

Utiliser la VM VirtualBox nommée `nosql2019`

- Dossier sous Windows : `\c$\Wms\` ; si non disponible, la récupérer sur le serveur `/softwares/sync/VMs/`
- OS : Ubuntu 64 bits
- RAM : 8 Go
- Augmenter la taille de la mémoire vidéo au maximum
- Login (root) : `nosql` ; mot de passe : `nosql`
- Activer le presse-papier partagé (bidirectionnel)
- Si l'affichage de la VM est en faible résolution, installer les Additions Invité (drivers) : Menu *Périphériques* puis *Insérer l'image CD des Additions Invité...* L'image d'un CD va alors être chargée et les drivers installés. Redémarrer si nécessaire la machine virtuelle et changer la résolution de l'affichage.

La VM contient PostgreSQL 10.6 et pgAdmin4.

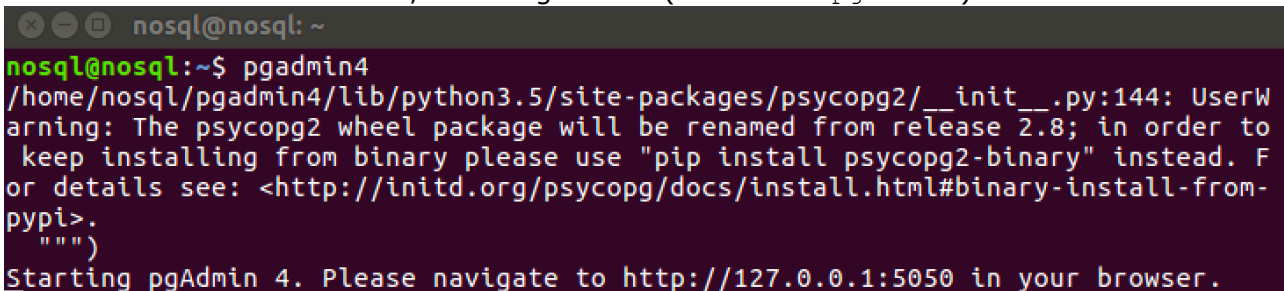
INDEX

BASE DE DONNEES

On considère la base de données dont le schéma est le suivant. **Cette base n'a ni contrainte (PK, FK,...), ni index.**

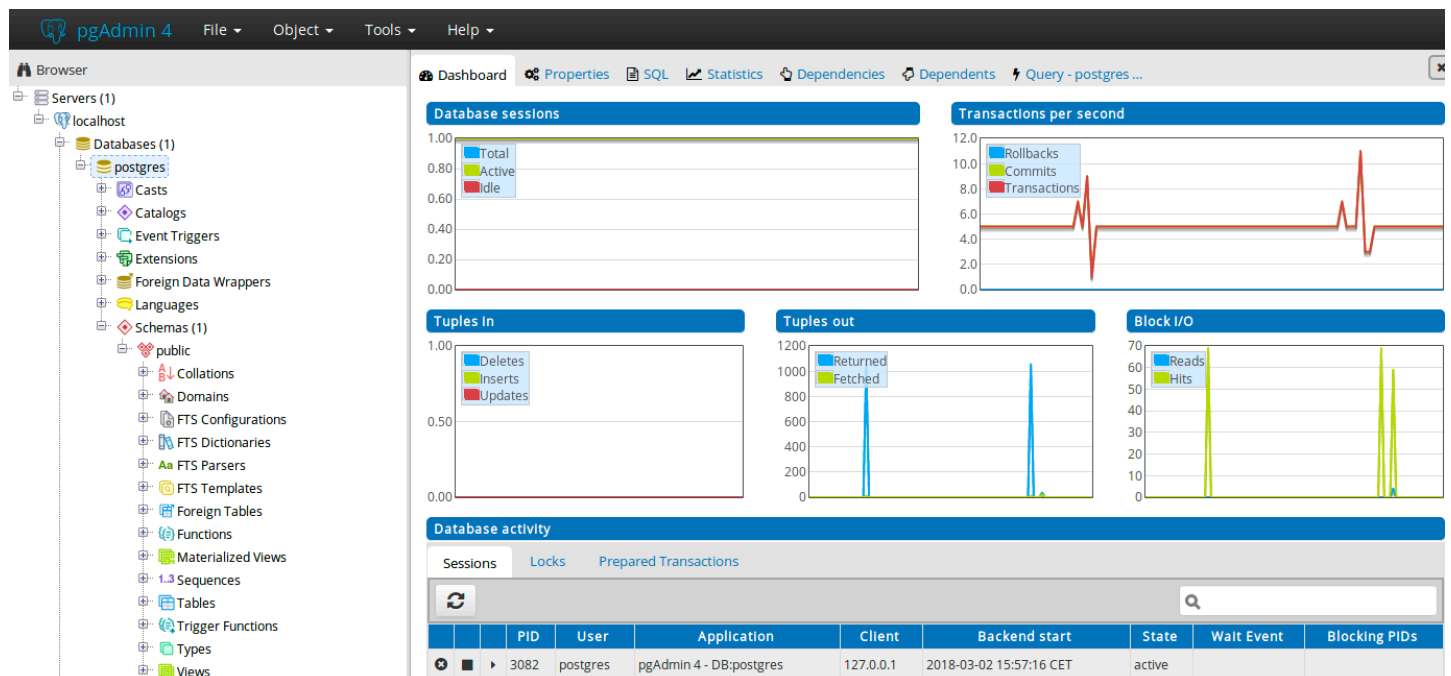
```
CREATE TABLE Film(  
    id integer,  
    pays varchar(30),  
    annee integer,  
    eur numeric  
);  
CREATE TABLE Realisateur(  
    id integer,  
    nom varchar(50)  
);  
CREATE TABLE Realise(  
    id_film integer,  
    id_real integer  
);  
CREATE TABLE Titres(  
    id_film integer,  
    titre varchar(200),  
    langue varchar(3)  
);  
CREATE SEQUENCE seq_film;  
CREATE SEQUENCE seq_real;
```

Dans une fenêtre *Terminal* de la VM, lancer PgAdmin4 (commande `pgadmin4`) :



```
nosql@nosql: ~  
nosql@nosql:~$ pgadmin4  
/home/nosql/pgadmin4/lib/python3.5/site-packages/psycopg2/__init__.py:144: UserWarning: The psycopg2 wheel package will be renamed from release 2.8; in order to keep installing from binary please use "pip install psycopg2-binary" instead. For details see: <http://initd.org/psycopg/docs/install.html#binary-install-from-pypi>.  
  """)  
Starting pgAdmin 4. Please navigate to http://127.0.0.1:5050 in your browser.
```

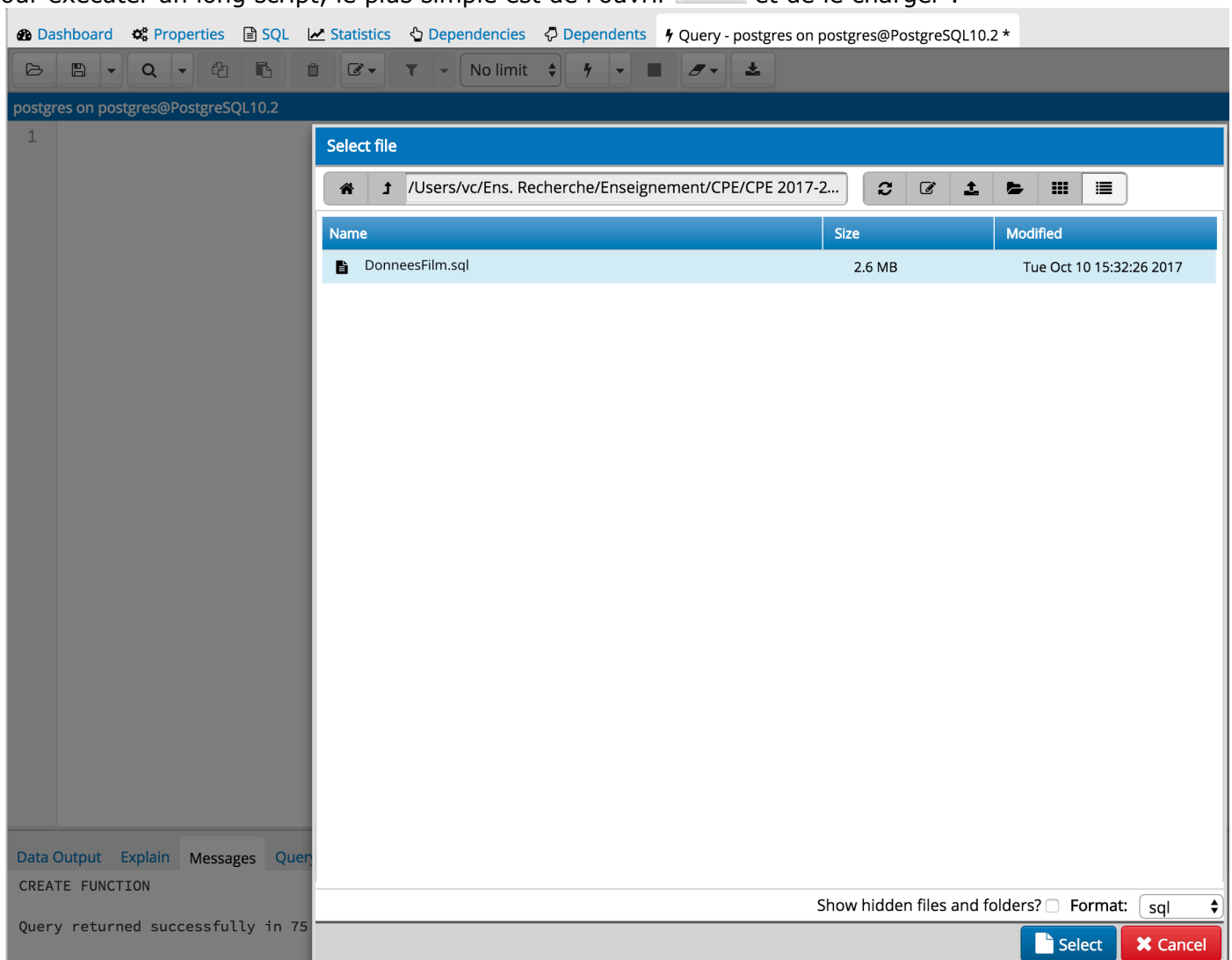
Dans un navigateur web, se connecter à `localhost:5050` pour charger l'interface.



Ouvrir une fenêtre SQL (menu *Tools* > *Query Tool*).

Exécutez les deux scripts (CreateBaseFilm.Sql puis DonneesFilm.sql) dans le schéma Public de la base de données postgres.

Pour exécuter un long script, le plus simple est de l'ouvrir  et de le charger :



Exécuter le code SQL .

Vérifiez le nombre de ligne dans chaque table. La base doit contenir 20247 enregistrements dans TITRES, 9706 dans FILM, 5976 dans REALISATEUR et 10245 dans REALISE.

postgres on postgres@localhost	
1	<code>select count(*) from titres</code>
Data Output Explain Messages Query History	
	count bigint
1	20247

INDEX & EXPLAIN

PLAN D'EXECUTION

Lorsqu'une commande SQL (`SELECT`, `UPDATE`, `INSERT` ou `DELETE`) est soumise à un SGBD, l'optimiseur doit trouver le meilleur chemin appelé *plan d'exécution*, pour accéder aux données référencées dans la commande avec à la fois un minimum d'opérations d'entrées/sorties et un minimum de temps de traitement.

Pour exécuter une commande SQL, un SGBD peut être amené à effectuer un certain nombre d'étapes. Chacune de ces étapes a pour but soit de retrouver les données directement dans la base de données soit de les préparer dans une forme quelconque pour les transmettre à une étape ultérieure ou à l'utilisateur final.

Cette combinaison d'étapes que l'optimiseur choisit pour exécuter la commande SQL constitue le *PLAN D'EXECUTION*.

L'outil *EXPLAIN* donne le plan d'exécution d'une requête. La description comprend :

- Le chemin d'accès utilisé,
- Les opérations physiques (tri, fusion, intersection,...),
- L'ordre des opérations ; il est représentable par un arbre ou une tabulation.

Cet outil permet de comprendre les actions du SGBD lors d'une requête `SELECT` afin d'améliorer la rapidité d'exécution.

Dans le cas de PostgreSQL, l'outil *EXPLAIN* fournit les indications suivantes :

- Coût estimé du lancement (temps passé avant que l'affichage de la sortie ne commence, c'est-à-dire pour faire le tri dans un noeud de tri) ; Ex. : `cost=0.00..458.00`
- Coût total estimé (si toutes les lignes doivent être récupérées, ce qui pourrait ne pas être le cas : par exemple une requête avec une clause `LIMIT` ne paiera pas le coût total du noeud d'entrée du noeud du plan Limit) ; Ex. : `cost=0.00..458.00`
- Nombre de lignes estimé en sortie par ce noeud de plan (encore une fois, seulement si exécuté jusqu'au bout) ; Ex. : `rows=10000`
- Largeur moyenne estimée (en octets) des lignes en sortie par ce noeud de plan. Ex. : `width=244`

Pour plus d'informations sur *EXPLAIN*, Cf. point 14.1. *Utiliser EXPLAIN* du manuel.

INDEX

Pour plus d'informations sur les index gérés par PostgreSQL, Cf. chapitre 11.

Vous pouvez utiliser la vue système `pg_indexes` de `pg_catalog` pour visualiser les index et leur description (type d'arbre indexé utilisé,...) du schéma public :

```
Select * from pg_indexes where schemaname='public';
```

REALISATION D'UN PLAN DE REQUETE

On détermine le plan d'une requête à l'aide de l'ordre :

`EXPLAIN SELECT...`

Exemple : `EXPLAIN select * from Realisateur;`

PLANS DE REQUETE

Le but de ces exercices est de se familiariser avec les plans de requêtes, en consultant ceux de PostgreSQL pour un ensemble de requêtes, impliquant ou non des index et des jointures. Les principales opérations (**Cf. Annexe**) que nous utiliserons sont :

- Parcours séquentiel d'une table : `SEQ SCAN`
- Parcours d'index : `INDEX SCAN`
- Jointure par boucles imbriquées : `NESTED LOOP`
- Jointure par Tri/Fusion : `SORT et MERGE`
- Jointure par hachage : `HASH JOIN`
- autres,...

TRAVAIL A REALISER

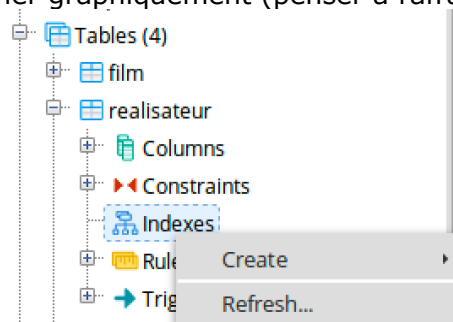
A. OBJECTIF & ETAPES A RESPECTER

Objectif : afficher les plans de requête pour les requêtes suivantes (cf. point B.), et tenter de déterminer l'interaction entre index et plans. Vous devrez au fur et à mesure tester les requêtes sans puis avec index et déterminer si cela améliore ou non la performance de la requête.

Vous rendrez un fichier explicatif à la fin du TP contenant l'ensemble de vos réponses en indiquant pourquoi (ou non) le résultat de l'EXPLAIN a changé, ainsi que la charge de calcul (coût de la requête `cost`) et le nombre d'enregistrements traités (`rows`). **Travail à faire en binôme.**

Etapas à respecter pour chaque requête :

1. `EXPLAIN SELECT...` (à exécuter sur une table sans index)
2. Copier (copie d'écran) ou recopier succinctement le plan d'exécution dans un éditeur et l'expliquer (regarder notamment le coût de la requête et les opérateurs)
3. Création d'un index sur le(s) champ(s)
 - Sur une FK ou un champ « normal », créer un INDEX NON UNIQUE : `CREATE INDEX IDX_NOMTABLE_NOM_CHAMP ON TABLE(CHAMP);`
 - Sur une PK, créer un INDEX UNIQUE (et non une PK même si cela revient au même) : `CREATE UNIQUE INDEX IDX_NOM_TABLE_NOM_CHAMP ON TABLE(CHAMP).`
4. `EXPLAIN SELECT...` (sur une table avec index cette fois !)
5. Copier (copie d'écran) ou recopier succinctement le plan d'exécution dans un éditeur et l'expliquer. L'index a-t-il rendu la requête plus performante ? Expliquer la différence avec le plan d'exécution obtenu lors des étapes 1 et 2.
6. Les étapes 3 à 5 sont à répéter autant de fois que nécessaire si plusieurs index sont à créer.
7. Supprimer l'index (ou les index) créé(s) (commande `DROP INDEX nom_index;`). Pour visualiser si les index ont été supprimés, utilisez la vue système `pg_indexes` : `Select * from pg_indexes where schemaname='public';` Cette vue ne doit afficher aucune ligne après suppression des index. Vous pouvez aussi le vérifier graphiquement (penser à rafraichir) :



8. **LORS DU PASSAGE A LA REQUETE SUIVANTE, TOUS LES INDEX DOIVENT AVOIR ETE SUPPRIMES.**

Exemple :

- i. Sans index : `EXPLAIN SELECT * FROM REALISATEUR`

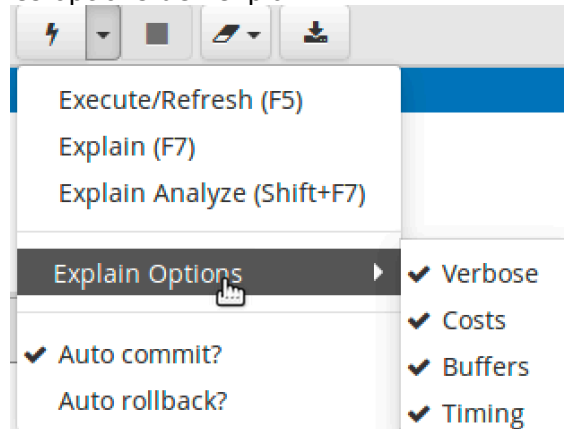
QUERY PLAN

text

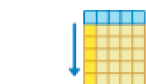
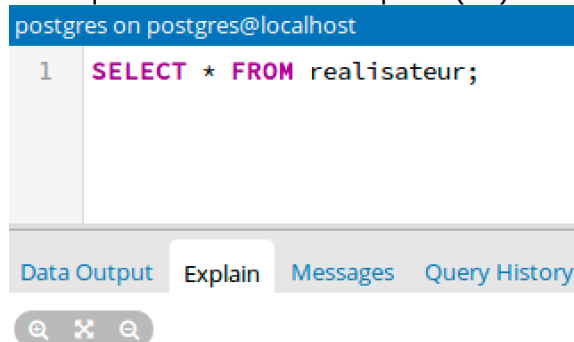
Seq Scan on realisateur (cost=0.00..97.76 rows=5976 width=19)

On peut aussi visualiser l'explain en mode graphique (surtout intéressant quand il y a des jointures) :

- Cocher toutes les options de l'explain



- Touche « F7 » ou cliquer sur le menu « Explain (F7) »



public.realisateur

Parallel Aware	false
Plan Width	19
Relation Name	realisateur
Output	id,nom
Alias	realisateur
Plan Rows	5976
Node Type	Seq Scan
Schema	public
Startup Cost	0
Total Cost	97.76

Explication : Accès séquentiel car pas d'index. Il ne serait de toute façon pas utilisé car on souhaite afficher tous les enregistrements d'une seule table (pas de WHERE, ni de jointure).

- ii. Avec INDEX UNIQUE (car ce serait la PK) sur le champ ID de realisateur
`CREATE UNIQUE INDEX IDX_REALISATEUR_ID ON REALISATEUR(ID);`

`EXPLAIN SELECT * FROM REALISATEUR;`

QUERY PLAN

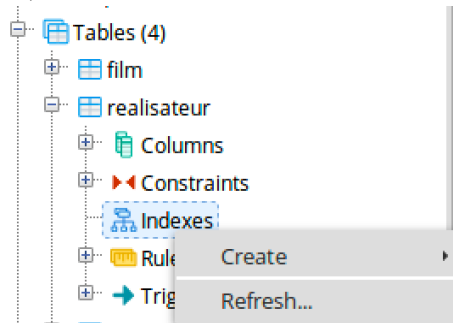
text

Seq Scan on realisateur (cost=0.00..97.76 rows=5976 width=19)

Explication : Pas de changement car on récupère tous les enregistrements de la table. En effet, il serait contre-productif de passer à chaque fois par l'index (soit 5976 fois) pour récupérer chaque ligne. Grace aux statistiques, PostgreSQL connaît en outre le nombre de lignes à récupérer.

iii. Suppression de l'index :

```
DROP INDEX IDX_REALISATEUR_ID;
```



B. REQUETES

Optimisation physique

Remarque : Il s'agit ici de déterminer si l'ajout d'un index a un impact sur les performances et dans quels cas.

1. `SELECT * FROM realisateur where ID=2800;`

- Testez sans index.
- Testez après l'ajout d'un index unique sur ID : `CREATE UNIQUE INDEX IDX_REALISATEUR_ID ON REALISATEUR (ID);`
- **Coût de la requête ? Expliquez. Comparez avec les résultats de l'exemple.**
- Supprimez l'index : `DROP INDEX IDX_REALISATEUR_ID;`

Remarque : si l'on créé une PK à la place de l'index unique sur ID, on a exactement le même plan d'exécution :

```
1 ALTER TABLE REALISATEUR ADD CONSTRAINT PK_REALISATEUR PRIMARY KEY (ID);
2 EXPLAIN SELECT * FROM REALISATEUR WHERE ID=2800;
```

Data Output	Explain	Messages	Notifications	Query History
<div> <div>QUERY PLAN</div> <div>text</div> <div> <div>1</div> <div>Index Scan using pk_realisateur on realisateur (cost=0.28..8.30 rows=1 width=19)</div> </div> <div> <div>2</div> <div>Index Cond: (id = 2800)</div> </div> </div>				

2. `SELECT * FROM realisateur where ID=2800 AND NOM ='spielberg';`

- Testez sans index
- Testez après l'ajout d'un index unique sur ID : `CREATE UNIQUE INDEX IDX_REALISATEUR_ID ON REALISATEUR (ID);`
- Ajoutez ensuite avec un index non unique sur NOM (il peut y avoir des doublons !) : `CREATE INDEX IDX_REALISATEUR_NOM ON REALISATEUR (NOM);` Testez. **Conclusion ?**
- Supprimez les 2 index :
`DROP INDEX IDX_REALISATEUR_NOM;`
`DROP INDEX IDX_REALISATEUR_ID;`
- Créez d'abord l'index non unique sur NOM puis l'index unique sur ID. Testez. **Quel index est le plus volumineux ? Qu'en déduire ?**

Indications :

- Pour connaître la taille de chaque index et table, vous pouvez exécuter la requête suivante :

```
SELECT N.nspname || '.' || C.relname AS "relation",
       CASE WHEN reltype = 0
            THEN pg_size_pretty(pg_total_relation_size(C.oid)) || ' (index)'
```

```

ELSE pg_size_pretty(pg_total_relation_size(C.oid)) || ' (' ||
pg_size_pretty(pg_relation_size(C.oid)) || ' data)'
END AS "size (data)"
FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
LEFT JOIN pg_tables T ON (T.tablename = C.relname)
LEFT JOIN pg_indexes I ON (I.indexname = C.relname)
LEFT JOIN pg_tablespace TS ON TS.spcname = T.tablespace
LEFT JOIN pg_tablespace XS ON XS.spcname = I.tablespace
WHERE nspname NOT IN ('pg_catalog','pg_toast','information_schema')
ORDER BY pg_total_relation_size(C.oid) DESC;

```

Si les index ne viennent pas d'être créés, penser à exécuter l'outil ANALYZE (bouton droit de la souris sur le nom de la BD puis Maintenance...) pour mettre à jour les statistiques.

Maintenance...

Options

Maintenance operation: **VACUUM** ANALYZE REINDEX CLUSTER

Vacuum

FULL ☐ No FREEZE ☐ No

ANALYZE ☒ No

Verbose Messages ☒ Yes

Exemple : sans index

relation text	size (data) text
public.titres	1160 kB (1136 kB data)
public.film	528 kB (496 kB data)
public.realise	392 kB (368 kB data)
public.realisateur	328 kB (304 kB data)
public.seq_film	8192 bytes (8192 bytes data)
public.seq_real	8192 bytes (8192 bytes data)

Exemple : quand index non unique sur nom et index unique sur id de la table realisateur

relation text	size (data) text
public.titres	1160 kB (1136 kB data)
public.realisateur	696 kB (304 kB data)
public.film	528 kB (496 kB data)
public.realise	392 kB (368 kB data)
public.idx_realisateur_nom	216 kB (index)
public.idx_realisateur_id	152 kB (index)
public.seq_real	8192 bytes (8192 bytes data)
public.seq_film	8192 bytes (8192 bytes data)

Si à la place de l'index unique, on crée une PK, on obtient la même chose. Il n'y a donc pas de différence en termes d'indexation entre PK et index unique.

	relation text	size (data) text
1	public.titres	1160 kB (1136 kB data)
2	public.realisateur	696 kB (304 kB data)
3	public.film	528 kB (496 kB data)
4	public.realise	392 kB (368 kB data)
5	public.idx_realisateur_nom	216 kB (index)
6	public.pk_realisateur	152 kB (index)
7	public.seq_real	8192 bytes (8192 bytes data)
8	public.seq_film	8192 bytes (8192 bytes data)

Le volume de la table `realisateur` est de 328 Ko ; les index représentent un volume de 368 Ko. La table `realisateur` n'ayant que 2 champs, les 2 index sont donc plus volumineux que la table, la différence étant imputable au rowid (en général stocké sur 8 octets) permettant de connaître dans quelle page du disque dur est stocké l'enregistrement.

Rappel : en moyenne 40% du volume d'une table doit être indexé.

- Oracle privilégie toujours l'index le moins volumineux quand il a le choix.
- Supprimez les 2 index.

3. `SELECT * FROM realisateur where ID=2800 OR NOM ='spielberg';`

Testez avant et après l'ajout d'un index unique sur ID. Ajoutez ensuite un index non unique sur NOM et testez.

Conclusion ? Comparez par rapport aux plans de la question précédente.

Supprimez les 2 index :

```
DROP INDEX IDX_REALISATEUR_NOM;
DROP INDEX IDX_REALISATEUR_ID;
```

Remarque : un index bitmap est une alternative à un index B-tree. Ils sont utilisés dans le contexte des entrepôts de données (datawarehouse), proposent un faible coût de stockage, sont rapides pour les opérations de lecture mais peu performants quand les MAJ sont nombreuses. Enfin, ils ne sont pas disponibles dans tous les SGBD. Cf. Annexe.

4. `SELECT * FROM realisateur where ID>1000;`

Testez avant et après l'ajout d'un index unique sur ID. Coût de la requête ? **Expliquer**
Supprimez l'index.

5. `SELECT * FROM realisateur where ID>4000;`

Testez seulement après l'ajout d'un index unique sur ID. **Expliquer les différences par rapport à la question précédente.**
Supprimez l'index.

6. `SELECT * FROM TITRES WHERE titre = 'Char';`

Testez après l'ajout d'un index unique sur (id_film, **titre**, langue) : `CREATE UNIQUE INDEX IDX_TITRES ON TITRES(id_film, titre, langue);`

Vous remarquerez ici qu'il s'agit d'un index unique composé de 3 champs (et que titre est en position n°2), ce qui est logique puisque ces 3 champs correspondent à la PK de TITRES. Contrairement à Oracle, PostgreSQL ne peut dans le cas présent utiliser l'index (problème !)

Supprimer l'index précédent : `DROP INDEX IDX_TITRES;`

Créez ensuite un index unique sur (**titre**, id_film, langue) : `CREATE UNIQUE INDEX IDX_TITRES ON TITRES(titre, id_film, langue);`

PostgreSQL passe-t'il par l'index ?

Supprimez l'index précédent et créez un index non unique (car FK) sur titre :

```
DROP INDEX IDX_TITRES;
CREATE INDEX IDX_TITRES_TITRE ON TITRES(titre);
```

Faire une synthèse de ces trois cas.

```
DROP INDEX IDX_TITRES_TITRE;
```


7. `SELECT * FROM TITRES WHERE titre = 'Char' AND ID_FILM=1000;`

Créer un index unique sur (id_film, titre, langue) puis réaliser le plan d'exécution.

`CREATE UNIQUE INDEX IDX_TITRES ON TITRES(id_film, titre, langue);`

Réaliser ensuite le plan d'exécution de la requête suivante :

`SELECT * FROM TITRES WHERE titre = 'Char' OR ID_FILM=1000;`

Qu'en déduire par rapport à la question précédente ?

`DROP INDEX IDX_TITRES;`

8. `SELECT * FROM realisateur WHERE substr(nom,1,2) = 'Sp';`

Testez après l'ajout d'un index non unique sur nom : `CREATE INDEX IDX_REALISATEUR_NOM ON REALISATEUR(NOM);`

Expliquer.

Supprimer l'index : `DROP INDEX IDX_REALISATEUR_NOM;`

Essayer ensuite de créer un index sur fonction (Cf. section 11.7. *Index sur des expressions* de la doc PostgreSQL en ligne) : `CREATE INDEX IDX_REALISATEUR_SUBSTR_NOM ON realisateur(substr(nom,1,2));`

Expliquer.

Supprimer l'index : `DROP INDEX IDX_REALISATEUR_SUBSTR_NOM;`

9. `SELECT t.* FROM Film f JOIN Titres t ON f.id = t.id_film;`

Testez avant et après l'ajout d'un index unique sur id de Film (affichez également l'explain en mode graphique). `CREATE UNIQUE INDEX IDX_FILM_ID ON FILM(id);`

Y a-t-il des modifications ?

Ajouter ensuite un index non unique sur titres(id_film) : `CREATE INDEX IDX_TITRES_ID_FILM ON TITRES(id_film);`

Y a-t-il des modifications ? Pensez au volume de données (et aux statistiques).

Suppression des index :

`DROP INDEX IDX_TITRES_ID_FILM;`

`DROP INDEX IDX_FILM_ID;`

10. `SELECT t.* FROM Film f JOIN Titres t ON f.id = t.id_film WHERE f.id=2800;`

Testez avant et après l'ajout d'un index unique sur id de Film (affichez également l'explain en mode graphique). Y a-t'il des modifications par rapport à la question précédente ?

Ajoutez un index non unique sur titres(id_film) : `CREATE INDEX IDX_TITRES_ID_FILM ON TITRES(id_film);`

Modifications (affichez également l'explain en mode graphique) ?

Expliquez sur quels types de champs ont été positionnés ces index (PK, FK ?). En déduire un principe d'optimisation.

11. `CREATE OR REPLACE VIEW films1995 AS
SELECT F.ID, F.ANNEE, R.NOM
FROM FILM F
JOIN REALISE A ON F.ID= A.ID_FILM
JOIN REALISATEUR R ON A.ID_REAL=R.ID
WHERE ANNEE = 1995;`

`SELECT * FROM films1995;`

Visualisez l'EXPLAIN sur le SELECT précédent (pas besoin d'ajouter des index). Que constatez-vous ?

12. Indexation d'un type TEXT

Vérifier qu'il n'y a plus d'index sur la table TITRES.

Modifier le type de la colonne titre (varchar -> text) : `ALTER TABLE titres ALTER titre TYPE text;`

Créer un index sur la colonne titre : `CREATE INDEX IDX_TITRES_TITRE ON TITRES(titre);`

Lancer l'outil ANALYZE (bouton droit de la souris sur le nom de la base puis *Maintenance*)

Réaliser le plan d'exécution pour la requête : `SELECT * FROM TITRES WHERE titre = 'Char';`

Comparer avec le coût de la question 6 quand il y avait uniquement un index non unique sur titres. Que constatez-vous ?

Supprimer ensuite l'index :

`DROP INDEX IDX_TITRES_TITRE;`

Optimisation de requêtes

Remarque : Il s'agit ici de déterminer si les différentes formes d'écriture d'une requête influent sur ses performances (indépendamment de la présence ou non d'index).

```
13.      SELECT id from film
          Where id >= all(select id from film);

      SELECT id from film f1
          Where not exists (select * from film f2 where f1.id<f2.id);

      SELECT MAX(id) from film;
```

Ces 3 requêtes donnent le même résultat.

SANS INDEX. Vérifier le coût de chaque requête. Classer par ordre d'efficacité.

14. Les 4 requêtes suivantes répondent à la question "Quels sont les réalisateurs (numéros) qui n'ont jamais réalisé de film ?" :

```
SELECT R.id
FROM Realisateur R
WHERE R.id NOT IN (
    SELECT id_real
    FROM Realise
);

SELECT R.id
FROM Realisateur R
WHERE NOT EXISTS (
    SELECT 'X'
    FROM Realise Re
    WHERE Re.id_real=R.id
);

SELECT R.id
FROM Realisateur R
    LEFT JOIN Realise Re ON R.id=Re.id_real
WHERE Re.id_real IS NULL;

SELECT R.id
FROM Realisateur R
    LEFT JOIN Realise Re ON R.id=Re.id_real
GROUP BY R.id
HAVING COUNT(Re.id_real)=0;
```

- Quels sont les plans d'exécution de ces requêtes (sans index) ? Laquelle est la moins coûteuse ?
- Sachant que sous Oracle 11g et 12c, les 4 requêtes ont le même coût (et le même plan d'exécution) et que sous Oracle 10g, EXISTS et OUTER JOIN ont pratiquement le même plan d'exécution et sont beaucoup plus performants que IN, qu'en déduire globalement (notamment quand on migre de version d'un même SGBD ou que l'on change de SGBD) ?
- Ces requêtes sont-elles optimisables via des index ? Combien et lesquels ?

```
15.      Select * from film where id between 2000 and 2001;
          Select * from film where id = 2000 or id= 2001;
          Select * from film where id in (2000, 2001);
          Select * from film where id = 2000
          UNION
          Select * from film where id = 2001;
```

Quelle requête est la plus performante (tester sans index) ? Qu'en déduire ?

16. Opérateur relationnel de « Division » : afficher les réalisateurs qui ont réalisé tous les films.

```
SELECT r.id, r.nom
FROM realisateur r
WHERE NOT EXISTS
```

```

(SELECT 'X'
FROM film f
WHERE NOT EXISTS
  (SELECT 'X'
   FROM realise rea
   WHERE rea.id_film=f.id AND rea.id_real=r.id));

SELECT r.id, r.nom
FROM realisateur r
  JOIN realise rea ON r.id=rea.id_real
GROUP BY r.id, r.nom
HAVING COUNT(DISTINCT rea.id_film) = (
  SELECT COUNT(*) FROM film
);

```

SANS INDEX. Vérifier le coût de chaque requête.

Remarque :

Plans d'exécution sous Oracle 11g :

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		3643	5511
FILTER			
Filter Predicates			
NOT EXISTS (SELECT 0 FROM FILM			
TABLE ACCESS (FULL)	REALISATEUR	5975	7
FILTER			
Filter Predicates			
NOT EXISTS (SELECT 0 FROM F			
TABLE ACCESS (FULL)	FILM	9706	2
TABLE ACCESS (FULL)	REALISE	1	7
Filter Predicates			
AND			
REA.ID_FILM=:B1			
REA.ID_REAL=:B2			

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		60	16
FILTER			
Filter Predicates			
COUNT(\$vm_col_1)= (SELECT CC			
HASH (GROUP BY)		60	16
VIEW	VM_NWVW_1	10245	16
HASH (GROUP BY)		10245	16
HASH JOIN		10245	15
Access Predicates			
R.ID=REA.ID_REAL			
TABLE ACCESS (FULL)	REALISATEUR	5975	7
TABLE ACCESS (FULL)	REALISE	10245	7
SORT (AGGREGATE)		1	
TABLE ACCESS (FULL)	FILM	9706	11

On se rend compte que la seconde écriture est beaucoup plus performante. Cependant, si le calcul était plus complexe dans le 2nd cas (plus de jointures) et le nombre de lignes plus important (seulement 6000 lignes dans la table réalisateur), le COUNT pourrait s'avérer plus gourmand que le NOT EXISTS car l'algorithme du EXISTS d'Oracle est censé être plus performant que celui de PostgreSQL.

REALISER ENSUITE UNE SYNTHESE GLOBALE DES PRINCIPES QUE VOUS POUVEZ TIRER DE CES QUELQUES ESSAIS IMPOSES ET EVENTUELLEMENT DE VOS PROPRES TESTS.

ANNEXE

Quelques OPERATIONS et OPTIONS du plan d'exécution

Opération	Option	Description
SORT		Trie l'ensemble de données sur les colonnes mentionnées dans la partie Sort Key. Cette opération a besoin d'une grande quantité de mémoire.

	GROUP BY	Clause <code>group by</code> regroupe les éléments du même groupe par un tri sur les valeurs des expressions définissant le regroupement
	JOIN	Un tri des n-uplets d'une relation préalable à une jointure par fusion : <code>MERGE JOIN</code>
	ORDER BY	Clause <code>order by</code>
	UNIQUE	Tri afin d'éliminer les doublons (clause <code>distinct</code> par exemple).
	AGGREGATE	Application d'une fonction d'agrégation
FILTER		Par exemple les <code>where</code> et <code>having</code>
MERGE JOIN		La jointure d'assemblage (ou de tri) combine deux listes triées, comme une fermeture éclair. Les 2 côtés de la jointure doivent être pré-triés. Utilisé avec l'opération <code>SORT JOIN</code> (tri)
HASH JOIN		La jointure de hachage charge les enregistrements candidats d'un côté de la jointure dans une table de hachage (marqué avec le mot Hash dans le plan) dont chaque enregistrement est ensuite testé avec l'autre côté de la jointure (l'autre relation est balayée complètement).
NESTED LOOPS		Joint deux tables en récupérant le résultat d'une table et en recherchant chaque ligne de la première table dans la seconde (la seconde table est accédée par une de ses clés).
SCAN	INDEX SCAN	On trouve le n-uplet connaissant son adresse. L'adresse du n-uplet est présente dans l'index B-tree ; accès direct ensuite au tuple de la table.
	INDEX ONLY SCAN	Idem précédent, mais il n'est pas nécessaire d'accéder à la table car l'index dispose de toutes les colonnes pour satisfaire la requête.
	SEQ SCAN	Balayage complet de la table, peut être efficace si la table est
BITMAP INDEX SCAN / BITMAP HEAP SCAN / RECHECK COND		Un <code>INDEX SCAN</code> standard récupère les pointeurs de ligne un par un dans l'index, et visite immédiatement la ligne pointée dans la table. Un parcours de bitmap récupère tous les pointeurs de ligne dans l'index en un coup, les trie en utilisant une structure bitmap en mémoire, puis visite les lignes dans la table dans l'ordre de leur emplacement physique.
HASHAGGREGATE		Utilise une table de hachage temporaire pour grouper les enregistrements. Ne requiert pas de données pré-triées. Elle utilise une grande quantité de mémoire. La sortie n'est pas triée.
GROUPAGGREGATE		Agrège un ensemble pré-trié suivant la clause <code>group by</code> . Cette opération ne place pas de grandes quantités de données en mémoire.

TABLES PARTITIONNEES

Documentation : <https://www.postgresql.org/docs/10/static/ddl-partitioning.html>

1. Création de la table partitionnée Ville

```
DROP TABLE IF EXISTS ville CASCADE;
CREATE TABLE ville (
    idville      bigserial not null,
    nom          varchar(50) not null,
    pays         varchar(50),
    nbhabitants  int
) PARTITION BY LIST (upper(pays));
```

Le partitionnement est effectué par pays.

2. Création de 4 partitions

- 1 seul niveau de partitionnement :

```
CREATE TABLE villeFR PARTITION OF ville FOR VALUES IN ('FRANCE');
CREATE TABLE villeDE PARTITION OF ville FOR VALUES IN ('ALLEMAGNE');
CREATE TABLE villeUS PARTITION OF ville FOR VALUES IN ('USA');

insert into ville (nom, pays, nbhabitants) values ('Annecy', 'France', 120000);
insert into ville (nom, pays, nbhabitants) values ('Lyon', 'France', 500000);
insert into ville (nom, pays, nbhabitants) values ('Berlin', 'Allemagne', 3500000);
insert into ville (nom, pays, nbhabitants) values ('New York', 'USA', 8500000);

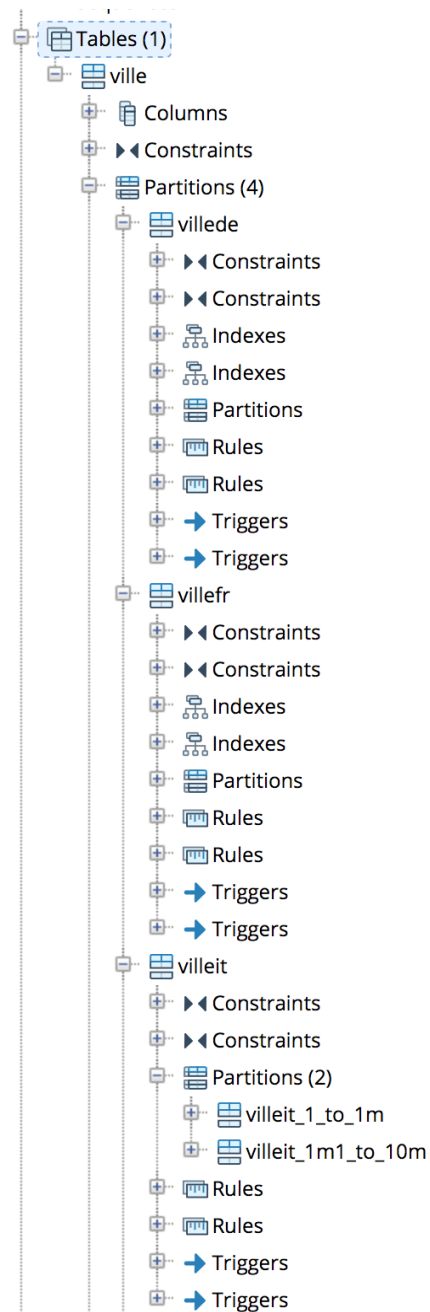
insert into ville (nom, pays, nbhabitants) values ('Rome', 'Italy', 3000000); -- ERREUR
la clé de partition n'est pas trouvée.
```

- 2 niveaux de partitionnement :

```
CREATE TABLE villeIT PARTITION OF ville FOR VALUES IN ('ITALY') PARTITION BY RANGE
(nbhabitants);
CREATE TABLE villeIT_1_to_1M
PARTITION OF villeIT FOR VALUES FROM (1) TO (1000000);
CREATE TABLE villeIT_1M1_to_10M
PARTITION OF villeIT FOR VALUES FROM (1000001) TO (10000000);

insert into ville (nom, pays, nbhabitants) values ('Rome', 'Italy', 3000000);
insert into ville (nom, pays, nbhabitants) values ('Bergame', 'Italy', 120000);
```

On peut voir les partitionnements dans PgAdmin4 :



3. Plan d'exécution

Réaliser le plan d'exécution pour les 2 cas suivants :

```
SELECT * FROM Ville WHERE nom = 'Lyon';
SELECT * FROM Ville WHERE nbhabitants=120000;
```

Remarque : si la table partitionnée est peu volumineuse, le coût d'une requête SELECT est plus important que sans partition (5 fois dans le premier cas, 4 fois dans le 2nd cas). Cependant, si la table partitionnée contient un très grand nombre de lignes, le rapport de coût s'inversera.

4. Suppression de la table partitionnée et de ses partitions

```
DROP TABLE ville CASCADE;
```