

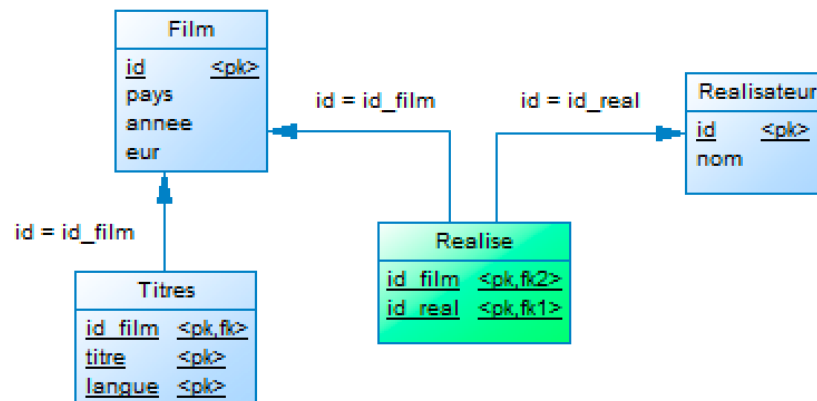
TP noté à réaliser en binôme et à rendre (CASIER sur l'e-campus).

INDEX.....	1
TABLES PARTITIONNEES.....	16

INDEX

PRESENTATION DE LA BASE DE DONNEES

On considère la base de données dont le schéma est le suivant :



Dans cette base, aucune contrainte (PK, FK,...), ni index n'a été créé.

Script :

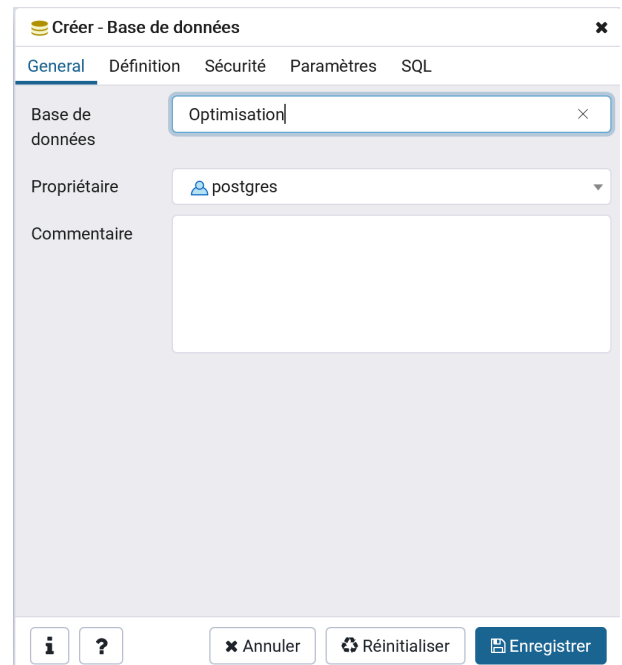
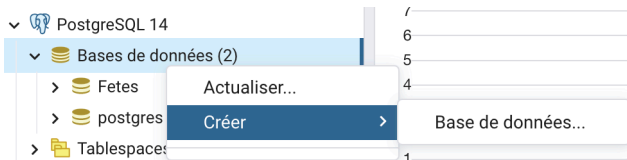
```

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;

```

Les index seront créés par la suite au fur et à mesure pour voir le lien entre indexation et performance. A la place des clés primaires, seront créés des index uniques (une clé primaire génère toujours un index unique pour assurer l'unicité des valeurs) car c'est l'index derrière la PK qui nous intéresse dans ce TP.

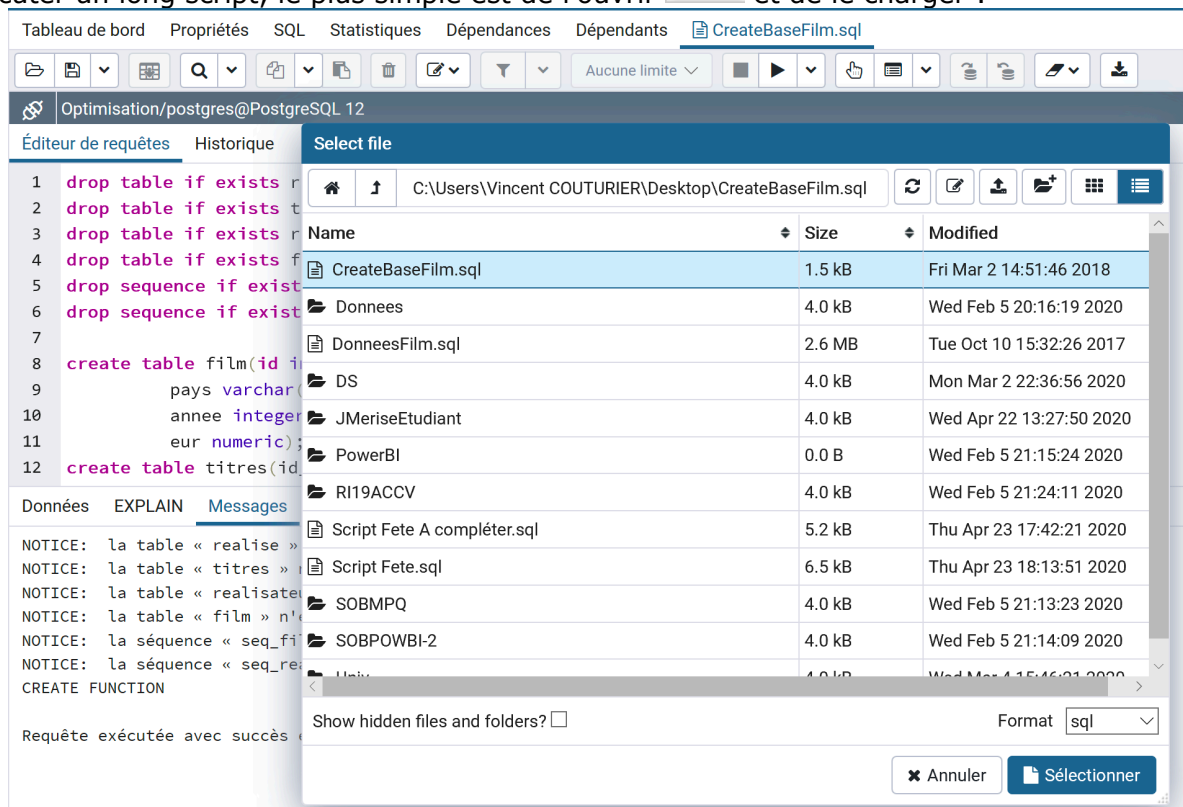
Créer une nouvelle base de données :




Ouvrir une fenêtre SQL (menu *Tools* > *Query Tool*) dans PgAdmin 4.

Exécutez les deux scripts (CreateBaseFilm.Sql puis DonneesFilm.sql) dans le schéma public de cette base de données.

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



Exécuter le code SQL .

Une fois les 2 scripts exécutés, fermer l'éditeur de requête (Query Tool) et en ouvrir un nouveau.

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>
<div> Data Output Explain Messages Query History </div>	
	<div> count bigint </div>
1	20247

INDEX & EXPLAIN SOUS POSTGRESQL

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.

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 afin d'améliorer la rapidité d'exécution.

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

- Coût estimé du lancement ou coût minimal estimé (coût de la requête avant que la récupération des données ne commence). Si l'on compare au fonctionnement d'autres SGBD, ce coût correspond au coût CPU de la requête (plus la requête sera complexe, plus ce coût minimal sera important). 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` aura un coût inférieur). Il inclut le coût CPU et le coût d'entrée/sortie permettant de récupérer les données. Ex. : `cost=0.00..458.00`
Vous vous baserez principalement sur ce coût pour comparer les requêtes.
- Nombre de lignes estimé en sortie (encore une fois, seulement si exécuté jusqu'au bout) ; Ex. : `rows=10000`
- Largeur moyenne estimée (en octets) des lignes en sortie. Ex. : `width=244`

Pour plus d'informations sur *EXPLAIN*, Cf. point 14.1. *Utiliser EXPLAIN* du manuel : <https://docs.postgresql.fr/14/performance-tips.html#using-explain>

INDEX

Pour plus d'informations sur les index gérés par PostgreSQL, Cf. chapitre 11 : <https://docs.postgresql.fr/14/indexes.html>

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 Films;`

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 verrons sont :

- Parcours séquentiel d'une table : `SEQ SCAN`
- Parcours d'index (accès direct en passant par un 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

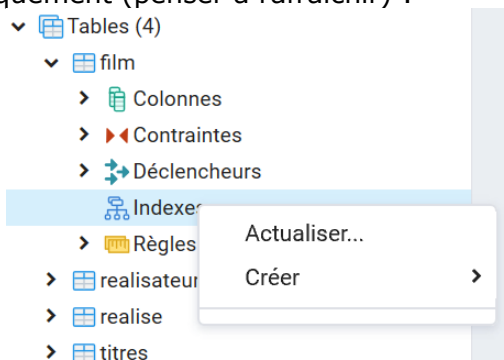
A. OBJECTIF & ETAPES A RESPECTER

Objectif : afficher les plans de requête pour les requêtes suivantes (cf. point B.), et tenter de déterminer l'interaction entre index et coût. 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 expliquant pourquoi le résultat de l'EXPLAIN a changé (ou pas), 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 (requêtes de la partie B) :

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, 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 REQUÊTE SUIVANTE, TOUS LES INDEX DOIVENT AVOIR ETE SUPPRIMES.

Exemple :

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

1 **EXPLAIN SELECT * FROM** film

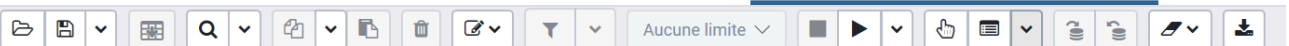
QUERY PLAN

text

1 Seq Scan on film (cost=0.00..159.06 rows=9706 width=17)

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

- Requête (supprimer EXPLAIN) : **SELECT * FROM** film
- Cocher toutes les options de l'explain



Optimisation/postgres@PostgreSQL 13

1 **SELECT * FROM** film

QUERY PLAN

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

1 **SELECT * FROM** film



public.film

public.film		✕
Node Type	Seq Scan	
Parallel Aware	false	
Async Capable	false	
Relation Name	film	
Schema	public	
Alias	film	
Startup Cost	0	
Total Cost	159.06	
Plan Rows	9706	
Plan Width	17	
Actual Startup Time	0.008	
Actual Total Time	1.827	
Actual Rows	9706	
Actual Loops	1	
Output	id,pays,annee,e	ur
Shared Hit Blocks	62	
Shared Read Blocks	0	
Shared Dirtied Blocks	0	
Shared Written Blocks	0	
Local Hit Blocks	0	
Local Read Blocks	0	
Local Dirtied Blocks	0	
Local Written Blocks	0	

Éditeur de requêtes

Historique

1

SELECT * FROM film

Données

EXPLAIN

Messages

Notifications

Graphique

Analyse

Statistiques

#	Noeud	Chronométrages		Lignes			Boucles
		Exclusif	Inclusif	Lignes X	Actuel	Plan	
1.	→ Seq Scan on public.film as film (cost=0..159.06 rows=9706 ...	1.827 ms	1.827 ms	↑ 1	9706	9706	1

Éditeur de requêtes

Historique

1

SELECT * FROM film

Données

EXPLAIN

Messages

Notifications

Graphique

Analyse

Statistiques

Statistique par type de nœud

Type de nœud	Total	Temps passé	% of query
Seq Scan	1	1.827 ms	100%

Statistiques par Relation

Nom de la relation	Nombre de balayages d'index	Temps total	% of query
Type de nœud	Total	Somme des temps	% of relation
public.film	1	1.827 ms	100%
Seq Scan	1	1.827 ms	100%

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

- ii. Avec INDEX UNIQUE (car ce serait la PK) sur le champ ID de film

```
CREATE UNIQUE INDEX idx_film_id ON film(id);
```

```
EXPLAIN SELECT * FROM film;
```

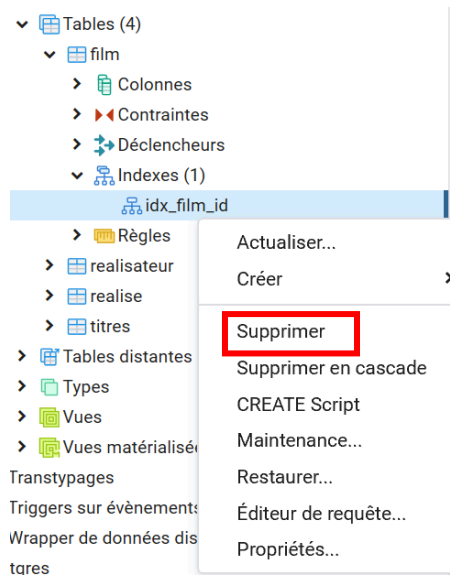
Éditeur de requêtes		Historique	
1	CREATE UNIQUE INDEX idx_film_id ON film(id);		
2			
3	EXPLAIN SELECT * FROM film;		
4			
Données		EXPLAIN	Messages Notifications
QUERY PLAN			
text			
1	Seq Scan on film (cost=0.00..159.06 rows=9706 width=17)		

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 9706 fois) pour récupérer chaque ligne. Grace aux statistiques, PostgreSQL connait en outre le nombre de lignes à récupérer.

- iii. Suppression de l'index :

```
DROP INDEX idx_film_id;
```

OU en mode graphique :



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 film WHERE id=5200;`

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

Remarque : si l'on crée une PK à la place de l'index unique sur ID, on a exactement le même plan d'exécution, ce qui prouve qu'un index est bien créé. Il s'agit d'un index unique qui assure l'unicité des valeurs (pas de doublon).

Éditeur de requêtes Historique

```

1 ALTER TABLE film
2     ADD CONSTRAINT pk_film PRIMARY KEY (id);
3
4 EXPLAIN SELECT * FROM film WHERE id=5200;
5

```

Données EXPLAIN Messages Notifications

	QUERY PLAN
1	Index Scan using pk_film on film (cost=0.29..8.30 rows=1 width=17)
2	Index Cond: (id = 5200)

2. `SELECT * FROM film WHERE id=5200 AND pays = 'CH/FR';`

- Testez sans index.
- Testez après l'ajout d'un index unique sur ID : `CREATE UNIQUE INDEX idx_film_id ON film(id);`
- Ajoutez ensuite avec un index non unique sur PAYS (il peut y avoir des doublons !) : `CREATE INDEX idx_film_pays ON film(pays);` Testez. **Conclusion ?**
- **Quel index est le plus volumineux (cf. indications ci-dessous) ? 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.

Exemple : sans index

	relation text	size (data) text
1	public.titres	1168 kB (1136 kB data)
2	public.film	536 kB (496 kB data)
3	public.realise	400 kB (368 kB data)
4	public.realisateur	336 kB (304 kB data)
5	public.seq_film	8192 bytes (8192 bytes data)
6	public.seq_real	8192 bytes (8192 bytes data)

On remarque que les séquences (numéro s'incrémentant automatiquement) occupent aussi de la place (mais marginale).

Taille de la table `film` : 536-496 (data) = 40 kB (place perdue)

Exemple : quand index non unique sur `pays` et index unique sur `id` de la table `film`

	relation text	size (data) text
1	public.titres	1168 kB (1136 kB data)
2	public.film	880 kB (496 kB data)
3	public.realise	400 kB (368 kB data)
4	public.realisateur	336 kB (304 kB data)
5	public.idx_film_id	232 kB (index)
6	public.idx_film_pays	112 kB (index)
7	public.seq_film	8192 bytes (8192 bytes data)
8	public.seq_real	8192 bytes (8192 bytes data)

Taille de la table `film` : 536 kB (cf. taille sans index) + 112 kB index sur `pays` + 232 kB index sur `id` = 880 kB. Ici, la taille des index représente environ 64% du volume de la table. N'oubliez pas que chaque enregistrement de l'index contient également le rowid

(en général stocké sur 8 octets) permettant de connaître dans quel bloc du disque dur est stocké l'enregistrement

Rappel : en moyenne 40% du volume d'une base de données doit être indexé.

Si à la place de l'index unique, on crée une PK, on obtient la même chose (même taille d'index). 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	1168 kB (1136 kB data)
2	public.film	880 kB (496 kB data)
3	public.realise	400 kB (368 kB data)
4	public.realisateur	336 kB (304 kB data)
5	public.pk_film	232 kB (index)
6	public.idx_film_pays	112 kB (index)
7	public.seq_real	8192 bytes (8192 bytes data)
8	public.seq_film	8192 bytes (8192 bytes data)

- **Oracle et SQL Server privilégient toujours l'index le moins volumineux quand ils ont le choix.**

- Supprimez les 2 index :

```
DROP INDEX idx_film_id;  
DROP INDEX idx_film_pays;
```

3.

```
SELECT * FROM film WHERE id=5200 OR pays = 'CH/FR';
```

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

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. Ici, PostgreSQL utilise ce type d'index et non un B-tree, mais peu importe il s'agit quand même d'un index. Cf. Annexe.

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

- Supprimez les 2 index :

```
DROP INDEX idx_film_id;  
DROP INDEX idx_film_pays;
```

4.

```
SELECT * FROM film WHERE id>2000;
```

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

5.

```
SELECT * FROM film WHERE id>8000;
```

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

Supprimez l'index.

6. Index multicolonne (composite)

Un index multicolonne est normalement utile quand on recherche très souvent sur un couple (ou trinome, etc.) de champs. Il est par exemple utile lorsque l'on effectue fréquemment une recherche combinée (par OR) sur le nom et le prénom d'une personne. Nous allons tester ici le comportement des index multicolonne de PostgreSQL.

- Ajoutez un index sur `film(pays, annee)` :

```
CREATE INDEX idx_film_pays_annee ON film(pays, annee);
```
- Testez les plans d'exécution des requêtes suivantes :

```
SELECT * FROM film WHERE pays = 'CH/FR';
SELECT * FROM film WHERE annee = 1991;
```

NB : seules 17 (sur 9706) lignes sont récupérées dans le cas de la 2^{ème} requête

```
SELECT * FROM film WHERE pays = 'CH/FR' OR annee = 1991;
SELECT * FROM film WHERE annee = 1991 OR pays = 'CH/FR';
```

- Supprimez l'index sur film(pays, annee) : `DROP INDEX idx_film_pays_annee;`
- Ajoutez un index sur film(annee, pays) cette fois (inversion de l'ordre des 2 champs) : `CREATE INDEX idx_film_annee_pays ON film(annee, pays);`
- Testez les plans d'exécution des requêtes suivantes :


```
SELECT * FROM film WHERE pays = 'CH/FR';
SELECT * FROM film WHERE annee = 1991;
SELECT * FROM film WHERE pays = 'CH/FR' OR annee = 1991;
SELECT * FROM film WHERE annee = 1991 OR pays = 'CH/FR';
```
- **Faire une synthèse de ces 2 cas** (index sur pays/annee puis annee/pays). Aide ici : <https://docs.postgresql.fr/14/indexes-multicolumn.html>

Remarque : sous Oracle, on obtient dans tous les cas un passage par l'index :

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			7	5
TABLE ACCESS	FILM	BY INDEX ROWID BATCHED	7	5
INDEX	IDX_FILM_PAYS_ANNEE	RANGE SCAN	7	2

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
TABLE ACCESS	FILM	BY INDEX ROWID BATCHED	1	2
INDEX	IDX_FILM_ANNEE_PAYS	RANGE SCAN	1	1

- Supprimez l'index sur film(annee, pays) : `DROP INDEX idx_film_annee_pays;`
- Créez 2 index monocolonne :


```
CREATE INDEX idx_film_pays ON film(pays);
CREATE INDEX idx_film_annee ON film(annee);
```
- Testez les plans d'exécution des requêtes suivantes :


```
SELECT * FROM film WHERE pays = 'CH/FR';
SELECT * FROM film WHERE annee = 1991;
SELECT * FROM film WHERE pays = 'CH/FR' OR annee = 1991;
SELECT * FROM film WHERE annee = 1991 OR pays = 'CH/FR';
```
- Taille des index :

Taille de l'index sur film(pays, annee)		Taille de l'index sur film(annee, pays)		Taille des 2 index : sur film(annee) et film(pays)	
public.idx_film_pays_annee	144 kB (index)	public.idx_film_annee_pays	136 kB (index)	public.idx_film_pays	112 kB (index)
				public.idx_film_annee	88 kB (index)

Malgré la taille plus réduite d'un index multicolonnes (par rapport à la création de 2 index), son utilisation est-elle pertinente ? Justifiez.

- Supprimez les index :


```
DROP INDEX idx_film_pays;
DROP INDEX idx_film_annee;
```

7. Index multicolonnes (SUITE)

Dans certains cas, la création d'index multi-colonnes est obligatoire, dans le cas notamment des tables de jointure (i.e. les associations dans le modèle conceptuel).

Realise
id_film <pk, fk2>
id_real <pk, fk1>

Par exemple, la table `Realise` a une primary key composée de 2 champs et donc un index unique sera créé sur ces 2 colonnes afin de s'assurer de l'unicité (un film peut être réalisé par plusieurs réalisateurs et un réalisateur peut réaliser plusieurs films, mais il ne peut y avoir de doublon sur le couple <film, réalisateur>). Cet index multicolonnes devrait (normalement) être utilisé dans le cas de la recherche sur une ou plusieurs de ces 2 colonnes.

Que préconisez-vous dans de tels cas (sachant en outre que `id_real` et `id_film` sont aussi FK) ?

Si vous n'êtes pas sûr de votre explication, testez le code suivant :

```
CREATE UNIQUE INDEX idx_realise ON realise(id_film, id_real);
EXPLAIN SELECT * FROM realise WHERE id_film=1;
EXPLAIN SELECT * FROM realise WHERE id_real=2;
```

```
CREATE INDEX idx_realise_idfilm ON realise(id_film);
CREATE INDEX idx_realise_idreal ON realise(id_real);
EXPLAIN SELECT * FROM realise WHERE id_film=1;
EXPLAIN SELECT * FROM realise WHERE id_real=2;
```

Puis supprimez les index.

```
DROP INDEX idx_realise;
DROP INDEX idx_realise_idfilm;
DROP INDEX idx_realise_idreal;
```

8. SELECT * FROM film WHERE SUBSTR(pays,1,2) = 'CH';

- Testez après l'ajout d'un index non unique sur nom : CREATE INDEX idx_film_pays ON film(pays);
- Supprimez l'index : DROP INDEX idx_film_pays;
- Essayez ensuite de créer un index sur fonction (Cf. section 11.7. *Index sur des expressions* de la doc PostgreSQL : <https://docs.postgresql.fr/14/indexes-expressional.html>) : CREATE INDEX idx_film_substr_pays ON film(substr(pays,1,2));

Expliquez.

- Supprimez l'index : DROP INDEX idx_film_substr_pays;

9. SELECT t.id_film, t.titre, t.langue, f.annee, f.pays
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 (car PK) : CREATE UNIQUE INDEX idx_film_id ON film(id);

Vous pourrez également afficher l'explain en mode graphique.

Y a-t-il des modifications sur le plan d'exécution entre avant et après ?

- 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 sur le plan d'exécution ? Pensez au volume de données récupérées (et donc aux statistiques).

- Supprimez des index :
DROP INDEX idx_titres_id_film;
DROP INDEX idx_film_id;

10. SELECT t.id_film, t.titre, t.langue, f.annee, f.pays
FROM Film f
JOIN Titres t ON f.id = t.id_film
WHERE f.pays='FR/BE';

- Testez sans index.

Quelles sont les modifications et pourquoi / à la question précédente sans index (pensez au volume de données récupéré)

- Le seul index qui devrait être positionné par défaut est celui sur la PK de Film (ID). Ajoutez un index unique puis tester.

```
CREATE UNIQUE INDEX idx_film_id ON film(id);
```

Quelles sont les modifications et pourquoi ?

- Indexez la condition de recherche (afin toujours d'optimiser l'accès à la table Film) : testez après l'ajout d'un index sur PAYS : CREATE INDEX idx_film_pays ON film(pays);

Quelles sont les modifications et pourquoi ?

- Ajoutez un index non unique sur TITRES(ID_FILM) : CREATE INDEX idx_titres_id_film ON titres(id_film);

Cette fois un index a été posé sur la 2^{de} condition de jointure (la clé étrangère),

Quelles sont les modifications et pourquoi ?

Expliquez sur quels types de champs ont été positionnés ces index (PK, FK, etc.). En déduire un principe d'optimisation (déjà abordé en Q7).

- Supprimez les index :
DROP INDEX idx_film_pays;
DROP INDEX idx_film_id;
DROP INDEX idx_titres_id_film;

```
11.      CREATE OR REPLACE VIEW films1995 AS
        SELECT id, annee
        FROM film
        WHERE annee = 1995;
```

```
        SELECT * FROM films1995;
```

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

Rappel : une vue est une table virtuelle ne contenant pas de données. A chaque fois que vous « appellerez » la vue (SELECT ... FROM unevue), l'ordre SELECT interne à la vue sera exécuté.

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).

```
12.      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.

13. Les 3 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;
```

- Quels sont les plans d'exécution de ces requêtes (sans index) ? Laquelle est la moins coûteuse ?
- Sachant que sous Oracle 11g et versions supérieures, les 3 requêtes ont exactement 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 ?

```
14.      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 ?

Remarque : sous Oracle, les 3 premières requêtes ont exactement le même coût.

15. 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.

Remarques :

- Plan d'exécution sous PostgreSQL 10 (NOT EXISTS) :

	QUERY PLAN text
1	Nested Loop Anti Join (cost=0.00..5792436113.15 rows=2988 width=19)
2	Join Filter: (NOT (alternatives: SubPlan 1 or hashed SubPlan 2))
3	-> Seq Scan on realisateur r (cost=0.00..97.76 rows=5976 width=19)
4	-> Materialize (cost=0.00..207.59 rows=9706 width=4)
5	-> Seq Scan on film f (cost=0.00..159.06 rows=9706 width=4)
6	SubPlan 1
7	-> Seq Scan on realise rea (cost=0.00..199.67 rows=1 width=0)
8	Filter: ((id_film = f.id) AND (id_real = r.id))
9	SubPlan 2
10	-> Seq Scan on realise rea_1 (cost=0.00..148.45 rows=10245 width=8)

- Plans d'exécution sous Oracle :

- NOT EXISTS :

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			3643	5502
FILTER				
Filter Predicates				
NOT EXISTS (SELECT 0 FROM FILM F WHERE NOT EXISTS (SELECT 0 FROM REALISE REA WHERE REA.ID_FILM=:B1 AND REA.ID_REAL=:B2))				
TABLE ACCESS	REALISATEUR	FULL	5975	7
FILTER				
Filter Predicates				
NOT EXISTS (SELECT 0 FROM REALISE REA WHERE REA.ID_FILM=:B1 AND REA.ID_REAL=:B2)				
TABLE ACCESS	FILM	FULL	9706	2
TABLE ACCESS	REALISE	FULL	1	7
Filter Predicates				
AND				
REA.ID_FILM=:B1				
REA.ID_REAL=:B2				

- COUNT :

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				15
FILTER			60	
Filter Predicates	COUNT(\$vm_col_1)= (SELECT COUNT(*) FROM FILM FILM)			
HASH		GROUP BY	60	15
VIEW	SYS.VM_NWWW_1		10245	15
HASH		GROUP BY	10245	15
HASH JOIN			10245	14
Access Predicates	R.ID=REA.ID_REAL			
TABLE ACCESS	REALISATEUR	FULL	5975	7
TABLE ACCESS	REALISE	FULL	10245	7
SORT		AGGREGATE	1	
TABLE ACCESS	FILM	FULL	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 :

- Synthèse Indexation : sur quels champs faut-il mettre des index ? Quelles conditions pour utiliser les index ? Faut-il créer des index mono-colonne ou multi-colonnes ? etc.
- Synthèse optimisation de requêtes : quel type de requête est préférable ? Quel impact si on change de version d'un même SGBD ou si l'on change de SGBD ? etc.

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 group by 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 : MERGE JOIN
	ORDER BY	Clause order by
	UNIQUE	Tri afin d'éliminer les doublons (clause distinct par exemple).
	AGGREGATE	Application d'une fonction d'agrégation
FILTER		Par exemple les where et having
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 SORT JOIN (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 petite.
BITMAP INDEX SCAN / BITMAP HEAP SCAN / RECHECK COND		Un INDEX SCAN 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 group by. Cette opération ne place pas de grandes quantités de données en mémoire.

Documentation : <https://docs.postgresql.fr/14/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);

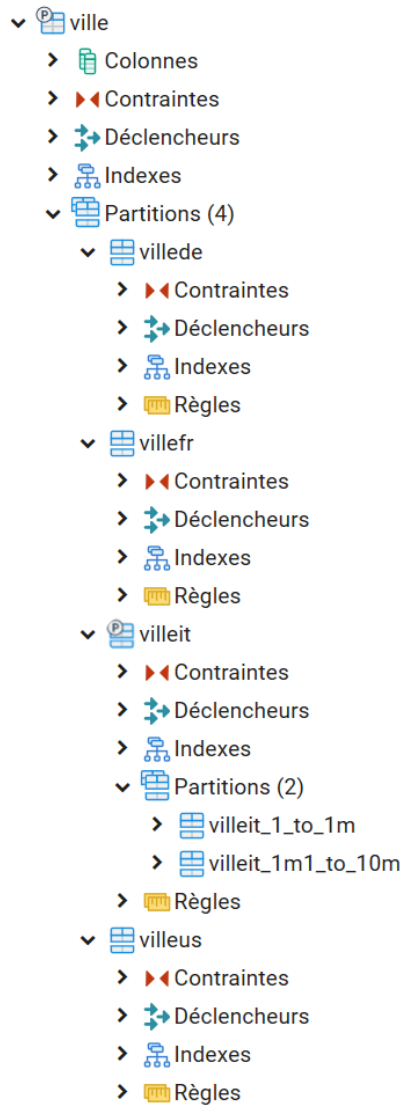
insert into ville (nom, pays, nbhabitants) values ('Rome', 'Italy', 3000000); -- ERREUR
la clé de partition n'est pas trouvée.
```

- 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';`

	QUERY PLAN
	text
1	Append (cost=0.00..68.15 rows=5 width=248)
2	-> Seq Scan on villede ville_1 (cost=0.00..13.63 rows=1 width=248)
3	Filter: ((nom)::text = 'Lyon'::text)
4	-> Seq Scan on villefr ville_2 (cost=0.00..13.63 rows=1 width=248)
5	Filter: ((nom)::text = 'Lyon'::text)
6	-> Seq Scan on villeit_1_to_1m ville_3 (cost=0.00..13.63 rows=1 width=248)
7	Filter: ((nom)::text = 'Lyon'::text)
8	-> Seq Scan on villeit_1m1_to_10m ville_4 (cost=0.00..13.63 rows=1 width=248)
9	Filter: ((nom)::text = 'Lyon'::text)
10	-> Seq Scan on villeus ville_5 (cost=0.00..13.63 rows=1 width=248)
11	Filter: ((nom)::text = 'Lyon'::text)

`SELECT * FROM Ville WHERE nbhabitants=120000;`

	QUERY PLAN
	text
1	Append (cost=0.00..54.52 rows=4 width=248)
2	-> Seq Scan on villede ville_1 (cost=0.00..13.63 rows=1 width=248)
3	Filter: (nbhabitants = 120000)
4	-> Seq Scan on villefr ville_2 (cost=0.00..13.63 rows=1 width=248)
5	Filter: (nbhabitants = 120000)
6	-> Seq Scan on villeit_1_to_1m ville_3 (cost=0.00..13.63 rows=1 width=248)
7	Filter: (nbhabitants = 120000)
8	-> Seq Scan on villeus ville_4 (cost=0.00..13.63 rows=1 width=248)
9	Filter: (nbhabitants = 120000)

Remarque : si la table partitionnée est peu volumineuse, le coût d'une requête *SELECT* est plus important que sans partition (par exemple, ici, le partitionnement implique 5 seq scans dans le cas de la 1^{ère} requête, 4 pour la 2^{nde}). Cependant, si la table partitionnée contient un très grand nombre de lignes, le rapport de coût s'inversera.

Coût sans partitionnement (6 lignes seulement) :

<div> <div>Éditeur de requêtes</div> <div>Historique</div> </div> <div> <div>1</div> <div>EXPLAIN SELECT * FROM Ville WHERE nom = 'Lyon';</div> </div> <div> <div>Données</div> <div>EXPLAIN</div> <div>Messages</div> <div>Notifications</div> </div> <div> <div>QUERY PLAN</div> <div>text</div> <div> <div>1</div> <div>Seq Scan on ville (cost=0.00..13.63 rows=1 width=248)</div> </div> <div> <div>2</div> <div>Filter: ((nom)::text = 'Lyon')::text</div> </div> </div>	<div> <div>Éditeur de requêtes</div> <div>Historique</div> </div> <div> <div>1</div> <div>EXPLAIN SELECT * FROM Ville WHERE nbhabitants=120000;</div> </div> <div> <div>Données</div> <div>EXPLAIN</div> <div>Messages</div> <div>Notifications</div> </div> <div> <div>QUERY PLAN</div> <div>text</div> <div> <div>1</div> <div>Seq Scan on ville (cost=0.00..13.63 rows=1 width=248)</div> </div> <div> <div>2</div> <div>Filter: (nbhabitants = 120000)</div> </div> </div>
--	--

4. Suppression de la table partitionnée et de ses partitions

DROP TABLE ville CASCADE;