Base de données avancées
Devoir maison

SAHIN Tolga 22100432
Gr3

1) Précisions sur l'exécution

Le projet est fonctionnel sur mon PC personnel, il a été codé sur les versions :
- Python 3.7.3
- Numpy : 1.16.5
- Pandas : 0.23.3
- Psycopg2 : 2.8.6 (dt dec pq3 ext lo64)

Donc, au cas où, une erreur à l'exécution se présente, c'est sûrement en cause de versions trop récentes. J'ai utilisé StringIO, qui ne marchera pas sur les dernières versions de Python, une fonction utilise pandas et numpy et leur version est quasi-interdépendante aussi.
En tout cas ça suit les mêmes versions qu'au CREMI normalement.

1.2) Structure du projet

Les fichiers pythons exécutables sont dans le répertoire code.
Il faut configurer la connexion sur sa base de données avec en modifiant les variables dans le dictionnaire params. (ligne 24 pour imports.py, ligne 8 pour requests.py)
Ensuite simplement, exécuter avec les commandes, ou un IDE :
- python3 imports.py
- python3 requests.py

En ordre, imports.py est le fichier de création de la base de données et d'importation de la base de données via les différents datasets. Le fichier import.py contient de nombreuses fonctions permettant d'importer génériquement des données en CSV ou XLS.

requests.py contient quelques requêtes répondant à la première question d'essayer certaines requêtes via psycopg2.

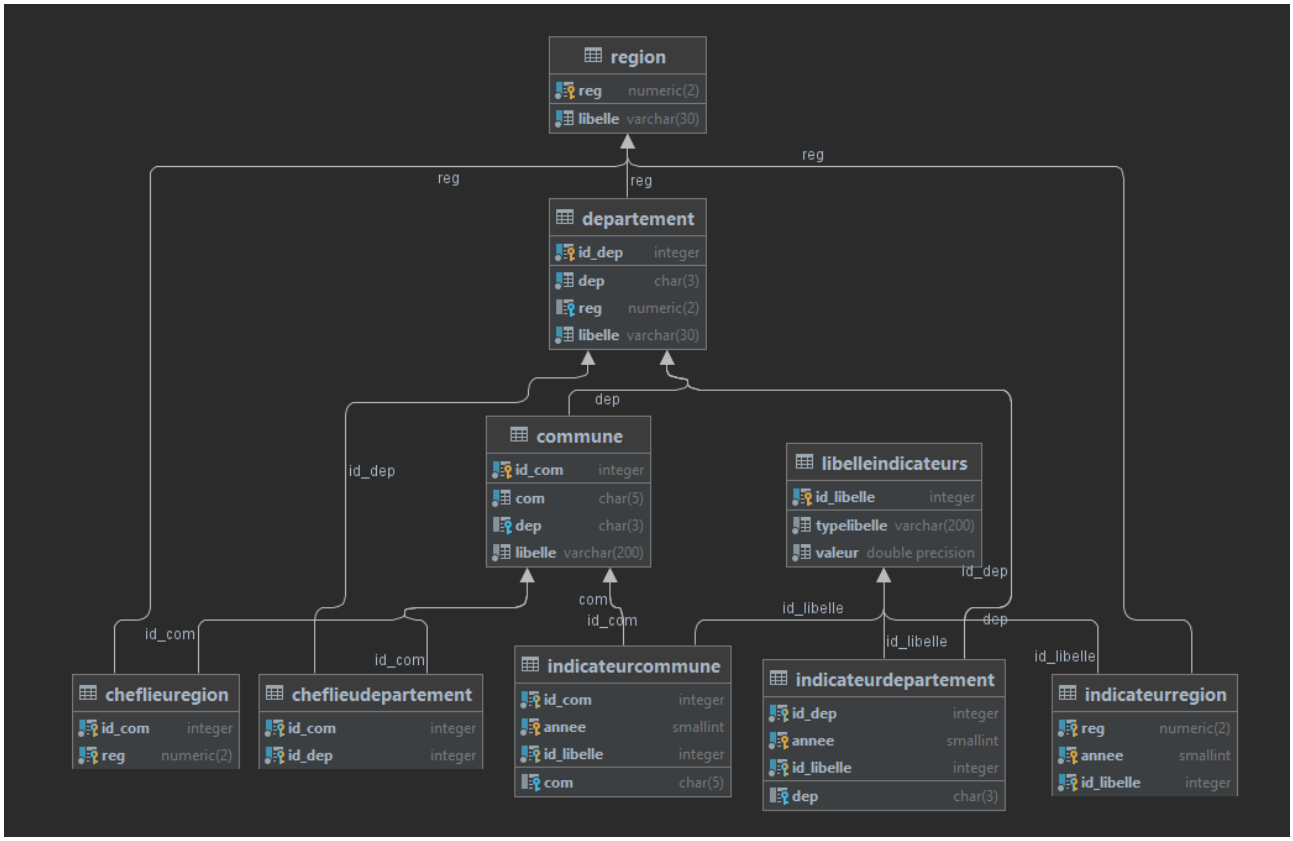
Les fichiers dans code/sql :
- create.sql est le fichier de création des tables.
- delete.sql est le fichier de délétion des tables.
- requests.sql est le fichier d'exécution des différentes requêtes.

Ce rapport contiendra les résultats des différentes implémentations et de brèves explications de la logique poursuivie.

2) Mise à jour du modèle depuis le dernier rendu

J'ai indiqué lors du premier rendu que ma motivation était de passer en SERIAL mes clés primaires afin d'améliorer la performance de la base. Donc, j'ai réalisé cela et fait en sorte que mes relations interagissent bien avec les clés primaires. Cela se détache de la logique du CSV qui travaille sur des clés en COM, REG, néanmoins, si on suppose connaître nos identifiants SERIALS, la logique de mon modèle est interprétable.

Voici ma dernière version du modèle (généré automatiquement par l'IDE Pycharm) qui reste fidèle à mon modèle choisi dans le premier rendu. Mais, avec l'intention d'améliorer la performance de certaines relations de dépendances.



2.1) Résultats obtenus pour les importations

Les résultats suivants seront très simplifiés au vu de la quantité de données. Mais, elles pourront expliquer brièvement l'intention suivie.

Importation de régions :

	reg	libelle
1	1	Guadeloupe
2	2	Martinique
3	3	Guyane
4	4	La Réunion
5	6	Mayotte

Importation de départements :

	id_dep	dep	reg	libelle
1	0	01	84	Ain
2	1	02	32	Aisne
3	2	03	84	Allier
4	3	04	93	Alpes-de-Haute-Provence
5	4	05	93	Hautes-Alpes

Importation de communes :

	id_com	com	dep	libelle
1	1	01001	01	L'Abergement-Clémenciat
2	2	01002	01	L'Abergement-de-Varey
3	3	01004	01	Ambérieu-en-Bugey
4	4	01005	01	Ambérieux-en-Dombes
5	5	01006	01	Ambléon

Importation de chefLieuRegion :

	id_com	reg
1	34841	1
2	34877	2
3	34904	3
4	34935	4
5	34956	6

Importation de chefLieuDepartement :

	id_com	id_dep
1	49	0
2	784	1
3	1381	2
4	1566	3
5	1760	4
6	1958	5

Importation de LibelleIndicateurs :

	id_libelle	typelibelle	valeur
1	1	Espérance de vie des hommes à la naissance pour les reg en 2018	80.6
2	2	Espérance de vie des hommes à la naissance pour les reg en 2018	78.9
3	3	Espérance de vie des hommes à la naissance pour les reg en 2018	78.5

	id_libelle	typelibelle	valeur
1501	1501	P18_POP	470
1502	1502	P18_POP	209
1503	1503	P18_POP	501
1504	1504	P18_POP	431
1505	1505	P18_POP	244

etc... pour plusieurs types de libellés

Importation d’ IndicateurCommune :

	id_com	annee	com	id_libelle
1	1	2018	01001	1200
2	2	2018	01002	1201
3	3	2018	01004	1202
4	4	2018	01005	1203
5	5	2018	01006	1204

Importation d' IndicateurRégion :

	id_dep	annee	dep	id_libelle
1	0	2015	01	144
2	1	2015	02	145
3	2	2015	03	146
4	3	2015	04	147
5	4	2015	05	148

Importation d' IndicateurDépartement :

	id_dep	annee	dep	id_libelle
1	0	2015	01	144
2	1	2015	02	145
3	2	2015	03	146
4	3	2015	04	147
5	4	2015	05	148

Un ordre de créatiou, d'importation est Région, Département, Commune.
Ensuite sans ordre précis, ChefLieuRegion, ChefLieuDepartement et LibelleIndicateurs.
Enfin, LibelleIndicateurs, puis sans ordre précis IndicateurRegion, IndicateurDepartement et IndicateurCommune.

2.3) Quelques choix techniques

Malgré les clés serials, j'ai préféré laisser COM et DEP qui pourrait être retirer entièrement, (si on le désire) ou mit sur une nouvelle table qui lierait des clés id à cela. Au final, ça revient à réutiliser COM et DEP mais sur de nouvelles tables, donc j'ai trouvé que ça complexifiait la chose sans réel intérêt de performance, ce qui fait que j'ai laissé leurs attributs qui simplifient certaines jointures ou conditions dans mon modèle. J'ai cherché quand même à comment les détacher de mon modèle, mais à par se détacher du CSV (et donc supposer que nos identifiants sont entièrement ceux de nos clés SERIALS). Chose que je préfère éviter car ça casserait la cohérence entre les données importées et le modèle, mais aussi, si on veut un département comme 2A, ça ne serait pas définie. Donc, j'ai fais le simple de choix de les garder symétriquement avec mes clés serials.

Le second choix est au niveau des indicateurs et des populations. Personnellement, j'ai vu les populations comme des indicateurs. Avec mon modèle, on peut définir pour les 3 tables (Commune, Région, Département) un indicateur et son libellé relié à une valeur (sur une autre table).

Donc, je sais qu'on pourrait faire ça autrement avec des attributs ou même des tables crées spécialement pour les populations pour chaque années. Mais, j'ai préféré rendre mon modèle plus générique et extensible à tout type de labels d'indicateurs. Plus tôt que de devoir me dire, que je vais devoir créer une table pour tel indicateur à chaque fois, (les populations, je les ai perçu comme des indicateurs aussi, simplement ça aura son propre type de libellé, par exemple, si on veut mettre une statistique quelconque ça ne poserait pas de problème aussi tant qu'on le labellise correctement). La limite est que ma table libellé aura beaucoup de lignes. Mais, en terme de performance, j'ai jugé que c'est mieux d'avoir plus de lignes sur une table qui en soi pour moi reste cohérente, que d'avoir tant de tables moins générique pour le même nombre de lignes.

Au niveau des requêtes, j'ai du bien logiquement m'adapter à mon modèle, et donc, si certaines choses comme la gestion des populations passera forcément par mes tables d'indicateurs.

3) Résultats obtenus pour les questions

Q1)

Les codes utilisés sont dans requests.py
Les différentes fonctions sont appelées dans le main tout en bas.
Les résultats seront encore une fois simplifié lorsque c'est nécessaire (donc les images).

Résultat obtenu pour une liste de numéro avec une région donné en paramètre de fonction.

Liste de départements pour la région numéro 28 :
Département : Calvados , avec le numéro : 14
Département : Eure , avec le numéro : 27
Département : Manche , avec le numéro : 50
Département : Orne , avec le numéro : 61
Département : Seine-Maritime , avec le numéro : 76

Résultat obtenu pour une liste de communes avec un nombre d'habitants n, pour une année x.

Liste de communes avec un nombre d'habitants d'au moins : 10000, en 2018 pour le département numéro 01 :
Commune : Ambérieu-en-Bugey , avec le numéro : 01004, avec tant d'habitants : 14204.0
Commune : Valserhône , avec le numéro : 01033, avec tant d'habitants : 16431.0
Commune : Bourg-en-Bresse , avec le numéro : 01053, avec tant d'habitants : 41248.0
Commune : Gex , avec le numéro : 01173, avec tant d'habitants : 13093.0
Commune : Miribel , avec le numéro : 01249, avec tant d'habitants : 10043.0
Commune : Oyonnax , avec le numéro : 01283, avec tant d'habitants : 22336.0
Commune : Saint-Genis-Pouilly , avec le numéro : 01354, avec tant d'habitants : 13243.0

Résultat obtenu pour l'affichage de n communes en (x années) avec la population maximale pour le département z.

Affichage de 3 communes en 2018 avec la population maximale pour le département numéro 01 :
Commune : Bourg-en-Bresse , avec le numéro : 01053, avec tant d'habitants : 41248.0
Commune : Oyonnax , avec le numéro : 01283, avec tant d'habitants : 22336.0
Commune : Valserhône , avec le numéro : 01033, avec tant d'habitants : 16431.0

Réciproquement, le minimum à la place du maximum.

Affichage de 3 communes en 2018 avec la population minimale pour le département numéro 01 :
Commune : Armix , avec le numéro : 01019, avec tant d'habitants : 27.0
Commune : Prémillieu , avec le numéro : 01311, avec tant d'habitants : 44.0
Commune : Flaxieu , avec le numéro : 01162, avec tant d'habitants : 66.0

Q2) Créer deux vues (cf commande CREATE OR REPLACE VIEW) qui donnent la population des départements et des régions sur toutes les années.

Ici, je vous propose deux implémentations différentes, même si j'ai gardé la seconde.

D'abord, j'ai compris que la population serait la somme des populations de toutes les années. Mais, j'ai trouvé ça un peu incohérent, car, ce n'est pas la population courante d'un département ou d'une région pour que je l'associe avec.

J'ai d'abord essayé ça :

DépartementsPopulations :

	id_dep	dep	population
1	0	01	2834775
2	1	02	2684696
3	2	03	1725840
4	3	04	754393
5	4	05	648901
6	5	06	5234573
7	6	07	1522041
8	7	08	1423436

Puis pour région, ça aurait été la somme des populations de la vue de départements.

- En tout cas, je trouvais pas cette association cohérente. Donc, j'ai opté pour une vue avec un affichage différé sur les années.

	id_dep	annee	population
1	0	1990	471019
2	0	1999	515270
3	0	2008	581355
4	0	2013	619497
5	0	2018	647634
6	1	1990	537200
7	1	1999	535434
8	1	2008	538726
9	1	2013	540020
10	1	2018	533316
11	2	1990	357710
12	2	1999	344721

Pour régions, j’ai utilisé la vue des départements, et j’ai fais la somme des populations différé sur les années aussi.

	reg	annee	population
1	1	1990	353431
2	1	1999	386566
3	1	2008	401784
4	1	2013	402119
5	1	2018	387629
6	2	1990	359572
7	2	1999	381427
8	2	2008	397693
9	2	2013	385551
10	2	2018	368783
11	3	1990	114678

Ensuite, pour tous les autres indicateurs, ça peut être mis facilement ensemble mais j’ai distingué cela des vues de population pour que ce soit plus atomique. Donc, j’ai créé deux nouvelles Vues indicateursRegions et indicateursDepartements. Elles suivent la même logique qu’au dessus sauf que ce sont les autres indicateurs hors population qui sont affichés.

IndicateursRegions :

	id_dep	annee	id_libelle	statistic
1	0	2008	141	5462.699999999983
2	0	2009	102	6645.5190246815555
3	0	2010	24	31007.700000000215
4	0	2010	50	33444.299999999756
5	0	2013	128	5630.367594909501
6	0	2014	63	1333.0307720741173
7	0	2014	76	4988.25742006159

IndicateursDepartements :

	id_dep	annee	id_libelle	statistic
1	0	2008	141	5462.699999999983
2	0	2009	102	6645.5190246815555
3	0	2010	24	31007.700000000215
4	0	2010	50	33444.299999999756
5	0	2013	128	5630.367594909501
6	0	2014	63	1333.0307720741173

3)

J’ai créer deux nouvelles tables de populations de régions et départements, pour pouvoir avoir des relations plus concrète entre région et les populations de régions pour chaque année.
Donc, pour cela, j’aurai besoin d’identifier dans région une population par un identifiant de la table qui contient les populations de région. Le même procédé pour les départements.

Sauf qu’une limite sur mes vues affichés précédemment était d’avoir un identifiant pour chaque année, vu que je vais devoir créer des clés uniques de population dans la table région et département.

Pour pallier à cela, j’ai effectué une sorte de transposée entre les lignes et les colonnes (index 1,2), à l’aide de la fonction crosstab(), puis j’ai stockée ça sur deux nouvelles vues.

Id	annee	value	→	Id	y	z
x1	y	1		x1	1	3
x1	z	3				

etc...

Du coup, dans région, je peux ajouter un identifiant assez facilement. J’ai implémenté la clé de cette table avec un SERIAL.

J’ai crée tout cela de manière procédurale, donc, d’abord deux procédures séparées pour factoriser, writeDptsPop() et writeRegPop().
Puis, writePopulations() est la procédure qui répondra à la question, en écrivant les populations pour chaque années.

DpsPopulations :

	id_dep	1990	1999	2008	2013	2018	id_pop
1	0	471019	515270	581355	619497	647634	1
2	1	537200	535434	538726	540020	533316	2
3	2	357710	344721	342807	343431	337171	3
4	3	130883	139561	157965	161916	164068	4
5	4	113300	121419	134205	139279	140698	5
6	5	971829	1011326	1084428	1080771	1086219	6
7	6	277581	286023	311452	320379	326606	7
8	7	296357	290130	284197	280907	271845	8
9	8	136455	137205	150201	152684	153066	9

Relié à région par la clé id_pop

	id_dep	dep	reg	libelle	id_pop
1	0	01	84	Ain	1
2	1	02	32	Aisne	2
3	2	03	84	Allier	3
4	3	04	93	Alpes-de-Haute-Provence	4
5	4	05	93	Hautes-Alpes	5
6	5	06	93	Alpes-Maritimes	6

Même procédé pour DptsRegions ...

4)

Un update, insert, delete est testé sur la table département et région.
Le trigger empêche bien ses commandes de se lancer, si on en essaie une nous aurons comme message d'erreur :

[P0001] ERREUR: This table is unmodifiable
Où : fonction PL/pgsql readonly(), ligne 3 à RAISE

Cette partie m'a fait réutilisé la procédure d'avant. J'ai plus ou moins relié chacune de mes vues et procédures. Donc, c'est assez factorisé mais aussi dépendants.
Donc, j'aurai retravaillé sur les procédures des questions d'avant afin de rendre la question 4 et 5 utilisable avec les précédentes procédures.

En tout cas, le trigger va réutiliser les procédures revus d'avant, pour arriver à cela, il faut désactiver le trigger d'écriture, à la fin de la fonction, on le réactive.

Un test simple avec une valeur très grande pour bien voir la modification.

```
UPDATE libelleindicateurs SET valeur = 1000000000000
WHERE id_libelle = 1200;

SELECT id_libelle, valeur FROM libelleindicateurs
where id_libelle = 1200;
```

Je gère toujours la population des communes avec ma table d'indicateurs, donc le trigger sur un update après mise à jour. Il portera sur l'attribut de valeur mais seulement si c'est une relation complète (ça a une commune, un département, une région) + c'est un type de libellé de population.

Avec id_libelle 1200 qui est le libellé pour la population de commune en 2018 du 1er département et 84ème région.

Dans dptPopulations :

	reg	1990	1999	2008	2013	2018	id_pop
1	84	6671915	6954285	7459092	7757595	10000079936…	15

Dans regPopulations :

	id_dep	1990	1999	2008	2013	2018	id_pop
1	0	471019	515270	581355	619497	10000006468…	1

5) Mes implémentations précédentes vérifient déjà l'idée mettre à jour les nouvelles données, et donc, probablement celle d'une nouvelle année. J'ai ajouté les années restantes en adaptant mes clés d'importation et ceux utilisés dans mes requêtes. Les procédures écrit bien automatiquement les données de population pour toute nouvelle année. Une différence est qu'il faut l'implémenter sur un trigger insert aussi.

Table regPopulations, insertion, mise à jour automatisées des populations sur les nouvelles années.

	reg	1968	1975	1982	1990	1999	2008	2013	2018	id_pop
1	1	305312	315848	317269	353431	386566	401784	402119	387629	1
2	2	320030	324832	328566	359572	381427	397693	385551	368783	2
3	3	44392	55125	73022	114678	157213	219266	244118	276128	3
4	4	416525	476675	515814	597823	706300	808250	835103	855961	4
5	11	9248631	9878565	10073059	10660554	10952011	11659260	11959807	12213447	5

Table depPopulations, insertion, mise à jour automatisées des populations sur les nouvelles années.

	id_dep	1968	1975	1982	1990	1999	2008	2013	2018	id_pop
1	0	339262	376477	418516	471019	515270	581355	619497	647634	1
2	1	525953	533807	533927	537200	535434	538726	540020	533316	2
3	2	386533	378406	369580	357710	344721	342807	343431	337171	3
4	3	104813	112178	119068	130883	139561	157965	161916	164068	4
5	4	91790	97358	105070	113300	121419	134205	139279	140698	5
6	5	722070	816681	881198	971829	1011326	1084428	1080771	1086219	6

Un point améliorable pour un utilisateur serait d'automatiser les années inconnus par la base et qui sont ajoutés uniquement à travers celle-ci. Au niveau du développeur, il y a simplement à préciser cette année en plus, là où les requêtes ont en besoin et toutes les lignes concernant cette année seront bien traitée.

6) (Le disque prend en cache après une première exécution, donc il faut se baser sur la première exécution).

```
EXPLAIN ANALYZE SELECT * FROM commune;
```

```
+-----------------------------------------------------------------------------------------------------------+
|QUERY PLAN                                                                                                 |
+-----------------------------------------------------------------------------------------------------------+
|Seq Scan on commune  (cost=0.00..600.65 rows=34965 width=27) (actual time=0.008..1.834 rows=34965 loops=1)|
|Planning Time: 0.081 ms                                                                                    |
|Execution Time: 2.743 ms                                                                                   |
```

→ L'exécution est lente 2,743 ms, on passe par un parcours séquentiel, donc ligne par ligne sur un grand nombre de données (34965), ce qui explique ce temps.

Alors que si l'on travaillerait sur un index (id_com ici)

```
EXPLAIN ANALYZE SELECT id_com FROM commune where id_com = 120;
```

```
+-----------------------------------------------------------------------------------------------------------------------------+
|QUERY PLAN                                                                                                                   |
+-----------------------------------------------------------------------------------------------------------------------------+
|Index Only Scan using commune_pkey on commune  (cost=0.29..8.31 rows=1 width=4) (actual time=0.014..0.015 rows=1 loops=1)|
|  Index Cond: (id_com = 120)                                                                                                 |
|  Heap Fetches: 1                                                                                                            |
|Planning Time: 0.073 ms                                                                                                      |
|Execution Time: 0.029 ms                                                                                                     |
```

→ Nous réalisons un scan index, le temps d'exécution est nettement diminué 0,029 ms. On ne traverse pas de manière séquentielle le tableau, on a juste cherché avec un index ce qui est nettement plus petit.

Sans index, on peut tester une requête avec la même logique :

```
EXPLAIN ANALYZE SELECT com FROM commune WHERE dep = '21';
```

```
+-----------------------------------------------------------------------------------------------+
|QUERY PLAN                                                                                     |
+-----------------------------------------------------------------------------------------------+
|Seq Scan on commune  (cost=0.00..688.06 rows=235 width=6) (actual time=1.247..3.236 rows=241 loops=1)|
|  Filter: (dep = '36'::bpchar)                                                                 |
|  Rows Removed by Filter: 34724                                                                |
|Planning Time: 0.061 ms                                                                        |
|Execution Time: 3.249 ms                                                                       |
```

→ Le parcours ici est séquentiel aussi, on cherche un par un ce qui est lent.

```
EXPLAIN ANALYZE
SELECT *
from region
	join regpopulations r on r.id_pop = region.id_pop
ORDER BY r."2018";
```

```
+-----------------------------------------------------------------------------------------------------------------------+
|QUERY PLAN                                                                                                             |
+-----------------------------------------------------------------------------------------------------------------------+
|Limit  (cost=23.05..23.05 rows=1 width=182) (actual time=0.047..0.048 rows=1 loops=1)
|
|  ->  Sort  (cost=23.05..24.70 rows=660 width=182) (actual time=0.047..0.047 rows=1 loops=1)
|
|        Sort Key: r."2013"                                                                                             |
|        Sort Method: top-N heapsort  Memory: 25kB                                                                      |
|        ->  Hash Join  (cost=1.38..19.75 rows=660 width=182) (actual time=0.029..0.036 rows=17 loops=1)
|
|              Hash Cond: (region.id_pop = r.id_pop)                                                                    |
|              ->  Seq Scan on region  (cost=0.00..16.60 rows=660 width=94) (actual time=0.011..0.012 rows=18
loops=1)            |
|              ->  Hash  (cost=1.17..1.17 rows=17 width=80) (actual time=0.012..0.012 rows=17 loops=1)
|
|                    Buckets: 1024  Batches: 1  Memory Usage: 10kB                                                      |
|                    ->  Seq Scan on regpopulations r  (cost=0.00..1.17 rows=17 width=80) (actual time=0.005..0.006
rows=17 loops=1)|
|Planning Time: 0.144 ms                                                                                               |
|Execution Time: 0.071 ms                                                                                              |
```

→ Une jointure sur des petites tables reste correct avec un temps d'exécution de 0,71 grace au tri sur la clé. Par contre, les scans sont triés mais séquentiels aussi, là par chance le nombre de lignes et léger mais si on en a beaucoup, il est inévitable que cette requête deviendrait lente et nécessiterait une optimisation par un index.

```
EXPLAIN ANALYZE
SELECT *
from region
	join regpopulations r on r.id_pop = region.id_pop
ORDER BY r."2018"
limit 1;
```

```
+-----------------------------------------------------------------------------------------------------------------------
------------+
|QUERY PLAN                                                                                                             |
+-----------------------------------------------------------------------------------------------------------------------
------------+
```

```
|Limit  (cost=23.05..23.05 rows=1 width=182) (actual time=0.046..0.046 rows=1 loops=1)
|
| ->  Sort  (cost=23.05..24.70 rows=660 width=182) (actual time=0.045..0.046 rows=1 loops=1)
|
|       Sort Key: r."2018"                                                                  |
|       Sort Method: top-N heapsort  Memory: 25kB
|
|       ->  Hash Join  (cost=1.38..19.75 rows=660 width=182) (actual time=0.027..0.034 rows=17
loops=1)                        |
|             Hash Cond: (region.id_pop = r.id_pop)                                                |
|             ->  Seq Scan on region  (cost=0.00..16.60 rows=660 width=94) (actual time=0.010..0.011
rows=18 loops=1)             |
|             ->  Hash  (cost=1.17..1.17 rows=17 width=80) (actual time=0.012..0.012 rows=17
loops=1)                        |
|                   Buckets: 1024  Batches: 1  Memory Usage: 10kB
|
|                   ->  Seq Scan on regpopulations r  (cost=0.00..1.17 rows=17 width=80) (actual
time=0.005..0.006 rows=17 loops=1)|
|Planning Time: 0.130 ms                                                                     |
|Execution Time: 0.068 ms                                                                    |
```

→ Si on limite, le nombre de données, on gagne légèrement plus de temps aussi.

Enfin testons une jointure sur notre plus grande table, celle-ci devrait être vraiment lente.

```
EXPLAIN ANALYZE
SELECT *
FROM indicateurcommune
       join libelleindicateurs l on l.id_libelle = indicateurcommune.id_libelle;
```

```
+-----------------------------------------------------------------------------------------------------------------
----------------+
|QUERY PLAN                                                                                                      |
+-----------------------------------------------------------------------------------------------------------------
----------------+
|Hash Join  (cost=9760.62..19180.38 rows=279584 width=36) (actual time=67.314..243.237
rows=279584 loops=1)                        |
|  Hash Cond: (indicateurcommune.id_libelle = l.id_libelle)
|
| ->  Seq Scan on indicateurcommune  (cost=0.00..4307.84 rows=279584 width=16) (actual
time=0.010..17.871 rows=279584 loops=1)        |
| ->  Hash  (cost=4604.83..4604.83 rows=280783 width=20) (actual time=66.789..66.790
rows=280783 loops=1)                        |
|       Buckets: 65536  Batches: 8  Memory Usage: 2442kB
|
|       ->  Seq Scan on libelleindicateurs l  (cost=0.00..4604.83 rows=280783 width=20) (actual
time=0.006..18.454 rows=280783 loops=1)|
|Planning Time: 0.162 ms                                                                         |
|Execution Time: 250.668 ms                                                                      |
```

Le scan est séquentiel, donc ligne par ligne sur 279584 joint à 280783, c'est juste beaucoup trop conséquent comme opération, le temps est énorme 250,668 ms qui représente littéralement 2 seconde et demi. Un ou des index sur ce type de requête sont nécessaire.

De nombreux autres requêtes sont écrites, les plus simples sont généralement assez rapides. Mais là où on voit les limites de vitesse sont lorsque la cardinalité est élevée, le coût du type sur lequel on manipule est élevé ou bien le nombre de jointures sur une table de taille n implique aussi un plus grand temps d'exécution.

7) Pour pallier à certains problèmes vu précédemment et non solvables par du tri. On va utiliser des index sur certains attributs utilisés dans nos requêtes.

On reprend la requête basique :

```
EXPLAIN ANALYZE SELECT com FROM commune WHERE dep = '36';
```

On utilise un index sur dep.

On passe de 3ms avant à 0,073 ce qui est une nette amélioration, et il y a plus de parcours séquentielle.

```
+-----------------------------------------------------------------------------------------------------------------+
|QUERY PLAN                                                                                                       |
+-----------------------------------------------------------------------------------------------------------------+
|Index Scan using idx_aabb on commune  (cost=0.29..13.65 rows=249 width=6) (actual time=0.021..0.059
rows=241 loops=1)|
|  Index Cond: (dep = '36'::bpchar)                                                                               |
|Planning Time: 0.081 ms                                                                                          |
|Execution Time: 0.073 ms                                                                                         |
+-----------------------------------------------------------------------------------------------------------------+
```

Puis, pour la jointure sur les années de 2018, en créant un index sur la colonne 2018. On peut simplifier aussi notre parcours séquentiel en index.

```
+-----------------------------------------------------------------------------------------------------------------------------------------
----+
|QUERY PLAN                                                                                                                             |
+-----------------------------------------------------------------------------------------------------------------------------------------
----+
|Limit  (cost=0.14..0.44 rows=1 width=182) (actual time=0.045..0.046 rows=1 loops=1)
|
|  ->  Nested Loop  (cost=0.14..198.94 rows=660 width=182) (actual time=0.045..0.045 rows=1 loops=1)
|
|        Join Filter: (region.id_pop = r.id_pop)                                                                                        |
|        Rows Removed by Join Filter: 3                                                                                                 |
|        ->  Index Scan using idx_dep2018 on regpopulations r  (cost=0.14..12.39 rows=17 width=80) (actual
time=0.033..0.033 rows=1 loops=1)|
|        ->  Materialize  (cost=0.00..19.90 rows=660 width=94) (actual time=0.008..0.009 rows=4 loops=1)
|
|              ->  Seq Scan on region  (cost=0.00..16.60 rows=660 width=94) (actual time=0.006..0.006 rows=4
loops=1)                    |
|Planning Time: 0.144 ms                                                                                                                |
|Execution Time: 0.065 ms                                                                                                               |
```

Le parcours est bien plus efficace, seulement 17 lignes avec un index scan. De plus, on gagne aussi un peu plus de temps.

Testons le parcours pour le nombre d'habitants, cette requête va être conséquente aussi, cherchons à l'optimiser.
Sans index, on a :

```
EXPLAIN ANALYZE
SELECT commune.COM AS numero_commune, libelle, valeur as population
```

```
FROM commune
        join indicateurcommune i on commune.id_com = i.id_com
        join libelleindicateurs l on i.id_libelle = l.id_libelle and valeur <= 5000;
```

```
+-----------------------------------------------------------------------------------------------------------------------
----------------------+
|QUERY PLAN                                                                                                             |
+-----------------------------------------------------------------------------------------------------------------------
----------------------+
|Hash Join  (cost=10978.38..20204.93 rows=265412 width=27) (actual time=86.874..304.099
rows=264957 loops=1)                                |
|  Hash Cond: (i.id_com = commune.id_com)
|
|  ->  Hash Join  (cost=9940.66..18470.42 rows=265412 width=12) (actual time=77.941..236.191
rows=264957 loops=1)                            |
|        Hash Cond: (i.id_libelle = l.id_libelle)
|
|        ->  Seq Scan on indicateurcommune i  (cost=0.00..4307.84 rows=279584 width=8) (actual
time=0.019..32.335 rows=279584 loops=1)        |
|        ->  Hash  (cost=5306.79..5306.79 rows=266550 width=12) (actual time=77.724..77.724
rows=266156 loops=1)                            |
|              Buckets: 131072  Batches: 8  Memory Usage: 2458kB
|
|              ->  Seq Scan on libelleindicateurs l  (cost=0.00..5306.79 rows=266550 width=12) (actual
time=0.014..36.780 rows=266156 loops=1)|
|                    Filter: (valeur <= '5000'::double precision)
|
|                    Rows Removed by Filter: 14627
|
|  ->  Hash  (cost=600.65..600.65 rows=34965 width=23) (actual time=8.775..8.775 rows=34965
loops=1)                                    |
|        Buckets: 65536  Batches: 1  Memory Usage: 2443kB
|
|        ->  Seq Scan on commune  (cost=0.00..600.65 rows=34965 width=23) (actual
time=0.010..3.949 rows=34965 loops=1)                    |
|Planning Time: 0.424 ms                                                                                                |
|Execution Time: 311.147 ms
|
```

Il y a encore une fois du parcours séquentiel ce qui faudrait éviter, surtout le temps d'exécution est lent (311.147 ms).

J'ai mis un index sur les valeurs de mes libellés. Mais, psql ne juge pas utile d'utiliser cet index et préfère suivre l'ordre séquentiel. Sachant que les autres sont des clés, je ne peux pas ajouter d'index à cette requête comme elle est écrite. J'en conclus que c'est une requête à optimiser ou bien il doit exister un moyen de l'indexer.

Enfin avec le premier test, on voit bien que les clés primaires et étrangères sont des index aussi. J'ai pu observé ça un peu plus en détail grâce à une requête trouvé sur stackoverflow qui indique tous les index de la base de donnée courante.

Par exemple on a :

```
+------+-------------------+-----------------------+
```

|Schema|Table |Index |
+------+------------------+----------------------+
public	cheflieudepartement	pkey_cheflieudepartement
public	cheflieuregion	pkey_cheflieuregion
public	commune	commune_com_key
public	commune	commune_pkey

Etc ..		

Donc, oui c’est bien aussi des index.

Encore une fois, il y a pas mal d’autres requêtes essayés à l’intérieur du fichier qui ne seront pas détaillé ici car ça suit essentiellement la même idée.

8) Il existe 4 niveaux d’isolations pour les transactions.

Par défaut, le Read Comitted Isolation Level : j’ai testé un insert tout simple avec via un des clients. Sauf que le second ne voit pas la modification à moins de rafraîchir la base de données.

```
BEGIN;
INSERT INTO commune(com, dep, libelle)
VALUES('52000', '01', 'test');
COMMIT;
```

Client A lance la requête :

34962	34962	97614	976	Ouangani
34963	34963	97615	976	Pamandzi
34964	34964	97616	976	Sada
34965	34965	97617	976	Tsingoni
34966	34966	52000	01	test

Client B ne la voit pas.

34963	34963	97615	976	Pamandzi
34964	34964	97616	976	Sada
34965	34965	97617	976	Tsingoni

- Repetable read

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
   UPDATE regpopulations SET "1975" = -1000 WHERE id_pop = 2;
   DELETE from regpopulations WHERE id_pop = 10;
COMMIT;
```

renvoie could not serialize access due to concurrent update sur le client B, sinon j’ai lu que c’est possible de voir les données concurrentes qui ont été commit avant le début de la transaction, mais celle pendant ne sont pas connues.

- Serializable Isolation Level

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   UPDATE regpopulations SET "1975" = -1000 WHERE id_pop = 2;
COMMIT;
```

J’ai effectué ça deux accès concurrent sur une valeur, sauf que serializable ne laisse pas passer deux commits non ordonnés. Donc, si A commit et que B aussi commit sur une table. Alors, seul l’un des deux pourra commit, l’autre a une erreur.

Le uncommited n’est pas recommandé en psql, donc, j’ai pas cherché à l’essayer mais c’est celui qui aurait pu permettre des accès concurrents sans réelle contraintes.