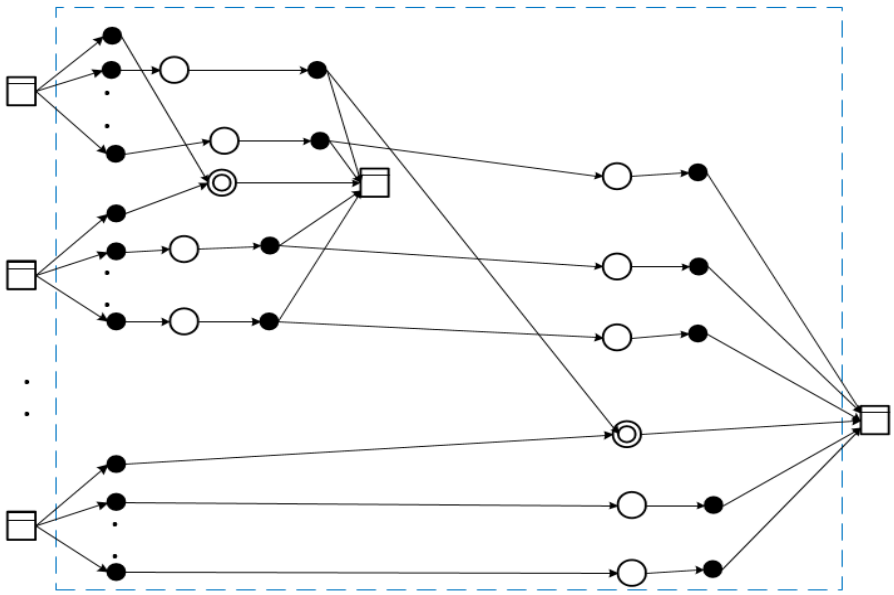

Figure 2: Un exemple des nœuds à l'intérieur d'un design object

Etant donné que le langage SQL Script est plus lent par rapport au langage C/C++, nous avons dû réfléchir des solutions afin d'accélérer la génération d'un graphe aléatoire. Et nous avons utilisé 2 idées suivantes:

- L'utilisation des motifs préconstruits pour charger plus vite le contenu des *design objects*:

  Nous avons réalisé un générateur de motifs qui est censé de générer **m** motifs. Le nombre **m** est déterminé à partir de l'intervalle [min_value, max_value] concernant le nombre de tables/vues en entrée et de l'intervalle [min_value, max_value] concernant le nombre de vues en sortie. L'avantage d'utiliser des motifs pre-construits est de gagner du temps pour les mêmes types de *design objects* au lieu de générer un nouveau contenu pour chacun. Par exemple, pour tous les *design object* qui prennent 2 tables en entrée et produit une vue en sortie, nous utilisons le même motif.

- Dupliquer un graphe aléatoire déjà généré pour avoir un graphe plus gros:

  Nous dupliquons un graphe aléatoire généré autant de fois que l'utilisateur souhaite et nous fusionnons l'un après l'autre pour obtenir un graphe plus gros.

Résultats

Il y a plusieurs paramètres d'entrée pour lancer la génération d'un graphe aléatoire. Comme notre but est de mesurer la performance des procédures stockées de Data Lineage selon quelques points, nous ne donnons pas tous les paramètres d'entrée en dessous.

Nous avons généré 7 graphes:

1) Un graph contenant 1.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend une table/vue en entrée ou produit une vue en sortie. Ce graph généré contient 8 niveaux de profondeur.
2) Un graph contenant 1.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend 20 tables/vues en entrée ou produit une vue en sortie. Ce graph généré contient 9 niveaux de profondeur.
3) Un graph contenant 1.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend 40 tables/vues en entrée ou produit une vue en sortie. Ce graph généré contient 9 niveaux de profondeur.
4) Un graph contenant 10.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend une table/vue en entrée ou produit une vue en sortie. Ce graph généré contient 80 niveaux de profondeur.
5) Un graph contenant 10.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend 20 tables/vues en entrée ou produit une vue en sortie. Ce graph généré contient 90 niveaux de profondeur.
6) Un graph contenant 10.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend 40 tables/vues en entrée ou produit une vue en sortie. Ce graph généré contient 90 niveaux de profondeur.
7) Un graph contenant 100.000 tables/vues et possédant seulement 1 seul motif de *design object* qui prend une table/vuesen entrée ou produit une vue en sortie. Ce graph généré contient 800 niveaux de profondeur.

Noter que quand le nombre d'entrée d'un motif augmente, le nombre de nœud utilisé dans le motif augmente aussi. C'est la raison pour laquelle on peut dire qu'il y a à peu près 7-8 millions de nœuds (des nœuds d'attributs, d'opération de calcul, d'opération de jointure, etc.) dans le 6ème graphe généré.

Le but de la génération de ces 7 graphes aléatoires est de mesurer la performance des procédures stockées de Data Lineage en 2 aspects:

- le nombre de tables/vues.
- le nombre de nœuds utilisé dans les contenus des *design objects*.

Et donc les questions qui se posent sont les suivantes:

1) Qu'est-ce qui se passe quand on augmente le nombre de tables/vues?
2) Qu'est-ce qui se passe quand on augmente le nombre de nœuds utilisés dans les contenues des *design objects*?

On peut répondre à la première question en consultant les résultats basés sur les 1er, 4ème et 7ème graphes générés. En exécutant la procédure stockée concernant l'analyse d'impact à partir d'une table située au 1er niveau, le temps d'exécution prend moins d'une seconde pour les 3 graphes générés.

On peut répondre à la 2ème question en consultant les résultats basés sur les 1er, 2ème et 3ème graphes générés. En exécutant la procédure stockée concernant l'analyse d'impact à partir d'une table située au 1er niveau, le temps d'exécution prend moins d'une seconde pour le 1er graphe généré, 7 secondes pour le 2ème graphe et 38 secondes pour le 3ème graphe. Donc, on peut dire qu'il y a un souci quand le nombre de nœuds utilisés dans les contenues des *design objects* augmente.

En plus de ces 3 graphes générés, nous avons exécuté la même procédure stockée à partir d'une table située au 1er niveau pour le 5ème graphe. Et cela a pris environ 55 minutes de l'exécuter. Par conséquent, il faudrait revoir l'implémentation du code ou changer la conception des algorithmes, car la durée de 55 minutes est un peu trop élevée.