Projet NoSql (partie I)

-

Rapport de Alizé Baudin

Le 17 mai 2023

I. Introduction

Voici le tableau que j'ai utilisé pour l'étude de donnée en Redis, PostgresSQL avec requête SQL et tableau JSON, et enfin sous Neo4j.

Plats	Repas classique	gC02	Repas classique bis	gC02
Entrée	Légume à la grecque	71.6	Tzatzíki	507.2
	- Légumes de saison	53.4	- yaourt	360
	- huile d'olive (1/2	18.2	- concombre	129
	cuillérée)		- huile d'olive (1/2 cuillérée)	18.2
Plat	Poulet au riz	953.5	Bifteck-frite	5630
	- poulet	774	- bifteck	5370
	- riz	84.6	- frite	260
	- beurre	94.9		
Dessert	Plateau de fromage	323	Tarte aux poires	150.2
	- fromage à pâte molle	107	- farine	46.9
	- fromage à pâte dure	140	- poire	71
	- pain	76	- huile (1cuillère)	32.2
Total		1350		6290

Plats	Repas végétarien	gC02
Entrée	Soupe de légume	68.5
	- Légumes de saison	53.4
	- huile (1/2 cuillérée)	15.1
Plat	Omelette aux pommes de	309.3
	terre	
	- deux œufs	276.7
	- pomme de terre	15.5
	- huile (1/2 cuillérée)	17.1
Dessert	Salade de fruits	129.4
	- fruit de saison	53.4
	- pain	76
Total		510

L'étude sera disposée en quatre partie, soit quatre parties associées à chaque logiciel de développement de base de données : Redis, PostgreSQL, MongoDB et Neo4j.

Dans chaque partie, nous allons répondre aux problématiques suivants selon

1. Afficher l'empreinte carbone de chaque ingrédient des deux menus

- 2. Afficher l'empreinte carbone de chaque plat de chaque menue et les ingrédients associés (avec leur emprunte carbone)
- 3. Afficher la composition et l'empreinte carbone de chacun des plats composant un menu (avec les détails sur la composition des plats comme sur les affichages précédents par exemple).
- 4. Afficher les plats (avec leur empreinte carbone) contenant un ingrédient donné.
- 5. Afficher les ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une empreinte inférieure à un seuil donné.

II. PostgreSQL

Cas requête SQL simple.

Pour comparer avec le modèle qui suivra, nous présenterons le format de la construction des bases de données.

Dans cette sous-partie, voici un exemple de construction de la base de données :

```
12 -- creation de la table des plats
13 CREATE TABLE plats (
14 id int PRIMARY KEY,
15 nom VARCHAR(255),
16 empreinte_carbone FLOAT
17 );

45 -- Plat principal :
1NSERT INTO plats (id, nom, empreinte_carbone)
47 VALUES
(12,'omelette_pomme_de_terre', 309.3),
(22,'poulet_au_riz', 952.1),
(32,'poulet_au_riz', 5630);

UPDATE plats
SET nom = 'bifteck_frite'
WHERE id = 32;

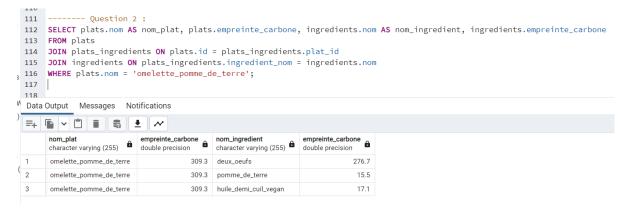
WHERE id = 32;
```

Question 1 : empreinte carbone d'un ingrédient donné.

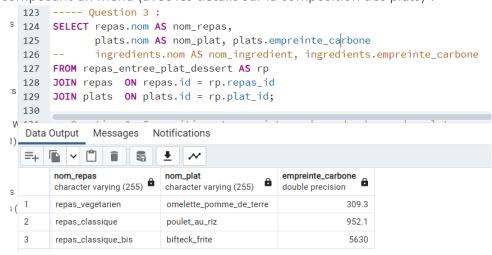
Pour récupérer l'empreinte carbone d'un ingrédient donné, nous utilisons la table 'ingredients' dans la base de données qui contient les informations sur les ingrédients et leurs empreintes carbone.



Question 2 : quelle est l'empreinte carbone d'un plat donné et quels sont les ingrédients composant un plat (avec leur empreinte carbone) ?



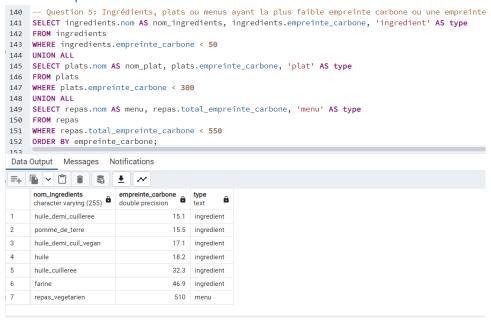
Question 3 : quelles sont la composition et l'empreinte carbone de chacun des plats composant un menu (avec les détails sur la composition des plats) ?



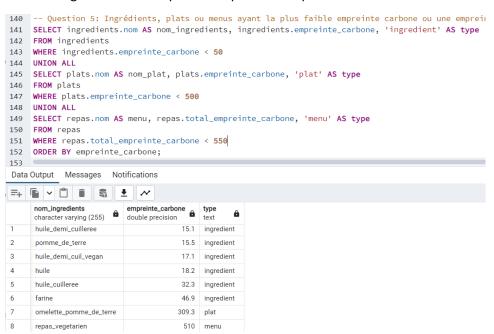
Question 4 : Afficher les plats (avec leur empreinte carbone) contenant un ingrédient donné.



Question 5 : afficher les ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une empreinte inférieure à un seuil donné.



Autre changement de seuil pour faire paraître les plats :



PostgreSQL sous un système de clé-valeur

Une fois que nous avons testé le format 'classique' en SQL, répondons aux questions pour un système clé-valeur. Les données sont ainsi sensiblement identiques à ce que l'on a précédemment.

Nous utilisons ici le système clé-valeur sous format JSON.

Voici dans un premier temps la création des tables :

```
1 -- Table "ingredients"
     CREATE TABLE ingredient (
 2
 3
        id SERIAL PRIMARY KEY,
        data JSONB
 4
     );
 5
 6
 7
     -- Table "entree"
 8
 9
     CREATE TABLE entree (
        id SERIAL PRIMARY KEY,
10
        name VARCHAR(255) NOT NULL,
11
12
        ingredients JSONB NOT NULL
13
     );
14
15
     -- Table "plat"
     CREATE TABLE plat (
16
        id SERIAL PRIMARY KEY,
17
        name VARCHAR(255) NOT NULL,
18
19
        ingredients JSONB NOT NULL
20
    );
22 -- Table "dessert"
23 CREATE TABLE dessert (
24
    id SERIAL PRIMARY KEY,
25
    name VARCHAR(255) NOT NULL,
26
    ingredients JSONB NOT NULL
27 );
28
29 -- Table "menu"
30 CREATE TABLE menu (
   id SERIAL PRIMARY KEY,
31
32
   name VARCHAR(255) NOT NULL,
33
    entree_id INT,
34
    plat_id INT,
35
    dessert_id INT,
36
    FOREIGN KEY (entree_id) REFERENCES entree(id),
    FOREIGN KEY (plat_id) REFERENCES plat(id),
37
38
    FOREIGN KEY (dessert_id) REFERENCES dessert(id)
```

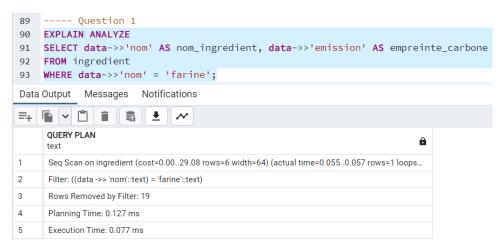
Pour des raisons pratiques, la table 'menu' ne change pas, mais que les autres tables ont un ajout de l'ingrédient en format JSON, soit en format documenté. Ainsi pour compléter la table 'menu' nous n'avons rien changé, contrairement aux autres tables, dont nous avons un exemple ci-dessous :

```
INSERT INTO ingredient (data) VALUES
42
43
       ('{"nom": "légume de saison", "emission": 53.4}'),
       ('{"nom": "huile d''olive (1/2 cuillérée)", "emission": 18.2}'),
45
       ('{"nom": "poulet", "emission": 774}'),
       ('{"nom": "riz", "emission": 84.6}'),
46
       ('{"nom": "beurre", "emission": 94.9}'),
47
       ('{"nom": "fromage à pâte molle", "emission": 107}'),
       ('{"nom": "fromage à pâte dure", "emission": 140}'),
49
50
       ('{"nom": "pain", "emission": 76}'),
       ('{"nom": "yaourt", "emission": 360}'),
51
       ('{"nom": "concombre", "emission": 129}'),
52
       ('{"nom": "bifteck", "emission": 5370}'),
53
       ('{"nom": "frite", "emission": 260}'),
54
55
      ('{"nom": "farine", "emission": 46.9}'),
56
      ('{"nom": "poire", "emission": 71}'),
57
      ('{"nom": "huile (1 cuillère)", "emission": 32.3}'),
58
       ('{"nom": "deux œufs", "emission": 276.7}'),
       ('{"nom": "pomme de terre", "emission": 15.5}'),
59
       ('{"nom": "huile ½ cuillérée", "emission": 17.1}'),
60
       ('{"nom": "fruits de saison", "emission": 53.4}'),
61
62
       ('{"nom": "pain", "emission": 76}');
66 -- Insertion des entrées
67 INSERT INTO entree (name, ingredients) VALUES
68
     ('légumes à la grecque', '{"légume de saison": 53.4, "huile d''olive (1/2 cuillérée)": 18.2}'),
     ('tzatziki', '{"yaourt": 360, "concombre": 129, "huile d''olive (1/2 cuillérée)": 18.2}'),
69
     ('soupe de légumes', '{"légume de saison": 53.4, "huile ½ cuillérée": 15.1}');
```

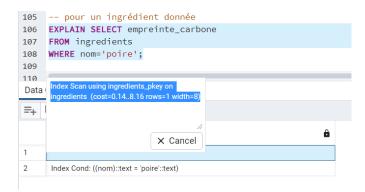
Pour ce faire, nous allons répondre aux questions sur la comparaison des performances entre le modèle classique et le modèle clé-valeur que l'on a ci-dessus.

Question 1 : empreinte carbone d'un ingrédient donné.

Sous PostgreSQL clé-valeur :



Sous PostgreSQL 'classique':



On remarque la comparaison avec le SQL 'classque', que le nombre de colonne et de ligne est moindre que pour le SQL clé-valeur. Cela s'explique par le fait que la structure initiale définit déjà une docuentation modulable, c'est-à-dire qui n'est pas agressif lors des requêtes mais déjà chargé. Ce qui n'est pas le cas avec le SQL 'classique'

Question 2 : quelle est l'empreinte carbone d'un plat donné et quels sont les ingrédients composant un plat (avec leur empreinte carbone) ?

Ecriture avec PostgreSQL clé-valeur :



Avec SQL classique sur PostgreSQL:



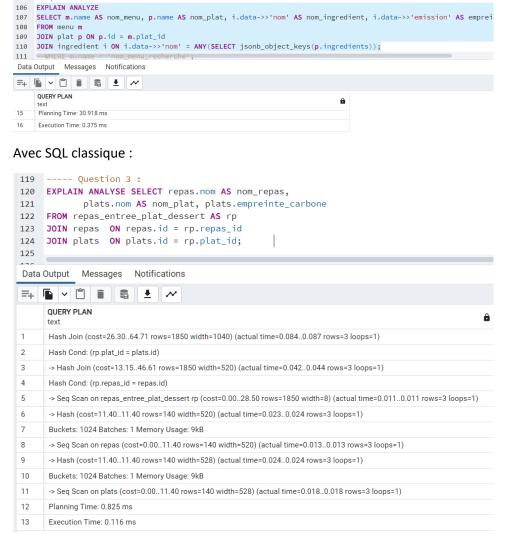
Ici la requête SQL simple, nous donne de meilleure performance, cela est dû au fait que l'on ne sélectionne que des plats et donc des requêtes à chargement 'lazy', de par la définition des clés

étrangères dans les tableaux. Cela veut dire que l'on ne modifie la structure de la table. Contrairement au cas de PostgreSQL clé-valeur, où les requête d'appel 'JOIN' est un chargement 'agressif', c'est à 'dire qui doit charger les éléments spécifiques de la documentation en format JSON à la table 'plat'.

Question 3 : quelles sont la composition et l'empreinte carbone de chacun des plats composant un menu (avec les détails sur la composition des plats) ?

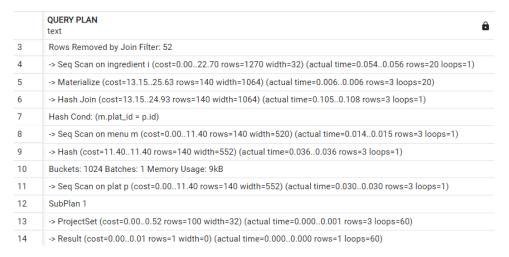
Avec SQL clé-valeur :

- Ouestion



Le temps d'exécution est encore une fois supérieur en système de clé valeur qu'en classique.

Cependant, comme nous le voyons ci-dessous, le temps à chaque requête est plus cours entre chaque étape pour scanner par exemple les tables.



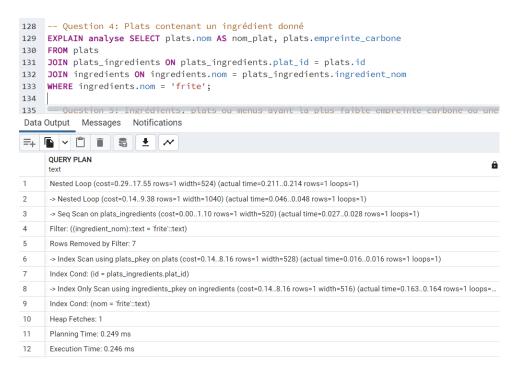
Le temps supplémentaire en système clé-valeur, vient donc bien du fait du chargement agressif pour le renvoi de la requête demandée, pour l'ensemble des menus qui n'est pas sous un format documenté (JSON)

Question 4 : Afficher les plats (avec leur empreinte carbone) contenant un ingrédient donné.

Avec un système clé-valeur :



Avec SQL simple:



Ici, nous commençons à voir l'intérêt d'un système clé valeur. En effet, le modèle clé valeur est ici plus performant en termes de temps. Cela s'explique par la structure initiale donnée, c'est-à-dire à la documentation donnée, qui permet de chargé les informations ciblés et de faire un chargement 'lazy'. En effet, contrairement à précédemment, les tables 'ingrédient' et 'plat' se communique entre elle par une documentation commune. Les informations sont ainsi chargées dans la mémoire.

Ce qui n'est pas le cas avec le SQL 'simple', qui fait un chargement agressif pour chercher l'élément 'frite' dans la table des ingrédients selon le plat donnée. Nouspuvons ainsi comparais le temps de traitement avec le format clé-valeur :

4	N - 1 - (- 0.00 070 00 - 100 111 500) (- 111 - 0.050 0.051 - 11 - 1)
1	Nested Loop (cost=0.00379.62 rows=420 width=580) (actual time=0.0580.061 rows=1 loops=1)
2	Join Filter: (SubPlan 1)
3	Rows Removed by Join Filter: 2
4	-> Seq Scan on plat p (cost=0.0011.40 rows=140 width=548) (actual time=0.0230.023 rows=3 loops=1)
5	-> Materialize (cost=0.0029.08 rows=6 width=32) (actual time=0.0070.008 rows=1 loops=3)
6	-> Seq Scan on ingredient i (cost=0.0029.05 rows=6 width=32) (actual time=0.0190.020 rows=1 loops=1)
7	Filter: ((data ->> 'nom'::text) = 'frite'::text)
8	Rows Removed by Filter: 19
9	SubPlan 1
10	-> ProjectSet (cost=0.000.52 rows=100 width=32) (actual time=0.0010.001 rows=2 loops=3)
11	-> Result (cost=0.000.01 rows=1 width=0) (actual time=0.0000.000 rows=1 loops=3)

Nous avons toujours un temps de traitement nettement plus court qu'avec le SQL 'simple'

Question 5 : afficher les ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une empreinte inférieure à un seuil donné.

Avec un système clé-valeur :

```
121 -- Ingrédients ayant la plus faible empreinte carbone
122 EXPLAIN ANALYZE
123 SELECT data->>'nom' AS nom_ingredient, data->>'emission' AS empreinte_carbone
124 FROM ingredient
125 WHERE data->>'emission' = (SELECT MIN(data->>'emission') FROM ingredient);
126
127
    -- Plats ayant la plus faible empreinte carbone
128 EXPLAIN ANALYZE
129 SELECT p.name AS nom_plat, i.data->>'nom' AS nom_ingredient, i.data->>'emission' AS empreinte_carbone
JOIN ingredient i ON i.data->>'nom' = ANY(SELECT jsonb_object_keys(p.ingredient))
132 WHERE i.data->>'emission' = (SELECT MIN(data->>'emission') FROM ingredient);
133
134
     -- Menus ayant la plus faible empreinte carbone
135 EXPLAIN ANALYZE
136 SELECT m.name AS nom_menu, p.name AS nom_plat, i.data->>'nom' AS nom_ingredient, i.data->>'emission' AS empreir
137 FROM menu m
138 JOIN plat p ON p.id = m.plat_id
139 JOIN ingredient i ON i.data->>'nom' = ANY(SELECT jsonb_object_keys(p.ingredient))
140 WHERE i.data->>'emission' = (SELECT MIN(data->>'emission') FROM ingredient)
   142 -- Ingrédients ayant une empreinte carbone inférieure à un seuil donné
   143 EXPLAIN ANALYZE
   144 SELECT data->>'nom' AS nom_ingredient, data->>'emission' AS empreinte_carbone
   145 FROM ingredient
   146 WHERE (data->>'emission')::numeric < 50
   147 UNION ALL
   148 SELECT p.name AS nom_plat, (i.data->>'emission')::text AS empreinte_carbone
   149 FROM plat p
   150 JOIN ingredient i ON i.data->>'nom' = ANY(SELECT jsonb_object_keys(p.ingredients))
   151 WHERE (i.data->>'emission')::numeric < 500
   152 UNION ALL
   153 SELECT m.name AS nom_menu, (i.data->>'emission')::text AS empreinte_carbone
   154 FROM menu m
   155 JOIN plat p ON p.id = m.plat_id
   156 JOIN ingredient i ON i.data->>'nom' = ANY(SELECT jsonb_object_keys(p.ingredients))
   157 WHERE (i.data->>'emission')::numeric < 550;</pre>
```

Nous avons pu inclure un ordre dans les seuils et déclaré un minimum d'émission de cardone dans chaque tableau. Ce qui n'a pas fonctionné avec le SQL simple.

Voici les performances pour le modèle clé-valeur sur la dernière requête :



Avec SQL simple:

10	Rows Removed by Filter: 2
11	Planning Time: 0.224 ms
12	Execution Time: 0.079 ms

Le temps long d'exécution pour le système clé-valeur est le fait que la table 'menu' n'est ps documenté en format JSON et la requête doit se retrancher dans du chargement agressif pour renvoyer la requête. Comme à la question 3, les performances en détail sous le système clé-valeur nous permets d'établir que les renvoie de requête entre table reste plus performant que lors de l'utilisation de SQL simple.

PostgreSQL avec un format JSON:

On a sensiblement la même chose que précédemment, mise à part sur la construction des tables ainsi que le remplissage de celles-ci qui est plus en détail. Voici un exemple avec la table 'ingredient'

```
70 INSERT INTO ingredient (nom, empreinte_carbone, data)
('beurre', 94.9, '{"type": "produit laitier", "catégorie": "beurre"}'),
               ('fromage_a_pate_molle', 107, '{"type": "produit laitier", "catégorie": "fromage à pâte molle"}'),
('fromage_a_pate_dure', 140, '{"type": "produit laitier", "catégorie": "fromage à pâte dure"}'),
('pain', 76, '{"type": "céréale", "catégorie": "pain"}'),
 77
 78
               ('yaourt', 360, '{"type": "produit laitier", "catégorie": "yaourt"}'),
               ('concombre', 129, '["type": "légume", "catégorie": "concombre"}'), ('bifteck', 5370, '{"type": "viande", "catégorie": "bœuf"}'),
 80
 81
                ('frite', 260, '{"type": "accompagnement", "catégorie": "frites"}'),
               ('farine', 46.9, '{"type": "céréale", "catégorie": "farine"}'),
 83
               ('Tarne', 40.9, '{"type": "cereate", "Categorie": "Tarne")',
('poire', 71, '{"type": "fruit", "categorie": "poire"}'),
('huile_cuilleree', 32.3, '{"type": "huile", "categorie": "cuillère à soupe"}'),
('huile_demi_cuilleree', 15.1, '{"type": "huile", "catégorie": "demi cuillère à soupe"}'),
('deux_oeufs', 276.7, '{"type": "œuf", "catégorie": "deux"}'),
('pomme_de_terre', 15.5, '{"type": "légume", "catégorie": "pomme de terre"}'),
 85
 87
 88
               ('huile_demi_cuil_vegan', 17.1, '{"type": "huile", "catégorie": "demi cuillère à soupe végétalienne"}'), ('fruits_saison', 53.4, '{"type": "fruit", "catégorie": "de saison"}');
1 ---- table JSON ingredient
 2 CREATE TABLE ingredient (
             nom TEXT PRIMARY KEY,
 4
              empreinte_carbone FLOAT,
 5
               data JSONB
 6);
 7
       --- création des tables JSON plat
 8
 9 CREATE TABLE entree (
            id INT PRIMARY KEY,
10
             nom TEXT,
11
12
             empreinte_carbone FLOAT,
               data JSONB
13
```

Nous allons donner ici les deux premières questions sous format JSON des requêtes et nous nous focaliserons surtout sur la différence de performance entre le modèle SQL et en format JSON sous PostgreSQL.

Lorsque vous exécutez la commande "EXPLAIN ANALYSE" devant vos requêtes en utilisant les formats JSON et SQL sous PostgreSQL, il est possible d'observer une différence de "planning time" entre les deux.

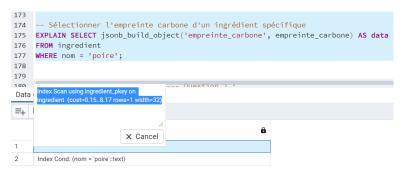
Il est important de noter que le "planning time" n'est qu'une partie du temps total d'exécution de la requête et ne reflète pas nécessairement la performance globale. D'autres facteurs tels que le temps d'exécution réel (execution time), le nombre de lignes affectées, les indices utilisés, etc., sont également des aspects importants à prendre en compte pour évaluer les performances d'une requête.

Question 1 : empreinte carbone d'un ingrédient donné.

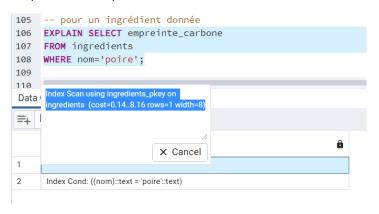
Dans ces requêtes, nous utilisons la fonction jsonb_build_object pour créer un objet JSON contenant uniquement la colonne empreinte_carbone de la table ingredients Le résultat est renvoyé sous forme de données JSON

→ Comparaison de temps :

Requête SQL JSON:



Requête SQL simple:



On remarque que pour une simple requête, la coût est plus élevé en JSON qu'en SQL simple. Cela est dû que l'on utilise la fonction jsonb_build_object.

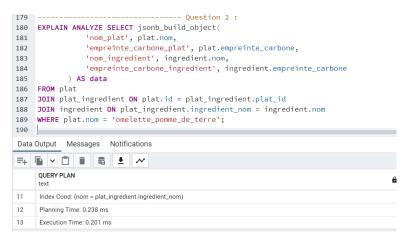
Question 2 : quelle est l'empreinte carbone d'un plat donné et quels sont les ingrédients composant un plat (avec leur empreinte carbone) ?

Nous avons de même, le même format que précédemment :

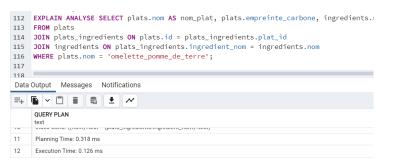


→ Comparaison de temps et de performance entre les deux méthodes :

JSON:



SQL:



Le temps d'exécution en SQL est plus court mais le temps d'exécution pour déterminer la requête est plus court en JSON. La différence de temps de planification (planning time) peut être attribuée aux différences inhérentes entre les deux formats de requêtes.

Lorsque nous utilisons le format SQL traditionnel, le moteur de requête PostgreSQL doit analyser la syntaxe SQL, vérifier la validité des objets référencés, déterminer les plans d'exécution possibles et effectuer d'autres opérations de planification spécifiques à SQL. Ces étapes supplémentaires peuvent prendre un certain temps et se traduire par un "planning time" plus élevé.

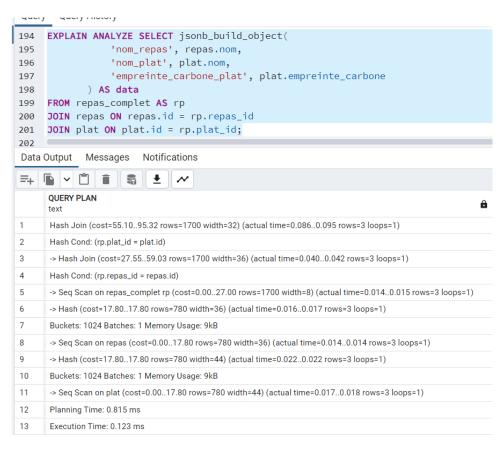
En revanche, sous le format JSON, on a directement une structure de données JSON à l'aide de fonctions JSON spécifiques de PostgreSQL, telles que jsonb_build_object. La planification des

opérations sur les données JSON peut être plus rapide car elles sont effectuées de manière plus directe et plus optimisée pour la manipulation de données JSON.

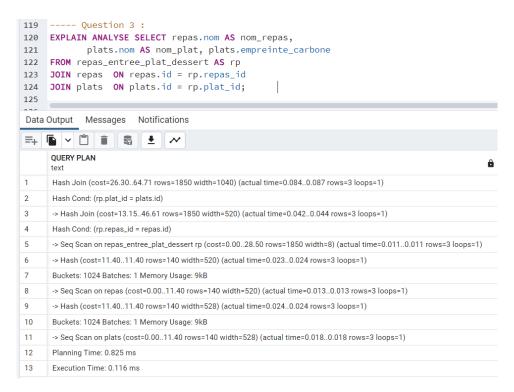
Question 3 : quelles sont la composition et l'empreinte carbone de chacun des plats composant un menu (avec les détails sur la composition des plats) ?

On va se limiter à la comparaison de performance entre les deux modèles.

JSON:



SQL simple:



En termes de coût et de durée, la requête SQL est gagnante sur ces point-là.

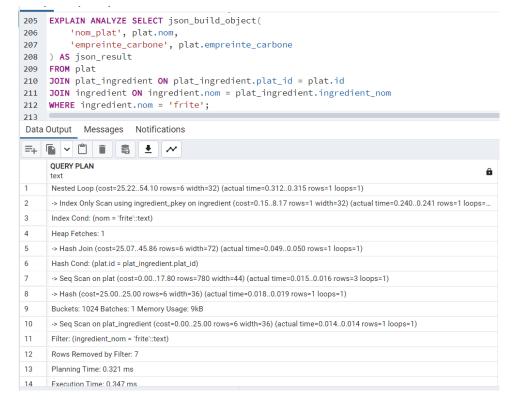
En revanche pour le planning time et le nombre de colonne ainsi que leur largeur, SQL en use deux fois plus. La requête SQL génère deux fois plus de colonnes avec des largeurs deux fois plus grande que lorsque l'on utilise le format JSON.

On voit plus clairement avec cette exemple le lien entre l'indicateur "planning time" et la gestion des colonnes avec les deux méthodes.

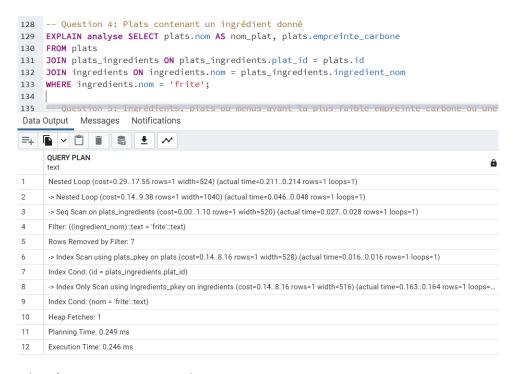
En petite donnée comme ici, on peut ainsi comprendre que le temps d'exécution est bâtue par SQL. Mais du des donnée massive, la requête SQL simple ne pourra juste pas finir le programme. Ce qui ne sera pas le cas avec la requête JSON, qui impose une structure initiale aux données.

Question 4 : Afficher les plats (avec leur empreinte carbone) contenant un ingrédient donné. De même.

JSON:



SQL:



Nous avons le même constat que pour la question 3.

Question 5 : afficher les ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une empreinte inférieure à un seuil donné.

Pour de soucis d'équité, nous avons retiré le 'order by' qui ne fonctionnais pas avec JSON. Cela n'affectait pas initialement la requête SQL qui l'ignorait déjà.

Nous suivons ici aussi exclusivement l'étude de différence de performance entre les modèles.

SQL:

```
EXPLAIN ANALYZE SELECT ingredients.nom AS nom_ingredients, ingredients.empreinte_carbone, 'ingredient' AS type
FROM ingredients
WHERE ingredients.empreinte_carbone < 50
UNION ALL
SELECT plats.nom AS nom_plat, plats.empreinte_carbone, 'plat' AS type
FROM plats
WHERE plats.empreinte_carbone < 500
UNION ALL
SELECT repas.nom AS menu, repas.total_empreinte_carbone, 'menu' AS type
FROM repas
WHERE repas.total_empreinte_carbone < 550;
```

	QUERY PLAN text
1	Append (cost=0.0037.37 rows=141 width=556) (actual time=0.0270.054 rows=8 loops=1)
2	-> Seq Scan on ingredients (cost=0.0011.75 rows=47 width=556) (actual time=0.0260.028 rows=6 loops=1)
3	Filter: (empreinte_carbone < '50'::double precision)
4	Rows Removed by Filter: 14
5	-> Seq Scan on plats (cost=0.0011.75 rows=47 width=556) (actual time=0.0120.013 rows=1 loops=1)
6	Filter: (empreinte_carbone < '500'::double precision)
7	Rows Removed by Filter: 2
8	-> Seq Scan on repas (cost=0.0011.75 rows=47 width=556) (actual time=0.0100.010 rows=1 loops=1)
9	Filter: (total_empreinte_carbone < '550'::double precision)
10	Rows Removed by Filter: 2
11	Planning Time: 0.224 ms
12	Execution Time: 0.079 ms

JSON:

```
216 EXPLAIN ANALYZE SELECT json_build_object(
         'nom_ingredient', ingredient.nom,
         \verb|'empreinte_carbone'|, ingredient.empreinte_carbone|,
218
219
         'type', 'ingredient'
220 ) AS data
221 FROM ingredient
222 WHERE ingredient.empreinte_carbone < 50
223 UNION ALL
224 SELECT json_build_object(
225
         'nom_plat', plat.nom,
226
         'empreinte_carbone', plat.empreinte_carbone,
         'type', 'plat'
227
228 ) AS data
229 FROM plat
230 WHERE plat.empreinte_carbone < 500
231 UNION ALL
232 SELECT json_build_object(
233
         'nom_menu', repas.nom,
         'empreinte_carbone', repas.empreinte_carbone,
234
235
         'type', 'menu'
236 ) AS data
237 FROM repas
238 WHERE repas.empreinte_carbone < 550;
239
```

	QUERY PLAN text
1	Append (cost=0.0073.45 rows=790 width=32) (actual time=0.0350.072 rows=8 loops=1)
2	-> Seq Scan on ingredient (cost=0.0020.80 rows=270 width=32) (actual time=0.0330.044 rows=6 loops=1)
3	Filter: (empreinte_carbone < '50'::double precision)
4	Rows Removed by Filter: 14
5	-> Seq Scan on plat (cost=0.0020.40 rows=260 width=32) (actual time=0.0120.013 rows=1 loops=1)
6	Filter: (empreinte_carbone < '500'::double precision)
7	Rows Removed by Filter: 2
8	-> Seq Scan on repas (cost=0.0020.40 rows=260 width=32) (actual time=0.0120.013 rows=1 loops=1)
9	Filter: (empreinte_carbone < '550'::double precision)
10	Rows Removed by Filter: 2
11	Planning Time: 0.282 ms
12	Execution Time: 0.098 ms

Les performances de la requête SQL simple son bien plus satisfaisantes que celles sous format JSON. Cela s'explique que dans ce cas précis, en SQL nous avons déjà formé les colonnes et qu'en JSON on utilise une fonction 'json_build_object' qui coûte plus cher que ce dont on a besoin puisque, que ce soit avec beaucoup de donnée ou avec peu de donnée, le formalisme reste le même dans les deux méthodes.

On en déduit que l'utilisation de 'json_build_object' est coûteux, puisqu'ici il réorganise quoi qu'il arrive les colonnes