

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, PostgreSQL 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	Tzatziki	507.2
	- Légumes de saison	53.4	- yaourt	360
	- huile d'olive (1/2 cuillère)	18.2	- concombre	129
			- huile d'olive (1/2 cuillère)	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 cuillère)	15.1
Plat	Omelette aux pommes de terre	309.3
	- deux œufs	276.7
	- pomme de terre	15.5
	- huile (1/2 cuillère)	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 menu et les ingrédients associés (avec leur empreinte 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 :
46 INSERT INTO plats (id, nom, empreinte_carbone)
47 VALUES
48     (12, 'omelette_pomme_de_terre', 309.3),
49     (22, 'poulet_au_riz', 952.1),
50     (32, 'poulet_au_riz', 5630);
51
52 UPDATE plats
53 SET nom = 'bifteck_frite'
54 WHERE id = 32;

```

Question 1 : empreinte carbone d'un ingrédient donné.

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

```

100 ---- les requêtes ----
101
102 ----- Question 1 :
103 SELECT empreinte_carbone
104 FROM ingredients;
105 -- pour un ingrédient donnée
106 SELECT empreinte_carbone
107 FROM ingredients
108 WHERE nom='poire';
109

```

Data Output Messages Notifications

empreinte_carbone double precision	
1	71

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

```

111 ----- Question 2 :
112 SELECT plats.nom AS nom_plat, plats.empreinte_carbone, ingredients.nom AS nom_ingredient, ingredients.empreinte_carbone
113 FROM plats
114 JOIN plats_ingredients ON plats.id = plats_ingredients.plat_id
115 JOIN ingredients ON plats_ingredients.ingredient_nom = ingredients.nom
116 WHERE plats.nom = 'omelette_pomme_de_terre';
117
118

```

	nom_plat character varying (255)	empreinte_carbone double precision	nom_ingredient character varying (255)	empreinte_carbone double precision
1	omelette_pomme_de_terre	309.3	deux_oeufs	276.7
2	omelette_pomme_de_terre	309.3	pomme_de_terre	15.5
3	omelette_pomme_de_terre	309.3	huile_demi_cuil_vegan	17.1

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) ?

```

123 ----- Question 3 :
124 SELECT repas.nom AS nom_repas,
125         plats.nom AS nom_plat, plats.empreinte_carbone
126 --      ingredients.nom AS nom_ingredient, ingredients.empreinte_carbone
127 FROM repas_entree_plat_dessert AS rp
128 JOIN repas ON repas.id = rp.repas_id
129 JOIN plats ON plats.id = rp.plat_id;
130

```

	nom_repas character varying (255)	nom_plat character varying (255)	empreinte_carbone double precision
1	repas_vegetarien	omelette_pomme_de_terre	309.3
2	repas_classique	poulet_au_riz	952.1
3	repas_classique_bis	bifteck_frite	5630

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

```

133 -- Question 4: Plats contenant un ingrédient donné
134 SELECT plats.nom AS nom_plat, plats.empreinte_carbone
135 FROM plats
136 JOIN plats_ingredients ON plats_ingredients.plat_id = plats.id
137 JOIN ingredients ON ingredients.nom = plats_ingredients.ingredient_nom
138 WHERE ingredients.nom = 'frite';
139

```

	nom_plat character varying (255)	empreinte_carbone double precision
1	bifteck_frite	5630

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

```
140 -- Question 5: Ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une empreinte
141 SELECT ingredients.nom AS nom_ingredients, ingredients.empreinte_carbone, 'ingredient' AS type
142 FROM ingredients
143 WHERE ingredients.empreinte_carbone < 50
144 UNION ALL
145 SELECT plats.nom AS nom_plat, plats.empreinte_carbone, 'plat' AS type
146 FROM plats
147 WHERE plats.empreinte_carbone < 300
148 UNION ALL
149 SELECT repas.nom AS menu, repas.total_empreinte_carbone, 'menu' AS type
150 FROM repas
151 WHERE repas.total_empreinte_carbone < 550
152 ORDER BY empreinte_carbone;
```

Data Output			
	nom_ingredients	empreinte_carbone	type
	character varying (255)	double precision	text
1	huile_demi_cuilleree	15.1	ingredient
2	pomme_de_terre	15.5	ingredient
3	huile_demi_cuil_vegan	17.1	ingredient
4	huile	18.2	ingredient
5	huile_cuilleree	32.3	ingredient
6	farine	46.9	ingredient
7	repas_vegetarien	510	menu

Autre changement de seuil pour faire paraître les plats :

```
140 -- Question 5: Ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une emprei
141 SELECT ingredients.nom AS nom_ingredients, ingredients.empreinte_carbone, 'ingredient' AS type
142 FROM ingredients
143 WHERE ingredients.empreinte_carbone < 50
144 UNION ALL
145 SELECT plats.nom AS nom_plat, plats.empreinte_carbone, 'plat' AS type
146 FROM plats
147 WHERE plats.empreinte_carbone < 500
148 UNION ALL
149 SELECT repas.nom AS menu, repas.total_empreinte_carbone, 'menu' AS type
150 FROM repas
151 WHERE repas.total_empreinte_carbone < 550
152 ORDER BY empreinte_carbone;
```

Data Output			
	nom_ingredients	empreinte_carbone	type
	character varying (255)	double precision	text
1	huile_demi_cuilleree	15.1	ingredient
2	pomme_de_terre	15.5	ingredient
3	huile_demi_cuil_vegan	17.1	ingredient
4	huile	18.2	ingredient
5	huile_cuilleree	32.3	ingredient
6	farine	46.9	ingredient
7	omelette_pomme_de_terre	309.3	plat
8	repas_vegetarien	510	menu

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"
2  CREATE TABLE ingredient (
3      id SERIAL PRIMARY KEY,
4      data JSONB
5  );
6
7
8  -- Table "entree"
9  CREATE TABLE entree (
10     id SERIAL PRIMARY KEY,
11     name VARCHAR(255) NOT NULL,
12     ingredients JSONB NOT NULL
13 );
14
15 -- Table "plat"
16 CREATE TABLE plat (
17     id SERIAL PRIMARY KEY,
18     name VARCHAR(255) NOT NULL,
19     ingredients JSONB NOT NULL
20 );
21
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 (
31     id SERIAL PRIMARY KEY,
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),
37     FOREIGN KEY (plat_id) REFERENCES plat(id),
38     FOREIGN KEY (dessert_id) REFERENCES dessert(id)
39 );

```

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 :

```

42 INSERT INTO ingredient (data) VALUES
43     ('{"nom": "légume de saison", "emission": 53.4}'),
44     ('{"nom": "huile d'olive (1/2 cuillère)", "emission": 18.2}'),
45     ('{"nom": "poulet", "emission": 774}'),
46     ('{"nom": "riz", "emission": 84.6}'),
47     ('{"nom": "beurre", "emission": 94.9}'),
48     ('{"nom": "fromage à pâte molle", "emission": 107}'),
49     ('{"nom": "fromage à pâte dure", "emission": 140}'),
50     ('{"nom": "pain", "emission": 76}'),
51     ('{"nom": "yaourt", "emission": 360}'),
52     ('{"nom": "concombre", "emission": 129}'),
53     ('{"nom": "bifteck", "emission": 5370}'),
54     ('{"nom": "frite", "emission": 260}'),
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}'),
59     ('{"nom": "pomme de terre", "emission": 15.5}'),
60     ('{"nom": "huile ½ cuillère", "emission": 17.1}'),
61     ('{"nom": "fruits de saison", "emission": 53.4}'),
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 cuillère)": 18.2}'),
69     ('tzatziki', '{"yaourt": 360, "concombre": 129, "huile d'olive (1/2 cuillère)": 18.2}'),
70     ('soupe de légumes', '{"légume de saison": 53.4, "huile ½ cuillère": 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 :

```

89 ----- Question 1
90 EXPLAIN ANALYZE
91 SELECT data->>'nom' AS nom_ingredient, data->>'emission' AS empreinte_carbone
92 FROM ingredient
93 WHERE data->>'nom' = 'farine';

```

Data Output		Messages	Notifications
<div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>🗑️</div> <div>🔄</div> <div>⬇️</div> <div>📈</div> </div> <div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div> </div>			
1	Seq Scan on ingredient (cost=0.00..29.08 rows=6 width=64) (actual time=0.055..0.057 rows=1 loops...		
2	Filter: ((data->>'nom')::text) = 'farine'::text)		
3	Rows Removed by Filter: 19		
4	Planning Time: 0.127 ms		
5	Execution Time: 0.077 ms		

Sous PostgreSQL 'classique' :

```

105 -- pour un ingrédient donnée
106 EXPLAIN SELECT empreinte_carbone
107 FROM ingredients
108 WHERE nom='poire';
109
110

```

Data

Index Scan using ingredients_pkey on ingredients (cost=0.14..8.16 rows=1 width=8)

X Cancel

1

2 Index Cond: ((nom)::text = 'poire'::text)

On remarque la comparaison avec le SQL 'classique', 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 documentation 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 :

```

96 --- Question 2
97 EXPLAIN ANALYZE
98 SELECT p.name AS nom_plat,
99 i.data->'nom' AS nom_ingredient,
100 i.data->'emission' AS empreinte_carbone
101 FROM plat p
102 JOIN ingredient i ON i.data->'nom' = ANY(SELECT jsonb_object_keys(p.ingredients))
103 WHERE p.name = 'omelette aux pommes de terre';

```

Data Output Messages Notifications

QUERY PLAN
text

11 Planning Time: 14.259 ms

12 Execution Time: 0.156 ms

Avec SQL classique sur PostgreSQL :

```

112 EXPLAIN ANALYZE SELECT plats.nom AS nom_plat, plats.empreinte_carbone, ingredients.i
113 FROM plats
114 JOIN plats_ingredients ON plats.id = plats_ingredients.plat_id
115 JOIN ingredients ON plats_ingredients.ingredient_nom = ingredients.nom
116 WHERE plats.nom = 'omelette_pomme_de_terre';
117
118

```

Data Output Messages Notifications

QUERY PLAN
text

11 Planning Time: 0.318 ms

12 Execution Time: 0.126 ms

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 :

```

105 -- Question 3
106 EXPLAIN ANALYZE
107 SELECT m.name AS nom_menu, p.name AS nom_plat, i.data->>'nom' AS nom_ingredient, i.data->>'emission' AS emprei
108 FROM menu m
109 JOIN plat p ON p.id = m.plat_id
110 JOIN ingredient i ON i.data->>'nom' = ANY(SELECT jsonb_object_keys(p.ingredients));
111 WHERE m.name = 'nom_menu_recherche';

```

	QUERY PLAN	
15	Planning Time: 30.918 ms	
16	Execution Time: 0.375 ms	

Avec SQL classique :

```

119 ----- Question 3 :
120 EXPLAIN ANALYZE SELECT repas.nom AS nom_repas,
121     plats.nom AS nom_plat, plats.empreinte_carbone
122 FROM repas_entree_plat_dessert AS rp
123 JOIN repas ON repas.id = rp.repas_id
124 JOIN plats ON plats.id = rp.plat_id;
125

```

	QUERY PLAN	
1	Hash Join (cost=26.30..64.71 rows=1850 width=1040) (actual time=0.084..0.087 rows=3 loops=1)	
2	Hash Cond: (rp.plat_id = plats.id)	
3	-> Hash Join (cost=13.15..46.61 rows=1850 width=520) (actual time=0.042..0.044 rows=3 loops=1)	
4	Hash Cond: (rp.repas_id = repas.id)	
5	-> Seq Scan on repas_entree_plat_dessert rp (cost=0.00..28.50 rows=1850 width=8) (actual time=0.011..0.011 rows=3 loops=1)	
6	-> Hash (cost=11.40..11.40 rows=140 width=520) (actual time=0.023..0.024 rows=3 loops=1)	
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
8	-> Seq Scan on repas (cost=0.00..11.40 rows=140 width=520) (actual time=0.013..0.013 rows=3 loops=1)	
9	-> Hash (cost=11.40..11.40 rows=140 width=528) (actual time=0.024..0.024 rows=3 loops=1)	
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
11	-> Seq Scan on plats (cost=0.00..11.40 rows=140 width=528) (actual time=0.018..0.018 rows=3 loops=1)	
12	Planning Time: 0.825 ms	
13	Execution Time: 0.116 ms	

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.

	QUERY PLAN text	
3	Rows Removed by Join Filter: 52	
4	-> Seq Scan on ingredient i (cost=0.00..22.70 rows=1270 width=32) (actual time=0.054..0.056 rows=20 loops=1)	
5	-> Materialize (cost=13.15..25.63 rows=140 width=1064) (actual time=0.006..0.006 rows=3 loops=20)	
6	-> Hash Join (cost=13.15..24.93 rows=140 width=1064) (actual time=0.105..0.108 rows=3 loops=1)	
7	Hash Cond: (m.plat_id = p.id)	
8	-> Seq Scan on menu m (cost=0.00..11.40 rows=140 width=520) (actual time=0.014..0.015 rows=3 loops=1)	
9	-> Hash (cost=11.40..11.40 rows=140 width=552) (actual time=0.036..0.036 rows=3 loops=1)	
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
11	-> Seq Scan on plat p (cost=0.00..11.40 rows=140 width=552) (actual time=0.030..0.030 rows=3 loops=1)	
12	SubPlan 1	
13	-> ProjectSet (cost=0.00..0.52 rows=100 width=32) (actual time=0.000..0.001 rows=3 loops=60)	
14	-> Result (cost=0.00..0.01 rows=1 width=0) (actual time=0.000..0.000 rows=1 loops=60)	

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 :

114

EXPLAIN ANALYZE

115

SELECT p.name AS nom_plat, i.data->>'nom' AS nom_ingredient, i.data->>'emission' AS empreinte_carbone

116

FROM plat p

117

JOIN ingredient i ON i.data->>'nom' = ANY(SELECT jsonb_object_keys(p.ingredients))

118

WHERE i.data->>'nom' = 'frite';

119

Data Output

Messages

Notifications

QUERY PLAN

text

12

Planning Time: 0.174 ms

13

Execution Time: 0.104 ms

Avec SQL simple :

128	-- Question 4: Plats contenant un ingrédient donné
129	EXPLAIN analyse SELECT plats.nom AS nom_plat, plats.empreinte_carbone
130	FROM plats
131	JOIN plats_ingredients ON plats_ingredients.plat_id = plats.id
132	JOIN ingredients ON ingredients.nom = plats_ingredients.ingredient_nom
133	WHERE ingredients.nom = 'frite';
134	
135	Question 5: Ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une

Data Output
Messages
Notifications

QUERY PLAN

text

1	Nested Loop (cost=0.29..17.55 rows=1 width=524) (actual time=0.211..0.214 rows=1 loops=1)
2	-> Nested Loop (cost=0.14..9.38 rows=1 width=1040) (actual time=0.046..0.048 rows=1 loops=1)
3	-> Seq Scan on plats_ingredients (cost=0.00..1.10 rows=1 width=520) (actual time=0.027..0.028 rows=1 loops=1)
4	Filter: ((ingredient_nom)::text = 'frite'::text)
5	Rows Removed by Filter: 7
6	-> Index Scan using plats_pkey on plats (cost=0.14..8.16 rows=1 width=528) (actual time=0.016..0.016 rows=1 loops=1)
7	Index Cond: (id = plats_ingredients.plat_id)
8	-> Index Only Scan using ingredients_pkey on ingredients (cost=0.14..8.16 rows=1 width=516) (actual time=0.163..0.164 rows=1 loops=1)
9	Index Cond: (nom = 'frite'::text)
10	Heap Fetches: 1
11	Planning Time: 0.249 ms
12	Execution Time: 0.246 ms

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. Nouspavons ainsi comparais le temps de traitement avec le format clé-valeur :

1	Nested Loop (cost=0.00..379.62 rows=420 width=580) (actual time=0.058..0.061 rows=1 loops=1)
2	Join Filter: (SubPlan 1)
3	Rows Removed by Join Filter: 2
4	-> Seq Scan on plat p (cost=0.00..11.40 rows=140 width=548) (actual time=0.023..0.023 rows=3 loops=1)
5	-> Materialize (cost=0.00..29.08 rows=6 width=32) (actual time=0.007..0.008 rows=1 loops=3)
6	-> Seq Scan on ingredient i (cost=0.00..29.05 rows=6 width=32) (actual time=0.019..0.020 rows=1 loops=1)
7	Filter: ((data -> 'nom'::text) = 'frite'::text)
8	Rows Removed by Filter: 19
9	SubPlan 1
10	-> ProjectSet (cost=0.00..0.52 rows=100 width=32) (actual time=0.001..0.001 rows=2 loops=3)
11	-> Result (cost=0.00..0.01 rows=1 width=0) (actual time=0.000..0.000 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
130 FROM plat p
131 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 empreinte_carbone
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;

```

Nous avons pu inclure un ordre dans les seuils et déclaré un minimum d'émission de carbone 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 :

```

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;
158

```

	Data Output	Messages	Notifications
	<div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>🗑️</div> <div>🔄</div> <div>⬇️</div> <div>📈</div> </div>		
	<div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div>		
34	Planning Time: 0.399 ms		
35	Execution Time: 13.531 ms		

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 pas documentée 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 permettent d'établir que le renvoi 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)
71 VALUES ('legume_de_saison', 53.4, '{"type": "légume", "catégorie": "de saison"}'),
72 ('huile', 18.2, '{"type": "huile", "catégorie": "général"}'),
73 ('poulet', 774, '{"type": "viande", "catégorie": "poulet"}'),
74 ('riz', 84.6, '{"type": "céréale", "catégorie": "de base"}'),
75 ('beurre', 94.9, '{"type": "produit laitier", "catégorie": "beurre"}'),
76 ('fromage_a_pate_molle', 107, '{"type": "produit laitier", "catégorie": "fromage à pâte molle"}'),
77 ('fromage_a_pate_dure', 140, '{"type": "produit laitier", "catégorie": "fromage à pâte dure"}'),
78 ('pain', 76, '{"type": "céréale", "catégorie": "pain"}'),
79 ('yaourt', 360, '{"type": "produit laitier", "catégorie": "yaourt"}'),
80 ('concombre', 129, '{"type": "légume", "catégorie": "concombre"}'),
81 ('bifteck', 5370, '{"type": "viande", "catégorie": "bœuf"}'),
82 ('frite', 260, '{"type": "accompagnement", "catégorie": "frites"}'),
83 ('farine', 46.9, '{"type": "céréale", "catégorie": "farine"}'),
84 ('poire', 71, '{"type": "fruit", "catégorie": "poire"}'),
85 ('huile_cuilleree', 32.3, '{"type": "huile", "catégorie": "cuillère à soupe"}'),
86 ('huile_demi_cuilleree', 15.1, '{"type": "huile", "catégorie": "demi cuillère à soupe"}'),
87 ('deux_oeufs', 276.7, '{"type": "œuf", "catégorie": "deux"}'),
88 ('pomme_de_terre', 15.5, '{"type": "légume", "catégorie": "pomme de terre"}'),
89 ('huile_demi_cuil_vegan', 17.1, '{"type": "huile", "catégorie": "demi cuillère à soupe végétalienne"}'),
90 ('fruits_saison', 53.4, '{"type": "fruit", "catégorie": "de saison"}');
```

```
1 ---- table JSON ingredient
2 CREATE TABLE ingredient (
3     nom TEXT PRIMARY KEY,
4     empreinte_carbone FLOAT,
5     data JSONB
6 );
7
8 --- création des tables JSON plat
9 CREATE TABLE entree (
10     id INT PRIMARY KEY,
11     nom TEXT,
12     empreinte_carbone FLOAT,
13     data JSONB
14 );
```

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.

```
174 -- Sélectionner l'empreinte carbone d'un ingrédient spécifique |
175 SELECT jsonb_build_object('empreinte_carbone', empreinte_carbone) AS data
176 FROM ingredient
177 WHERE nom = 'poire';
178
179
180
```

Data Output Messages Notifications

	data jsonb
1	{\"empreinte_carbone\": 71}

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 :

```
173
174 -- Sélectionner l'empreinte carbone d'un ingrédient spécifique
175 EXPLAIN SELECT jsonb_build_object('empreinte_carbone', empreinte_carbone) AS data
176 FROM ingredient
177 WHERE nom = 'poire';
178
179
180
```

Data

Index Scan using ingredient_pkey on ingredient (cost=0.15..8.17 rows=1 width=32)

1	
2	Index Cond: (nom = 'poire')::text

Requête SQL simple :

```
105 -- pour un ingrédient donnée
106 EXPLAIN SELECT empreinte_carbone
107 FROM ingredients
108 WHERE nom='poire';
109
110
```

Data

Index Scan using ingredients_pkey on ingredients (cost=0.14..8.16 rows=1 width=8)

1	
2	Index Cond: ((nom)::text = 'poire')::text

On remarque que pour une simple requête, le coût est plus élevé en JSON qu'en SQL simple. Cela est dû au fait 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 :

180	----- Question 2 :
181	SELECT jsonb_build_object(
182	'nom_plat', plat.nom,
183	'empreinte_carbone_plat', plat.empreinte_carbone,
184	'nom_ingredient', ingredient.nom,
185	'empreinte_carbone_ingredient', ingredient.empreinte_carbone
186) AS data
187	FROM plat
188	JOIN plat_ingredient ON plat.id = plat_ingredient.plat_id
189	JOIN ingredient ON plat_ingredient.ingredient_nom = ingredient.nom
190	WHERE plat.nom = 'omelette_pomme_de_terre';
191	
192	

Data Output	Messages	Notifications
-------------	----------	---------------

	data	jsonb
1	{	'nom_plat': 'omelette_pomme_de_terre', 'nom_ingredient': 'deux_oeufs', 'empreinte_carbone_plat': 309.3, 'empreinte_carbone_ingredient': 276.7}
2	{	'nom_plat': 'omelette_pomme_de_terre', 'nom_ingredient': 'pomme_de_terre', 'empreinte_carbone_plat': 309.3, 'empreinte_carbone_ingredient': 15.5}
3	{	'nom_plat': 'omelette_pomme_de_terre', 'nom_ingredient': 'huile_demi_cuil_vegan', 'empreinte_carbone_plat': 309.3, 'empreinte_carbone_ingredient': ...

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

JSON :

179	----- Question 2 :
180	EXPLAIN ANALYZE SELECT jsonb_build_object(
181	'nom_plat', plat.nom,
182	'empreinte_carbone_plat', plat.empreinte_carbone,
183	'nom_ingredient', ingredient.nom,
184	'empreinte_carbone_ingredient', ingredient.empreinte_carbone
185) AS data
186	FROM plat
187	JOIN plat_ingredient ON plat.id = plat_ingredient.plat_id
188	JOIN ingredient ON plat_ingredient.ingredient_nom = ingredient.nom
189	WHERE plat.nom = 'omelette_pomme_de_terre';
190	

Data Output	Messages	Notifications
-------------	----------	---------------

	QUERY PLAN
11	Index Cond: (nom = plat_ingredient.ingredient_nom)
12	Planning Time: 0.238 ms
13	Execution Time: 0.201 ms

SQL :

112	EXPLAIN ANALYZE SELECT plats.nom AS nom_plat, plats.empreinte_carbone, ingredients.
113	FROM plats
114	JOIN plats_ingredients ON plats.id = plats_ingredients.plat_id
115	JOIN ingredients ON plats_ingredients.ingredient_nom = ingredients.nom
116	WHERE plats.nom = 'omelette_pomme_de_terre';
117	
118	

Data Output	Messages	Notifications
-------------	----------	---------------

	QUERY PLAN
11	Planning Time: 0.318 ms
12	Execution Time: 0.126 ms

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 :

194	EXPLAIN ANALYZE SELECT jsonb_build_object(195 'nom_repas', repas.nom, 196 'nom_plat', plat.nom, 197 'empreinte_carbone_plat', plat.empreinte_carbone 198) AS data 199 FROM repas_complet AS rp 200 JOIN repas ON repas.id = rp.repas_id 201 JOIN plat ON plat.id = rp.plat_id; 202
Data Output Messages Notifications	
<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>	
	QUERY PLAN text
1	Hash Join (cost=55.10..95.32 rows=1700 width=32) (actual time=0.086..0.095 rows=3 loops=1)
2	Hash Cond: (rp.plat_id = plat.id)
3	-> Hash Join (cost=27.55..59.03 rows=1700 width=36) (actual time=0.040..0.042 rows=3 loops=1)
4	Hash Cond: (rp.repas_id = repas.id)
5	-> Seq Scan on repas_complet rp (cost=0.00..27.00 rows=1700 width=8) (actual time=0.014..0.015 rows=3 loops=1)
6	-> Hash (cost=17.80..17.80 rows=780 width=36) (actual time=0.016..0.017 rows=3 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB
8	-> Seq Scan on repas (cost=0.00..17.80 rows=780 width=36) (actual time=0.014..0.014 rows=3 loops=1)
9	-> Hash (cost=17.80..17.80 rows=780 width=44) (actual time=0.022..0.022 rows=3 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on plat (cost=0.00..17.80 rows=780 width=44) (actual time=0.017..0.018 rows=3 loops=1)
12	Planning Time: 0.815 ms
13	Execution Time: 0.123 ms

SQL simple :

119	----- Question 3 :
120	EXPLAIN ANALYSE SELECT repas.nom AS nom_repas,
121	plats.nom AS nom_plat, plats.empreinte_carbone
122	FROM repas_entree_plat_dessert AS rp
123	JOIN repas ON repas.id = rp.repas_id
124	JOIN plats ON plats.id = rp.plat_id;
125	
Data Output Messages Notifications	
<div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> </div>	
QUERY PLAN	
text	
1	Hash Join (cost=26.30..64.71 rows=1850 width=1040) (actual time=0.084..0.087 rows=3 loops=1)
2	Hash Cond: (rp.plat_id = plats.id)
3	-> Hash Join (cost=13.15..46.61 rows=1850 width=520) (actual time=0.042..0.044 rows=3 loops=1)
4	Hash Cond: (rp.repas_id = repas.id)
5	-> Seq Scan on repas_entree_plat_dessert rp (cost=0.00..28.50 rows=1850 width=8) (actual time=0.011..0.011 rows=3 loops=1)
6	-> Hash (cost=11.40..11.40 rows=140 width=520) (actual time=0.023..0.024 rows=3 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB
8	-> Seq Scan on repas (cost=0.00..11.40 rows=140 width=520) (actual time=0.013..0.013 rows=3 loops=1)
9	-> Hash (cost=11.40..11.40 rows=140 width=528) (actual time=0.024..0.024 rows=3 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on plats (cost=0.00..11.40 rows=140 width=528) (actual time=0.018..0.018 rows=3 loops=1)
12	Planning Time: 0.825 ms
13	Execution Time: 0.116 ms

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 bâtué 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 :

205	EXPLAIN ANALYZE SELECT json_build_object(
206	'nom_plat', plat.nom,
207	'empreinte_carbone', plat.empreinte_carbone
208) AS json_result
209	FROM plat
210	JOIN plat_ingredient ON plat_ingredient.plat_id = plat.id
211	JOIN ingredient ON ingredient.nom = plat_ingredient.ingredient_nom
212	WHERE ingredient.nom = 'frite';
213	
Data Output Messages Notifications	
<div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> </div>	
<div> <div>QUERY PLAN</div> <div>text</div> <div></div> </div>	
1	Nested Loop (cost=25.22..54.10 rows=6 width=32) (actual time=0.312..0.315 rows=1 loops=1)
2	-> Index Only Scan using ingredient_pkey on ingredient (cost=0.15..8.17 rows=1 width=32) (actual time=0.240..0.241 rows=1 loops=...
3	Index Cond: (nom = 'frite'::text)
4	Heap Fetches: 1
5	-> Hash Join (cost=25.07..45.86 rows=6 width=72) (actual time=0.049..0.050 rows=1 loops=1)
6	Hash Cond: (plat.id = plat_ingredient.plat_id)
7	-> Seq Scan on plat (cost=0.00..17.80 rows=780 width=44) (actual time=0.015..0.016 rows=3 loops=1)
8	-> Hash (cost=25.00..25.00 rows=6 width=36) (actual time=0.018..0.019 rows=1 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 9kB
10	-> Seq Scan on plat_ingredient (cost=0.00..25.00 rows=6 width=36) (actual time=0.014..0.014 rows=1 loops=1)
11	Filter: (ingredient_nom = 'frite'::text)
12	Rows Removed by Filter: 7
13	Planning Time: 0.321 ms
14	Execution Time: 0.347 ms

SQL :

128	-- Question 4: Plats contenant un ingrédient donné
129	EXPLAIN analyse SELECT plats.nom AS nom_plat, plats.empreinte_carbone
130	FROM plats
131	JOIN plats_ingredients ON plats_ingredients.plat_id = plats.id
132	JOIN ingredients ON ingredients.nom = plats_ingredients.ingredient_nom
133	WHERE ingredients.nom = 'frite';
134	
135	-- Question 5: Ingrédients, plats ou menus ayant la plus faible empreinte carbone ou une
Data Output Messages Notifications	
<div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> </div>	
<div> <div>QUERY PLAN</div> <div>text</div> <div></div> </div>	
1	Nested Loop (cost=0.29..17.55 rows=1 width=524) (actual time=0.211..0.214 rows=1 loops=1)
2	-> Nested Loop (cost=0.14..9.38 rows=1 width=1040) (actual time=0.046..0.048 rows=1 loops=1)
3	-> Seq Scan on plats_ingredients (cost=0.00..1.10 rows=1 width=520) (actual time=0.027..0.028 rows=1 loops=1)
4	Filter: ((ingredient_nom)::text = 'frite'::text)
5	Rows Removed by Filter: 7
6	-> Index Scan using plats_pkey on plats (cost=0.14..8.16 rows=1 width=528) (actual time=0.016..0.016 rows=1 loops=1)
7	Index Cond: (id = plats_ingredients.plat_id)
8	-> Index Only Scan using ingredients_pkey on ingredients (cost=0.14..8.16 rows=1 width=516) (actual time=0.163..0.164 rows=1 loops=...
9	Index Cond: (nom = 'frite'::text)
10	Heap Fetches: 1
11	Planning Time: 0.249 ms
12	Execution Time: 0.246 ms

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_ingredient, 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=0.00..37.37 rows=141 width=556) (actual time=0.027..0.054 rows=8 loops=1)
2	-> Seq Scan on ingredients (cost=0.00..11.75 rows=47 width=556) (actual time=0.026..0.028 rows=6 loops=1)
3	Filter: (empreinte_carbone < '50'::double precision)
4	Rows Removed by Filter: 14
5	-> Seq Scan on plats (cost=0.00..11.75 rows=47 width=556) (actual time=0.012..0.013 rows=1 loops=1)
6	Filter: (empreinte_carbone < '500'::double precision)
7	Rows Removed by Filter: 2
8	-> Seq Scan on repas (cost=0.00..11.75 rows=47 width=556) (actual time=0.010..0.010 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(
217     'nom_ingredient', ingredient.nom,
218     'empreinte_carbone', ingredient.empreinte_carbone,
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,
227     'type', 'plat'
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,
234     'empreinte_carbone', repas.empreinte_carbone,
235     'type', 'menu'
236 ) AS data
237 FROM repas
238 WHERE repas.empreinte_carbone < 550;
239
```

	QUERY PLAN	
	text	
1	Append (cost=0.00..73.45 rows=790 width=32) (actual time=0.035..0.072 rows=8 loops=1)	
2	-> Seq Scan on ingredient (cost=0.00..20.80 rows=270 width=32) (actual time=0.033..0.044 rows=6 loops=1)	
3	Filter: (empreinte_carbone < '50'::double precision)	
4	Rows Removed by Filter: 14	
5	-> Seq Scan on plat (cost=0.00..20.40 rows=260 width=32) (actual time=0.012..0.013 rows=1 loops=1)	
6	Filter: (empreinte_carbone < '500'::double precision)	
7	Rows Removed by Filter: 2	
8	-> Seq Scan on repas (cost=0.00..20.40 rows=260 width=32) (actual time=0.012..0.013 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