## **Formation PERF2**

# **Indexation & SQL Avancé**



# Table des matières

		Sur ce	document	1
		Chers l	lectrices & lecteurs,	1
		À prop	os de DALIBO	1
		Remer	ciements	2
		Forme	de ce manuel	2
		Licence	e Creative Commons CC-BY-NC-SA	2
		Marque	es déposées	3
		-	ns de PostgreSQL couvertes	3
1/	Tech	niques	d'indexation	5
	1.1	Introdu	uction	6
		1.1.1	Objectifs	6
		1.1.2	Introduction aux index	6
		1.1.3	Utilités d'un index	7
		1.1.4	Index et lectures	8
		1.1.5	Index: inconvénients	9
		1.1.6	Index : contraintes pratiques à la création	11
		1.1.7	Types d'index dans PostgreSQL	13
	1.2	Fonctio	onnement d'un index	15
		1.2.1	Structure d'un index	15
		1.2.2	Un index n'est pas magique	17
		1.2.3	Index B-tree	17
		1.2.4	Exemple de structure d'index	19
		1.2.5	Index multicolonnes	21
		1.2.6	Nœuds des index	24
	1.3	Métho	dologie de création d'index	26
		1.3.1	L'index ? Quel index ?	26
		1.3.2	Index et clés étrangères	27
	1.4	Index i	nutilisé	28
		1.4.1	Index utilisable mais non utilisé	28
		1.4.2	Index inutilisable à cause d'une fonction	30
		1.4.3	Index inutilisable à cause d'un LIKE '%'	32
		1.4.4	Index inutilisable car invalide	33
	1.5	Indexa	tion B-tree avancée	34
		1.5.1	Index partiels	34
		1.5.2	Index partiels: cas d'usage	37
		1.5.3	Index partiels: utilisation	38
		1.5.4	Index fonctionnels: principe	38
		1.5.5	Index fonctionnels : conditions	39
		1.5.6	Index fonctionnels: maintenance	44
		1.5.7	Index couvrants: principe	45
		158	Classes d'onérateurs	47

		1.5.9 Conclusion	49
	1.6	Quiz	50
	1.7	Installation de PostgreSQL depuis les paquets communautaires	51
		1.7.1 Sur Rocky Linux 8 ou 9	51
		1.7.2 Sur Debian / Ubuntu	54
		1.7.3 Accès à l'instance depuis le serveur même (toutes distributions)	56
	1.8	Travaux pratiques	58
		1.8.1 Index « simples »	58
		1.8.2 Sélectivité	59
		1.8.3 Index partiels	59
		1.8.4 Index fonctionnels	60
		1.8.5 Cas d'index non utilisés	60
	1.9	Travaux pratiques (solutions)	62
		1.9.1 Index « simples »	62
		1.9.2 Sélectivité	65
		1.9.3 Index partiels	66
		1.9.4 Index fonctionnel	68
		1.9.5 Cas d'index non utilisés	69
21	Inde	exation avancée	73
-,	2.1	Index Avancés	74
	2.2	Index B-tree (rappels)	75
	2.3	Index GIN	76
	2.0	2.3.1 GIN: définition & données non structurées	76
		2.3.2 GIN et les tableaux	77
		2.3.3 GIN pour les JSON et les textes	80
		2.3.4 GIN & données scalaires	81
		2.3.5 GIN: mise à jour	83
	2.4	Index GiST	84
		2.4.1 GiST: cas d'usage	84
		2.4.2 GiST & KNN	85
		2.4.3 GiST & Contraintes d'exclusion	87
	2.5	GIN, GiST & pg_trgm	90
	2.6	Indexation multicolonne: GIN, GiST & bloom	92
	2.7	Index BRIN	97
	2.8	Index hash	106
	2.9	Outils	108
			108
		2.9.2 Identifier les prédicats et des requêtes liées	109
		2.9.3 Extension HypoPG	
		2.9.4 Étude des index à créer	
	2.10	Quiz	
		Travaux pratiques	
		2.11.1 Indexation de motifs avec les varchar_patterns et pg_trgm	
		2.11.2 Index GIN comme bitmap	

		2.11.3 Index GIN et critères multicolonnes	115
		2.11.4 HypoPG	115
	2.12	Travaux pratiques (solutions)	117
		2.12.1 Indexation de motifs avec les varchar_patterns et pg_trgm	117
		2.12.2 Index GIN comme bitmap	121
		2.12.3 Index GIN et critères multicolonnes	123
		2.12.4 HypoPG	125
3/	Exte		131
	3.1	Préambule	
	3.2	pg_trgm	
	3.3	pg_stat_statements	
		3.3.1 pg_stat_statements: mise en place	
		3.3.2 pg_stat_statements: exemple 1	
		3.3.3 pg_stat_statements: exemple 2	
		3.3.4 pg_stat_statements: exemple 3	
	3.4	auto_explain	
	3.5	pg_buffercache	146
	3.6	pg_prewarm	
	3.7	Langages procéduraux	
		3.7.1 Avantages & inconvénients	
	3.8	hll	156
	3.9	Quiz	158
	3.10	Travaux pratiques	159
		3.10.1 Indexation de pattern avec les varchar_patterns et pg_trgm	159
		3.10.2 auto_explain	160
		3.10.3 pg_stat_statements	160
		3.10.4 PL/Python, import de page web et compression	
		3.10.5 PL/Perl et comparaison de performances	162
		3.10.6 hll	164
	3.11	Travaux pratiques (solutions)	166
		3.11.1 Indexation de pattern avec les varchar_patterns et pg_trgm	166
		3.11.2 auto_explain	
		3.11.3 pg_stat_statements	
		3.11.4 PL/Python, import de page web et compression	176
		3.11.5 PL/Perl et comparaison de performances	179
		3.11.6 hll	183
4/		· ·	189
	4.1	Principe & intérêts du partitionnement	
	4.0	4.1.1 Pourquoi partitionner?	
	4.2	Partitionnement applicatif	
	4.0	4.2.1 Partitionnement applicatif	
	4.3	Partitionnement par héritage	
		4.3.1 Partitionnement par héritage	193

	4.4	Partitio	onnement déclaratif
		4.4.1	Partitionnement déclaratif
		4.4.2	Partitionnement par liste
		4.4.3	Partitionnement par liste: implémentation
		4.4.4	Partitionnement par intervalle
		4.4.5	Partitionnement par intervalle: implémentation
		4.4.6	Partitionnement par hachage
		4.4.7	Partitionnement par hachage: principe
		4.4.8	Clé de partitionnement multicolonne
		4.4.9	Sous-partitionnement
		4.4.10	Partition par défaut
		4.4.11	Attacher une partition
		4.4.12	Détacher une partition
		4.4.13	Changer la définition d'une partition
		4.4.14	Supprimer une partition
		4.4.15	Fonctions de gestion et vues système
		4.4.16	Clé primaire et clé de partitionnement
		4.4.17	Indexation
		4.4.18	Planification & performances
		4.4.19	Opérations de maintenance
		4.4.20	Sauvegardes
		4.4.21	Limitations du partitionnement déclaratif et versions
	4.5	Tables	distantes & sharding
	4.6	Extensi	ions & outils
	4.7	Conclu	rsion
	4.8	Quiz .	
	4.9	Travau	x pratiques
		4.9.1	Partitionnement
		4.9.2	Partitionner pendant l'activité
	4.10	Travau	x pratiques (solutions)
		4.10.1	Partitionnement
		4.10.2	Partitionner pendant l'activité
5/		es avand	
	5.1	UUID.	
		5.1.1	UUID : principe
		5.1.2	UUID : avantages
		5.1.3	UUID: inconvénients
		5.1.4	UUID : utilisation sous PostgreSQL
		5.1.5	UUID : une indexation facile
	<b>-</b> -	5.1.6	UUID:résumé
	5.2		tableaux
		5.2.1	Tableaux: principe
		5.2.2	Tableaux : recherche de valeurs
		5.2.3	Tableaux: performances

	5.2.4	Tableaux: indexation	267
5.3	Types	composés	269
	5.3.1	Types composés: généralités	269
5.4	hstore		271
	5.4.1	hstore: principe	271
	5.4.2	hstore: exemple	272
5.5	JSON		274
	5.5.1	JSON & PostgreSQL	274
	5.5.2	Type json	275
	5.5.3	Type jsonb	276
	5.5.4	Validation du format JSON	276
	5.5.5	JSON: Exemple d'utilisation	279
	5.5.6	JSON: construction	280
	5.5.7	JSON: Affichage des attributs	280
	5.5.8	Modifier un JSON	282
	5.5.9	JSON, tableaux et agrégation	284
	5.5.10	Conversions jsonb / relationnel (1)	286
	5.5.11	Conversions jsonb / relationnel (2): JSON_TABLE	289
	5.5.12	JSON: performances	293
	5.5.13	Recherche dans un champ JSON (1)	295
	5.5.14	Recherche dans un champ JSON (2): SQL/JSON et JSONPath	295
	5.5.15	Recherche dans un champ JSON (3): Exemples SQL/JSON & JSONPath	296
	5.5.16	jsonb: indexation (1/2)	298
	5.5.17	jsonb: indexation (2/2)	299
	5.5.18	Pour aller plus loin	300
5.6	XML .		301
	5.6.1	XML: présentation	301
5.7	Objets	binaires	303
	5.7.1	Objets binaires: les types	303
	5.7.2	bytea	304
	5.7.3	Large Object	306
5.8	Quiz .		308
5.9	Travau	x pratiques	
	5.9.1	UUID	
	5.9.2	jsonb	310
	5.9.3	Large Objects	
5.10		x pratiques (solutions)	
		UUID	
		jsonb: lecture de champs	
		jsonb:index GIN jsonb_path_ops	
		jsonb et tableaux	
		Accès JSONPath	
		Index fonctionnel, colonne générée et JSON	
		jsonb et mise à jour	
	5.10.8	Large Objects	340

6/	Fond	ctionnalités avancées pour la performance	341
	6.1	Préambule	342
		6.1.1 Au menu	342
	6.2	Tables temporaires	343
		6.2.1 Tables temporaires : utilité	343
		6.2.2 Tables temporaires : limites et paramétrage	343
	6.3	Tables non journalisées (unlogged)	346
		6.3.1 Tables non journalisées : utilité	346
		6.3.2 Tables non journalisées: mise en place	347
		6.3.3 Bascule d'une table en/depuis unlogged	347
	6.4	Colonnes générées	349
		6.4.1 Colonnes générées : principe	349
		6.4.2 Colonnes générées vs DEFAULT	349
	6.5	JIT	353
		6.5.1 JIT: la compilation à la volée	353
		6.5.2 JIT : qu'est-ce qui est compilé ?	354
		6.5.3 JIT: algorithme « naïf »	355
		6.5.4 Quand le JIT est-il utile?	356
	6.6	Recherche plein texte	357
		6.6.1 Full Text Search: principe	357
		6.6.2 Full Text Search: exemple	358
		6.6.3 Full Text Search : dictionnaires	360
		6.6.4 Full Text Search: stockage & indexation	363
		6.6.5 Full Text Search sur du JSON	365
	6.7	Quiz	367
	6.8	Travaux pratiques	368
		6.8.1 Tables non journalisées	368
		6.8.2 Indexation Full Text	369
	6.9	Travaux pratiques (solutions)	370
		6.9.1 Tables non journalisées	370
		6.9.2 Indexation Full Text	373
Le	s forn	nations Dalibo	375
		Cursus des formations	375
		Les livres blancs	376
		Téléchargement gratuit	376

#### Sur ce document

Formation PERF2	
Indexation & SQL Avancé	
25.03	
N/A	
https://dali.bo/perf2_pdf	
https://dali.bo/perf2_epub	
https://dali.bo/perf2_html	
https://dali.bo/perf2_slides	

Vous trouverez en ligne les différentes versions complètes de ce document. Les solutions de TP ne figurent pas forcément dans la version imprimée, mais sont dans les versions numériques (PDF ou HTML).

#### Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme... Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com¹!

#### À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

<sup>&</sup>lt;sup>1</sup>mailto:formation@dalibo.com

#### Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

#### Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

#### **Licence Creative Commons CC-BY-NC-SA**

Cette formation est sous licence **CC-BY-NC-SA<sup>2</sup>**. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

### Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur http://creativecommons.org/licenses/by-nc-sa/2.0 /fr/legalcode

<sup>&</sup>lt;sup>2</sup>http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

## Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées³ par PostgreSQL Community Association of Canada.

### **Versions de PostgreSQL couvertes**

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

<sup>&</sup>lt;sup>3</sup>https://www.postgresql.org/about/policies/trademarks/

## 1/ Techniques d'indexation



Photo de Maksym Kaharlytskyi<sup>1</sup>, Unsplash licence

¹https://unsplash.com/@qwitka

#### 1.1 INTRODUCTION



- Qu'est-ce qu'un index ?Comment indexer une base ?Les index B-tree dans PostgreSQL

#### 1.1.1 Objectifs



- Comprendre ce qu'est un index
  Maîtriser le processus de création d'index
  Connaître les différents types d'index B-tree et leurs cas d'usages

#### 1.1.2 Introduction aux index



- Uniquement destinés à l'optimisation
   À gérer d'abord par le développeur
   Markus Winand : SQL Performance Explained

Les index ne sont pas des objets qui font partie de la théorie relationnelle. Ils sont des objets physiques qui permettent d'accélérer l'accès aux données. Et comme ils ne sont que des moyens d'optimisation des accès, les index ne font pas non plus partie de la norme SQL. C'est d'ailleurs pour cette raison que la syntaxe de création d'index est si différente d'une base de données à une autre.

La création des index est à la charge du développeur ou du DBA, leur création n'est pas automatique, sauf exception.

Pour Markus Winand, c'est d'abord au développeur de poser les index, car c'est lui qui sait comment ses données sont utilisées. Un DBA d'exploitation n'a pas cette connaissance, mais il connaît généralement mieux les différents types d'index et leurs subtilités, et voit comment les requêtes réagissent en production. Développeur et DBA sont complémentaires dans l'analyse d'un problème de performance.

Le site de Markus Winand, Use the index, Luke<sup>2</sup>, propose une version en ligne de son livre *SQL Performance Explained*, centré sur les index B-tree (les plus courants). Une version française est par ailleurs disponible sous le titre *SQL*: au cœur des performances.

#### 1.1.3 Utilités d'un index



- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire tout récupérer dans l'index (*Index Only Scan*)
- Un index facilite:
  - certains tris
  - certains agrégats
- Obligatoires et automatique pour clés primaires & unicité
  - conseillé pour clés étrangères (FK)

Les index ne changent pas le résultat d'une requête, mais l'accélèrent. L'index permet de pointer l'endroit de la table où se trouve une donnée, pour y accéder directement. Parfois c'est toute une plage de l'index, voire sa totalité, qui sera lue, ce qui est généralement plus rapide que lire toute la table.

Le cas le plus favorable est l'*Index Only Scan* : toutes les données nécessaires sont contenues dans l'index, lui seul sera lu et PostgreSQL ne lira pas la table elle-même.

PostgreSQL propose différentes formes d'index :

- index classique sur une seule colonne d'une table ;
- index composite sur plusieurs colonnes d'une table ;
- index partiel, en restreignant les données indexées avec une clause WHERE;
- index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table;
- index couvrants, contenant plus de champs que nécessaire au filtrage, pour ne pas avoir besoin de lire la table, et obtenir un *Index Only Scan*.

La création des index est à la charge du développeur. Seules exceptions : ceux créés automatiquement quand on déclare des contraintes de clé primaire ou d'unicité. La création est alors automatique.

Les contraintes de clé étrangère imposent qu'il existe déjà une clé primaire sur la table pointée, mais ne crée pas d'index sur la table portant la clé.

<sup>&</sup>lt;sup>2</sup>https://use-the-index-luke.com

#### 1.1.4 Index et lectures



```
Un index améliore les SELECT

- Sans index:

=# SELECT * FROM test WHERE id = 10000;
Temps : 1760,017 ms

- Avec index:

=# CREATE INDEX idx_test_id ON test (id);

=# SELECT * FROM test WHERE id = 10000;
Temps : 27,711 ms
```

L'index est une structure de données qui permet d'accéder rapidement à l'information recherchée. À l'image de l'index d'un livre, pour retrouver un thème rapidement, on préférera utiliser l'index du livre plutôt que lire l'intégralité du livre jusqu'à trouver le passage qui nous intéresse. Dans une base de données, l'index a un rôle équivalent. Plutôt que de lire une table dans son intégralité, la base de données utilisera l'index pour ne lire qu'une faible portion de la table pour retrouver les données recherchées.

Pour la requête d'exemple (avec une table de 20 millions de lignes), on remarque que l'optimiseur n'utilise pas le même chemin selon que l'index soit présent ou non. Sans index, PostgreSQL réalise un parcours séquentiel de la table :

```
QUERY PLAN

Gather (cost=1000.00..193661.66 rows=1 width=4)
Workers Planned: 2

-> Parallel Seq Scan on test (cost=0.00..192661.56 rows=1 width=4)
Filter: (id = 10000)
```

Lorsqu'il est présent, PostgreSQL l'utilise car l'optimiseur estime que son parcours ne récupérera qu'une seule ligne sur les 20 millions que compte la table :

```
QUERY PLAN

Index Only Scan using idx_test_id on test (cost=0.44..8.46 rows=1 width=4)

Index Cond: (id = 10000)
```

Mais l'index n'accélère pas seulement la simple lecture de données, il permet également d'accélérer les tris et les agrégations, comme le montre l'exemple suivant sur un tri :

```
EXPLAIN SELECT id FROM test
WHERE id BETWEEN 1000 AND 1200 ORDER BY id DESC;
```

#### QUERY PLAN

#### 1.1.5 Index: inconvénients

Soit une table test2 telle que:



- L'index n'est pas gratuit!
- Ralentit les écritures
  - maintenance
- Place disque
- Compromis à trouver

La présence d'un index ralentit les écritures sur une table. En effet, il faut non seulement ajouter ou modifier les données dans la table, mais il faut également maintenir le ou les index de cette table.

Les index dégradent surtout les temps de réponse des insertions. Les mises à jour et les suppressions (UPDATE et DELETE) tirent en général parti des index pour retrouver les lignes concernées par les modifications. Le coût de maintenance de l'index est secondaire par rapport au coût de l'accès aux données.

```
CREATE TABLE test2 (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    valeur INTEGER,
    commentaire TEXT
);

La table est chargée avec pour seul index présent celui sur la clé primaire:

INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 100000000
Durée : 35253,228 ms (00:35,253)
```

Un index supplémentaire est créé sur une colonne de type entier :

```
CREATE INDEX idx_test2_valeur ON test2 (valeur);
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 44410,775 ms (00:44,411)
```

Un index supplémentaire est encore créé, mais cette fois sur une colonne de type texte :

```
CREATE INDEX idx_test2_commentaire ON test2 (commentaire);
INSERT INTO test2 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 207075,335 ms (03:27,075)
```

On peut comparer ces temps à l'insertion dans une table similaire dépourvue d'index :

```
CREATE TABLE test3 AS SELECT * FROM test2;
INSERT INTO test3 (valeur, commentaire)
SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 14758,503 ms (00:14,759)
```

La table test2 a été vidée préalablement pour chaque test.

Enfin, la place disque utilisée par ces index n'est pas négligeable :

Pour ces raisons, on ne posera pas des index systématiquement avant de se demander s'ils seront utilisés. L'idéal est d'étudier les plans de ses requêtes et de chercher à optimiser.

#### 1.1.6 Index : contraintes pratiques à la création



```
- Lourd...

-- bloque les écritures !

CREATE INDEX ON matable ( macolonne ) ;

-- ne bloque pas, peut échouer

CREATE INDEX CONCURRENTLY ON matable ( macolonne ) ;

- Si fragmentation :

REINDEX INDEX nomindex ;

REINDEX TABLE CONCURRENTLY nomtable ;

- Paramètres :

- maintenance_work_mem (sinon: fichier temporaire!)

- max_parallel_maintenance_workers
```

#### Création d'un index :

Bien sûr, la durée de création de l'index dépend fortement de la taille de la table. PostgreSQL va lire toutes les lignes et trier les valeurs rencontrées. Ce peut être lourd et impliquer la création de fichiers temporaires.

Si l'on utilise la syntaxe classique, toutes les écritures sur la table sont bloquées (mises en attente) pendant la durée de la création de l'index (verrou *ShareLock*). Les lectures restent possibles, mais cette contrainte est parfois rédhibitoire pour les grosses tables.

#### **Clause CONCURRENTLY:**

Ajouter le mot clé CONCURRENTLY permet de rendre la table accessible en écriture. Malheureusement, cela nécessite au minimum deux parcours de la table, et donc alourdit et ralentit la construction de l'index. Dans quelques cas défavorables (entre autres l'interruption de la création de l'index), la création échoue et l'index existe mais est invalide :

L'index est inutilisable et doit être supprimé et recréé, ou bien réindexé. Pour les détails, voir la documentation officielle<sup>3</sup>.

Une supervision peut détecter des index invalides avec cette requête, qui ne doit jamais rien ramener :

```
SELECT indexrelid::regclass AS index, indrelid::regclass AS table
FROM pg_index
WHERE indisvalid = false;
```

#### Réindexation:

Comme les tables, les index sont soumis à la fragmentation. Celle-ci peut cependant monter assez haut sans grande conséquence pour les performances. De plus, le nettoyage des index est une des étapes des opérations de VACUUM<sup>4</sup>.

Une reconstruction de l'index est automatique lors d'un VACUUM FULL de la table.

Certaines charges provoquent une fragmentation assez élevée, typiquement les tables gérant des files d'attente. Une réindexation reconstruit totalement l'index. Voici quelques variantes de l'ordre :

```
REINDEX INDEX pgbench_accounts_bid_idx; -- un seul index
REINDEX TABLE pgbench_accounts; -- tous les index de la table
REINDEX (VERBOSE) DATABASE pgbench; -- tous ceux de la base, avec détails

Il existe là aussi une clause CONCURRENTLY:

REINDEX (VERBOSE) INDEX CONCURRENTLY pgbench_accounts_bid_idx;
```

(En cas d'échec, on trouvera là aussi des index invalides, suffixés avec \_\_ccnew , à côté des index préexistants toujours fonctionnels et que PostgreSQL n'a pas détruits.)

#### Paramètres :

La rapidité de création d'un index dépend essentiellement de la mémoire accordée, définie dans maintenance\_work\_mem. Si elle ne suffit pas, le tri se fera dans des fichiers temporaires plus lents. Sur les serveurs modernes, le défaut de 64 Mo est ridicule, et on peut monter aisément à :

```
SET maintenance_work_mem = '2GB';
```

Attention de ne pas saturer la mémoire en cas de création simultanée de nombreux gros index (lors d'une restauration avec pg\_restore notamment).

Si le serveur est bien doté en CPU, la parallélisation de la création d'index peut apporter un gain en temps appréciable. La valeur par défaut est :

```
SET max_parallel_maintenance_workers = 2 ;
```

et devrait même être baissée sur les plus petites configurations.

 $<sup>{\</sup>it ^3} https://docs.postgresql.fr/current/sql-create index.html \#SQL-CREATEINDEX-CONCURRENTLY$ 

<sup>&</sup>lt;sup>4</sup>https://dali.bo/m5\_html#fonctionnement-de-vacuum

#### 1.1.7 Types d'index dans PostgreSQL



- Défaut : B-tree classique (équilibré)
- UNIQUE (préférer la contrainte)
- Mais aussi multicolonne, fonctionnel, partiel, couvrant
- Index spécialisés : hash, GiST, GIN, BRIN, HNSW....

Par défaut un CREATE INDEX créera un index de type B-tree, de loin le plus courant. Il est stocké sous forme d'arbre équilibré, avec de nombreux avantages :

- les performances se dégradent peu avec la taille de l'arbre (les temps de recherche sont en O(log(n)), donc fonction du logarithme du nombre d'enregistrements dans l'index);
- l'accès concurrent est excellent, avec très peu de contention entre processus qui insèrent simultanément.

Toutefois les B-tree ne permettent de répondre qu'à des questions très simples, portant sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre tout de même la majorité des cas.

#### Contrainte d'unicité et index :

Un index peut être déclaré UNIQUE pour provoquer une erreur en cas d'insertion de doublons. Mais on préférera généralement déclarer une *contrainte* d'unicité (notion fonctionnelle), qui techniquement, entraînera la création d'un index.

Par exemple, sur cette table personne:

```
$ CREATE TABLE personne (id int, nom text);
```

\$ \d personne

	Туре	cle « public.persor   Collationnement	NULL-able	•
id	integer text		   	   

on peut créer un index unique :

```
$ CREATE UNIQUE INDEX ON personne (id);
```

\$ \d personne

La contrainte d'unicité est alors implicite. La suppression de l'index se fait sans bruit :

```
DROP INDEX personne_id_idx;
```

Définissons une contrainte d'unicité sur la colonne plutôt qu'un index :

```
ALTER TABLE personne ADD CONSTRAINT unique_id UNIQUE (id);
```

\$ \d personne

Un index est également créé. La contrainte empêche sa suppression :

```
DROP INDEX unique_id ;
```

```
ERREUR: n'a pas pu supprimer index unique_id car il est requis par contrainte
unique_id sur table personne
ASTUCE: Vous pouvez supprimer contrainte unique_id sur table personne à la
place.
```

Le principe est le même pour les clés primaires.

#### Indexation avancée :

Il faut aussi savoir que PostgreSQL permet de créer des index B-tree :

- sur plusieurs colonnes;
- sur des résultats de fonction;
- sur une partie des valeurs indexées ;
- intégrant des champs non indexés mais souvent récupérés avec les champs indexés (index couvrants).

D'autres types d'index que B-tree existent, destinés à certains types de données ou certains cas d'optimisation précis.

#### 1.2 FONCTIONNEMENT D'UN INDEX



#### 1.2.1 Structure d'un index



- Structure associant des clés (termes) à des localisations (pages)
- Structure de données spécialisée, plusieurs types
- Séparée de la table
- Analogies:
  - fiches en carton des bibliothèques avant l'informatique (B-tree)
  - index d'un livre technique (GIN)

Les fiches en carton des anciennes bibliothèques sont un bon équivalent du type d'index le plus courant utilisé par les bases de données en général et PostgreSQL en particulier : le B-tree.

Lorsque l'on recherche des ouvrages dans la bibliothèque, il est possible de parcourir l'intégralité du bâtiment pour chercher les livres qui nous intéressent. Ceci prend énormément de temps. La biblio-

thèque peut être triée, mais ce tri ne permet pas forcément de trouver facilement le livre. Ce type de recherche trouve son analogie sous la forme du parcours complet d'une table (Seq Scan).

Une deuxième méthode pour localiser l'ouvrage consiste à utiliser un index. Sur fiche carton ou sous forme informatique, cet index associe par exemple le nom d'auteur à un ensemble de références (emplacements dans les rayonnages) où celui-ci est présent. Ainsi, pour trouver les œuvres de Proust avec l'index en carton, il suffit de parcourir les fiches, dont l'intégralité tient devant l'utilisateur. La fiche indique des références dans plusieurs rayons et il faudra aller se déplacer pour trouver les œuvres, en allant directement aux bons rayons.

Dans une base de données, le fonctionnement d'un index est très similaire. En effet, comme dans une bibliothèque, l'index est une structure de données à part, qui n'est pas strictement nécessaire à l'exploitation des informations, et qui est principalement utilisée pour la recherche dans l'ensemble de données. Cette structure de données possède un coût de maintenance, dans les deux cas : toute modification des données entraîne des modifications de l'index afin de le maintenir à jour. Et un index qui n'est pas à jour peut provoquer de gros problèmes. Dans le doute, on peut jeter l'index et le recréer de zéro sans problème d'intégrité des données originales.

Il peut y avoir plusieurs index suivant les besoins. L'index trié par auteur ne permet pas de trouver un livre dont on ne connaît que le titre (sauf à lire toutes les fiches). Il faut alors un autre index classé par titre.

Pour filer l'analogie : un index peut être multicolonne (les fiches en carton triées par auteur le sont car elles contiennent le titre, et pas que la référence dans les rayons). L'index peut répondre à une demande à lui seul : il suffit pour compter le nombre de livres de Marcel Proust (c'est le principe des *Index Only Scans*). Une fiche d'un index peut contenir des informations supplémentaires (dates de publication, éditeur...) pour faciliter d'autres recherches sans aller dans les rayons (index « couvrant »).

Dans la réalité comme dans une base de données, il y a un dilemme quand il faut récupérer de très nombreuses données : soit aller chercher de nombreux livres un par un dans les rayons, soit balayer tous les livres systématiquement dans l'ordre où ils viennent pour éviter trop d'allers-retours.

Autres types d'index non informatiques similaires au B-tree :

- les tables décennales de l'État civil, qui pointent vers un endroit précis des registres des actes de naissance, mariage ou décès d'une commune ;
- l'index d'un catalogue papier.

L'index d'un livre technique ou d'un livre de recettes cible des parties des données et non les données elles-mêmes (comme le titre). Il s'approche plus d'un autre type d'index, le GIN, qui existe aussi dans PostgreSQL.

Un annuaire téléphonique papier présente les données sous un mode strictement ordonné. Cette intégration entre table et index n'a pas d'équivalent sous PostgreSQL mais existe dans d'autres moteurs de bases de données.

#### 1.2.2 Un index n'est pas magique



- Un index ne résout pas tout
  Importance de la conception du schéma de données
  Importance de l'écriture de requêtes SQL correctes

Bien souvent, la création d'index est vue comme le remède à tous les maux de performance subis par une application. Il ne faut pas perdre de vue que les facteurs principaux affectant les performances vont être liés à la conception du schéma de données, et à l'écriture des requêtes SQL.

Pour prendre un exemple caricatural, un schéma EAV (Entity-Attribute-Value, ou entité-clé-valeur) ne pourra jamais être performant, de part sa conception. Bien sûr, dans certains cas, une méthodologie pertinente d'indexation permettra d'améliorer un peu les performances, mais le problème réside là dans la conception même du schéma. Il est donc important dans cette phase de considérer la manière dont le modèle va influer sur les méthodes d'accès aux données, et les implications sur les performances.

De même, l'écriture des requêtes elles-mêmes conditionnera en grande partie les performances observées sur l'application. Par exemple, la mauvaise pratique (souvent mise en œuvre accidentellement via un ORM) dite du « N+1 » ne pourra être corrigée par une indexation correcte : celle-ci consiste à récupérer une collection d'enregistrement (une requête) puis d'effectuer une requête pour chaque enregistrement afin de récupérer les enregistrements liés (N requêtes). Dans ce type de cas, une jointure est bien plus performante. Ce type de comportement doit encore une fois être connu de l'équipe de développement, car il est plutôt difficile à détecter par une équipe d'exploitation.

De manière générale, avant d'envisager la création d'index supplémentaires, il convient de s'interroger sur les possibilités de réécriture des requêtes, voire du schéma.

#### 1.2.3 Index B-tree



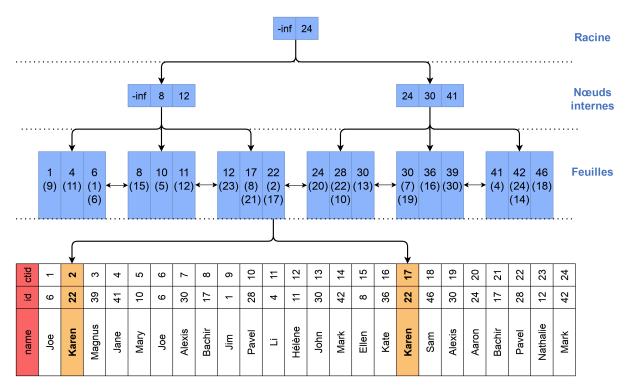
- Type d'index le plus courantet le plus simple
- Utilisable pour les contraintes d'unicité
  Supporte les opérateurs : < , <= , = , >= , >
- Supporte le tri
- Ne peut pas indexer des colonnes de plus de 2,6 ko

L'index B-tree est le plus simple conceptuellement parlant. Sans entrer dans les détails, un index B-tree est par définition équilibré: ainsi, quelle que soit la valeur recherchée, le coût est le même lors du parcours d'index. Ceci ne veut pas dire que toute requête impliquant l'index mettra le même temps! En effet, si chaque clé n'est présente qu'une fois dans l'index, celle-ci peut être associée à une multitude de valeurs, qui devront alors être cherchées dans la table.

L'algorithme utilisé par PostgreSQL pour ce type d'index suppose que chaque page peut contenir au moins trois valeurs. Par conséquent, chaque valeur ne peut excéder un peu moins d'½ de bloc, soit environ 2,6 ko. La valeur en question correspond donc à la totalité des données de toutes les colonnes de l'index pour une seule ligne. Si l'on tente de créer ou maintenir un index sur une table ne satisfaisant pas ces prérequis, une erreur sera renvoyée, et la création de l'index (ou l'insertion/mise à jour de la ligne) échouera. Ces champs sont souvent des longs textes ou des champs composés dont on cherchera plutôt des parties, et un index B-tree n'est de toute façon pas adapté. Si un index de type B-tree est tout de même nécessaire sur les colonnes en question, pour des recherches sur l'intégralité de la ligne, les index de type hash sont plus adaptés (mais ils ne supportent que l'opérateur = ).

## 1.2.4 Exemple de structure d'index





Ce schéma présente une vue très simplifiée d'une table (en blanc, avec ses champs id et name) et d'un index B-tree sur id (en bleu), tel que le créerait :

CREATE INDEX mon\_index ON ma\_table (id) ;

Un index B-tree peut contenir trois types de nœuds :

- la racine : elle est unique c'est la base de l'arbre ;
- des nœuds internes : il peut y en avoir plusieurs niveaux ;
- des feuilles : elles contiennent :
  - les valeurs indexées (triées!);
  - les valeurs incluses (si applicable);
  - les positions physiques (ctid), ici entre parenthèses et sous forme abrégée, car la forme réelle est (numéro de bloc, position de la ligne dans le bloc);
  - l'adresse de la feuille précédente et de la feuille suivante.

La racine et les nœuds internes contiennent des enregistrements qui décrivent la valeur minimale de chaque bloc du niveau inférieur et leur adresse (ctid).

Lors de la création de l'index, il ne contient qu'une feuille. Lorsque cette feuille se remplit, elle se divise en deux et un nœud racine est créé au-dessus. Les feuilles se remplissent ensuite progressivement et se séparent en deux quand elles sont pleines. Ce processus remplit progressivement la racine. Lorsque la racine est pleine, elle se divise en deux nœuds internes, et une nouvelle racine est crée au-dessus. Ce processus permet de garder un arbre équilibré.

Recherchons le résultat de :

```
SELECT name FROM ma_table WHERE id = 22
```

en passant par l'index.

- En parcourant la racine, on cherche un enregistrement dont la valeur est strictement supérieure à la valeur que l'on recherche. Ici, 22 est plus petit que 24 : on explore donc le nœud de gauche.
- Ce nœud référence trois nœuds inférieurs (ici des feuilles). On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud: pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre. Ici, 22 est plus grand que 12, on explore donc le nœud de droite au niveau inférieur.
- Un arbre B-tree peut bien évidemment avoir une profondeur plus grande, auquel cas l'étape précédente est répétée.
- Une fois arrivé sur une feuille, il suffit de la parcourir pour récupérer l'ensemble des positions physiques des lignes correspondants au critère. Ici, la feuille nous indique qu'à la valeur 22 correspondent deux lignes aux positions 2 et 17. Lorsque la valeur recherchée est supérieure ou égale à la plus grande valeur du bloc, PostgreSQL va également lire le bloc suivant. Ce cas de figure peut se produire si PostgreSQL a divisé une feuille en deux avant ou même pendant la recherche que nous exécutons. Ce serait par exemple le cas si on cherchait la valeur 30.
- Pour trouver les valeurs de name, il faut aller chercher dans la table même les lignes aux positions trouvées dans l'index. D'autre part, les informations de visibilité des lignes doivent aussi être trouvées dans la table. (Il existe des cas où la recherche peut éviter cette dernière étape : ce sont les *Index Only Scan*.)

Même en parcourant les deux structures de données, si la valeur recherchée représente une assez petite fraction des lignes totales, le nombre d'accès disques sera donc fortement réduit. En revanche, au lieu d'effectuer des accès séquentiels (pour lesquels les disques durs classiques sont relativement performants), il faudra effectuer des accès aléatoires, en *sautant* d'une position sur le disque à une autre. Le choix est fait par l'optimiseur.

Supposons désormais que nous souhaitions exécuter une requête sans filtre, mais exigeant un tri, du type :

```
SELECT id FROM ma_table ORDER BY id ;
```

L'index peut nous aider à répondre à cette requête. En effet, toutes les feuilles sont liées entre elles, et permettent ainsi un parcours ordonné. Il nous suffit donc de localiser la première feuille (la plus à gauche), et pour chaque clé, récupérer les lignes correspondantes. Une fois les clés de la feuille traitées, il suffit de suivre le pointeur vers la feuille suivante et de recommencer.

L'alternative consisterait à parcourir l'ensemble de la table, et trier toutes les lignes afin de les obtenir dans le bon ordre. Un tel tri peut être très coûteux, en mémoire comme en temps CPU. D'ailleurs,

de tels tris débordent très souvent sur disque (via des fichiers temporaires) afin de ne pas garder l'intégralité des données en mémoire.

Pour les requêtes utilisant des opérateurs d'inégalité, on voit bien comment l'index peut là aussi être utilisé. Par exemple, pour la requête suivante :

```
SELECT * FROM ma_table WHERE id <= 10 AND id >= 4 ;
```

Il suffit d'utiliser la propriété de tri de l'index pour parcourir les feuilles, en partant de la borne inférieure, jusqu'à la borne supérieure.

Dernière remarque : ce schéma ne montre qu'une entrée d'index pour 22, bien qu'il pointe vers deux lignes. En fait, il y avait bien deux entrées pour 22 avant PostgreSQL 13. Depuis cette version, PostgreSQL sait dédupliquer les entrées pour économiser de la place.

#### 1.2.5 Index multicolonnes



- Possibilité d'indexer plusieurs colonnes :

```
CREATE INDEX ON ma_table (id, name) ;
```

- Ordre des colonnes **primordial** 
  - accès direct aux premières colonnes de l'index
  - pour les autres, PostgreSQL lira tout l'index ou ignorera l'index

Il est possible de créer un index sur plusieurs colonnes. Il faut néanmoins être conscient des requêtes supportées par un tel index. Admettons que l'on crée une table d'un million de lignes avec un index sur trois champs :

```
CREATE TABLE t1 (c1 int, c2 int, c3 int, c4 text);
INSERT INTO t1 (c1, c2, c3, c4)
SELECT i*10,j*5,k*20, 'text'||i||j||k
FROM generate_series (1,100) i
CROSS JOIN generate_series(1,100) j
CROSS JOIN generate_series(1,100) k;

CREATE INDEX ON t1 (c1, c2, c3);

VACUUM ANALYZE t1;
-- Figer des paramètres pour l'exemple
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1;
SET random_page_cost TO 4;
```

L'index est optimal pour répondre aux requêtes portant sur les premières colonnes de l'index :

```
QUERY PLAN

Index Scan using t1_c1_c2_c3_idx on t1 (cost=0.42..8.45 rows=1 width=22)
Index Cond: ((c1 = 1000) AND (c2 = 500) AND (c3 = 2000))

Et encore plus quand l'index permet de répondre intégralement au contenu de la requête:
```

Et encore plus quand i muex permet de repondre integralement au contenu de la requet

```
QUERY PLAN

Index Only Scan using t1_c1_c2_c3_idx on t1 (cost=0.42..6.33 rows=95 width=12)

Index Cond: ((c1 = 1000) AND (c2 = 500))
```

Mais si les premières colonnes de l'index ne sont pas spécifiées, alors l'index devra être parcouru en grande partie.

Cela reste plus intéressant que parcourir toute la table, surtout si l'index est petit et contient toutes les données du SELECT. Mais le comportement dépend alors de nombreux paramètres, comme les statistiques, les estimations du nombre de lignes ramenées et les valeurs relatives de seq\_page\_cost

```
SET random_page_cost TO 0.1 ; SET seq_page_cost TO 0.1 ; -- SSD

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

QUERY PLAN

Index Scan using t1_c1_c2_c3_idx on t1 (...) (...)
Index Cond: (c3 = 2000)
Buffers: shared hit=3899
Planning:
Buffers: shared hit=15
Planning Time: 0.218 ms
Execution Time: 67.081 ms
```

Noter que tout l'index a été lu.

et random\_page\_cost :

Mais pour limiter les aller-retours entre index et table, PostgreSQL peut aussi décider d'ignorer l'index et de parcourir directement la table :

```
SET random_page_cost TO 4 ; SET seq_page_cost TO 1 ; -- défaut (disque mécanique)

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

QUERY PLAN

Seq Scan on t1 (cost=0.00..18871.00 rows=9600 width=22) (...)

Filter: (c3 = 2000)

Rows Removed by Filter: 990000

Buffers: shared hit=6371

Planning Time: 0.178 ms

Execution Time: 114.572 ms
```

Concernant les *range scans* (requêtes impliquant des opérateurs d'inégalité, tels que <, <=, >), celles-ci pourront être satisfaites par l'index de manière quasi optimale si les opérateurs d'inégalité sont appliqués sur la dernière colonne requêtée, et de manière sub-optimale s'ils portent sur les premières colonnes.

Cet index pourra être utilisé pour répondre aux requêtes suivantes de manière optimale :

```
SELECT * FROM t1 WHERE c1 = 20 ;

SELECT * FROM t1 WHERE c1 = 20 AND c2 = 50 AND c3 = 400 ;

SELECT * FROM t1 WHERE c1 = 10 AND c2 <= 4 ;
```

Il pourra aussi être utilisé, mais de manière bien moins efficace, pour les requêtes suivantes, qui bénéficieraient d'un index sur un ordre alternatif des colonnes :

```
SELECT * FROM t1 WHERE c1 = 100 AND c2 >= 80 AND c3 = 40 ; SELECT * FROM t1 WHERE c1 < 100 AND c2 = 100 ;
```

Le plan de cette dernière requête est :

```
Bitmap Heap Scan on t1 (cost=2275.98..4777.17 rows=919 width=22) (...)

Recheck Cond: ((c1 < 100) AND (c2 = 100))

Heap Blocks: exact=609

Buffers: shared hit=956

→ Bitmap Index Scan on t1_c1_c2_c3_idx (cost=0.00..2275.76 rows=919 width=0)

→ (...)

Index Cond: ((c1 < 100) AND (c2 = 100))

Buffers: shared hit=347

Planning Time: 0.227 ms

Execution Time: 15.596 ms
```

Les index multicolonnes peuvent aussi être utilisés pour le tri comme dans les exemples suivants. Il n'y a pas besoin de trier (ce peut être très coûteux) puisque les données de l'index sont triées. Ici le cas est optimal puisque l'index contient toutes les données nécessaires :

```
SELECT * FROM t1 ORDER BY c1;
SELECT * FROM t1 ORDER BY c1, c2;
SELECT * FROM t1 ORDER BY c1, c2, c3;
```

Le plan de cette dernière requête est :

```
Index Scan using t1_c1_c2_c3_idx on t1 (cost=0.42..55893.66 rows=1000000 width=22) \hookrightarrow (...)

Buffers: shared hit=1003834
Planning Time: 0.282 ms
Execution Time: 425.520 ms
```

Il est donc nécessaire d'avoir une bonne connaissance de l'application (ou de passer du temps à observer les requêtes consommatrices) pour déterminer comment créer des index multicolonnes pertinents pour un nombre maximum de requêtes.

#### 1.2.6 Nœuds des index



- Index Scan
- Bitmap Scan
- Index Only Scan
  - idéal pour les performances
- et les variantes parallélisées

L'optimiseur a le choix entre plusieurs parcours pour utiliser un index, principalement suivant la quantité d'enregistrements à récupérer :

#### 1.2.6.1 Index Scan

Un *Index Scan* est optimal quand il y a peu d'enregistrements à récupérer. Noter qu'il comprend l'accès à l'index *et* celui à la table ensuite.

#### 1.2.6.2 Bitmap Scan

Le *Bitmap Scan* est utile quand il y a plus de lignes, ou quand on veut lire plusieurs index d'une même table pour satisfaire plusieurs conditions de filtre.

Il se décompose en deux nœuds : un *Bitmap Index Scan* qui récupère des blocs d'index, et un *Bitmap Heap Scan* qui va chercher les blocs dans la table.

Typiquement, ce nœud servira pour des recherches de plages de valeurs ou de grandes quantités de lignes.



Une bonne corrélation des données avec leur emplacement physique favorise un *Index Scan* par rapport à un *Bitmap Scan*, et un *Bitmap Scan* par rapport à un *Seq Scan*. Une estimation de cette corrélation est consultable, pour chaque colonne de chaque table, dans la vue pg\_stats, champ correlation (des valeurs proches de -1 ou 1 indiquent une bonne corrélation).

#### 1.2.6.3 Index Only Scan

L'*Index Only Scan* est utile quand les champs de la requête correspondent aux colonnes de l'index. Ce nœud permet d'éviter la lecture de tout ou partie de la table et est donc très performant.

Autre intérêt de l'*Index Only Scan* : les enregistrements cherchés sont contigus dans l'index (puisqu'il est trié), et le nombre d'accès disque est bien plus faible. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

Si peu de champs de la table sont impliqués dans la requête, il faut penser à viser un *Index Only Scan*.

#### 1.2.6.4 Parallélisation

Chacun de ses nœuds a une version parallélisable si l'index est assez grand et que l'optimiseur pense que paralléliser est utile. Il apparaît alors un nœud *Gather* pour rassembler les résultats des différents workers.

## 1.3 MÉTHODOLOGIE DE CRÉATION D'INDEX



- On indexe pour une requête
   ou idéalement une collection de requêtes
   Et pas « une table »

La première chose à garder en tête est que l'on indexe pas le schéma de données, c'est-à-dire les tables, mais en fonction de la charge de travail supportée par la base, c'est-à-dire les requêtes. En effet, comme nous l'avons vu précédemment, tout index superflu a un coût global pour la base de données, notamment pour les opérations DML.

#### 1.3.1 L'index ? Quel index ?



- Identifier les requêtes nécessitant un index
   Créer les index permettant de répondre à ces requêtes
   Valider le fonctionnement, en rejouant la requête avec :

  EXPLAIN (ANALYZE, BUFFERS)

La méthodologie elle-même est assez simple. Selon le principe qu'un index sert à une (ou des) requête(s), la première chose à faire consiste à identifier celle(s)-ci. L'équipe de développement est dans une position idéale pour réaliser ce travail : elle seule peut connaître le fonctionnement global de l'application, et donc les colonnes qui vont être utilisées, ensemble ou non, comme cible de filtres ou de tris. Au delà de la connaissance de l'application, il est possible d'utiliser des outils tels que pg-Badger, pg\_stat\_statements et PoWA pour identifier les requêtes particulièrement consommatrices, et qui pourraient donc potentiellement nécessiter un index. Ces outils seront présentés plus loin dans cette formation.

Une fois les requêtes identifiées, il est nécessaire de trouver les index permettant d'améliorer cellesci. Ils peuvent être utilisés pour les opérations de filtrage (clause WHERE ), de tri (clauses ORDER BY , GROUP BY ) ou de jointures. Idéalement, l'étude portera sur l'ensemble des requêtes, afin notamment de pouvoir décider d'index multicolonnes pertinents pour le plus grand nombre de requêtes, et éviter ainsi de créer des index redondants.

## 1.3.2 Index et clés étrangères



- Indexation des colonnes faisant référence à une autre
- Performances des DMLPerformances des jointures

De manière générale, l'ensemble des colonnes étant la source d'une clé étrangère devraient être indexées, et ce pour deux raisons.

La première concerne les jointures. Généralement, lorsque deux tables sont liées par des clés étrangères, il existe au moins certaines requêtes dans l'application joignant ces tables. La colonne « cible » de la clé étrangère est nécessairement indexée, c'est un prérequis dû à la contrainte unique nécessaire à celle-ci. Il est donc possible de la parcourir de manière triée.

La colonne source devrait être indexée elle aussi : en effet, il est alors possible de la parcourir de manière ordonnée, et donc de réaliser la jointure selon l'algorithme Merge Join (comme vu lors du module sur les plans d'exécution<sup>5</sup>), et donc d'être beaucoup plus rapide. Un tel index accélérera de la même manière les Nested Loop, en permettant de parcourir l'index une fois par ligne de la relation externe au lieu de parcourir l'intégralité de la table.

De la même manière, pour les DML sur la table cible, cet index sera d'une grande aide : pour chaque ligne modifiée ou supprimée, il convient de vérifier, soit pour interdire soit pour « cascader » la modification, la présence de lignes faisant référence à celle touchée.

S'il n'y a qu'une règle à suivre aveuglément ou presque, c'est bien celle-ci : les colonnes faisant partie d'une clé étrangère doivent être indexées!

Deux exceptions : les champs ayant une cardinalité très faible et homogène (par exemple, un champ homme/femme dans une population équilibrée) ; et ceux dont on constate l'inutilité après un certain temps, par des valeurs à zéro dans pg\_stat\_user\_indexes .

Indexation & SQL Avancé

<sup>&</sup>lt;sup>5</sup>https://dali.bo/j0\_html

## 1.4 INDEX INUTILISÉ



C'est souvent tout à fait normal

- Utiliser l'index est-il rentable ?
- La requête est-elle compatible ?Bug de l'optimiseur : rare

C'est l'optimiseur SQL qui choisit si un index doit ou non être utilisé. Il est tout à fait possible que PostgreSQL décide qu'utiliser un index donné n'en vaut pas la peine par rapport à d'autres chemins. Il faut aussi savoir identifier les cas où l'index ne peut pas être utilisé.

L'optimiseur possède forcément quelques limitations. Certaines sont un compromis par rapport au temps que prendrait la recherche systématique de toutes les optimisations imaginables. Il y aussi le problème des estimations de volumétries, qui sont d'autant plus difficiles que la requête est complexe.

Quant à un vrai bug, si le cas peut être reproduit, il doit être remonté aux développeurs de PostgreSQL. D'expérience, c'est rarissime.

#### 1.4.1 Index utilisable mais non utilisé



- L'optimiseur pense qu'il n'est pas rentable
  - sélectivité trop faible
  - meilleur chemin pour remplir d'autres critères
  - index redondant
  - Index Only Scan nécessite un VACUUM fréquent
- Les estimations de volumétries doivent être assez bonnes!
  - statistiques récentes, précises

Il existe plusieurs raisons pour que PostgreSQL néglige un index.

#### Sélectivité trop faible, trop de lignes :

Comme vu précédemment, le parcours d'un index implique à la fois des lectures sur l'index, et des lectures sur la table. Au contraire d'une lecture séquentielle de la table (Seg Scan), l'accès aux données via l'index nécessite des lectures aléatoires. Ainsi, si l'optimiseur estime que la requête nécessitera de

parcourir une grande partie de la table, il peut décider de ne pas utiliser l'index : l'utilisation de celui-ci serait alors trop coûteux.

Autrement dit, l'index n'est pas assez discriminant pour que ce soit la peine de faire des allers-retours entre lui et la table. Le seuil dépend entre autres des volumétries de la table et de l'index et du rapport entre les paramètres random\_page\_cost et seq\_page\_cost (respectivement 4 et 1 pour un disque dur classique peu rapide, et souvent 1 et 1 pour du SSD, voire moins).

#### Il y a un meilleur chemin:

Un index sur un champ n'est qu'un chemin parmi d'autres, en aucun cas une obligation, et une requête contient souvent plusieurs critères sur des tables différentes. Par exemple, un index sur un filtre peut être ignoré si un autre index permet d'éviter un tri coûteux, ou si l'optimiseur juge que faire une jointure avant de filtrer le résultat est plus performant.

#### Index redondant:

Il existe un autre index doublant la fonctionnalité de celui considéré. PostgreSQL favorise naturellement un index plus petit, plus rapide à parcourir. À l'inverse, un index plus complet peut favoriser plusieurs filtres, des tris, devenir couvrant...

#### **VACUUM trop ancien:**

Dans le cas précis des *Index Only Scan*, si la table n'a pas été récemment nettoyée, il y aura trop d'allers-retours avec la table pour vérifier les informations de visibilité (*heap fetches*). Un VACUUM permet de mettre à jour la *Visibility Map* pour éviter cela.

# Statistiques périmées :

Il peut arriver que l'optimiseur se trompe quand il ignore un index. Des statistiques périmées sont une cause fréquente. Pour les rafraîchir :

```
ANALYZE (VERBOSE) nom_table;
```

Si cela résout le problème, ce peut être un indice que l'autovacuum ne passe pas assez souvent (voir pg\_stat\_user\_tables.last\_autoanalyze). Il faudra peut-être ajuster les paramètres autovacuum\_analyze\_scale\_factor ou autovacuum\_analyze\_threshold sur les tables.

#### Statistiques pas assez fines:

Les statistiques sur les données peuvent être trop imprécises. Le défaut est un histogramme de 100 valeurs, basé sur 300 fois plus de lignes. Pour les grosses tables, augmenter l'échantillonnage sur les champs aux valeurs peu homogènes est possible :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500 ;
```

La valeur 500 n'est qu'un exemple. Monter beaucoup plus haut peut pénaliser les temps de planification. Ce sera d'autant plus vrai si on applique cette nouvelle valeur globalement, donc à tous les champs de toutes les tables (ce qui est certes le plus facile).

#### Estimations de volumétries trompeuses :

Par exemple, une clause WHERE sur deux colonnes corrélées (ville et code postal par exemple), mène à une sous-estimation de la volumétrie résultante par l'optimiseur, car celui-ci ignore le lien entre

les deux champs. Vous pouvez demander à PostgreSQL de calculer cette corrélation avec l'ordre CREATE STATISTICS (voir le module de formation J2<sup>6</sup> ou la documentation officielle<sup>7</sup>).

## Compatibilité:

Il faut toujours s'assurer que la requête est écrite correctement et permet l'utilisation de l'index.

Un index peut être inutilisable à cause d'une fonction plus ou moins explicite, ou encore d'un mauvais typage. Il arrive que le critère de filtrage ne peut remonter sur la table indexée à cause d'un CTE matérialisé (explicitement ou non), d'un DISTINCT, ou d'une vue complexe.

Nous allons voir quelques problèmes classiques.

#### 1.4.2 Index inutilisable à cause d'une fonction



- Pas le bon type (CAST plus ou moins explicite)

```
EXPLAIN SELECT * FROM clients WHERE client_id = 3::numeric;
   - Utilisation de fonctions, comme:
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYYY')='2014';
```

Voici quelques exemples d'index incompatible avec la clause WHERE:

#### Mauvais type:

Cela peut paraître contre-intuitif, mais certains transtypages ne permettent pas de garantir que les résultats d'un opérateur (par exemple l'égalité) seront les mêmes si les arguments sont convertis dans un type ou dans l'autre. Cela dépend des types et du sens de conversion. Dans les exemples suivants, le champ client\_id est de type bigint. PostgreSQL réussit souvent à convertir, mais ce n'est pas toujours parfait.

<sup>7</sup>https://docs.postgresql.fr/current/sql-createstatistics.html

#### Autres exemples:

- Dans une jointure, si les deux champs joints n'ont pas le même type, il est possible que de simples index ne soient pas utilisables, ou un seul d'entre eux. Il faudra corriger l'incohérence, ou créer des index fonctionnels incluant le transtypage.
- Un index B-tree sur un tableau ou un JSON ne peut servir pour une recherche sur un de ses éléments. Il faudra s'orienter vers un index plus spécialisé, par exemple GIN ou GiST.

#### Utilisation de fonction :

Si une fonction est appliquée sur la colonne à indexer, comme dans cet exemple classique :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYYY')='2014';
alors PostgreSQL n'utilisera pas l'index sur ma_date. Il faut réécrire la requête ainsi:
SELECT * FROM ma_table WHERE ma_date >='2014-01-01' AND ma_date<'2015-01-01';</pre>
```

Dans l'exemple suivant, on cherche les commandes dont la date tronquée au mois correspond au 1er janvier, c'est-à-dire aux commandes dont la date est entre le 1er et le 31 janvier. Pour un humain, la logique est évidente, mais l'optimiseur n'en a pas connaissance.

```
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_trunc('month', date_commande) = '2015-01-01';
                             QUERY PLAN
Gather (cost=1000.00..8160.96 rows=5000 width=51)
       (actual time=17.282..192.131 rows=4882 loops=1)
Workers Planned: 3
Workers Launched: 3
-> Parallel Seq Scan on commandes (cost=0.00..6660.96 rows=1613 width=51)
                           (actual time=17.338..177.896 rows=1220 loops=4)
     Filter: (date_trunc('month'::text,
                                    (date_commande)::timestamp with time zone)
                         = '2015-01-01 00:00:00+01'::timestamp with time zone)
     Rows Removed by Filter: 248780
 Planning time: 0.215 ms
 Execution time: 196.930 ms
```

Il faut plutôt écrire:

```
EXPLAIN ANALYZE
SELECT * FROM commandes
WHERE date_commande BETWEEN '2015-01-01' AND '2015-01-31';
                          QUERY PLAN
Index Scan using commandes_date_commande_idx on commandes
                                     (cost=0.42..118.82 rows=5554 width=51)
                              (actual time=0.019..0.915 rows=4882 loops=1)
   Index Cond: ((date_commande >= '2015-01-01'::date)
           AND (date_commande <= '2015-01-31'::date))
Planning time: 0.074 ms
Execution time: 1.098 ms
```

Dans certains cas, la réécriture est impossible (fonction complexe, code non modifiable...). Nous verrons qu'un index fonctionnel peut parfois être la solution.

Ces exemples semblent évidents, mais il peut être plus compliqué de trouver dans l'urgence la cause du problème dans une grande requête d'un schéma mal connu.

#### 1.4.3 Index inutilisable à cause d'un LIKE '...%'



```
SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';
   - Solution:
CREATE INDEX idx1 ON ma_table (col_varchar varchar_pattern_ops) ;
```

Si vous avez un index « normal » sur une chaîne texte, certaines recherches de type LIKE n'utiliseront pas l'index. En effet, il faut bien garder à l'esprit qu'un index est basé sur un opérateur précis. Ceci est généralement indiqué correctement dans la documentation, mais pas forcément très intuitif.

Si un opérateur non supporté pour le critère de tri est utilisé, l'index ne servira à rien :

```
CREATE INDEX ON fournisseurs (commentaire);
EXPLAIN ANALYZE SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';
                           QUERY PLAN
Seq Scan on fournisseurs (cost=0.00..225.00 rows=1 width=45)
                          (actual time=0.045..1.477 rows=47 loops=1)
   Filter: (commentaire ~~ 'ipsum%'::text)
  Rows Removed by Filter: 9953
Planning time: 0.085 ms
Execution time: 1.509 ms
```

Nous verrons qu'il existe d'autre classes d'opérateurs, permettant d'indexer correctement la requête précédente, et que varchar\_pattern\_ops est l'opérateur permettant d'indexer la requête précédente.

#### 1.4.4 Index inutilisable car invalide



Dans le cas où un index a été construit avec la clause CONCURRENTLY, nous avons vu qu'il peut arriver que l'opération échoue et l'index existe mais reste invalide, et donc inutilisable. Le problème ne se pose pas pour un échec de REINDEX ... CONCURRENTLY, car l'ancienne version de l'index est toujours là et utilisable.

# 1.5 INDEXATION B-TREE AVANCÉE



De nombreuses possibilités d'indexation avancée :

- Index partiels
- Index fonctionnels
- Index couvrants
- Classes d'opérateur

# 1.5.1 Index partiels



- N'indexe qu'une partie des données :

```
\begin{cal} \textbf{CREATE INDEX on} evenements & \textbf{(type)} & \textbf{WHERE} & \textbf{traite IS FALSE} \end{cal} ;
```

- Ne sert que si la clause est logiquement équivalente!
  - ou partie de la clause (inégalités, IN)
- Intérêt : index beaucoup plus petit

Un index partiel est un index ne couvrant qu'une partie des enregistrements. Ainsi, l'index est beaucoup plus petit. En contrepartie, il ne pourra être utilisé que si sa condition est définie dans la requête.

Pour prendre un exemple simple, imaginons un système de « queue », dans lequel des événements sont entrés, et qui disposent d'une colonne traite indiquant si oui ou non l'événement a été traité. Dans le fonctionnement normal de l'application, la plupart des requêtes ne s'intéressent qu'aux événements non traités :

```
WHEN i % 3 = 1 THEN 'EXPEDITION'
            ELSE 'COMMANDE'
   FROM generate_series(1, 10000) as i);
-- et 10 non encore traités
INSERT INTO evenements (id, traite, type) (
   SELECT i,
       false,
       CASE WHEN i % 3 = 0 THEN 'FACTURATION'
            WHEN i % 3 = 1 THEN 'EXPEDITION'
            ELSE 'COMMANDE'
   FROM generate_series(10001, 10010) as i);
\d evenements
                Table « public.evenements »
Colonne | Type | Collationnement | NULL-able | Par défaut
id | integer |
                                    | not null |
traite | boolean |
                                    not null
        | text |
                                    | not null
tvpe
payload | text
Index :
    "evenements_pkey" PRIMARY KEY, btree (id)
```

Typiquement, différents applicatifs vont être intéressés par des événements d'un certain type, mais les événements déjà traités ne sont quasiment jamais accédés, du moins via leur état (une requête portant sur traite IS true sera exceptionnelle et ramènera l'essentiel de la table : un index est inutile).

Ainsi, on peut souhaiter indexer le type d'événement, mais uniquement pour les événements non traités :

```
CREATE INDEX index_partiel on evenements (type) WHERE NOT traite ;
```

Si on recherche les événements dont le type est « FACTURATION », sans plus de précision, l'index ne peut évidemment pas être utilisé :

```
QUERY PLAN

Seq Scan on evenements (cost=0.00..183.12 rows=50 width=69)
Filter: (type = 'FACTURATION'::text)
```

En revanche, si la condition sur l'état de l'événement est précisée, l'index sera utilisé :

Sur ce jeu de données, on peut comparer la taille de deux index, partiels ou non :

Un index composé sur (is\_traite, type) | serait efficace, mais inutilement gros.

# Clauses de requête et clause d'index :



Attention! Les clauses de l'index et du WHERE doivent être **logiquement équivalentes**! (et de préférence identiques)

Par exemple, dans les requêtes précédentes, un critère traite IS FALSE à la place de NOT traite n'utilise pas l'index (en effet, il ne s'agit pas du même critère à cause de NULL : NULL = false renvoie NULL , mais NULL IS false renvoie false).

Par contre, des conditions mathématiquement plus restreintes que l'index permettent son utilisation :

Mais cet index partiel ne sera pas utilisé pour un critère précédant 2015.

De la même manière, si un index partiel contient une liste de valeurs, IN () ou NOT IN (), il est en principe utilisable :

# 1.5.2 Index partiels: cas d'usage



- Données *chaudes* et *froides*
- Index dédié à une requête avec une condition fixe

Le cas typique d'utilisation d'un index partiel est celui de l'exemple précédent : une application avec des données *chaudes*, fréquemment accédées et traitées, et des données *froides*, qui sont plus destinées à de l'historisation ou de l'archivage. Par exemple, un système de vente en ligne aura probablement intérêt à disposer d'index sur les commandes dont l'état est différent de clôturé : en effet, un tel système effectuera probablement des requêtes fréquemment sur les commandes qui sont en cours de traitement, en attente d'expédition, en cours de livraison mais très peu sur des commandes déjà livrées, qui ne serviront alors plus qu'à de l'analyse statistique.

De manière générale, tout système est susceptible de bénéficier des index partiels s'il doit gérer des données à état dont seul un sous-ensemble de ces états est activement exploité par les requêtes à optimiser. Par exemple, toujours sur cette même table, des requêtes visant à faire des statistiques sur les expéditions pourraient tirer parti de cet index :

```
CREATE INDEX index_partiel_expes ON evenements (id) WHERE type = 'EXPEDITION';

EXPLAIN SELECT count(id) FROM evenements WHERE type = 'EXPEDITION';

QUERY PLAN

Aggregate (cost=106.68..106.69 rows=1 width=8)

-> Index Only Scan using index_partiel_expes on evenements (cost=0.28..98.34

-> rows=3337 width=4)
```

Nous avons mentionné précédemment qu'un index est destiné à satisfaire une requête ou un ensemble de requêtes. Donc, si une requête présente fréquemment des critères de ce type :

```
WHERE une_colonne = un_parametre_variable
AND une_autre_colonne = une_valeur_fixe
```

alors il peut être intéressant de créer un index partiel pour les lignes satisfaisant le critère :

```
WHERE une_autre_colonne = une_valeur_fixe
```

Ces critères sont généralement très liés au fonctionnel de l'application : du point de vue de l'exploitation, il est souvent difficile d'identifier des requêtes dont une valeur est toujours fixe. Encore une fois, l'appropriation des techniques d'indexation par l'équipe de développement permet d'améliorer grandement les performances de l'application.

# 1.5.3 Index partiels: utilisation



- Éviter les index de type :

En général, un index partiel doit indexer une colonne différente de celle qui est filtrée (et donc connue). Ainsi, dans l'exemple précédent, la colonne indexée (type) n'est pas celle de la clause where. On pose un critère, mais on s'intéresse aux types d'événements ramenés. Un autre index partiel pourrait porter sur id where not traite pour simplement récupérer une liste des identifiants non traités de tous types.

L'intérêt est d'obtenir un index très ciblé et compact, et aussi d'économiser la place disque et la charge CPU de maintenance. Il faut tout de même que les index partiels soient notablement plus petits que les index « génériques » (au moins de moitié). Avec des index partiels spécialisés, il est possible de « précalculer » certaines requêtes critiques en intégrant leurs critères de recherche exacts.

# 1.5.4 Index fonctionnels: principe



- Un index sur a est inutilisable pour :

```
SELECT ... WHERE upper(a)='DUPOND'
- Indexer le résultat de la fonction:
CREATE INDEX mon_idx ON ma_table (upper(a));
```

À partir du moment où une clause WHERE applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement.

C'est comme demander à un dictionnaire Anglais vers Français : « Quels sont les mots dont la traduction en français est 'fenêtre' ? ». Le tri du dictionnaire ne correspond pas à la question posée. Il nous faudrait un index non plus sur les mots anglais, mais sur leur traduction en français.

C'est exactement ce que font les index fonctionnels : ils indexent le résultat d'une fonction appliquée à l'enregistrement.

L'exemple classique est l'indexation insensible à la casse : on crée un index sur UPPER (ou LOWER) de la chaîne à indexer, et on recherche les mots convertis à la casse souhaitée.

#### 1.5.5 Index fonctionnels: conditions



- Critère identique à la fonction dans l'index
- Fonction impérativement IMMUTABLE !
  - délicat avec les conversions de dates/heures
- Ne pas espérer d'Index Only Scan

Il est facile de créer involontairement des critères comportant des fonctions, notamment avec des conversions de type ou des manipulations de dates. Il a été vu plus haut qu'il vaut mieux placer la transformation du côté de la constante. Par exemple, la requête suivante retourne toutes les commandes de l'année 2011, mais la fonction extract est appliquée à la colonne date\_commande (type date) et l'index est inutilisable.

L'optimiseur ne peut donc pas utiliser un index :

#### **QUERY PLAN**

```
Index Scan using commandes_date_commande_idx on commandes
   Index Cond: ((date_commande >= '2011-01-01'::date) AND (date_commande <=
    '2011-12-31'::date))</pre>
```

C'est la solution la plus propre.

Mais dans d'autres cas, une telle réécriture de la requête sera impossible ou très délicate. On peut alors créer un index fonctionnel, dont la définition doit être **strictement** celle du WHERE:

#### Fonction immutable:

Cependant, n'importe quelle fonction d'indexation n'est pas utilisable, ou pas pour tous les types. La fonction d'indexation doit être notée IMMUTABLE : cette propriété indique à PostgreSQL que la fonction retournera toujours le **même résultat** quand elle est appelée avec les **mêmes arguments**.

En d'autres termes : le résultat de la fonction ne doit dépendre :

- ni du contenu de la base (pas de SELECT donc);
- ni de la configuration, ni de l'environnement (variables d'environnement, paramètres de session, fuseau horaire, formatage...);
- ni du temps ( now() ou clock\_timestamp() sont interdits, et indirectement les calculs d'âge);
- ni d'une autre fonction non-déterministe (comme random()) ou plus généralement non immutable.

Sans ces restrictions, l'endroit dans lequel la donnée est insérée dans l'index serait potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Pour revenir à l'exemple précédent : pour calculer l'année, on peut aussi imaginer un index avec la fonction to\_char, une autre fonction hélas fréquemment utilisée pour les conversions de date. Au moment de la création d'un tel index, PostgreSQL renvoie l'erreur suivante :

```
CREATE INDEX annee_commandes_idx2
ON commandes ((to_char(date_commande,'YYYYY')::int));
ERROR: functions in index expression must be marked IMMUTABLE
En effet, to_char() n'est pas immutable, juste « stable » et cela dans toutes ses variantes:
```

```
magasin=# \df+ to_char
```

#### Liste des fonctions

```
|...résultat| Type de données des paramètres |...|Volatibilité|...
                    _____+
...to_char | text
                  | bigint, text
                                                    stable
                 | double precision, text
                                                   | | stable
...to_char | text
...to_char | text
                 | integer, text
                                                   | | stable
...to_char | text
                 | interval, text
                                                   | | stable
                                                   | | stable
...to_char | text
                 | numeric, text
...to_char | text
                 | real, text
                                                   | | stable
                                                                 | ...
                  | timestamp without time zone, text| | stable
...to_char | text
...to_char | text
                 | timestamp with time zone, text | | stable
(8 lignes)
```

La raison est que to\_date accepte des paramètres de formatage qui dépendent de la session (nom du mois, virgule ou point décimal...). Ce n'est pas une très bonne fonction pour convertir une date ou heure en nombre.

La fonction extract, elle, est bien immutable quand il s'agit de convertir commande.date\_commande de date vers une année, comme dans l'exemple plus haut.

```
\sf extract (text, date)
CREATE OR REPLACE FUNCTION pg_catalog."extract"(text, date)
RETURNS numeric
LANGUAGE internal
IMMUTABLE PARALLEL SAFE STRICT
AS $function$extract_date$function$
```

De même, extract est immutable avec une entrée de type timestamp without time zone.

Les choses se compliquent si l'on manipule des heures avec fuseau horaire. En effet, il est conseillé de toujours privilégier la variante timestamp with time zone. Cette fois, l'index fonctionnel basé avec extract va poser problème :

```
DROP INDEX annee_commandes_idx ;
-- Nouvelle table d'exemple avec date_commande comme timestamp with time zone
-- La conversion introduit implicitement le fuseau horaire de la session
CREATE TABLE commandes2 (LIKE commandes INCLUDING ALL);
ALTER TABLE commandes2 ALTER COLUMN date_commande TYPE timestamp with time zone ; INSERT INTO commandes2 SELECT \star FROM commandes ;
-- Reprise de l'index fonctionnel précédent
CREATE INDEX annee_commandes2_idx
ON commandes2(extract('year' from date_commande) );
ERROR: functions in index expression must be marked IMMUTABLE
En effet la fonction extract n'est pas immutable pour le type timestamp with time zone :
magasin=# \sf extract (text, timestamp with time zone)
CREATE OR REPLACE FUNCTION pg_catalog."extract"(text, timestamp with time zone)
 RETURNS numeric
 LANGUAGE internal
 STABLE PARALLEL SAFE STRICT
AS $function$extract_timestamptz$function$
```

Pour certains *timestamps* autour du Nouvel An, l'année retournée dépend du fuseau horaire. Le problème se poserait bien sûr aussi si l'on extrayait les jours ou les mois.

Il est possible de « tricher » en figeant le fuseau horaire dans une fonction pour obtenir un type intermédiaire timestamp without time zone, qui ne posera pas de problème :

```
CREATE INDEX annee_commandes2_idx
ON commandes2(extract('year' from (
   date_commande AT TIME ZONE 'Europe/Paris' )::timestamp
   ));
```

Ce contournement impose de modifier le critère de la requête. Tant qu'on y est, il peut être plus clair d'enrober l'appel dans une fonction que l'on définira immutable.

Le nom de la fonction est aussi une indication pour les utilisateurs dans d'autres fuseaux.

Certes, on a ici modifié le code de la requête, mais il est parfois possible de contourner ce problème en passant par des vues qui masquent la fonction.

Signalons enfin la fonction date\_part : c'est une alternative possible à extract, avec les mêmes soucis et contournement.

À partir de PostgreSQL 16, une autre possibilité existe avec date\_trunc car la variante avec timestamp without time zone est devenue immutable:

```
Index Scan using annee_commandes2_paris_idx3 on commandes2
   Index Cond: (date_trunc('year'::text, date_commande, 'Europe/Paris'::text) =
   '2021-01-01 00:00:00+01'::timestamp with time zone)
```

#### **Index Only Scan:**

Obtenir un *Index Only Scan* est une optimisation importante pour les requêtes critiques avec peu de champs sur la table. Hélas, en raison d'une limitation du planificateur, les index fonctionnels ne donnent pas lieu à un *Index Only Scan*:

```
EXPLAIN (COSTS OFF)

SELECT annee_paris (date_commande) FROM commandes2

WHERE annee_paris (date_commande) > 2021;

QUERY PLAN

Index Scan using annee_commandes2_paris_idx on commandes2

Index Cond: ((EXTRACT(year FROM (date_commande AT TIME ZONE

'Europe/Paris'::text)))::integer > 2021)
```

Plus insidieusement, le planificateur peut choisir un *Index Only Scan...* sur la colonne sur laquelle porte la fonction !

```
QUERY PLAN

Aggregate (cost=28520.40..28520.41 rows=1 width=8)

-> Index Only Scan using commandes2_date_commande_idx on commandes2

(cost=0.42..18520.41 rows=999999 width=8)
```

Ce qui entraîne au moins un gaspillage de CPU pour réexécuter les fonctions sur chaque ligne.

Sacrifier un peu d'espace disque pour une colonne générée et son index (non fonctionnel) peut s'avérer une solution :

```
-- Attention, cette commande réécrit la table

ALTER TABLE commandes2 ADD COLUMN annee_paris smallint

GENERATED ALWAYS AS ( annee_paris (date_commande) ) STORED;

CREATE INDEX commandes2_annee_paris_idx ON commandes2 (annee_paris);

-- Prise en compte des statistiques et des lignes mortes sur la table réécrite

VACUUM ANALYZE commandes2;

EXPLAIN SELECT count( annee_paris ) FROM commandes2;

QUERY PLAN

Finalize Aggregate (cost=14609.10..14609.11 rows=1 width=8)

-> Gather (cost=14608.88..14609.09 rows=2 width=8)

Workers Planned: 2

-> Partial Aggregate (cost=13608.88..13608.89 rows=1 width=8)

-> Parallel Index Only Scan using commandes2_annee_paris_idx on

commandes2 (cost=0.42..12567.20 rows=416672 width=2)
```

#### 1.5.6 Index fonctionnels: maintenance



- Ne pas oublier ANALYZE après création d'un index fonctionnel
  - les statistiques peuvent même être l'intérêt majeur (<v14)
- La fonction ne doit jamais tomber en erreur
- Si modification de la fonction
  - réindexation

#### Statistiques:

Après la création de l'index fonctionnel, un ANALYZE nom\_table est conseillé: en effet, l'optimiseur ne peut utiliser les statistiques déjà connues pour le résultat d'une fonction. Par contre, PostgreSQL peut créer des statistiques sur le résultat de la fonction pour chaque ligne. Ces statistiques seront visibles dans la vue système pg\_stats (tablename contient le nom de l'index, et non celui de la table!).

Ces statistiques à jour sont d'ailleurs un des intérêts de l'index fonctionnel, même si l'index lui-même est superflu. Dans ce cas, à partir de PostgreSQL 14, on pourra utiliser CREATE STATISTICS sur l'expression pour ne pas avoir à créer et maintenir un index entier.

#### **Avertissements:**



La fonction ne doit jamais tomber en erreur! Il ne faut pas tester l'index qu'avec les données en place, mais aussi avec toutes celles susceptibles de se trouver dans le champ concerné. Sinon, il y aura des refus d'insertion ou de mise à jour. Des ANALYZE ou VACUUM pourraient aussi échouer, avec de gros problèmes sur le long terme.



Si le contenu de la fonction est modifié avec CREATE OR REPLACE FUNCTION, il faudra impérativement réindexer, car PostgreSQL ne le fera pas automatiquement. Sans cela, les résultats des requêtes différeront selon qu'elles utiliseront ou non l'index!

# 1.5.7 Index couvrants: principe



- But: obtenir un Index Only Scan

```
CREATE UNIQUE INDEX clients_idx1
ON clients (id_client) INCLUDE (nom_client);

- Répondent à la clause WHERE
- ET contiennent toutes les colonnes demandées par la requête:

SELECT id_client,nom_client FROM clients WHERE id_client > 100;
- ...si l'index n'est pas trop gros
```

# 1.5.7.1 Principe des index couvrants

Un index couvrant (covering index) cherche à favoriser le nœud d'accès le plus rapide, l'Index Only Scan: il contient non seulement les champs servant de critères de recherche, mais aussi tous les champs résultats. Ainsi, il n'y a plus besoin d'interroger la table.

Les index couvrants peuvent être explicitement déclarés avec la clause INCLUDE :

- à comparer à un index multicolonne

Dans cet exemple, il n'y a pas eu d'accès à la table. L'index est unique mais contient aussi la colonne valeur.



Noter le VACUUM, nécessaire pour garantir que la visibility map de la table est à jour et permet ainsi un *Index Only Scan* sans aucun accès à la table (clause *Heap Fetches* à 0).

Par abus de langage, on peut dire d'un index multicolonne sans clause INCLUDE qu'il est « couvrant » s'il répond complètement à la requête.

Dans les versions antérieures à la 11, on émulait cette fonctionnalité en incluant les colonnes dans des index multicolonne :

```
CREATE INDEX t_idx ON t (id, valeur) ;
```

Cette technique reste tout à fait valable dans les versions suivantes, car l'index multicolonne (complètement trié) peut servir de manière optimale à d'autres requêtes. Il peut même être plus petit que celui utilisant INCLUDE.

Un intérêt de la clause INCLUDE est de se greffer sur des index uniques ou de clés et d'économiser un nouvel index et un peu de place. Accessoirement, il évite le tri des champs dans la clause INCLUDE.

# 1.5.7.2 Inconvénients & limitation des index couvrants

Il faut garder à l'esprit que l'ajout de colonnes à un index (couvrant ou multicolonne) augmente sa taille. Cela peut avoir un impact sur les performances des requêtes qui n'utilisent pas les colonnes supplémentaires. Il faut également être vigilant à ce que la taille des enregistrements avec les colonnes incluses ne dépassent pas 2,6 ko. Au-delà de cette valeur, les insertions ou mises à jour échouent.

Enfin, la déduplication (apparue en version 13) n'est pas active sur les index couvrants, ce qui a un impact supplémentaire sur la taille de l'index sur le disque et en cache. Ça n'a pas trop d'importance si l'index principal contient surtout des valeurs différentes, mais s'il y en a beaucoup moins que de lignes, il serait dommage de perdre l'intérêt de la déduplication. Là encore, le planificateur peut ignorer l'index s'il est trop gros. Il faut tester avec les données réelles, et comparer avec un index multicolonne (dédupliqué).

Les méthodes d'accès aux index doivent inclure le support de cette fonctionnalité. C'est le cas pour le B-tree ou le GiST, et pour le SP-GiST en version 14.

# 1.5.8 Classes d'opérateurs



- Un index utilise des opérateurs de comparaison
- Texte : différentes collations = différents tris... complexes
  - Index inutilisable sur:

```
WHERE col_varchar LIKE 'chaine%'
```

- Solution: opérateur varchar\_pattern\_ops :
  - force le tri caractère par caractère, sans la collation

```
CREATE INDEX idx1
ON ma_table (col_varchar varchar_pattern_ops)
```

- Plus généralement :
  - nombreux autres opérateurs pour d'autres types d'index

Un opérateur sert à indiquer à PostgreSQL comment il doit manipuler un certain type de données. Il y a beaucoup d'opérateurs par défaut, mais il est parfois possible d'en prendre un autre.

Pour l'indexation, il est notamment possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison.

Le cas d'utilisation le plus fréquent dans PostgreSQL est la comparaison de chaîne LIKE 'chaine%'. L'indexation texte « classique » utilise la collation par défaut de la base (en France, généralement fr\_FR.UTF-8 ou en\_US.UTF-8) ou la collation de la colonne de la table si elle diffère. Cette collation contient des notions de tri. Les règles sont différentes pour chaque collation. Et ces règles sont complexes.

Par exemple, le **ß** allemand se place entre **ss** et **t** (et ce, même en français). En danois, le tri est très particulier car le **å** et le **aa** apparaissent après le **z**.

```
-- Cette collation doit exister sur le système

CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz'))

SELECT * FROM ls ORDER BY x COLLATE "da_DK";

x
----
s
s
ss
ß
t
zz
å
aa
```

Il faut être conscient que cela a une influence sur le résultat d'un filtrage :

```
WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls
WHERE x > 'z' COLLATE "da_DK";

x
----
aa
å
zz
```

Il serait donc très complexe de réécrire le LIKE en un BETWEEN, comme le font habituellement tous les SGBD: col\_texte LIKE 'toto%' peut être réécrit comme coltexte >= 'toto' and coltexte < 'totp' en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur Unicode par exemple. Même si l'index est dans la bonne collation, il n'est pas facilement utilisable:

```
CREATE INDEX ON textes (livre);

EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

QUERY PLAN

Gather (cost=1000.00..525328.76 rows=75173 width=123)
Workers Planned: 2

-> Parallel Seq Scan on textes (cost=0.00..516811.46 rows=31322 width=123)
Filter: (livre ~~ 'Les misérables%'::text)

La classe d'opérateurs varchar_pattern_ops sert à changer ce comportement:
```

```
CREATE INDEX ON ma_table (col_varchar varchar_pattern_ops) ;
```

Ce nouvel index est alors construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient. Il devient alors trivial pour l'optimiseur de faire la réécriture :

Cela convient pour un LIKE 'critère%', car le début est fixe, et l'ordre de tri n'influe pas sur le résultat. (Par contre cela ne permet toujours pas d'indexer LIKE %critère%).) Noter la clause Filter qui filtre en deuxième intention ce qui a pu être trouvé dans l'index.

Il existe quelques autres cas d'utilisation d'opclass alternatives, notamment pour utiliser d'autres types d'index que B-tree. Deux exemples :

```
- indexation d'un JSON (type jsonb ) par un index GIN :
```

```
CREATE INDEX ON stock_jsonb USING gin (document_jsonb jsonb_path_ops);
```

- indexation de trigrammes de textes avec le module pg\_trgm et des index GiST :

CREATE INDEX ON livres USING gist (text\_data gist\_trgm\_ops);

Pour plus de détails à ce sujet, se référer à la section correspondant aux classes d'opérateurs<sup>8</sup>.



Ne mettez pas systématiquement | varchar\_pattern\_ops | dans tous les index de chaînes de caractère. Cet opérateur est adapté au LIKE 'critères' mais ne servira pas pour un tri sur la chaîne (ORDER BY). Selon les requêtes et volumétries, les deux index peuvent être nécessaires.

#### 1.5.9 Conclusion



- Responsabilité de l'indexation
- Compréhension des mécanismesDifférents types d'index, différentes stratégies

L'indexation d'une base de données est souvent un sujet qui est traité trop tard dans le cycle de l'application. Lorsque celle-ci est gérée à l'étape du développement, il est possible de bénéficier de l'expérience et de la connaissance des développeurs. La maîtrise de cette compétence est donc idéalement transverse entre le développement et l'exploitation.

Le fonctionnement d'un index B-tree est somme toute assez simple, mais il est important de bien l'appréhender pour comprendre les enjeux d'une bonne stratégie d'indexation.

PostgreSQL fournit aussi d'autres types d'index moins utilisés, mais très précieux dans certaines situations: BRIN, GIN, GiST, etc.

<sup>8</sup>https://www.postgresql.org/docs/current/static/indexes-opclass.html

# **1.6 QUIZ**



# 1.7 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

# 1.7.1 Sur Rocky Linux 8 ou 9



**ATTENTION**: Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages postgresql, postgresql-client et postgresql-server! L'utilisation des dépôts du PGDG est fortement conseillée.

# Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 17);
- la distribution (ici Rocky Linux 8);
- l'architecture (ici x86\_64, la plus courante).

#### Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
# dnf -qy module disable postgresql
```

# Installation de PostgreSQL 17 (client, serveur, librairies, extensions):

```
# dnf install -y postgresql17-server postgresql17-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql17-llvmjit
```

#### Création d'une première instance :

Il est conseillé de déclarer PG\_SETUP\_INITDB\_OPTIONS, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb
# cat /var/lib/pgsql/17/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

Success. You can now start the database server using:

/usr/pgsql-17/bin/pg\_ctl -D /var/lib/pgsql/17/data/ -l logfile start

#### Chemins:

Objet	Chemin
Binaires	/usr/pgsql-17/bin
Répertoire de l'utilisateur <b>postgres</b>	/var/lib/pgsql
PGDATA par défaut	/var/lib/pgsql/17/data
Fichiers de configuration	dans PGDATA/
Traces	dans PGDATA/log

# **Configuration:**

Modifier postgresql.conf est facultatif pour un premier lancement.

# Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-17
# systemctl stop postgresql-17
# systemctl status postgresql-17
# systemctl reload postgresql-17
# systemctl restart postgresql-17
```

# Test rapide de bon fonctionnement et connexion à psql:

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

# Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-17
```

#### Ouverture du firewall pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que listen\_addresses doit être également modifié dans postgresql.conf.)

#### Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans dnf install ...) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de /var/lib/pgsql automatiquement créé à l'installation. Il faudra juste modifier les ports dans les postgresql.conf pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des PGDATA différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sousrépertoire de /var/lib/pgsqsl/17/ (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple infocentre :

- Création du fichier service de la deuxième instance :

- Modification de ce dernier fichier avec le nouveau chemin :

Environment=PGDATA=/var/lib/pgsql/17/infocentre

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb postgresql-17-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de /var/lib/pgsql/17/infocentre/postgresql.conf (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-17-infocentre
# systemctl [enable|disable] postgresql-17-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

## 1.7.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence: https://apt.postgresql.org/

# Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y gnupg2 postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt /etc/apt/sources.list.d/pgdg.list adapté à la distribution en place.

# Installation de PostgreSQL 17:

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans /etc/postgresql-common/createcluster.conf , paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-17 postgresql-client-17
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut main ).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

#### Chemins:

Objet	Chemin
Binaires	/usr/lib/postgresql/17/bin/
Répertoire de l'utilisateur <b>postgres</b>	/var/lib/postgresql
PGDATA de l'instance par défaut	/var/lib/postgresql/17/main
Fichiers de configuration	dans /etc/postgresql/17/main/
Traces	dans /var/log/postgresql/

# Configuration

Modifier postgresql.conf est facultatif pour un premier essai.

#### Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 17 main [start|stop|reload|status|restart]
```

# Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans /etc/postgresql/17/main/start.conf.

#### Ouverture du firewall:

Debian et Ubuntu n'installent pas de firewall par défaut.

#### Statut des instances du serveur :

```
# pg_lsclusters
```

# Test rapide de bon fonctionnement et connexion à psql:

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

#### Destruction d'une instance :

```
# pg_dropcluster 17 main
```

# Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

 optionnellement, /etc/postgresql-common/createcluster.conf permet de mettre en place tout d'entrée les checksums, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 17 infocentre
```

Il est également possible de préciser certains paramètres du fichier postgresql.conf, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 17 infocentre \
    --port=12345 \
    --datadir=/PGDATA/17/infocentre \
    --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
    -- --data-checksums --waldir=/ssd/postgresql/17/infocentre/journaux
    - adapter au besoin /etc/postgresql/17/infocentre/postgresql.conf;
    - démarrage:
# pg_ctlcluster 17 infocentre start
```

# 1.7.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par sudo est nécessaire :

```
$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à trust le type de la connexion en local dans le pg\_hba.conf :

```
local all postgres trust
```

La connexion en tant qu'utilisateur postgres (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (17.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans pg\_hba.conf, paramétrer une authentification par mot de passe pour les accès depuis localhost (déjà en place sous Debian):

```
# IPv4 local connections:
host all all 127.0.0.1/32 scram-sha-256
# IPv6 local connections:
host all all ::1/128 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur postgres de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (17.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier .pgpass dans le répertoire personnel doit contenir les informations sur cette connexion :

localhost:5432:\*:postgres:motdepassetrèslong

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

Pour n'avoir à taper que psql, on peut définir ces variables d'environnement dans la session voire dans ~/.bashrc :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

# Rappels:

- en cas de problème, consulter les traces (dans /var/lib/pgsql/17/data/log ou /var/log/postgresql/);
- toute modification de pg\_hba.conf ou postgresql.conf impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-17
root:~# pg_ctlcluster 17 main reload
postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

# 1.8 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/j4\_solutions.

Tous les TP se basent sur la configuration par défaut de PostgreSQL, sauf précision contraire.

Cette série de question utilise la base **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, magasin et facturation. Penser au search\_path.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1;
SET random_page_cost TO 4;
```

# 1.8.1 Index « simples »



But: Mettre en avant un cas d'usage d'un index « simple »

Considérons le cas d'usage d'une recherche de commandes par date. Le besoin fonctionnel est le suivant : renvoyer l'intégralité des commandes passées au mois de janvier 2014.

Créer la requête affichant l'intégralité des commandes passées au mois de janvier 2014.

Afficher le plan de la requête, en utilisant EXPLAIN (ANALYZE, BUFFERS). Que constate-t-on?

Nous souhaitons désormais afficher les résultats à l'utilisateur par ordre de date croissante.

Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan. Que constate-t-on?

Maintenant, nous allons essayer d'optimiser ces deux requêtes.

Créer un index permettant de répondre à ces requêtes.

Afficher de nouveau le plan des deux requêtes. Que constate-t-on?

Maintenant, étudions l'impact des index pour une opération de jointure. Le besoin fonctionnel est désormais de lister toutes les commandes associées à un client (admettons, dont le client\_id vaut 3), avec les informations du client lui-même.

```
Écrire la requête affichant commandes.nummero_commande et clients.type_client pour client_id = 3. Afficher son plan. Que constate-t-on?
```

Créer un index pour accélérer cette requête.

Afficher de nouveau son plan. Que constate-t-on?

#### 1.8.2 Sélectivité



**But** : Comprendre la sélectivité des index.

Écrire une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').

Ajouter un index sur la colonne type\_client, et rejouer les requêtes précédentes.

Afficher leurs plans d'exécution. Que se passe-t-il? Pourquoi?

# 1.8.3 Index partiels



But: Mettre en avant un cas d'usage d'un index partiel

Sur la base fournie pour les TPs, les lots non livrés sont constamment requêtés. Notamment, un système d'alerte est mis en place afin d'assurer un suivi qualité sur les lots expédié depuis plus de 3 jours (selon la date d'expédition), mais non réceptionné (date de réception à NULL).

Écrire la requête correspondant à ce besoin fonctionnel (il est normal qu'elle ne retourne rien).

Afficher le plan d'exécution.

Quel index partiel peut-on créer pour optimiser?

Afficher le nouveau plan d'exécution et vérifier l'utilisation du nouvel index.

#### 1.8.4 Index fonctionnels



But: Cas d'usage d'un index fonctionnel

Pour répondre aux exigences de stockage, l'application a besoin de pouvoir trouver rapidement les produits dont le volume est compris entre certaines bornes (nous négligeons ici le facteur de forme, qui est problématique dans le cadre d'un véritable stockage en entrepôt!).

Écrire une requête permettant de renvoyer l'ensemble des produits (table magasin.produits) dont le volume ne dépasse pas 1 litre (les unités de longueur sont en mm, 1 litre = 1 000 000 mm<sup>3</sup>).

Quel index permet d'optimiser cette requête ? (Utiliser une fonction est possible, mais pas obligatoire.)

# 1.8.5 Cas d'index non utilisés



But : Mettre en avant des cas d'index inutilisés

Un développeur cherche à récupérer les commandes dont le numéro d'expédition est 190774 avec cette requête :

SELECT \* FROM lignes\_commandes WHERE numero\_lot\_expedition = '190774'::numeric ;

Afficher le plan de la requête.

Créer un index pour améliorer son exécution.

L'index est-il utilisé? Quel est le problème?

# **DALIBO Formations**

Écrire une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

Créer un index pour améliorer l'exécution de cette requête.

Pourquoi celui-ci n'est-il pas utilisé ? (Conseil : regarder la vue pg\_stats)

Faire le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

# 1.9 TRAVAUX PRATIQUES (SOLUTIONS)

Tout d'abord, nous positionnons le search\_path pour chercher les objets des schémas magasin et facturation :

```
SET search_path = magasin, facturation;
```

# 1.9.1 Index « simples »

Considérons le cas d'usage d'une recherche de commandes par date. Le besoin fonctionnel est le suivant : renvoyer l'intégralité des commandes passées au mois de janvier 2014.

Créer la requête affichant l'intégralité des commandes passées au mois de janvier 2014.

Pour renvoyer l'ensemble de ces produits, la requête est très simple :

```
SELECT * FROM commandes date_commande
WHERE date_commande >= '2014-01-01'
AND date_commande < '2014-02-01';</pre>
```

Afficher le plan de la requête, en utilisant EXPLAIN (ANALYZE, BUFFERS) . Que constate-t-on?

Le plan de celle-ci est le suivant :

Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan. Que constate-t-on?

```
Ajoutons la clause ORDER BY:
```

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01'
ORDER BY date_commande;</pre>
```

#### QUERY PLAN

On constate ici que lors du parcours séquentiel, 980 796 lignes ont été lues, puis écartées car ne correspondant pas au prédicat, nous laissant ainsi avec un total de 19 204 lignes. Les valeurs précises peuvent changer, les données étant générées aléatoirement. De plus, le tri a été réalisé en mémoire. On constate de plus que 10 158 blocs ont été parcourus, ici depuis le cache, mais ils auraient pu l'être depuis le disque.

Créer un index permettant de répondre à ces requêtes.

Création de l'index :

```
CREATE INDEX idx_commandes_date_commande ON commandes(date_commande);
```

Afficher de nouveau le plan des deux requêtes. Que constate-t-on?

Le temps d'exécution a été réduit considérablement : la requête est 25 fois plus rapide. On constate notamment que seuls 254 blocs ont été parcourus.

Pour la requête avec la clause ORDER BY, nous obtenons le plan d'exécution suivant :

```
QUERY PLAN

Index Scan using idx_commandes_date_commande on commandes

(cost=0.42..822.60 rows=19674 width=50)
```

```
(actual time=0.032..3.378 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
                AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
Planning time: 0.516 ms
 Execution time: 4.049 ms
Celui-ci est identique! En effet, l'index permettant un parcours trié, l'opération de tri est ici « gra-
tuite ».
  Écrire la requête affichant commandes.nummero_commande et clients.type_client pour
   client_id = 3 . Afficher son plan. Que constate-t-on?
EXPLAIN (ANALYZE, BUFFERS) SELECT numero_commande, type_client FROM commandes
      INNER JOIN clients ON commandes.client_id = clients.client_id
    WHERE clients.client_id = 3;
                                 QUERY PLAN
 ______
Nested Loop (cost=0.29..22666.42 rows=11 width=101)
              (actual time=8.799..80.771 rows=14 loops=1)
  Buffers: shared hit=10161
  -> Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.017..0.018 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=3
   -> Seg Scan on commandes (cost=0.00..22658.00 rows=11 width=50)
                              (actual time=8.777..80.734 rows=14 loops=1)
         Filter: (client_id = 3)
         Rows Removed by Filter: 999986
         Buffers: shared hit=10158
Planning time: 0.281 ms
 Execution time: 80.853 ms
  Créer un index pour accélérer cette requête.
CREATE INDEX ON commandes (client_id) ;
  Afficher de nouveau son plan. Que constate-t-on?
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
      INNER JOIN clients on commandes.client_id = clients.client_id
     WHERE clients.client_id = 3;
                              QUERY PLAN
Nested Loop (cost=4.80..55.98 rows=11 width=101)
              (actual time=0.064..0.189 rows=14 loops=1)
   Buffers: shared hit=23
   -> Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.032..0.032 rows=1 loops=1)
         Index Cond: (client_id = 3)
```

```
Buffers: shared hit=6

-> Bitmap Heap Scan on commandes (cost=4.51..47.56 rows=11 width=50) (actual time=0.029..0.147 rows=14 loops=1)

Recheck Cond: (client_id = 3)
Heap Blocks: exact=14
Buffers: shared hit=17
-> Bitmap Index Scan on commandes_client_id_idx (cost=0.00..4.51 rows=11 width=0) (actual time=0.013..0.013 rows=14 loops=1)

Index Cond: (client_id = 3)
Buffers: shared hit=3

Planning time: 0.486 ms

Execution time: 0.264 ms
```

On constate ici un temps d'exécution divisé par 160 : en effet, on ne lit plus que 17 blocs pour la commande (3 pour l'index, 14 pour les données) au lieu de 10 158.

# 1.9.2 Sélectivité

Écrire une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').

Les requêtes:

```
SELECT * FROM clients WHERE type_client = 'P';
SELECT * FROM clients WHERE type_client = 'E';
```

Ajouter un index sur la colonne type\_client, et rejouer les requêtes précédentes.

Pour créer l'index :

```
CREATE INDEX ON clients (type_client);
```

Afficher leurs plans d'exécution. Que se passe-t-il ? Pourquoi ?

Les plans d'éxécution :

## QUERY PLAN

L'optimiseur sait estimer, à partir des statistiques (consultables via la vue pg\_stats), qu'il y a approximativement 89 000 clients particuliers, contre 8 000 clients entreprise.

Dans le premier cas, la majorité de la table sera parcourue, et renvoyée : il n'y a aucun intérêt à utiliser l'index.

Dans l'autre, le nombre de lignes étant plus faible, l'index est bel et bien utilisé (via un *Bitmap Scan*, ici).

# 1.9.3 Index partiels

Sur la base fournie pour les TPs, les lots non livrés sont constamment requêtés. Notamment, un système d'alerte est mis en place afin d'assurer un suivi qualité sur les lots expédié depuis plus de 3 jours (selon la date d'expédition), mais non réceptionné (date de réception à NULL).

Écrire la requête correspondant à ce besoin fonctionnel (il est normal qu'elle ne retourne rien).

La requête est la suivante :

```
SELECT * FROM lots
    WHERE date_reception IS NULL
AND    date_expedition < now() - '3d'::interval;</pre>
```

Afficher le plan d'exécution.

Le plans (ci-dessous avec ANALYZE) opère un *Seq Scan* parallélisé, lit et rejette toutes les lignes, ce qui est évidemment lourd :

```
QUERY PLAN
```

```
Gather (cost=1000.00..17764.65 rows=1 width=43) (actual time=28.522..30.993 rows=0

→ loops=1)
Workers Planned: 2
Workers Launched: 2

-> Parallel Seq Scan on lots (cost=0.00..16764.55 rows=1 width=43) (actual

→ time=24.887..24.888 rows=0 loops=3)
Filter: ((date_reception IS NULL) AND (date_expedition < (now() - '3

→ days'::interval)))
Rows Removed by Filter: 335568
```

Planning Time: 0.421 ms Execution Time: 31.012 ms

Quel index partiel peut-on créer pour optimiser?

On peut optimiser ces requêtes sur les critères de recherche à l'aide des index partiels suivants :

```
CREATE INDEX ON lots (date_expedition) WHERE date_reception IS NULL;
```

Afficher le nouveau plan d'exécution et vérifier l'utilisation du nouvel index.

```
EXPLAIN (ANALYZE)

SELECT * FROM lots

WHERE date_reception IS NULL

AND date_expedition < now() - '3d'::interval;

QUERY PLAN

Index Scan using lots_date_expedition_idx on lots (cost=0.13..4.15 rows=1 width=43) (actual time=0.008..0.009 rows=0 loops=1)

Index Cond: (date_expedition < (now() - '3 days'::interval))

Planning Time: 0.243 ms

Execution Time: 0.030 ms
```

Il est intéressant de noter que seul le test sur la condition indexée (date\_expedition) est présent dans le plan : la condition date\_reception IS NULL est implicitement validée par l'index partiel.

Attention, il peut être tentant d'utiliser une formulation de la sorte pour ces requêtes :

```
SELECT * FROM lots
WHERE date_reception IS NULL
AND now() - date_expedition > '3d'::interval;
```

D'un point de vue logique, c'est la même chose, mais l'optimiseur n'est pas capable de réécrire cette requête correctement. Ici, le nouvel index sera tout de même utilisé, le volume de lignes satisfaisant au critère étant très faible, mais il ne sera pas utilisé pour filtrer sur la date :

La ligne importante et différente ici concerne le Filter en lieu et place du Index Cond du plan précédent. Ici tout l'index partiel (certes tout petit) est lu intégralement et les lignes testées une à une.

C'est une autre illustration des points vus précédemment sur les index non utilisés.

# 1.9.4 Index fonctionnel

Ce TP utilise la base magasin.

Écrire une requête permettant de renvoyer l'ensemble des produits (table magasin.produits) dont le volume ne dépasse pas 1 litre (les unités de longueur sont en mm, 1 litre = 1 000 000 mm<sup>3</sup>).

Concernant le volume des produits, la requête est assez simple :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000 ;
```

Quel index permet d'optimiser cette requête ? (Utiliser une fonction est possible, mais pas obligatoire.)

L'option la plus simple est de créer l'index de cette façon, sans avoir besoin d'une fonction :

```
CREATE INDEX ON produits((longueur * hauteur * largeur));
```

En général, il est plus propre de créer une fonction. On peut passer la ligne entière en paramètre pour éviter de fournir 3 paramètres. Il faut que cette fonction soit IMMUTABLE pour être indexable :

```
CREATE OR REPLACE function volume (p produits)
RETURNS numeric
AS $$
SELECT p.longueur * p.hauteur * p.largeur;
$$ language SQL
PARALLEL SAFE
IMMUTABLE;
```

(Elle est même PARALLEL SAFE pour la même raison qu'elle est IMMUTABLE : elle dépend uniquement des données de la table.)

On peut ensuite indexer le résultat de cette fonction :

```
CREATE INDEX ON produits (volume(produits));
```

Il est ensuite possible d'écrire la requête de plusieurs manières, la fonction étant ici écrite en SQL et non en PL/pgSQL ou autre langage procédural :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000 ;
SELECT * FROM produits WHERE volume(produits) < 1000000 ;</pre>
```

En effet, l'optimiseur est capable de « regarder » à l'intérieur de la fonction SQL pour déterminer que les clauses sont les mêmes, ce qui n'est pas vrai pour les autres langages.

En revanche, la requête suivante, où la multiplication est faite dans un ordre différent, n'utilise pas l'index :

```
SELECT * FROM produits WHERE largeur * longueur * hauteur < 10000000 ;
```

et c'est notamment pour cette raison qu'il est plus propre d'utiliser la fonction.

De part l'origine « relationnel-objet » de PostgreSQL, on peut même écrire la requête de la manière suivante :

```
SELECT * FROM produits WHERE produits.volume < 1000000;
```

# 1.9.5 Cas d'index non utilisés

Afficher le plan de la requête.

Le moteur fait un parcours séquentiel et retire la plupart des enregistrements pour n'en conserver que 6.

Créer un index pour améliorer son exécution.

```
CREATE INDEX ON lignes_commandes (numero_lot_expedition);
```

L'index est-il utilisé? Quel est le problème?

L'index n'est pas utilisé à cause de la conversion bigint vers numeric. Il est important d'utiliser les bons types :

Sans conversion la requête est bien plus rapide. Faites également le test sans index, le *Seq Scan* sera également plus rapide, le moteur n'ayant pas à convertir toutes les lignes parcourues.

Écrire une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

# QUERY PLAN

Créer un index pour améliorer l'exécution de cette requête.

CREATE INDEX ON lignes\_commandes(quantite);

Pourquoi celui-ci n'est-il pas utilisé ? (Conseil : regarder la vue pg\_stats)

La table pg\_stats nous donne des informations de statistiques. Par exemple, pour la répartition des valeurs pour la colonne quantite:

Ces quelques lignes nous indiquent qu'il y a 10 valeurs distinctes et qu'il y a environ 10 % d'enregistrements correspondant à chaque valeur.

Avec le prédicat quantite BETWEEN 1 and 8, le moteur estime récupérer environ 80 % de la table. Il est donc bien plus coûteux de lire l'index et la table pour récupérer 80 % de la table. C'est pourquoi le moteur fait un *Seq Scan* qui moins coûteux.

Faire le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

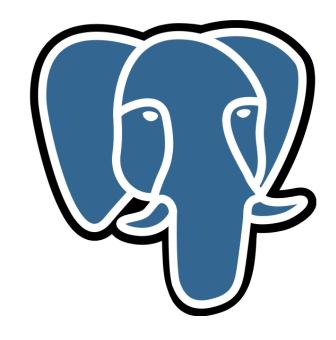
# **DALIBO Formations**

```
(actual time=194.250..194.250 rows=1254886 loops=1)
    Index Cond: ((quantite >= 1) AND (quantite <= 4))
    Buffers: shared read=3431
Planning time: 0.271 ms
Execution time: 648.414 ms
(9 rows)</pre>
```

Cette fois, la sélectivité est différente et le nombre d'enregistrements moins élevé. Le moteur passe donc par un parcours d'index.

Cet exemple montre qu'on indexe selon une requête et non selon une table.

# 2/ Indexation avancée



# 2.1 INDEX AVANCÉS



De nombreuses fonctionnalités d'indexation sont disponibles dans PostgreSQL:

- Index B-tree
  - multicolonnes
  - fonctionnels
  - partiels
  - couvrants
- Classes d'opérateurs
- Indexation GIN, GiST, BRIN, hash, bloom
- Indexation de motifs
- Indexation multicolonne

PostgreSQL fournit de nombreux types d'index, afin de répondre à des problématiques de recherches complexes.

# 2.2 INDEX B-TREE (RAPPELS)



- Index B-tree fonctionnel, multicolonne, partiel, couvrant:
  - requête cible :

```
SELECT col4 FROM ma_table
WHERE col3<12 and f(col1)=7 and col2 LIKE 'toto%';
   - index dédié:
CREATE INDEX idx_adv ON ma_table (f(col1), col2 varchar_pattern_ops)
INCLUDE (col4) WHERE col3<12;</pre>
```

- Rappel: un index est coûteux à maintenir!

Rappelons que l'index classique est créé comme ceci :

```
CREATE INDEX mon_index ON ma_table(ma_colonne) ;
```

B-tree (en arbre équilibré) est le type d'index le plus fréquemment utilisé.

Toutes les fonctionnalités vues précédemment peuvent être utilisées simultanément. Il est parfois tentant de créer des index très spécialisés grâce à toutes ces fonctionnalités, comme dans l'exemple ci-dessus. Mais il ne faut surtout pas perdre de vue qu'un index est une structure lourde à mettre à jour, comparativement à une table. Une table avec un seul index est environ 3 fois plus lente qu'une table nue, et chaque index ajoute le même surcoût. Il est donc souvent plus judicieux d'avoir des index pouvant répondre à plusieurs requêtes différentes, et de ne pas trop les spécialiser. Il faut trouver un juste compromis entre le gain à la lecture et le surcoût à la mise à jour.

# 2.3 INDEX GIN



Un autre type d'index

- Définition
- Utilisation avec des données non structurées
- Utilisation avec des données scalaires
- Mise à jour

Les index B-tree sont les plus utilisés, mais PostgreSQL propose d'autres types d'index. Le type GIN est l'un des plus connus.

# 2.3.1 GIN: définition & données non structurées



GIN: Generalized Inverted iNdex

- Index inversé généralisé
  - les champs sont décomposés en éléments (par API)
  - l'index associe une valeur à la liste de ses adresses
- Pour chaque entrée du tableau
  - liste d'adresses où le trouver
- Utilisation principale: données non scalaires
  - tableaux, listes, non structurées...

Un index inversé est une structure classique, utilisée le plus souvent dans l'indexation *Full Text*. Le principe est de décomposer un document en sous-structures, et ce sont ces éléments qui seront indexées. Par exemple, un document sera décomposé en la liste de ses mots, et chaque mot sera une clé de l'index. Cette clé fournira la liste des documents contenant ce mot. C'est l'inverse d'un index B-tree classique, qui va lister chacune des occurrences d'une valeur et y associer sa localisation.

Par analogie: dans un livre de cuisine, un index classique permettrait de chercher « Crêpes au caramel à l'armagnac » et « Sauce caramel et beurre salé », alors qu'un index GIN contiendrait « caramel », « crêpes », « armagnac », « sauce », « beurre »...

Pour plus de détails sur la structure elle-même, cet article Wikipédia<sup>1</sup> est une lecture conseillée.

<sup>&</sup>lt;sup>1</sup>https://fr.wikipedia.org/wiki/Index\_invers%C3%A9

Dans l'implémentation de PostgreSQL, un index GIN est construit autour d'un index B-tree des éléments indexés, et à chacun est associé soit une simple liste de pointeurs vers la table (posting list) pour les petites listes, soit un pointeur vers un arbre B-tree contenant ces pointeurs (posting tree). La pending list est une optimisation des écritures (voir plus bas).

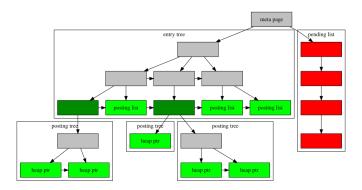


Figure 2/.1: Schéma des index GIN (documentation officielle de PostgreSQL, licence PostgreSQL)

Les index GIN de PostgreSQL sont « généralisés », car ils sont capables d'indexer n'importe quel type de données, à partir du moment où on leur fournit les différentes fonctions d'API permettant le découpage et le stockage des différents *items* composant la donnée à indexer. En pratique, ce sont les opérateurs indiqués à la création de l'index, parfois implicites, qui contiendront la logique nécessaire.

Les index GIN sont des structures lentes à la mise à jour. Par contre, elles sont extrêmement efficaces pour les interrogations multicritères, ce qui les rend très appropriées pour l'indexation *Full Text*, des champs jsonb ...

# 2.3.2 GIN et les tableaux



Quels tableaux contiennent une valeur donnée?

# GIN et champ structuré:

Comme premier exemple d'indexation d'un champ structuré, prenons une table listant des voitures<sup>2</sup>, dont un champ caracteristiques contient une liste d'attributs séparés par des virgules. Ceci ne respecte bien entendu pas la première forme normale.

Cette liste peut être transformée facilement en tableau avec regexp\_split\_to\_array. Des opérateurs de manipulation peuvent alors être utilisés, comme : (a) (contient), (d) (contenu par), (a) des éléments en communs). Par exemple, pour chercher les voitures possédant deux caractéristiques données, la requête est :

immatriculation	modele	caracteristiques
XB-025-PH RC-561-BI	clio   megane 	toit ouvrant,climatisation regulateur de vitesse,boite automatique, toit ouvrant,climatisation,
LU-190-K0 SV-193-YR FG-432-FZ	megane   megane   kangoo	toit ouvrant,climatisation,4 roues motrices climatisation,abs,toit ouvrant climatisation,jantes aluminium,regulateur de vitesse, toit ouvrant

# avec ce plan:

```
QUERY PLAN
```

```
Seq Scan on voitures (cost=0.00..1406.20 rows=1 width=96)
Filter: (regexp_split_to_array(caracteristiques, ','::text) @> '{"toit
ouvrant",climatisation}'::text[])
```

Pour accélérer la recherche, le tableau de textes peut être directement indexé avec un index GIN, ici un index fonctionnel :

```
CREATE INDEX idx_attributs_array ON voitures
USING gin (regexp_split_to_array(caracteristiques,','));
```

On ne précise pas d'opérateur, celui par défaut pour les tableaux convient.

## Le plan devient :

# GIN et tableau:

GIN supporte nativement les tableaux des types scalaires (int, float, text, date ...):

<sup>&</sup>lt;sup>2</sup>https://dali.bo/tp\_voitures

CREATE TABLE tablo (i int, a int[]);

```
INSERT INTO tablo SELECT i, ARRAY[i, i+1] FROM generate_series(1,100000) i;
Un index B-tree classique permet de rechercher un tableau identique à un autre, mais pas de chercher
un tableau qui contient une valeur scalaire à l'intérieur du tableau :
EXPLAIN (COSTS OFF) SELECT * FROM tablo WHERE a = ARRAY[42,43];
                   QUERY PLAN
 Bitmap Heap Scan on tablo
   Recheck Cond: (a = '{42,43}'::integer[])
   -> Bitmap Index Scan on tablo_a_idx
         Index Cond: (a = '{42,43}'::integer[])
SELECT * FROM tablo WHERE a @> ARRAY[42] ;
i |
       а
41 | {41,42}
42 | {42,43}
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT * FROM tablo WHERE a @> ARRAY[42];
                          QUERY PLAN
______
Seq Scan on tablo (actual time=0.023..19.322 rows=2 loops=1)
   Filter: (a @> '{42}'::integer[])
   Rows Removed by Filter: 99998
  Buffers: shared hit=834
Planning:
  Buffers: shared hit=6 dirtied=1
Planning Time: 0.107 ms
 Execution Time: 19.337 ms
L'indexation GIN permet de chercher des valeurs figurant à l'intérieur des champs indexés :
CREATE INDEX ON tablo USING gin (a);
pour un résultat beaucoup plus efficace :
Bitmap Heap Scan on tablo (actual time=0.010..0.010 rows=2 loops=1)
   Recheck Cond: (a @> '{42}'::integer[])
  Heap Blocks: exact=1
  Buffers: shared hit=5
  -> Bitmap Index Scan on tablo_a_idx1 (actual time=0.007..0.007 rows=2 loops=1)
         Index Cond: (a @> '{42}'::integer[])
         Buffers: shared hit=4
 Planning:
   Buffers: shared hit=23
 Planning Time: 0.121 ms
 Execution Time: 0.023 ms
```

# 2.3.3 GIN pour les JSON et les textes



- JSON:
  - opérateur jsonb\_path\_ops ou jsonb\_ops
- Indexation de trigrammes
  - extensions pg\_trgm (opérateur dédié gin\_trgm\_ops )
- Recherche Full Text
  - indexe les vecteurs

Le principe est le même pour des JSON, s'ils sont bien stockés dans un champ de type jsonb. Les recherches de l'existence d'une clé à la racine du document ou tableau JSON sont réalisées avec les opérateurs ?, ? | et ?&.

Les recherches sur la présence d'une valeur dans un document JSON avec les opérateurs @> , @? ou @@ peuvent être réalisées avec la classe d'opérateur par défaut (json\_ops), mais aussi la classe d'opérateur jsonb\_path\_ops, donc au choix :

```
CREATE INDEX idx_prs ON personnes USING gin (proprietes jsonb_ops) ;
CREATE INDEX idx_prs ON personnes USING gin (proprietes jsonb_path_ops) ;
```

La classe jsonb\_path\_ops est plus performante pour ce genre de recherche et génère des index plus compacts lorsque les clés apparaissent fréquemment dans les données. Par contre, elle ne permet pas d'effectuer efficacement des recherches de structure JSON vide du type : { "a" : {} }. Dans ce dernier cas, PostgreSQL devra faire, au mieux, un parcours de l'index complet, au pire un parcours séquentiel de la table. Le choix de la meilleure classe pour l'index dépend fortement de la typologie des données.

La documentation officielle<sup>3</sup> entre plus dans le détail.

L'extension pg\_trgm utilise aussi les index GIN, pour permettre des recherches de type :

```
SELECT * FROM ma_table
WHERE ma_col_texte LIKE '%ma_chaine1%ma_chaine2%';
```

<sup>&</sup>lt;sup>3</sup>https://docs.postgresql.fr/current/datatype-json.html#JSON-INDEXING

L'extension fournit un opérateur dédié pour l'indexation (voir plus loin).

La recherche *Full Text* est généralement couplée à un index GIN pour indexer les tsvector (voir le module T1<sup>4</sup>).

# 2.3.4 GIN & données scalaires



- Données scalaires aussi possibles
  - avec l'extension btree\_gin
- GIN compresse quand les données se répètent
  - alternative à bitmap!

Grâce à l'extension btree\_gin, fournie avec PostgreSQL, l'indexation GIN peut aussi s'appliquer aux scalaires. Il est ainsi possible d'indexer un ensemble de colonnes, par exemple d'entiers ou de textes (voir exemple plus loin). Cela peut servir dans les cas où une requête multicritères peut porter sur de nombreux champs différents, dont aucun n'est obligatoire. Un index B-tree est là moins adapté, voire inutilisable.

Un autre cas d'utilisation traditionnel est celui des index dits *bitmap*. Les index bitmap sont très compacts, mais ne permettent d'indexer que peu de valeurs différentes. Un index bitmap est une structure utilisant 1 bit par enregistrement pour chaque valeur indexable. Par exemple, on peut définir un index bitmap sur le sexe : deux valeurs seulement (voire quatre si on autorise NULL ou non-binaire) sont possibles. Indexer un enregistrement nécessitera donc un ou deux bits. Le défaut des index *bitmap* est que l'ajout de nouvelles valeurs est très peu performant car l'index nécessite d'importantes réécritures à chaque ajout. De plus, l'ajout de données provoque une dégradation des performances puisque la taille par enregistrement devient bien plus grosse.

Les index GIN permettent un fonctionnement sensiblement équivalent au *bitmap* : chaque valeur indexable contient la liste des enregistrements répondant au critère, et cette liste a l'intérêt d'être compressée. Par exemple, sur une base créée avec pgbench, de taille 100, avec les options par défaut :

```
CREATE EXTENSION btree_gin;

CREATE INDEX pgbench_accounts_gin_idx on pgbench_accounts USING gin (bid);

EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT * FROM pgbench_accounts WHERE bid=5;

QUERY PLAN

Bitmap Heap Scan on pgbench_accounts (actual time=7.505..19.931 rows=100000 loops=1)
Recheck Cond: (bid = 5)

4https://dali.bo/t1_html
```

Dans ce cas précis qui renvoie de nombreuses lignes, l'utilisation du GIN est même aussi efficace que le B-tree ci-dessous, car l'index GIN est mieux compressé et a besoin de lire moins de blocs :

```
CREATE INDEX pgbench_accounts_btree_idx ON pgbench_accounts (bid);
EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT * FROM pgbench_accounts WHERE bid=5;
```

### QUERY PLAN

-----

```
Index Scan using pgbench_accounts_btree_idx on pgbench_accounts (actual
    time=0.008..19.138 rows=100000 loops=1)
    Index Cond: (bid = 5)
```

Planning:

Buffers: shared hit=18 Planning Time: 0.117 ms Execution Time: 23.369 ms

Buffers: shared hit=1728

L'index GIN est en effet environ 5 fois plus compact que le B-tree dans cet exemple simple, avec 100 valeurs distinctes de bid:

Ce ne serait pas le cas avec des valeurs toutes différentes. (Avant la version 13, la différence de taille est encore plus importante car les index B-tree ne disposent pas encore de la déduplication des clés.)

Avant de déployer un index GIN, il faut vérifier l'impact du GIN sur les performances en insertions et mises à jour et l'impact sur les requêtes.

# 2.3.5 GIN: mise à jour



- Création lourde
  - maintenance\_work\_mem élevé
- Mise à jour lente
  - d'où l'option fastupdate
  - ... à désactiver si temps de réponse instable!

Les index GIN sont lourds à créer et à mettre à jour. Une valeur élevée de maintenance\_work\_mem est conseillée.

L'option fastupdate permet une mise à jour bien plus rapide. Elle est activée par défaut. PostgreSQL stocke alors les mises à jour de l'index dans une *pending list* qui est intégrée en bloc, notamment lors d'un VACUUM. Sa taille peut être modifiée par le paramètre gin\_pending\_list\_limit, par défaut à 4 Mo, et au besoin surchargeable sur chaque index.

L'inconvénient de cette technique est que le temps de réponse de l'index devient instable : certaines recherches peuvent être très rapides et d'autres très lentes. Le seul moyen d'accélérer ces recherches est de désactiver fastupdate. Cela permet en plus d'éviter la double écriture dans les journaux de transactions. Mais il y a un impact : les mises à jour de l'index sont bien plus lentes.

Il faut donc faire un choix. Si l'on conserve l'option fastupdate, il faut surveiller la fréquence de passage de l'autovacuum. L'appel manuel à la fonction gin\_clean\_pending\_list() est une autre option.

Pour les détails, voir la documentation<sup>5</sup>.

<sup>&</sup>lt;sup>5</sup>https://www.postgresql.org/docs/current/static/gin-implementation.html#GIN-FAST-UPDATE%3E

# 2.4 INDEX GIST



GiST : Generalized Search Tree

- Arbre de recherche généralisé
- Indexation non plus des valeurs mais de la véracité de prédicats
  Moins performants que B-tree (moins sélectifs)
- Moins lourds que GIN

Initialement, les index GiST sont un produit de la recherche de l'université de Berkeley. L'idée fondamentale est de pouvoir indexer non plus les valeurs dans l'arbre B-tree, mais plutôt la véracité d'un prédicat : « ce prédicat est vrai sur telle sous-branche ». On dispose donc d'une API permettant au type de données d'informer le moteur GiST d'informations comme : « quel est le résultat de la fusion de tel et tel prédicat » (pour pouvoir déterminer le prédicat du nœud parent), quel est le surcoût d'ajout de tel prédicat dans telle ou telle partie de l'arbre, comment réaliser un split (découpage) d'une page d'index, déterminer la distance entre deux prédicats, etc.

Tout ceci est très virtuel, et rarement re-développé par les utilisateurs. Par contre, certaines extensions et outils intégrés utilisent ce mécanisme.

Il faut retenir qu'un index GiST est moins performant qu'un B-tree si ce dernier est possible.

L'utilisation se recoupant en partie avec les index GIN, il faut noter que :

- les index GiST ont moins tendance à se fragmenter que les GIN, même si cela dépend énormément du type de mises à jour ;
- les index GiST sont moins lourds à maintenir;
- mais généralement moins performants.

# 2.4.1 GiST: cas d'usage



Le GiST indexe à peu près n'importe quoi

- géométries (PostGIS)intervalles, adresses IP, FTS...

D'autres usages recouvrent en partie ceux de GIN.

Un index GiST permet d'indexer n'importe quoi, quelle que soit la dimension, le type, tant qu'on peut utiliser des prédicats sur ce type.

Il est disponible pour les types natifs suivants :

- géométriques (box, circle, point, poly): le projet PostGIS utilise les index GIST massivement, pour répondre efficacement à des questions complexes telles que « quelles sont les routes qui coupent le Rhône? », « quelles sont les villes adjacentes à Toulouse? », « quels sont les restaurants situés à moins de 3 km de la Nationale 12? »;
- range (d'int, de timestamp ...);
- adresses IP/CIDR.
- Full Text Search.

Une partie des cas d'utilisation des index GIST recouvre ceux des index GIN. Nous verrons que les index GIST sont notamment utilisés par :

- les contraintes d'exclusion;
- les recherches multicolonnes;
- l'extension pg\_trgm (dans ce cas, GiST est moins efficace que GIN pour la recherche exacte, mais permet de rapidement trouver les N enregistrements les plus proches d'une chaîne donnée, sans tri, et est plus compact).

## 2.4.2 **GIST & KNN**



- KNN = K-Nearest neighbours (K-plus proches voisins)
  - c'est-à-dire:

```
SELECT ...
ORDER BY ma_colonne <-> une_référence LIMIT 10 ;
```

- Très utile pour la recherche de mots ressemblants, géographique
  - Exemple:

```
SELECT    p, p <-> point(18,36)
FROM    mes_points
ORDER BY p <-> point(18, 36)
LIMIT    4;
```

Les index GiST supportent les requêtes de type *K-plus proche voisins*, et permettent donc de répondre extrêmement rapidement à des requêtes telles que :

- quels sont les dix restaurants les plus proches d'un point particulier?
- quels sont les 5 mots ressemblant le plus à « éphélant » ? afin de proposer des corrections à un utilisateur ayant commis une faute de frappe (par exemple).

Une convention veut que l'opérateur distance soit généralement nommé « <-> », mais rien n'impose ce choix.

On peut prendre l'exemple d'indexation ci-dessus, avec le type natif point :

```
CREATE TABLE mes_points (p point);
INSERT INTO mes_points (SELECT point(i, j)
FROM generate_series(1, 100) i, generate_series(1,100) j WHERE random() > 0.8);
CREATE INDEX ON mes_points USING gist (p);
```

Pour trouver les 4 points les plus proches du point ayant pour coordonnées (18,36), on peut utiliser la requête suivante :

p	?column?
(18,37)	1
(18,35)	1
(16,36)	2
(16,35)	2.23606797749979

Cette requête utilise bien l'index GiST créé plus haut :

Les index SP-GiST sont compatibles avec ce type de recherche depuis la version 12.

# 2.4.3 GiST & Contraintes d'exclusion



Contrainte d'exclusion : une extension du concept d'unicité

- Unicité:
  - n-uplet1 = n-uplet2 interdit dans une table
- Contrainte d'exclusion :
  - n-uplet1 op n-uplet2 interdit dans une table
  - op est n'importe quel opérateur indexable par GiST
  - Exemple:

```
CREATE TABLE circles
   ( c circle,
        EXCLUDE USING gist (c WITH &&));
```

- Exemple : réservations de salles

# **Premiers exemples:**

Les contraintes d'unicité sont une forme simple de contraintes d'exclusion. Si on prend l'exemple :

```
CREATE TABLE foo (
   id int,
   nom text,
   EXCLUDE (id WITH =)
);
```

cette déclaration est équivalente à une contrainte UNIQUE sur foo.id, mais avec le mécanisme des contraintes d'exclusion. Ici, la contrainte s'appuie toujours sur un index B-tree. Les NULL sont toujours permis, exactement comme avec une contrainte UNIQUE.

On peut également poser une contrainte unique sur plusieurs colonnes :

```
CREATE TABLE foo (
   nom text,
   naissance date,
   EXCLUDE (nom WITH =, naissance WITH =)
);
```

# Intérêt:

L'intérêt des contraintes d'exclusion est qu'on peut utiliser des index d'un autre type que les B-tree, comme les GiST ou les hash, et surtout des opérateurs autres que l'égalité, ce qui permet de couvrir des cas que les contraintes habituelles ne savent pas traiter.

Par exemple, une contrainte UNIQUE ne permet pas d'interdire que deux enregistrements de type intervalle aient des bornes qui se chevauchent. Cependant, il est possible de le faire avec une contrainte d'exclusion.

# **Exemples:**

L'exemple suivante implémente la contrainte que deux objets de type circle (cercle) ne se chevauchent pas. Or ce type ne s'indexe qu'avec du GiST :

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
INSERT INTO circles(c) VALUES ('10, 4, 10');
INSERT INTO circles(c) VALUES ('8, 3, 8');

ERROR: conflicting key value violates exclusion constraint "circles_c_excl"
DETAIL: Key (c)=(<(8,3),8>) conflicts with existing key (c)=(<(10,4),10>).
```

Un autre exemple très fréquemment proposé est celui de la réservation de salles de cours sur des plages horaires qui ne doivent pas se chevaucher:

```
CREATE TABLE reservation
  salle
           TEXT,
  professeur TEXT,
  durant tstzrange);
CREATE EXTENSION btree_gist ;
ALTER TABLE reservation ADD CONSTRAINT test_exclude EXCLUDE
USING gist (salle WITH =, durant WITH &&);
INSERT INTO reservation (professeur, salle, durant) VALUES
( 'marc', 'salle techno', '[2010-06-16 09:00:00, 2010-06-16 10:00:00)');
INSERT INTO reservation (professeur,salle,durant) VALUES
( 'jean', 'salle techno', '[2010-06-16 10:00:00, 2010-06-16 11:00:00)');
INSERT INTO reservation (professeur,salle,durant) VALUES
( 'jean', 'salle informatique', '[2010-06-16 10:00:00, 2010-06-16 11:00:00)');
INSERT INTO reservation (professeur, salle, durant) VALUES
( 'michel', 'salle techno', '[2010-06-16 10:30:00, 2010-06-16 11:00:00)');
ERROR: conflicting key value violates exclusion constraint "test_exclude"
DETAIL: Key (salle, durant)=(salle techno,
              ["2010-06-16 10:30:00+02","2010-06-16 11:00:00+02"))
         conflicts with existing key
         (salle, durant)=(salle techno,
                          ["2010-06-16 10:00:00+02","2010-06-16 11:00:00+02")).
```

On notera que, là encore, l'extension btree\_gist permet d'utiliser l'opérateur = avec un index GiST, ce qui nous permet d'utiliser = dans une contrainte d'exclusion.

Cet exemple illustre la puissance du mécanisme. Il est quasiment impossible de réaliser la même opération sans contrainte d'exclusion, à part en verrouillant intégralement la table, ou en utilisant le mode d'isolation *serializable*, qui a de nombreuses implications plus profondes sur le fonctionnement de l'application.

# Autres fonctionnalités :

# **DALIBO Formations**

Enfin, précisons que les contraintes d'exclusion supportent toutes les fonctionnalités avancées que l'on est en droit d'attendre d'un système comme PostgreSQL : mode différé (*deferred*), application de la contrainte à un sous-ensemble de la table (permet une clause WHERE), ou utilisation de fonctions/expressions en place de références de colonnes.

# 2.5 GIN, GIST & PG\_TRGM



- Indexation des recherches LIKE '%critère%'
- Similarité basée sur des trigrammes

```
CREATE EXTENSION pg_trgm;
SELECT similarity('bonjour','bnojour');
similarity
-----
0.333333

- Indexation (GIN ou GiST):

CREATE INDEX test_trgm_idx ON test_trgm
USING gist (text_data gist_trgm_ops);
```

Ce module permet de décomposer en trigramme les chaînes qui lui sont proposées :

Une fois les trigrammes indexés, on peut réaliser de la recherche floue, ou utiliser des clauses MIKE malgré la présence de jokers (%) n'importe où dans la chaîne. À l'inverse, les indexations simples, de type B-tree, ne permettent des recherches efficaces que dans un cas particulier: si le seul joker de la chaîne est à la fin de celle ci (LIKE 'hello%' par exemple). Contrairement à la Full Text Search, la recherche par trigrammes ne réclame aucune modification des requêtes.

```
CREATE EXTENSION pg_trgm;

CREATE TABLE test_trgm (text_data text);

INSERT INTO test_trgm(text_data)
VALUES ('hello'), ('hello everybody'),
('helo young man'),('hallo!'),('HELLO !');
INSERT INTO test_trgm SELECT 'hola' FROM generate_series(1,1000);

CREATE INDEX test_trgm_idx ON test_trgm
    USING gist (text_data gist_trgm_ops);

SELECT text_data FROM test_trgm
WHERE text_data like '%hello%';
    text_data
```

```
hello
hello everybody

Cette dernière requête passe par l'index test_trgm_idx, malgré le % initial:

EXPLAIN (ANALYZE)

SELECT text_data FROM test_trgm
WHERE text_data like '%hello%';

QUERY PLAN

Index Scan using test_trgm_gist_idx on test_trgm
(cost=0.41..0.63 rows=1 width=8) (actual time=0.174..0.204 rows=2 loops=1)
Index Cond: (text_data ~~ '%hello%'::text)
Rows Removed by Index Recheck: 1
Planning time: 0.202 ms
Execution time: 0.250 ms
```

On peut aussi utiliser un index GIN (comme pour le *Full Text Search*). Les index GIN ont l'avantage d'être plus efficaces pour les recherches exhaustives. Mais l'indexation pour la recherche des k éléments les plus proches (on parle de recherche k-NN) n'est disponible qu'avec les index GIST.

```
SELECT text_data, text_data <-> 'hello'
FROM test_trgm
ORDER BY text_data <-> 'hello'
LIMIT 4;
```

nous retourne par exemple les deux enregistrements les plus proches de « hello » dans la table test\_trgm.

# 2.6 INDEXATION MULTICOLONNE: GIN, GIST & BLOOM



- Multicolonne dans n'importe quel ordre
- GIN ou GiST
  - extensions btree\_gist ou btree\_gin
- Ou bloom?
- Quel est le meilleur?
  - ça dépend...

GiST comme GIN sont intéressants si on a besoin d'indexer plusieurs colonnes, sans trop savoir quelles colonnes seront interrogées.

Pour indexer des scalaires, il faut utiliser les extensions btree\_gist ou btree\_gin.

```
CREATE EXTENSION IF NOT EXISTS btree_gist ;
CREATE EXTENSION IF NOT EXISTS btree_gin ;
```

Ce premier jeu de données utilisera des données qui se répètent beaucoup (de basse cardinalité) et les requêtes ramèneront de nombreuses lignes :

```
CREATE TABLE demo_gist (n int, i int, j int, k int, l int, filler char(50) default ' ');

CREATE TABLE demo_gin (n int, i int, j int, k int, l int, filler char(50) default ' ');

CREATE INDEX demo_gist_idx ON demo_gist USING gist (i,j,k,l);

CREATE INDEX demo_gin_idx ON demo_gin USING gin (i,j,k,l);

INSERT INTO demo_gist

SELECT n, mod(n,37) AS i, mod(n,53) AS j, mod (n, 97) AS k, mod(n,229) AS l

FROM generate_series (1,1000000) n;

INSERT INTO demo_gin

SELECT n, mod(n,37) AS i, mod(n,53) AS j, mod (n, 97) AS k, mod(n,229) AS l

FROM generate_series (1,1000000) n;
```

Même en ne fournissant pas la première colonne des index, les index GIN et GiST sont utilisables :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)

SELECT * FROM demo_gist WHERE j=17 AND l=17;

QUERY PLAN

Bitmap Heap Scan on demo_gist (actual time=1.615..1.662 rows=83 loops=1)
Recheck Cond: ((j = 17) AND (l = 17))
Heap Blocks: exact=83
Buffers: shared hit=434
```

```
-> Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=1.607..1.607 rows=83
→ loops=1)
         Index Cond: ((j = 17) \text{ AND } (l = 17))
         Buffers: shared hit=351
Planning Time: 0.026 ms
 Execution Time: 1.673 ms
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_gin WHERE j=17 AND l=17
Bitmap Heap Scan on demo_gin (actual time=0.436..0.491 rows=83 loops=1)
   Recheck Cond: ((j = 17) \text{ AND } (l = 17))
  Heap Blocks: exact=83
  Buffers: shared hit=100
   -> Bitmap Index Scan on demo_gin_i_j_k_l_idx (actual time=0.427..0.427 rows=83
→ loops=1)
         Index Cond: ((j = 17) \text{ AND } (l = 17))
         Buffers: shared hit=17
Planning:
   Buffers: shared hit=1
Planning Time: 0.031 ms
 Execution Time: 0.503 ms
```

Le GIN est ici plus efficace car le nombre de blocs d'index balayés est plus bas. En effet, avec une basse cardinalité, la compression du GIN joue à plein. (Pour ces tables identiques de 97 Mo sous PostgreSQL 16, l'index GiST fait 62 Mo, et le GIN 19 Mo seulement.)

Si la première colonne i était systématiquement fournie à la requête, on pourrait se contenter d'un index B-tree (30 Mo ici) ; mais il serait peu efficace pour les autres requêtes : la requête précédente donnerait souvent lieu à un *Seq Scan* parallélisé, bien que parfois un *Bitmap Scan* puisse apparaître avec des performances satisfaisantes.

Toujours dans ce cas précis, le temps de création de l'index GIN est aussi meilleur que le GiST d'un facteur deux au moins (et équivalent au B-tree), mais ce temps est très sensible à la valeur de maintenance\_work\_mem.

À l'inverse, le GiST est bien plus performant que le GIN dans d'autres types de requêtes commme celleci avec BETWEEN:

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)

SELECT * FROM demo_gist

WHERE j BETWEEN 17 AND 21 AND l BETWEEN 17 AND 21 ;

QUERY PLAN

Bitmap Heap Scan on demo_gist (actual time=8.419..8.895 rows=2059 loops=1)
Recheck Cond: ((j >= 17) AND (j <= 21) AND (l >= 17) AND (l <= 21))
Heap Blocks: exact=757
Buffers: shared hit=1744
-> Bitmap Index Scan on demo_gist_i_j_k_l_idx (actual time=8.355..8.355

rows=2059 loops=1)
    Index Cond: ((j >= 17) AND (j <= 21) AND (l >= 17) AND (l <= 21))
Buffers: shared hit=987
Planning Time: 0.030 ms
Execution Time: 8.958 ms
```

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM demo_gin
WHERE j BETWEEN 17 AND 21 AND l BETWEEN 17 AND 21 ;
                                OUERY PLAN
_____
                                           _____
 Bitmap Heap Scan on demo_gin (actual time=68.281..69.313 rows=2059 loops=1)
   Recheck Cond: ((j >= 17) \text{ AND } (j <= 21) \text{ AND } (l >= 17) \text{ AND } (l <= 21))
   Heap Blocks: exact=757
   Buffers: shared hit=1760
   -> Bitmap Index Scan on demo_gin_i_j_k_l_idx (actual time=68.174..68.174
 → rows=2059 loops=1)
         Index Cond: ((j \ge 17) \text{ AND } (j \le 21) \text{ AND } (l \ge 17) \text{ AND } (l \le 21))
         Buffers: shared hit=1003
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.056 ms
 Execution Time: 69.435 ms
Un index GiST permet aussi l'Index Scan et parfois l'Index Only Scan, au contraire d'un index GIN, qui
se limitera toujours à un Bitmap Scan. L'intérêt varie selon les requêtes :
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT j,l FROM demo_gist WHERE j=17 AND l=17 ;
                                QUERY PLAN
 Index Only Scan using demo_gist_i_j_k_l_idx on demo_gist (actual time=0.043..1.563
→ rows=83 loops=1)
   Index Cond: ((j = 17) \text{ AND } (l = 17))
   Heap Fetches: 0
   Buffers: shared hit=352
 Planning Time: 0.024 ms
 Execution Time: 1.572 ms
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT j,l FROM demo_gin WHERE j=17 AND l=17 ;
                                QUERY PLAN
 Bitmap Heap Scan on demo_gin (actual time=0.699..0.796 rows=83 loops=1)
   Recheck Cond: ((j = 17) \text{ AND } (l = 17))
   Heap Blocks: exact=83
   Buffers: shared hit=100
   -> Bitmap Index Scan on demo_gin_i_j_k_l_idx (actual time=0.683..0.683 rows=83
 → loops=1)
         Index Cond: ((j = 17) \text{ AND } (l = 17))
         Buffers: shared hit=17
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.069 ms
 Execution Time: 0.819 ms
```

Si la cardinalité est élevée (les données sont toutes différentes), l'index GIN perd son avantage en taille. En effet, si l'on remplace les données précédentes par un jeu de données toutes différentes :

```
TRUNCATE TABLE demo_gin;
TRUNCATE TABLE demo_gist;
INSERT INTO demo_gin
SELECT n, n AS i, 100e6+n AS j, 200e6+n AS k, 300e6+n AS l
FROM generate_series (1,1000000) n;
INSERT INTO demo_gist
SELECT n, n AS i, 100e6+n AS j, 200e6+n AS k, 300e6+n AS l
FROM generate_series (1,1000000) n;
```

la taille de l'index GIN passe à 218 Mo (plus que la table), alors que l'index GIST ne monte qu'à 85 Mo (et un index B-tree resterait à 30 Mo). Cela ne rend pas forcément l'index GIN moins efficace dans l'absolu, mais a un effet défavorable sur le cache.

### Index bloom:

Il existe encore un type d'index, rarement utilisé : l'index Bloom<sup>6</sup>, qui réclame l'installation de l'extension bloom (livrée avec PostgreSQL). Cet index est basé sur les filtres bloom<sup>7</sup>, de nature probabiliste. Les lignes retournées par l'index doivent être revérifiées dans la table, ce qui impose un *Recheck* systématique. L'index est rapide à générer, et encore plus petit que les index ci-dessus (15 Mo ici), mais cet index doit être complètement parcouru. Les performances sont donc moins bonnes qu'avec GiST ou GIN. Il est aussi limité aux entiers et chaînes de caractères, et à une recherche sur l'égalité (donc il est inadapaté aux critères BETWEEN et LIKE). Si ses performances suffisent, un index bloom peut être utile dans le cas où de nombreuses colonnes sont susceptibles d'être interrogées, car il évite de créer d'autres index B-tree ou GIN coûteux en place. Sur une table identique à celles ci-dessus, le plan est :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)

SELECT * FROM demo_bloom WHERE j=100000017 AND l=300000017 ;

QUERY PLAN

Bitmap Heap Scan on demo_bloom (actual time=3.732..3.753 rows=1 loops=1)
Recheck Cond: ((j = 100000017) AND (l = 300000017))
Rows Removed by Index Recheck: 27
Heap Blocks: exact=28
Buffers: shared hit=1961 read=28
-> Bitmap Index Scan on demo_bloom_i_j_k_l_idx (actual time=3.721..3.722 rows=28

cloops=1)
Index Cond: ((j = 100000017) AND (l = 300000017))
Buffers: shared hit=1933 read=28
Planning Time: 0.042 ms
Execution Time: 3.767 ms
```

# Choix du type d'index pour une indexation multicolonne :

Les exemples ci-dessus viennent d'une instance avec la configuration par défaut de PostgreSQL. Rappelons que les paramètres effective\_io\_concurrency, seq\_page\_cost et random\_page\_cost (liés aux disques) et effective\_cache\_size (lié à la mémoire), influent fortement sur le choix d'un parcours *Index Scan*, *Bitmap Scan* ou *Seq Scan* quand la requête balaie beaucoup de lignes. La répar-

<sup>&</sup>lt;sup>6</sup>https://docs.postgresql.fr/current/bloom.html

<sup>&</sup>lt;sup>7</sup>https://fr.wikipedia.org/wiki/Filtre\_de\_Bloom

# **DALIBO Formations**

tition physique des données joue aussi (sur l'efficacité du cache comme sur celle des *Index Scan* et *Bitmap Scan*.) Le type de données à indexer, leur longueur, l'opérateur... a aussi une importance.

Au final, pour une indexation multicolonne d'une table, il faudra tester les types d'index disponibles avec la charge, le paramétrage et les requêtes réelles, des données réelles, et arbitrer en tenant compte des tailles des index (donc de l'impact sur le cache), des durées de génération, des performances pures et des performances en pratique acceptables...

# 2.7 INDEX BRIN



BRIN: Block Range INdex

```
CREATE INDEX brin_demo_brin_idx ON brin_demo USING brin (age)
WITH (pages_per_range=16) ;
```

- Valeurs corrélées à leur emplacement physique
- Calcule des plages de valeur par groupe de blocs
  - index très compact
- Pour:
  - grosses volumétries
  - corrélation entre emplacement et valeur
  - retrier la table avec CLUSTER ?

Un index BRIN ne stocke pas les valeurs de la table, mais quelles plages de valeurs se rencontrent dans un ensemble de blocs. Cela réduit la taille de l'index et permet d'exclure un ensemble de blocs lors d'une recherche.

# 2.7.0.1 Exemple

Soit une table brin\_demo de 2 millions de personnes, triée par âge :

Un index BRIN va contenir une plage des valeurs pour chaque bloc. Dans notre exemple, l'index contiendra la valeur minimale et maximale de plusieurs blocs. La conséquence est que ce type d'index prend très peu de place et il peut facilement tenir en RAM (réduction des opérations des disques). Il est aussi plus rapide à construire et maintenir.

```
CREATE INDEX brin_demo_btree_idx ON brin_demo USING btree (age);
CREATE INDEX brin_demo_brin_idx ON brin_demo USING brin (age);
```

=# \di+ brin\_demo\*

List		

Schema	Name		Туре	•	Table	Persistence	'	•
	brin_demo_brin_idx	·  -			brin_demo   brin_demo	permanent	48 kB   13 MB	

On peut consulter le contenu de cet index<sup>8</sup>, et constater que chacune de ses entrées liste les valeurs de la par paquet de 128 blocs (cette valeur peut se changer):

```
CREATE EXTENSION IF NOT EXISTS pageinspect ;
```

FROM brin\_page\_items(get\_raw\_page('brin\_demo\_brin\_idx', 2), 'brin\_demo\_brin\_idx');

itemoffset	blknum	attnum	allnulls	hasnulls	placeholder	value
1		1	f	f	f	{1 2}
2	128	1	f	f	f	[ {2 3}
3	256	1	f	f	f	{3 4}
4	384	1	f	f	f	{4 6}
5	512	1	f	f	f	{6 7}
6	640	1	f	f	f	{7 8}
7	768	1	f	f	f	[ 8 10}
8	896	1	f	f	f	{10 11}
9	1024	1	f	f	f	{11 12}
•••						
66	8320	1	f	f	f	<b> </b> {85 86}
67	8448	1	f	f	f	<b> </b> {86 88}
68	8576	1	f	f	f	{88 89}
69	8704	1	f	f	f	{89 90}
70	8832	1	f	f	f	{90 90}

La colonne blknum indique le début de la tranche de blocs, et value la plage de valeurs rencontrées. Les personnes de 87 ans sont donc présentes uniquement entre les blocs 8448 à 8575, ce qui se vérifie :

Testons une requête avec uniquement ce petit index BRIN:

```
DROP INDEX brin_demo_btree_idx ;
```

```
EXPLAIN (ANALYZE,BUFFERS,COSTS OFF) SELECT count(*) FROM brin_demo WHERE age = 87 ;
```

```
QUERY PLAN
```

Aggregate (actual time-4 220 4 220 rews-1 leane-1)

```
Aggregate (actual time=4.838..4.839 rows=1 loops=1)
Buffers: shared hit=130
```

<sup>-&</sup>gt; Bitmap Heap Scan on brin\_demo (actual time=0.241..3.530 rows=22212 loops=1) Recheck Cond: (age = 87)

<sup>8</sup>https://docs.postgresql.fr/current/pageinspect.html#id-1.11.7.33.8

On constate que l'index n'est consulté que sur 2 blocs. Le nœud *Bitmap Index Scan* renvoie les 128 blocs contenant des valeurs entre 86 et 88, et ces blocs sont récupérés dans la table (*heap*). 6716 lignes sont ignorées, et 22 212 conservées. Le temps de 4,8 ms est bon.

Certes, un index B-tree, bien plus gros, aurait fait encore mieux dans ce cas précis, qui est modeste. Mais plus la table est énorme, plus une requête en ramène une proportion importante, plus les allerretours entre index et table sont pénalisants, et plus l'index BRIN devient compétitif, en plus de rester très petit.

Par contre, si la table se fragmente, même un peu, les lignes d'un même age se retrouvent réparties dans toute la table, et l'index BRIN devient bien moins efficace :

```
UPDATE brin_demo
       age=age+0
WHERE random()>0.99; -- environ 20000 lignes
VACUUM brin_demo ;
UPDATE brin_demo
      age=age+0
WHERE age > 80 AND random()>0.90; -- environ 22175 lignes
VACUUM ANALYZE brin_demo ;
FROM brin_page_items(get_raw_page('brin_demo_brin_idx', 2), 'brin_demo_brin_idx');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder | value
                           1 | f
1 | f
                                        | f
| f
| f
| f
                                                   | f
         1 |
                  0 |
                                                                 | {1 .. 81}
                128 |
                         1 | f
         2 |
                                       | f
                                                  | f
                                                                 [ {2 .. 81}
          3 |
                256
                          1 | f
                                                   l f
                                                                 | {3 .. 81}
                                                                 | {4 .. 81}
                384
                                        | f
         45 l
                5632
                           1 | f
                                                    | f
                                                                 | {58 .. 87}
                           1 | f
         46
                5760
                                        | f
                                                    | f
                                                                 | {59 .. 87}
                           1 | f
                                         | f
                                                                 | {60 .. 87}
         47
                5888
                                                    | f
                           1 | f
         56 l
                7040
                                        | f
                                                    | f
                                                                 | {72 .. 89}
                           1 | f
                                        | f
         57
                7168 l
                                                    | f
                                                                 | {73 .. 89}
                                                   | f
               7296
                           1 | f
                                        | f
                                                                 | {75 .. 89}
         58
         67
                8448
                           1 | f
                                        | f
                                                                 | {86 .. 88}
                                        | f
                                                                 | {88 .. 89}
                8576 |
                          1 | f
                                                   | f
         69 |
                8704 |
                                        | f
                                                   | f
                           1 | f
                                                                 | {89 .. 90}
                                        | f
                                                   | f
                           1 | f
         70 |
                8832 |
                                                                 | {1 .. 90}
```

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT count(*) FROM brin_demo WHERE age = 87;
                               QUERY PLAN
Aggregate (actual time=71.053..71.055 rows=1 loops=1)
   Buffers: shared hit=3062
   -> Bitmap Heap Scan on brin_demo (actual time=2.451..69.851 rows=22303 loops=1)
         Recheck Cond: (age = 87)
         Rows Removed by Index Recheck: 664141
         Heap Blocks: lossy=3060
         Buffers: shared hit=3062
         -> Bitmap Index Scan on brin_demo_brin_idx (actual time=0.084..0.084
   rows=30600 loops=1)
               Index Cond: (age = 87)
               Buffers: shared hit=2
Planning:
  Buffers: shared hit=1
Planning Time: 0.069 ms
 Execution Time: 71.102 ms
```

3060 blocs et 686 444 lignes, la plupart inutiles, ont été lus dans la table (en gros, un tiers de celles-ci). Cela ne devient plus très intéressant par rapport à un parcours complet de la table.

Pour rendre son intérêt à l'index, il faut reconstruire la table avec les données dans le bon ordre avec la commande CLUSTER 9. Hélas, c'est une opération au moins aussi lourde et bloquante qu'un VACUUM FULL. De plus, le tri de la table ne peut se faire par l'index BRIN, et il faut recréer un index B-tree au moins le temps de l'opération.

```
CREATE INDEX brin_demo_btree_idx ON brin_demo USING btree (age);
CLUSTER brin_demo USING brin_demo_btree_idx;
SELECT *
FROM brin_page_items(get_raw_page('brin_demo_brin_idx', 2),'brin_demo_brin_idx');
itemoffset | blknum | attnum | allnulls | hasnulls | placeholder | value
        1 |
              0 | 1 | f
                                   | f
                                             | f
                                                          | {1 .. 2}
              128 |
        2 |
                        1 | f
                                    | f
                                              | f
                                                           | {2 .. 3}
                        1 | f
                                    | f
                                              | f
                                                         | {88 .. 89}
        68 | 8576 |
                                   69 |
             8704
                        1 | f
                                                          | {89 .. 90}
        70 |
              8832 |
                        1 | f
                                                          | {90 .. 90}
```

On revient alors à la situation de départ.

# 2.7.0.2 Utilité d'un index BRIN

Pour qu'un index BRIN soit utile, il faut donc :

- que les données soit naturellement triées dans la table, et le restent (série temporelle, décisionnel avec imports réguliers...);
- que la table soit à « insertion seule » pour conserver l'ordre physique ;

<sup>&</sup>lt;sup>9</sup>https://docs.postgresql.fr/current/sql-cluster.html

- ou que l'on ait la disponibilité nécessaire pour reconstruire régulièrement la table avec CLUSTER et éviter que les performances se dégradent au fil du temps.

Sous ces conditions, les BRIN sont indiqués si l'on a des problèmes de volumétrie, ou de temps d'écritures dus aux index B-tree, ou pour éviter de partitionner une grosse table dont les requêtes ramènent une grande proportion.

#### 2.7.0.3 Index BRIN sur clé composée

Prenons un autre exemple avec plusieurs colonnes et un type text:

```
CREATE TABLE test (id serial PRIMARY KEY, val text);
INSERT INTO test (val) SELECT md5(i::text) FROM generate_series(1, 10000000) i;
```

La colonne id sera corrélée (c'est une séquence), la colonne md5 ne sera pas du tout corrélée. L'index BRIN porte sur les deux colonnes :

```
CREATE INDEX test_brin_idx ON test USING brin (id,val);
```

Pour une table de 651 Mo, l'index ne fait ici que 104 ko.

Pour voir son contenu:

```
SELECT itemoffset, blknum, attnum,value
FROM brin_page_items(get_raw_page('test_brin_idx', 2),'test_brin_idx')
LIMIT 4;
```

La colonne attnum correspond au numéro d'attribut du champ dans la table. L'id est bien corrélé aux numéros de bloc, contrairement à la colonne val. Ce que nous confirme bien la vue pg\_stats:

```
SELECT tablename, attname, correlation
FROM pg_stats WHERE tablename='test' ORDER BY attname;
```

		correlation
test	id val	1   0.00528745

Si l'on teste la requête suivante, on s'aperçoit que PostgreSQL effectue un parcours complet (*Seq Scan*) de façon parallélisée ou non, et n'utilise donc pas l'index BRIN. Pour comprendre pourquoi, essayons de l'y forcer :

```
SET enable_seqscan TO off ;
SET max_parallel_workers_per_gather TO 0;
EXPLAIN (BUFFERS, ANALYZE) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
   AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                               QUERY PLAN
______
Bitmap Heap Scan on test (cost=721.46..234055.46 rows=2642373 width=37) (actual

    time=2.558..1622.646 rows=2668675 loops=1)

  Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
            AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))</pre>
  Rows Removed by Index Recheck: 7331325
  Heap Blocks: lossy=83334
  Buffers: shared hit=83349
  -> Bitmap Index Scan on test_brin_idx (cost=0.00..60.86 rows=10000000 width=0)
Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
               AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
        Buffers: shared hit=15
Planning:
  Buffers: shared hit=1
Planning Time: 0.079 ms
Execution Time: 1703.018 ms
```

83 334 blocs sont lus (651 Mo) soit l'intégralité de la table! Il est donc logique que PostgreSQL préfère d'entrée un Seq Scan (parcours complet).

Pour que l'index BRIN soit utile pour ce critère, il faut là encore trier la table avec une commande CLUSTER, ce qui nécessite un index B-tree:

Notons au passage que cet index B-tree est presque aussi gros que notre table!

Après la commande CLUSTER, notre table est bien corrélée avec val (mais plus avec id):

```
CLUSTER test USING test_btree_idx;
ANALYZE test;
SELECT tablename, attname, correlation
FROM pg_stats WHERE tablename='test' ORDER BY attname;

tablename | attname | correlation
-----test | id | -0.0023584804
test | val | 1
```

La requête après le cluster utilise alors l'index BRIN :

```
SET enable_seqscan TO on ;
EXPLAIN (BUFFERS, ANALYZE) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
   AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                             QUERY PLAN
______
                                -----
Bitmap Heap Scan on test (cost=712.28..124076.96 rows=2666839 width=37) (actual

    time=1.460..540.250 rows=2668675 loops=1)

  Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
            AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
  Rows Removed by Index Recheck: 19325
  Heap Blocks: lossy=22400
  Buffers: shared hit=22409
  -> Bitmap Index Scan on test_brin_idx (cost=0.00..45.57 rows=2668712 width=0)
Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
        Buffers: shared hit=9
Planning:
  Buffers: shared hit=18 read=2
  I/O Timings: read=0.284
Planning Time: 0.468 ms
 Execution Time: 630.124 ms
```

22 400 blocs sont lus dans la table, soit 175 Mo. Dans la table triée, l'index BRIN devient intéressant. Cette remarque vaut aussi si PostgreSQL préfère un *Index Scan* au plan précédent (notamment si random\_page\_cost vaut moins du 4 par défaut).

On supprime notre index BRIN et on garde l'index B-tree :

La durée est ici similaire, mais le nombre de blocs lus est double, ce qui est une conséquence de la taille de l'index.

#### 2.7.0.4 Création

À partir de PostgreSQL 17, la création d'un index BRIN peut être parallélisée. Comme cet index est intéressant surtout sur de grosses tables, la durée est sérieusement réduite. Le nombre de processus parallèles est réglé par max\_parallel\_maintenance\_workers. Dans les versions précédentes, un seul processeur est malheureusement utilisable lors de la création.

#### 2.7.0.5 Maintenance et consultation

Les lignes ajoutées à la table après la création de l'index ne sont pas forcément intégrées au résumé tout de suite. Il faut faire attention à ce que le VACUUM passe assez souvent<sup>10</sup>. Cela dit, la maintenance d'un BRIN lors d'écritures est plus légères qu'un gros B-tree.

Pour modifier la granularité de l'index BRIN, il faut utiliser le paramètre pages\_per\_range à la création :

```
CREATE INDEX brin_demo_brin_idx ON brin_demo
USING brin (age) WITH (pages_per_range=16);
```

Les calculs de plage de valeur se feront alors par paquets de 16 blocs, ce qui est plus fin tout en conservant une volumétrie dérisoire. Sur la table <a href="brin\_demo">brin\_demo</a>, l'index ne fait toujours que 56 ko. Par contre, la requête d'exemple parcourra un peu moins de blocs inutiles. La plage est à ajuster en fonction de la finesse des données et de leur répartition.

Pour consulter la répartition des valeurs comme nous l'avons fait plus haut, il faut utiliser pageins-pect<sup>11</sup>. Pour une table brin\_demo dix fois plus grosse, et des plages de 16 blocs :

On retrouve le paramètre de plages de 16 blocs, et la range map commence au bloc 5. Par ailleurs, pg\_class.relpages indique 20 blocs. Nous avons donc les bornes des pages de l'index à consulter:

		. '		•		placeholder   +	
18			1		f		{1 1}
18	274	16	1	f	f	f	{1 1}
18	275	32	1	f	f	f	{1 1}
18	276	48	1	f	f	f	{1 1}

 $<sup>^{10}</sup> https://docs.postgresql.fr/current/brin-intro.html \#BRIN-OPERATION$ 

<sup>&</sup>lt;sup>11</sup>https://docs.postgresql.fr/current/pageinspect.html#id-1.11.7.33.8

•••						
19	224	88432	1   f	f	f	{90 90}
19	225	88448	1   f	f	f	{90 90}
19	226	88464	1   f	f	f	{90 90}
19	227	88480	1   f	f	f	{90 90}
(5531 li	gnes)	•	•		•	•

## 2.7.0.6 Plus d'informations sur les BRIN

- Conférence d'Adrien Nayrat au PGDay France 2016 (Lille, 31 mai 2016) : vidéo<sup>12</sup>, texte<sup>13</sup>.
- Documentation officielle<sup>14</sup>.

<sup>12</sup>https://youtu.be/g3tSRyeN1TY 13https://kb.dalibo.com/conferences/index\_brin/index\_brin\_pgday

<sup>14</sup>https://docs.postgresql.fr/current/brin.html

#### 2.8 INDEX HASH



- Basés sur un hash
- Tous types de données, quelle que soit la taille
- Ne gèrent que =
  - donc ni < , > , != ...
- Mais plus compacts

Les index hash contiennent des hachages de tout type de données. Cela leur permet d'être relativement petits, même pour des données de gros volume, et d'être une alternative aux index B-tree qui ne peuvent pas indexer des objets de plus de 2,7 ko.

Une conséquence de ce principe est qu'il est impossible de parcourir des plages de valeurs, seule l'égalité **exacte** à un critère peut être recherchée. Cet exemple utilise la base du projet Gutenberg<sup>15</sup>:

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)

SELECT * FROM textes
-- attention au nombre exact d'espaces

WHERE contenu = ' Maître corbeau, sur un arbre perché';

QUERY PLAN

Index Scan using textes_contenu_hash_idx on textes (actual time=0.049..0.050 rows=1

loops=1)
Index Cond: (contenu = ' Maître corbeau, sur un arbre perché'::text)
Buffers: shared hit=3
Planning Time: 0.073 ms
Execution Time: 0.072 ms
```

Les index hash restent plus longs à créer que des index B-tree. La création n'est pas parallélisée. Ils ne sont plus petits qu'eux que si les champs indexés sont gros. Par exemple, dans la même table texte, de 3 Go, le nom de l'œuvre (livre) est court, mais une ligne de texte (contenu) peut faire 3 ko (ici, on a dû purger les trois lignes trop longues pour être indexées par un B-tree):

```
SELECT pg_size_pretty(pg_relation_size(indexname::regclass)) AS taille, indexdef FROM pg_indexes
WHERE indexname like 'texte%';

taille | indexdef | inde
```

<sup>&</sup>lt;sup>15</sup>https://dali.bo/tp\_gutenberg

#### **DALIBO Formations**

1033 MB | CREATE INDEX textes\_livre\_hash\_idx ON public.textes USING hash (livre)
155 MB | CREATE INDEX textes\_livre\_idx ON public.textes USING btree (livre

→ varchar\_pattern\_ops)

On réservera donc les index hash à l'indexation de grands champs, éventuellement binaires, notamment dans des requêtes recherchant la présence d'un objet. Ils peuvent vous éviter de gérer vous-même un hachage du champ.

Les index hash n'étaient pas journalisés avant la version 10, leur utilisation y était donc une mauvaise idée (corruption à chaque arrêt brutal, pas de réplication...). Ils étaient aussi peu performants par rapport à des index B-tree. Ceci explique le peu d'utilisation de ce type d'index jusqu'à maintenant.

#### **2.9 OUTILS**



- Pour identifier des requêtes
  Pour identifier des prédicats et des requêtes liées
  Pour valider un index

Différents outils permettent d'aider le développeur ou le DBA à identifier plus facilement les index à créer. On peut classer ceux-ci en trois groupes, selon l'étape de la méthodologie à laquelle ils s'appliquent.

Tous les outils suivants sont disponibles dans les paquets diffusés par le PGDG sur yum.postgresql.org16 ou apt.postgresql.org<sup>17</sup>.

### 2.9.1 Identifier les requêtes



- pgBadgerpg\_stat\_statementsPoWA

Pour identifier les requêtes les plus lentes, et donc potentiellement nécessitant une réécriture ou un nouvel index, pgBadger<sup>18</sup> permet d'analyser les logs une fois ceux-ci configurés pour tracer toutes les requêtes. Des exemples figurent dans notre formation DBA1<sup>19</sup>.

Pour une vision cumulative, voire temps réel, de ces requêtes, l'extension pg\_stat\_statements, fournie avec les « contrib » de PostgreSQL, permet de garder trace des N requêtes les plus fréquemment exécutées, et calcule le temps d'exécution total de chacune d'entre elles, ainsi que les accès au cache de PostgreSQL ou au système de fichiers. Son utilisation est détaillée dans notre module X2<sup>20</sup>.

Le projet PoWA<sup>21</sup> exploite ces statistiques en les historisant, et en fournissant une interface web permettant de les exploiter.

<sup>&</sup>lt;sup>16</sup>https://yum.postgresql.org

<sup>&</sup>lt;sup>17</sup>https://apt.postgresql.org

<sup>&</sup>lt;sup>18</sup>https://pgbadger.darold.net

<sup>&</sup>lt;sup>19</sup>https://dali.bo/h1\_html#pgbadger

<sup>&</sup>lt;sup>20</sup>https://dali.bo/x2\_html#pg\_stat\_statements

<sup>&</sup>lt;sup>21</sup>https://powa.readthedocs.io/en/latest/

#### 2.9.2 Identifier les prédicats et des requêtes liées



- Extension pg\_qualstats
   avec PoWa

Pour identifier les prédicats (clause WHERE ou condition de jointure à identifier en priorité), l'extension pg\_qualstats<sup>22</sup> permet de pousser l'analyse offerte par pg\_stat\_statements au niveau du prédicat lui-même. Ainsi, on peut détecter les requêtes filtrant sur les mêmes colonnes, ce qui peut aider notamment à déterminer des index multicolonnes ou des index partiels.

De même que pg\_stat\_statements, cette extension peut être historisée et exploitée par le biais du projet PoWA.

## 2.9.3 Extension HypoPG



- Extension PostgreSQL
   Création d'index hypothétiques pour tester leur intérêt
   avant de les créer pour de vrai

  - Limitations : surtout B-Tree, statistiques

Cette extension est disponible sur GitHub<sup>23</sup> et dans les paquets du PGDG. Il existe trois fonctions principales et une vue :

- hypopg\_create\_index()pour créer un index hypothétique;
- hypopg\_drop\_index()
   pour supprimer un index hypothétique particulier ou hypopg\_reset() pour tous les supprimer;
- hypopg\_list\_indexes pour les lister.

Un index hypothétique n'existe que dans la session, ni en mémoire ni sur le disque, mais le planificateur le prendra en compte dans un EXPLAIN simple (évidemment pas un EXPLAIN ANALYZE ). En quittant la session, tous les index hypothétiques restants et créés sur cette session sont supprimés.

<sup>&</sup>lt;sup>22</sup>https://github.com/powa-team/pg\_qualstats

<sup>&</sup>lt;sup>23</sup>https://github.com/HypoPG/hypopg

L'exemple suivant est basé sur la base dont le script peut être téléchargé sur https://dali.bo/tp\_employes\_services.

```
CREATE EXTENSION hypopg;
EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';
                                QUERY PLAN
Gather (cost=1000.00..8263.14 rows=1 width=41)
  Workers Planned: 2
   -> Parallel Seq Scan on employes_big (cost=0.00..7263.04 rows=1 width=41)
         Filter: ((prenom)::text = 'Gaston'::text)
SELECT * FROM hypopg_create_index('CREATE INDEX ON employes_big(prenom)');
indexrelid |
                       indexname
     24591 | <24591>btree_employes_big_prenom
EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';
                         QUERY PLAN
Index Scan using <24591>btree_employes_big_prenom on employes_big
                               (cost=0.05..4.07 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
SELECT * FROM hypopg_list_indexes;
                       indexname | nspname | relname | amname
indexrelid |
     24591 | <24591>btree_employes_big_prenom | public | employes_big | btree
SELECT * FROM hypopg_reset();
hypopg_reset
(1 row)
CREATE INDEX ON employes_big(prenom);
EXPLAIN SELECT * FROM employes_big WHERE prenom='Gaston';
                   QUERY PLAN
Index Scan using employes_big_prenom_idx on employes_big
                       (cost=0.42..4.44 rows=1 width=41)
   Index Cond: ((prenom)::text = 'Gaston'::text)
```

Le cas idéal d'utilisation est l'index B-Tree sur une colonne. Un index fonctionnel est possible, mais, faute de statistiques disponibles avant la création réelle de l'index, les estimations peuvent être fausses. Les autres types d'index sont moins bien ou non supportées.

110

#### 2.9.4 Étude des index à créer



- PoWA peut utiliser HypoPG

Le projet PoWA propose une fonctionnalité, encore rudimentaire, de suggestion d'index à créer, en se basant sur HypoPG, pour répondre à la question « Quel serait le plan d'exécution de ma requête si cet index existait ? ».

L'intégration d'HypoPG dans PoWA permet là aussi une souplesse d'utilisation, en présentant les plans espérés avec ou sans les index suggérés.

Ensuite, en ouvrant l'interface de PoWA, on peut étudier les différentes requêtes, et les suggestions d'index réalisées par l'outil. À partir de ces suggestions, on peut créer les nouveaux index, et enfin relancer le bench pour constater les améliorations de performances.

# **2.10 QUIZ**



# 2.11 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/j5\_solutions.

Tous les TP se basent sur la configuration par défaut de PostgreSQL, sauf précision contraire.

#### 2.11.1 Indexation de motifs avec les varchar\_patterns et pg\_trgm



**But**: Indexer des motifs à l'aide de l'opérateur varchar\_pattern\_ops et de l'extension pg\_trgm

Ces exercices nécessitent une base contenant une quantité de données importante.

On utilisera donc le contenu de livres issus du projet Gutenberg. La base **gutenberg** est disponible en deux versions : complète sur https://dali.bo/tp\_gutenberg (dump de 0,5 Go, table de 21 millions de lignes dans 3 Go) ou https://dali.bo/tp\_gutenberg10 pour un extrait d'un dizième. Le dump peut se restaurer par exemple dans une nouvelle base, et contient juste une table nommée textes.

```
curl -kL https://dali.bo/tp_gutenberg -o /tmp/gutenberg.dmp
createdb gutenberg
pg_restore -d gutenberg /tmp/gutenberg.dmp
# le message sur le schéma public exitant est normale
rm -- /tmp/gutenberg.dmp
```

Pour obtenir des plans plus lisibles, on désactive JIT et parallélisme :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
```

Créer un index simple sur la colonne contenu de la table.

Rechercher un enregistrement commençant par « comme disent » : l'index est-il utilisé ?

Créer un index utilisant la classe text\_pattern\_ops . Refaire le test.

On veut chercher les lignes finissant par « Et vivre ». Indexer reverse (contenu) et trouver les lignes.

Installer l'extension pg\_trgm, puis créer un index GIN spécialisé de recherche dans les chaînes. Rechercher toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis in-

#### sensible.

Si vous avez des connaissances sur les expression rationnelles, utilisez aussi ces trigrammes pour des recherches plus avancées. Les opérateurs sont :

opérateur	fonction
~	correspondance sensible à la casse
~*	correspondance insensible à la casse
!~	non-correspondance sensible à la casse
!~*	non-correspondance insensible à la casse

Rechercher toutes les lignes contenant « Fantine » OU « Valjean » : on peut utiliser une expression rationnelle.

Rechercher toutes les lignes mentionnant à la fois « Fantine » ET « Valjean ». Une formulation d'expression rationnelle simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

#### 2.11.2 Index GIN comme bitmap



But: Comparer l'utilisation des index B-tree et GIN

Ce TP utilise la base **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin**:

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, magasin et facturation. Penser au search\_path.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

Créer deux index sur lignes\_commandes(quantite) :

- un de type B-tree
- et un GIN.

puis deux autres sur lignes\_commandes(fournisseur\_id).

Comparer leur taille.

Comparer l'utilisation des deux types d'index avec EXPLAIN avec des requêtes sur fournisseur\_id = 1014, puis sur quantite = 4.

#### 2.11.3 Index GIN et critères multicolonnes



**But** : Comparer l'utilisation des index B-tree et GIN sur des critères multicolonnes

Créer une table avec 4 colonnes de 50 valeurs :

```
CREATE UNLOGGED TABLE ijkl
AS SELECT i,j,k,l
FROM generate_series(1,50) i
CROSS JOIN generate_series (1,50) j
CROSS JOIN generate_series(1,50) k
CROSS JOIN generate_series (1,50) l;
```

Créer un index B-tree et un index GIN sur ces 4 colonnes.

Comparer l'utilisation pour des requêtes portant sur i , j , k & l , puis i & k , puis j & l .

#### 2.11.4 HypoPG



But : Créer un index hypothétique avec HypoPG

Pour la clarté des plans, désactiver le JIT.

```
Créer la table suivante, où la clé i est très déséquilibrée :

CREATE UNLOGGED TABLE log10 AS

SELECT i,j, i+10 AS k, now() AS d, lpad(' ',300,' ') AS filler

FROM generate_series (0,7) i,
```

LATERAL (SELECT \* FROM generate\_series(1, power(10,i)::bigint ) j ) jj ;

(Ne pas oublier VACUUM ANALYZE .)

On se demande si créer un index sur i, (i,j) ou (j,i) serait utile pour les deux requêtes suivantes :

```
SELECT i, min(j), max(j) FROM log10 GROUP BY i ; SELECT max(j) FROM log10 WHERE i = 6 ;
```

- Installer l'extension hypopg (les paquets dépendent de votre version de PostgreSQL et de la distribution, par exemple hypopg\_17 ou postgresql-17-hypopg).
- Créer des index hypothétiques (y compris un partiel) et choisir un seul index.

Comparer le plan de la deuxième requête avant et après la création réelle de l'index.

Créer un index fonctionnel hypothétique pour faciliter la requête suivante :

```
SELECT k FROM log10 WHERE mod(j,99) = 55;
```

Quel que soit le résultat, le créer quand même et voir s'il est utilisé.

# 2.12 TRAVAUX PRATIQUES (SOLUTIONS)

# 2.12.1 Indexation de motifs avec les varchar\_patterns et pg\_trgm

Créer un index simple sur la colonne contenu de la table.

```
CREATE INDEX ON textes(contenu);
```

Il y aura une erreur si la base **gutenberg** est dans sa version complète, un livre de Marcel Proust dépasse la taille indexable maximale :

```
ERROR: index row size 2968 exceeds maximum 2712 for index "textes_contenu_idx" ASTUCE: Values larger than 1/3 of a buffer page cannot be indexed. Consider a function index of an MD5 hash of the value, or use full text indexing.
```

Pour l'exercice, on supprime ce livre avant d'indexer la colonne :

```
DELETE FROM textes where livre = 'Les Demi-Vierges, Prévost, Marcel';
CREATE INDEX ON textes(contenu);
```

Rechercher un enregistrement commençant par « comme disent » : l'index est-il utilisé ?

Le plan exact peut dépendre de la version de PostgreSQL, du paramétrage exact, d'éventuelles modifications à la table. Dans beaucoup de cas, on obtiendra :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
VACUUM ANALYZE textes;
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
                             QUERY PLAN
 Seq Scan on textes (cost=0.00..669657.38 rows=1668 width=124)
            (actual time=305.848..6275.845 rows=47 loops=1)
   Filter: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Filter: 20945503
 Planning Time: 1.033 ms
 Execution Time: 6275.957 ms
C'est un Seq Scan : l'index n'est pas utilisé!
Dans d'autres cas, on aura ceci (avec PostgreSQL 12 et la version complète de la base ici) :
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
                             QUERY PLAN
 Index Scan using textes_contenu_idx on textes (...)
   Index Cond: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Index Recheck: 110
   Buffers: shared hit=28 read=49279
   I/O Timings: read=311238.192
 Planning Time: 0.352 ms
```

Execution Time: 313481.602 ms

C'est un Index Scan mais il ne faut pas crier victoire : l'index est parcouru entièrement (50 000 blocs !). Il ne sert qu'à lire toutes les valeurs de contenu en lisant moins de blocs que par un Seq Scan de la table. Le choix de PostgreSQL entre lire cet index et lire la table dépend notamment du paramétrage et des tailles respectives.

Le problème est que l'index sur contenu utilise la collation C et non la collation par défaut de la base, généralement en\_US.UTF-8 ou fr\_FR.UTF-8. Pour contourner cette limitation, PostgreSQL fournit deux classes d'opérateurs : varchar\_pattern\_ops pour varchar et text\_pattern\_ops pour text.

Créer un index utilisant la classe text\_pattern\_ops . Refaire le test.

On constate que comme l'ordre choisi est l'ordre ASCII, l'optimiseur sait qu'après « comme disent », c'est « comme disenu » qui apparaît dans l'index.

Noter que Index Cond contient le filtre utilisé pour l'index (réexprimé sous forme d'inégalités en collation C) et Filter un filtrage des résultats de l'index.

On veut chercher les lignes finissant par « Et vivre ». Indexer reverse (contenu) et trouver les lignes.

Cette recherche n'est possible avec un index B-Tree qu'en utilisant un index sur fonction :

```
CREATE INDEX ON textes(reverse(contenu) text_pattern_ops);

Il faut ensuite utiliser ce reverse systématiquement dans les requêtes:

EXPLAIN (ANALYZE)

SELECT * FROM textes WHERE reverse(contenu) LIKE reverse('%Et vivre');

QUERY PLAN

Index Scan using textes_reverse_idx on textes
```

```
(cost=0.56..377770.76 rows=104728 width=123)
                           (actual time=0.083..0.098 rows=2 loops=1)
   Index Cond: ((reverse(contenu) ~>=~ 'erviv tE'::text)
            AND (reverse(contenu) ~<~ 'erviv tF'::text))</pre>
   Filter: (reverse(contenu) ~~ 'erviv tE%'::text)
 Planning Time: 1.903 ms
 Execution Time: 0.421 ms
On constate que le résultat de reverse (contenu) a été directement utilisé par l'optimiseur. La re-
quête est donc très rapide. On peut utiliser une méthode similaire pour la recherche insensible à la
casse, en utiliser lower() ou upper().
Toutefois, ces méthodes ne permettent de filtrer qu'au début ou à la fin de la chaîne, ne permettent
qu'une recherche sensible ou insensible à la casse, mais pas les deux simultanément, et imposent aux
développeurs de préciser reverse, lower, etc. partout.
  Installer l'extension pg_trgm, puis créer un index GIN spécialisé de recherche dans les chaînes.
  Rechercher toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis in-
  sensible.
Pour installer l'extension pg_trgm :
CREATE EXTENSION pg_trgm;
Pour créer un index GIN sur la colonne contenu :
CREATE INDEX idx_textes_trgm ON textes USING gin (contenu gin_trgm_ops);
Recherche des lignes contenant « Valjean » de façon sensible à la casse :
EXPLAIN (ANALYZE)
SELECT * FROM textes WHERE contenu LIKE '%Valjean%';
                                   QUERY PLAN
 Bitmap Heap Scan on textes (cost=77.01..6479.68 rows=1679 width=123)
                       (actual time=11.004..14.769 rows=1213 loops=1)
   Recheck Cond: (contenu ~~ '%Valjean%'::text)
   Rows Removed by Index Recheck: 1
   Heap Blocks: exact=353
   -> Bitmap Index Scan on idx_textes_trgm
                                 (cost=0.00..76.59 rows=1679 width=0)
                        (actual time=10.797..10.797 rows=1214 loops=1)
         Index Cond: (contenu ~~ '%Valjean%'::text)
 Planning Time: 0.815 ms
 Execution Time: 15.122 ms
Puis insensible à la casse :
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ILIKE '%Valjean%';
```

QUERY PLAN

Bitmap Heap Scan on textes (cost=77.01..6479.68 rows=1679 width=123)

On constate que l'index a été nettement plus long à créer, et que la recherche est plus lente. La contrepartie est évidemment que les trigrammes sont infiniment plus souples. On constate aussi que le LIKE a dû encore filtrer 1 enregistrement après le parcours de l'index : en effet l'index trigramme est insensible à la casse, il ramène donc trop d'enregistrements, et une ligne avec « VALJEAN » a dû être filtrée.

Rechercher toutes les lignes contenant « Fantine » OU « Valjean » : on peut utiliser une expression rationnelle.

Rechercher toutes les lignes mentionnant à la fois « Fantine » ET « Valjean ». Une formulation d'expression rationnelle simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

#### 2.12.2 Index GIN comme bitmap

Ce TP utilise la base **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin**:

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
Les données sont dans deux schémas, magasin et facturation. Penser au search_path.
Pour ce TP, figer les paramètres suivants :
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
  Créer deux index sur lignes_commandes (quantite) :
     - un de type B-tree
     - et un GIN.

    puis deux autres sur lignes_commandes(fournisseur_id).

SET search_path TO magasin ;
Il est nécessaire d'utiliser l'extension btree_gin afin d'indexer des types scalaires (int ...) avec un
GIN, sinon vous aurez cette erreur:
CREATE INDEX ON lignes_commandes USING gin (quantite);
ERROR: data type bigint has no default operator class for access method "gin"
HINT: You must specify an operator class for the index
    or define a default operator class for the data type.
CREATE EXTENSION btree_gin;
CREATE INDEX lignes_commandes_quantite_gin ON lignes_commandes
    USING gin (quantite);
CREATE INDEX lignes_commandes_quantite_btree ON lignes_commandes
    (quantite) ; -- implicitement B-tree
CREATE INDEX lignes_commandes_fournisseur_id_gin ON lignes_commandes
    USING gin (fournisseur_id) ;
CREATE INDEX lignes_commandes_fournisseur_id_btree ON lignes_commandes
    (fournisseur_id) ; -- implicitement B-tree
```

Comparer leur taille.

Ces index sont compressés, ainsi la clé n'est indexée qu'une fois. Le gain est donc intéressant dès lors que la table comprend des valeurs identiques.

(Ces valeurs ont été obtenues avec PostgreSQL 13. Une version antérieure affichera des index B-tree nettement plus gros, car le stockage des valeurs dupliquées y est moins efficace. Les index GIN seront donc d'autant plus intéressants.)

Noter qu'il y a peu de valeurs différentes de quantite, et beaucoup plus de fournisseur\_id, ce qui explique les différences de tailles d'index:

Les index GIN sont donc plus compacts. Sont-il plus efficace à l'utilisation?

```
Comparer l'utilisation des deux types d'index avec EXPLAIN avec des requêtes sur fournisseur_id = 1014, puis sur quantite = 4.
```

Pour récupérer une valeur précise avec peu de valeurs, PostgreSQL préfère les index B-Tree:

```
EXPLAIN (ANALYZE, BUFFERS)
 SELECT * FROM lignes_commandes
 WHERE fournisseur_id = 1014 ;
                                OUERY PLAN
Bitmap Heap Scan on lignes_commandes (cost=20.97..5469.07 rows=1618 width=74)
                                       (actual time=0.320..9.132 rows=1610 loops=1)
   Recheck Cond: (fournisseur_id = 1014)
  Heap Blocks: exact=1585
  Buffers: shared hit=407 read=1185
  -> Bitmap Index Scan on lignes_commandes_fournisseur_id_btree
                                       (cost=0.00..20.56 rows=1618 width=0)
                                       (actual time=0.152..0.152 rows=1610 loops=1)
         Index Cond: (fournisseur_id = 1014)
         Buffers: shared hit=3 read=4
Planning:
   Buffers: shared hit=146 read=9
Planning Time: 2.269 ms
Execution Time: 9.250 ms
```

À l'inverse, la recherche sur les quantités ramène plus de valeurs, et, là, PostgreSQL estime que le GIN est plus favorable :

```
EXPLAIN (ANALYZE, BUFFERS)
 SELECT * FROM lignes_commandes WHERE quantite = 4;
                                 QUERY PLAN
Bitmap Heap Scan on lignes_commandes (cost=2844.76..48899.60 rows=308227 width=74)
                                   (actual time=37.904..293.243 rows=313674 loops=1)
  Recheck Cond: (quantite = 4)
  Heap Blocks: exact=42194
  Buffers: shared hit=532 read=41712 written=476
   -> Bitmap Index Scan on lignes_commandes_quantite_gin
                                       (cost=0.00..2767.70 rows=308227 width=0)
                                     (actual time=31.164..31.165 rows=313674 loops=1)
         Index Cond: (quantite = 4)
         Buffers: shared hit=50
Planning:
  Buffers: shared hit=13
Planning Time: 0.198 ms
Execution Time: 305.030 ms
```

En cas de suppression de l'index GIN, l'index B-Tree reste utilisable. Il sera plus long à lire. Cependant, en fonction de sa taille, de celle de la table, de la valeur de random\_page\_cost et seq\_page\_cost, PostgreSQL peut décider de ne pas l'utiliser.

#### 2.12.3 Index GIN et critères multicolonnes

```
Créer une table avec 4 colonnes de 50 valeurs:

CREATE UNLOGGED TABLE ijkl

AS SELECT i,j,k,l

FROM generate_series(1,50) i

CROSS JOIN generate_series (1,50) j

CROSS JOIN generate_series(1,50) k

CROSS JOIN generate_series (1,50) l;
```

Les 264 Mo de cette table contiennent 6,25 millions de lignes.

Comme après tout import, ne pas oublier d'exécuter un VACUUM :

VACUUM ijkl;

Créer un index B-tree et un index GIN sur ces 4 colonnes.

Là encore, btree\_gin est obligatoire pour indexer un type scalaire, et on constate que l'index GIN est plus compact :

```
CREATE INDEX ijkl_btree on ijkl (i,j,k,l) ;
CREATE EXTENSION btree_gin ;
CREATE INDEX ijkl_gin on ijkl USING gin (i,j,k,l) ;
```

```
Comparer l'utilisation pour des requêtes portant sur i , j , k & l , puis i & k , puis j & l .
```

Le premier critère porte idéalement sur toutes les colonnes de l'index B-tree : celui-ci est très efficace et réclame peu d'accès. Toutes les colonnes retournées font partie de l'index : on profite donc en plus d'un Index Only Scan :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)

SELECT * FROM ijkl WHERE i=10 AND j=20 AND k=30 AND l=40;

QUERY PLAN

Index Only Scan using ijkl_btree on ijkl (... rows=1 loops=1)

Index Cond: ((i = 10) AND (j = 20) AND (k = 30) AND (l = 40))

Heap Fetches: 0

Buffers: shared hit=1 read=3

Planning Time: 0.330 ms

Execution Time: 0.400 ms
```

Tant que la colonne i est présente dans le critère, l'index B-tree reste avantageux même s'il doit balayer plus de blocs (582 ici !) :

```
# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM ijkl WHERE i=10 AND k=30;

QUERY PLAN

Index Only Scan using ijkl_btree on ijkl (... rows=2500 loops=1)
   Index Cond: ((i = 10) AND (k = 30))
   Heap Fetches: 0
   Buffers: shared hit=102 read=480 written=104
Planning Time: 0.284 ms
Execution Time: 29.452 ms
```

Par contre, dès que la première colonne de l'index B-tree (i) manque, celui-ci devient beaucoup moins intéressant (quoique pas inutilisable, mais il y a des chances qu'il faille le parcourir complètement). L'index GIN devient alors intéressant par sa taille réduite :

```
# EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * from ijkl WHERE j=20 AND k=40;

QUERY PLAN

Bitmap Heap Scan on ijkl (... rows=2500 loops=1)
Recheck Cond: ((j = 20) AND (k = 40))
Heap Blocks: exact=713
Buffers: shared hit=119 read=670
-> Bitmap Index Scan on ijkl_gin (... rows=2500 loops=1)
```

```
Index Cond: ((j = 20) AND (k = 40))
Buffers: shared hit=76
Planning Time: 0.382 ms
Execution Time: 28.987 ms
```

L'index GIN est obligé d'aller vérifier la visibilité des lignes dans la table, il ne supporte pas les *Index Only Scan*.

Sans lui, la seule alternative serait un Seq Scan qui parcourrait les 33 784 blocs de la table :

```
DROP INDEX ijkl_gin ;
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
 SELECT * from ijkl WHERE j=20 AND k=40 ;
                           QUERY PLAN
______
Gather (actual time=34.861..713.474 rows=2500 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=2260 read=31524
  -> Parallel Seq Scan on ijkl (... rows=833 loops=3)
        Filter: ((j = 20) \text{ AND } (k = 40))
        Rows Removed by Filter: 2082500
        Buffers: shared hit=2260 read=31524
Planning Time: 1.464 ms
 Execution Time: 713.796 ms
2.12.4 HypoPG
```

Pour la clarté des plans, désactiver le JIT.

```
SET jit TO off;
```

```
Créer la table suivante, où la clé i est très déséquilibrée:

CREATE UNLOGGED TABLE log10 AS
SELECT i,j, i+10 AS k, now() AS d, lpad(' ',300,' ') AS filler
FROM generate_series (0,7) i,
LATERAL (SELECT * FROM generate_series(1, power(10,i)::bigint ) j ) jj;

(Ne pas oublier VACUUM ANALYZE.)

CREATE UNLOGGED TABLE log10 AS
SELECT i,j, i+10 AS k, now() AS d, lpad(' ',300,' ') AS filler
FROM generate_series (0,7) i,
LATERAL (SELECT * FROM generate_series(1, power(10,i)::bigint ) j ) jj;

VACUUM ANALYZE log10 ;
```

Cette table fait 11,1 millions de lignes et presque 4 Go.

On se demande si créer un index sur i, (i,j) ou (j,i) serait utile pour les deux requêtes suivantes :

```
SELECT i, min(j), max(j) FROM log10 GROUP BY i ;
SELECT max(j) FROM log10 WHERE i = 6 ;
```

- Installer l'extension hypopg (les paquets dépendent de votre version de PostgreSQL et de la distribution, par exemple hypopg\_17 ou postgresql-17-hypopg).
- Créer des index hypothétiques (y compris un partiel) et choisir un seul index.

D'abord installer le paquet de l'extension. Sur Rocky Linux et autres dérivés Red Hat :

```
sudo dnf install hypopg_17
```

Sur Debian, Ubuntu et dérivés :

```
sudo apt install postgresql-17-hypopg
```

Puis installer l'extension dans la base concernée :

```
CREATE EXTENSION hypopg;
```

Création des différents index hypothétiques qui pourraient servir :

On vérifie qu'ils sont tous actifs dans cette session :

```
SELECT * FROM hypopg_list_indexes;
```

indexrelid	indexname	nspname	relname	amname
	   <78053>btree_log10_i   <78054>btree_log10_i_j	public   public   public		btree btree
	<78055>btree_log10_j_i	public	log10	btree
78056	<78056>btree_log10_j	public	log10	btree

```
EXPLAIN SELECT i, min(j), max(j) FROM log10 GROUP BY i;
                             QUERY PLAN
 ______
Finalize GroupAggregate (cost=1000.08..392727.62 rows=5 width=20)
  Group Key: i
  -> Gather Merge (cost=1000.08..392727.49 rows=10 width=20)
        Workers Planned: 2
        -> Partial GroupAggregate (cost=0.06..391726.32 rows=5 width=20)
              Group Key: i
              -> Parallel Index Only Scan
                  using <78054>btree_log10_i_j on log10
                  (cost=0.06..357004.01 rows=4629634 width=12)
EXPLAIN SELECT max(j) FROM log10 WHERE i = 6;
                            OUERY PLAN
Result (cost=0.08..0.09 rows=1 width=8)
  InitPlan 1 (returns $0)
    -> Limit (cost=0.05..0.08 rows=1 width=8)
          -> Index Only Scan Backward using <78056>btree_log10_j on log10
                                (cost=0.05..31812.59 rows=969631 width=8)
                Index Cond: (j IS NOT NULL)
Les deux requêtes n'utilisent pas le même index. Le partiel ( <78056>btree_log10_j ) ne conviendra
évidemment pas à toutes les requêtes, on voit donc ce qui se passe sans lui :
SELECT * FROM hypopg_drop_index(78056);
hypopg_drop_index
t
EXPLAIN SELECT max(j) FROM log10 WHERE i = 6 ;
                             QUERY PLAN
Result (cost=0.10..0.11 rows=1 width=8)
  InitPlan 1 (returns $0)
    -> Limit (cost=0.06..0.10 rows=1 width=8)
          -> Index Only Scan Backward using <78054>btree_log10_i_j on log10
                                     (cost=0.06..41660.68 rows=969631 width=8)
                Index Cond: ((i = 6) AND (j IS NOT NULL))
C'est presque aussi bon. L'index sur (i,j) semble donc convenir aux deux requêtes.
  Comparer le plan de la deuxième requête avant et après la création réelle de l'index.
Bien sûr, un EXPLAIN (ANALYZE) négligera ces index qui n'existent pas réellement :
  EXPLAIN (ANALYZE, TIMING OFF)
  SELECT max(j) FROM log10 WHERE i = 6;
                             QUERY PLAN
 _____
                                      _____
```

```
Finalize Aggregate
                          (cost=564931.67..564931.68 rows=1 width=8)
                                              (actual rows=1 loops=1)
   -> Gather (cost=564931.46..564931.67 rows=2 width=8)
                                    (actual rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         -> Partial Aggregate (cost=563931.46..563931.47 rows=1 width=8)
                                                     (actual rows=1 loops=3)
               -> Parallel Seq Scan on log10
                                 (cost=0.00..562921.43 rows=404013 width=8)
                                                (actual rows=333333 loops=3)
                      Filter: (i = 6)
                      Rows Removed by Filter: 3370370
 Planning Time: 0.414 ms
 Execution Time: 2322.879 ms
CREATE INDEX ON log10 (i,j);
Et le nouveau plan est cohérent avec l'estimation d'HypoPG:
 Result (cost=0.60..0.61 rows=1 width=8) (actual rows=1 loops=1)
   InitPlan 1 (returns $0)
     -> Limit (cost=0.56..0.60 rows=1 width=8) (actual rows=1 loops=1)
           -> Index Only Scan Backward using log10_i_j_idx on log10
                (cost=0.56..34329.16 rows=969630 width=8) (actual rows=1 loops=1)
                  Index Cond: ((i = 6) AND (j IS NOT NULL))
                  Heap Fetches: 0
 Planning Time: 1.070 ms
 Execution Time: 0.239 ms
  Créer un index fonctionnel hypothétique pour faciliter la requête suivante :
  SELECT k FROM log10 WHERE mod(j,99) = 55;
  Quel que soit le résultat, le créer quand même et voir s'il est utilisé.
Si on simule la présence de cet index fonctionnel :
SELECT * FROM hypopg_create_index ('CREATE INDEX ON log10 ( mod(j,99))');
EXPLAIN SELECT k FROM log10 WHERE mod(j,99) = 55;
                                  QUERY PLAN
 Gather (cost=1000.00..581051.11 rows=55556 width=4)
   Workers Planned: 2
   -> Parallel Seq Scan on log10 (cost=0.00..574495.51 rows=23148 width=4)
         Filter: (mod(j, '99'::bigint) = 55)
on constate que l'optimiseur le néglige.
Si on le crée quand même, sans oublier de mettre à jour les statistiques :
CREATE INDEX ON log10 ( mod(j,99) );
ANALYZE log10;
on constate qu'il est alors utilisé :
```

La différence tient à la volumétrie attendue qui a doublé après l'ANALYZE : il y a à présent des statistiques sur les résultats de la fonction qui n'étaient pas disponibles sans la création de l'index, comme on peut le constater avec :

```
SELECT * FROM pg_stats WHERE tablename = 'log10';
NB: on peut penser aussi à un index couvrant:

SELECT * FROM
hypopg_create_index (
   'CREATE INDEX ON log10 ( mod(j,99) ) INCLUDE(k)'
);
```

en fonction des requêtes réelles à optimiser.

# 3/ Extensions PostgreSQL pour la performance



# 3.1 PRÉAMBULE



Ce module présente des extensions plus spécifiquement destinées à améliorer les performances.

# 3.2 PG\_TRGM



- Indexation des recherches LIKE '%critère%'
- Similarité basée sur des trigrammes

```
CREATE EXTENSION pg_trgm;
SELECT similarity('bonjour','bnojour');

similarity
-----
0.333333

- Indexation (GIN ou GiST):

CREATE INDEX test_trgm_idx ON test_trgm
USING gist (text_data gist_trgm_ops);
```

Ce module permet de décomposer en trigramme les chaînes qui lui sont proposées :

Une fois les trigrammes indexés, on peut réaliser de la recherche floue, ou utiliser des clauses MARE malgré la présence de jokers (%) n'importe où dans la chaîne. À l'inverse, les indexations simples, de type B-tree, ne permettent des recherches efficaces que dans un cas particulier : si le seul joker de la chaîne est à la fin de celle ci (LIKE 'hello%' par exemple). Contrairement à la Full Text Search, la recherche par trigrammes ne réclame aucune modification des requêtes.

```
hello
hello everybody

Cette dernière requête passe par l'index test_trgm_idx, malgré le % initial:

EXPLAIN (ANALYZE)

SELECT text_data FROM test_trgm
WHERE text_data like '%hello%';

QUERY PLAN

Index Scan using test_trgm_gist_idx on test_trgm
(cost=0.41..0.63 rows=1 width=8) (actual time=0.174..0.204 rows=2 loops=1)
Index Cond: (text_data ~~ '%hello%'::text)
Rows Removed by Index Recheck: 1
Planning time: 0.202 ms
Execution time: 0.250 ms
```

On peut aussi utiliser un index GIN (comme pour le *Full Text Search*). Les index GIN ont l'avantage d'être plus efficaces pour les recherches exhaustives. Mais l'indexation pour la recherche des k éléments les plus proches (on parle de recherche k-NN) n'est disponible qu'avec les index GIST.

```
SELECT text_data, text_data <-> 'hello'
FROM test_trgm
ORDER BY text_data <-> 'hello'
LIMIT 4;
```

nous retourne par exemple les deux enregistrements les plus proches de « hello » dans la table test\_trgm.

# 3.3 PG\_STAT\_STATEMENTS



Capture en temps réel des requêtes

- Normalisation
- Indicateurs:
  - nombre d'exécutions,
  - nombre d'enregistrements retournés
  - temps cumulé d'exécution et d'optimisation
  - lectures/écritures en cache, demandées au système, tris
  - temps de lecture/écriture ( track\_io\_timing )
  - écritures dans les journaux de transactions (v13)
  - temps de planning (désactivé par défaut, v13)
  - utilisation du JIT (v15)

Cette extension est fournie avec PostgreSQL et est parmi les plus populaires et les plus utiles.

Une fois installé, pg\_stat\_statements capture, à chaque exécution de requête, tous les compteurs ci-dessus et d'autres associés à cette requête (champ query ), ci-dessous avec PostgreSQL 17 :

postgres=#	\d	pg_	stat_	_statements
------------	----	-----	-------	-------------

postgres-# \u pg_stat_s	Vue « public.pg_st	at_statements »		
Colonne	Type	Collationnement	NULL-able	Par défaut
userid	oid		 	,
dbid	oid		İ	İ
toplevel	boolean		İ	İ
queryid	bigint	ĺ	Ì	İ
query	text	İ	İ	ĺ
plans	bigint			
total_plan_time	double precision	İ		
min_plan_time	double precision			
max_plan_time	double precision	İ		
mean_plan_time	double precision	İ		
stddev_plan_time	double precision	1		
calls	bigint			
total_exec_time	double precision	1		
min_exec_time	double precision			
max_exec_time	double precision			
mean_exec_time	double precision	1		
stddev_exec_time	double precision			
rows	bigint			
shared_blks_hit	bigint			
shared_blks_read	bigint	1		
shared_blks_dirtied	bigint	1		
shared_blks_written	bigint			
local_blks_hit	bigint			
local_blks_read	bigint	1		
local_blks_dirtied	bigint	1		
local_blks_written	bigint	1		
temp_blks_read	bigint			

Quelques champs peuvent manquer ou porter un autre nom dans les versions précédentes.

Les requêtes d'une même base et d'un même utilisateur sont normalisées (reconnues comme identiques même avec des paramètres différents).

Les champs sont détaillés dans https://dali.bo/h2\_html#pg\_stat\_statements.

#### 3.3.1 pg\_stat\_statements: mise en place



```
shared_preload_libraries = 'pg_stat_statements'

CREATE EXTENSION pg_stat_statements ; -- dans 1 ou plusieurs bases
SELECT * FROM pg_stat_statements ;
```

- Vue en mémoire partagée (volumétrie contrôlée)
- Pas d'échantillonnage, seulement des compteurs cumulés
  - pg\_stat\_statements\_reset() ou PoWA

#### 3.3.1.1 Installation et réinitialisation

Ce module nécessite un espace en mémoire partagée. Pour l'installer, il faut donc renseigner le paramètre suivant avant de redémarrer l'instance :

```
shared_preload_libraries = 'pg_stat_statements'
```

Il faut installer l'extension dans au moins une base (dont une à laquelle les développeurs auront aussi accès, car l'information les concerne au premier chef) :

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements ;
```

La vue pg\_stat\_statements retourne un instantané des compteurs au moment de l'interrogation depuis l'installation, depuis le dernier arrêt brutal, ou depuis le dernier appel à la fonction pg\_stat\_statements\_reset(). Cette dernière fonction permet de réinitialiser les compteurs pour une base, un utilisateur, une requête, ou tout.

Deux méthodes d'utilisation sont donc possibles :

- effectuer un *reset* au début d'une période, puis interroger la vue pg\_stat\_statements à la fin de cette période ;
- capturer à intervalles réguliers le contenu de pg\_stat\_statements et visualiser les changements dans les compteurs : le projet PoWA¹ a été développé à cet effet.

La requête étant déjà analysée, cette opération supplémentaire n'ajoute qu'un faible surcoût (de l'ordre de 5 % sur une requête extrêmement courte), fixe, pour chaque requête.

Les données de l'extension sont stockées dans le PGDATA, sous pg\_stat\_tmp (même pour les versions récentes de PostgreSQL qui ne l'utilisent plus pour le stats collector), et un arrêt brutal peut mener à la perte du contenu.

### 3.3.1.2 Paramétrage

pg\_stat\_statements | possède quelques paramètres<sup>2</sup>.

Dès lors que l'extension est chargée en mémoire, la capture des compteurs est enclenchée, sauf si le paramètre pg\_stat\_statements.track est positionné à none. Celui-ci permet donc d'activer cette capture à la demande, sans qu'il soit nécessaire de redémarrer l'instance, ce qui peut s'avérer utile pour une instance avec beaucoup de requêtes très courtes (de type OLTP), et dont la rapidité est un élément critique : pour une telle instance, le surcoût lié à pg\_stat\_statements peut être jugé trop important pour que cette capture soit activée en permanence.

Sur un serveur chargé, il est déconseillé de réduire pg\_stat\_statements.max (nombre de requêtes différentes suivies, à 5000 par défaut), car le coût d'une désallocation n'est pas négligeable<sup>3</sup>.

<sup>&</sup>lt;sup>1</sup>https://powa.readthedocs.io/en/latest/

<sup>&</sup>lt;sup>2</sup>https://docs.postgresql.fr/current/pgstatstatements.html#id-1.11.7.40.9

<sup>&</sup>lt;sup>3</sup>https://yhuelf.github.io/2021/09/30/pg\_stat\_statements\_bottleneck.html

#### 3.3.2 pg stat\_statements: exemple 1



Requêtes les plus longues en temps cumulé :

La requête ci-dessus affiche les dix requêtes les plus longues en cumulé (même avec des paramètres différents), le nombre d'appels, le temps total, le temps moyen par appel. Les temps sont en millisecondes.

NB: pour une instance en version 12 ou antérieure, utiliser le champ total\_time, qui inclut aussi le temps de planification.

#### 3.3.3 pg\_stat\_statements: exemple 2



Requêtes les plus fréquemment appelées :

Cette requête affiche les dix requêtes les plus fréquentes en nombre d'appels, et le temps moyen. Exemple de sortie, avec un peu de formatage :

```
ORDER BY s.calls DESC
 LIMIT 10 \gx
-[ RECORD 1 ]---+---
rolname
              postgres
datname
              postgres
       329 021
calls
total_exec_time | 00:00:01.617168
avg_time | 00:00:00.000005
query | SELECT pg_postma
              | SELECT pg_postmaster_start_time()
-[ RECORD 2 ]---+---
rolname
             postgres
datname
             postgres
calls | 316 192
total_exec_time | 24:19:01.780477
avg_time | 00:00:00.276863
              | SELECT
query
                           count(datid) as databases,
                           pg_size_pretty(sum(pg_database_size(
                               pg_database.datname))::bigint) as total_size,
                           to_char(now(),$1) as time,
                           sum(xact_commit)::BIGINT as total_commit,
                           sum(xact_rollback)::BIGINT as total_rollback
                       FROM pg_database
                       JOIN pg_stat_database ON (pg_database.oid = pg_stat_data.
               .base.datid)
                    WHERE datistemplate = $2
-[ RECORD 3 ]---+-----
rolname
             | postgres
            | postgres
| 316 192
datname
calls
total_exec_time | 00:01:22.127931
avg_time | 00:00:00.00026
              | SELECT CASE sum(blks_hit+blks_read)
query
                         WHEN $1 THEN $2
                         ELSE trunc(sum(blks_hit)/sum(blks_hit+blks_read)*$3)::.
               |.float
                       END AS hitratio
                      FROM pg_stat_database
-[ RECORD 4 ]---+---
rolname | postgres
       | postgres
| 316 192
datname
calls
total_exec_time | 00:00:02.82872
avg_time | 00:00:00.000009
query
              | SELECT buffers_alloc FROM pg_stat_bgwriter
-[ RECORD 5 ]---+-----
rolname | postgres
datname | postgres
calls | 316 192
total_exec_time | 00:18:08.125136
avg_time | 00:00:00.003441
query
              | SELECT COUNT(*) AS nb FROM pg_stat_activity WHERE state != $1
-[ RECORD 6 ]---+---
             postgres
datname
              pgbench_300_hdd
```

```
| 79 534
total_exec_time | 00:03:44.82423
avg_time | 00:00:00.002827
             | select wait_event, wait_event_type, query from pg_stat_activity .
query
             .where state =$1 and pid = $2
-[ RECORD 7 ]---+-----
rolname | temboard_agent datname | postgres calls | 75 028
total_exec_time | 00:00:00.368735
avg_time | 00:00:00.000005
query | SFLECT_ng_nostmg
query
             | SELECT pg_postmaster_start_time()
-[ RECORD 8 ]---+-----
rolname | temboard_agent datname | postgres calls | 72 091
calls
              72 091
total_exec_time | 00:04:02.992142
avg_time | 00:00:00.003371
query | SELECT COUNT(*) AS nb FROM pg_stat_activity WHERE state != $1
-[ RECORD 9 ]---+-----
rolname | temboard_agent datname | postgres calls | 72 091
total_exec_time | 05:47:55.416569
avg_time | 00:00:00.28957
query
              | SELECT
                           count(datid) as databases,
                           pg_size_pretty(sum(pg_database_size(
                               pg_database.datname))::bigint) as total_size,
                           to_char(now(),$1) as time,
                           sum(xact_commit)::BIGINT as total_commit,
                           sum(xact_rollback)::BIGINT as total_rollback
                       FROM pg_database
                       JOIN pg_stat_database ON (pg_database.oid = pg_stat_data.
               l.base.datid)
                      WHERE datistemplate = $2
-[ RECORD 10 ]--+-----
rolname | temboard_agent datname | postgres calls | 72 091
total_exec_time | 00:00:17.817369
avg_time | 00:00:00.000247
query
              | SELECT CASE sum(blks_hit+blks_read)
                         WHEN $1 THEN $2
                         ELSE trunc(sum(blks_hit)/sum(blks_hit+blks_read)*$3)::.
              |.float
                       END AS hitratio
                       FROM pg_stat_database
```

On voit qu'il y a beaucoup de requêtes de supervision, ce qui est logique. Il est donc conseillé de dédier un utilisateur à la supervision pour pouvoir filtrer aisément.

## 3.3.4 pg\_stat\_statements: exemple 3



Requêtes les plus consommatrices et hit ratio :

Cette requête calcule le *hit ratio*, c'est-à-dire la proportion des blocs lus depuis le cache de PostgreSQL, pour les cinq plus grosses requêtes en temps cumulé. Dans l'idéal, ce ratio serait à 100 %.

# 3.4 AUTO\_EXPLAIN



- Tracer les plans des requêtes lentes automatiquement
- Contrib officielle
- Mise en place globale (traces):
  - globale:

```
shared_preload_libraries='auto_explain' -- redémarrage !
ALTER DATABASE erp SET auto_explain.log_min_duration = '3s';
   - session:
LOAD 'auto_explain';
SET auto_explain.log_analyze TO true;
```

L'outil auto\_explain est habituellement activé quand on a le sentiment qu'une requête devient subitement lente à certains moments, et qu'on suspecte que son plan diffère entre deux exécutions. Elle permet de tracer dans les journaux applicatifs, voire dans la console, le plan de la requête dès qu'elle dépasse une durée configurée.

C'est une « contrib » officielle de PostgreSQL (et non une extension). Tracer systématiquement le plan d'exécution d'une requête souvent répétée prend de la place, et est assez coûteux. C'est donc un outil à utiliser parcimonieusement. En général on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre auto\_explain.log\_min\_duration. Par défaut, ce paramètre vaut -1 pour ne tracer aucun plan.

Comme dans un EXPLAIN classique, on peut activer les options (par exemple ANALYZE ou TIMING avec, respectivement, un SET auto\_explain.log\_analyze TO true; ou un SET auto\_explain.log\_timing TO true; ) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'EXPLAIN, notamment : auto\_explain.log\_buffers, auto\_explain.log\_settings.

Quant à auto\_explain.sample\_rate, il permet de ne tracer qu'un échantillon des requêtes (voir la documentation<sup>4</sup>).

Pour utiliser auto\_explain globalement, il faut charger la bibliothèque au démarrage dans le fichier postgresql.conf via le paramètre shared\_preload\_libraries.

```
shared_preload_libraries='auto_explain'
```

<sup>&</sup>lt;sup>4</sup>https://docs.postgresql.fr/current/auto-explain.html

Après un redémarrage de l'instance, il est possible de configurer les paramètres de capture des plans d'exécution par base de données. Dans l'exemple ci-dessous, l'ensemble des requêtes sont tracées sur la base de données bench, qui est utilisée par pgbench.

```
ALTER DATABASE bench SET auto_explain.log_min_duration = '0';
ALTER DATABASE bench SET auto_explain.log_analyze = true;
```



Attention, l'activation des traces complètes sur une base de données avec un fort volume de requêtes peut être très coûteux.

Un benchmark pgbench est lancé sur la base de données bench avec 1 client qui exécute 1 transaction par seconde pendant 20 secondes :

```
pgbench -c1 -R1 -T20 bench
```

Les plans d'exécution de l'ensemble les requêtes exécutées par pgbench sont alors tracés dans les traces de l'instance.

```
2021-07-01 13:12:55.790 CEST [1705] LOG: duration: 0.041 ms plan:
    Query Text: SELECT abalance FROM pgbench_accounts WHERE aid = 416925;
    Index Scan using pgbench_accounts_pkey on pgbench_accounts
        (cost=0.42..8.44 rows=1 width=4) (actual time=0.030..0.032 rows=1 loops=1)
       Index Cond: (aid = 416925)
2021-07-01 13:12:55.791 CEST [1705] LOG: duration: 0.123 ms plan:
   Query Text: UPDATE pgbench_tellers SET tbalance = tbalance + -3201 WHERE tid = 19;
   Update on pgbench_tellers (cost=0.00..2.25 rows=1 width=358)
                               (actual time=0.120..0.121 rows=0 loops=1)
       -> Seq Scan on pgbench_tellers (cost=0.00..2.25 rows=1 width=358)
                                         (actual time=0.040..0.058 rows=1 loops=1)
            Filter: (tid = 19)
           Rows Removed by Filter: 99
2021-07-01 13:12:55.797 CEST [1705] LOG: duration: 0.116 ms plan:
   Query Text: UPDATE pgbench_branches SET bbalance = bbalance + -3201 WHERE bid = 5;
   Update on pgbench_branches (cost=0.00..1.13 rows=1 width=370)
                                (actual time=0.112..0.114 rows=0 loops=1)
       -> Seq Scan on pgbench_branches (cost=0.00..1.13 rows=1 width=370)
                               (actual time=0.036..0.038 rows=1 loops=1)
            Filter: (bid = 5)
            Rows Removed by Filter: 9
[...]
```

Pour utiliser auto\_explain uniquement dans la session en cours, il faut penser à descendre au niveau de message LOG (défaut de auto\_explain). On procède ainsi :

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0;
SET auto_explain.log_analyze = true;
SET client_min_messages to log;
SELECT count(*)
    FROM pg_class, pg_index
    WHERE oid = indrelid AND indisunique;
```

```
LOG: duration: 1.273 ms plan:
Query Text: SELECT count(*)
           FROM pg_class, pg_index
          WHERE oid = indrelid AND indisunique;
Aggregate (cost=38.50..38.51 rows=1 width=8)
    (actual time=1.247..1.248 rows=1 loops=1)
  -> Hash Join (cost=29.05..38.00 rows=201 width=0)
          (actual time=0.847..1.188 rows=198 loops=1)
        Hash Cond: (pg_index.indrelid = pg_class.oid)
        -> Seq Scan on pg_index (cost=0.00..8.42 rows=201 width=4)
                         (actual time=0.028..0.188 rows=198 loops=1)
              Filter: indisunique
              Rows Removed by Filter: 44
           Hash (cost=21.80..21.80 rows=580 width=4)
           (actual time=0.726..0.727 rows=579 loops=1)
              Buckets: 1024 Batches: 1 Memory Usage: 29kB
              -> Seq Scan on pg_class (cost=0.00..21.80 rows=580 width=4)
                                (actual time=0.016..0.373 rows=579 loops=1)
count
  198
```

auto\_explain est aussi un moyen de suivre les plans au sein de fonctions. Par défaut, un plan n'indique les compteurs de blocs *hit*, *read*, *temp*... que de l'appel global à la fonction.

Une fonction simple en PL/pgSQL est définie pour récupérer le solde le plus élevé dans la table pgbench\_accounts :

```
CREATE OR REPLACE function f_max_balance() RETURNS int AS $$
    DECLARE
        acct_balance int;
BEGIN
        SELECT max(abalance)
        INTO acct_balance
        FROM pgbench_accounts;
        RETURN acct_balance;
    END;
$$ LANGUAGE plpgsql ;
```

Un simple EXPLAIN ANALYZE de l'appel de la fonction ne permet pas d'obtenir le plan de la requête SELECT max(abalance) FROM pgbench\_accounts contenue dans la fonction:

```
EXPLAIN (ANALYZE, VERBOSE) SELECT f_max_balance();
```

```
QUERY PLAN
```

```
Result (cost=0.00..0.26 rows=1 width=4) (actual time=49.214..49.216 rows=1 loops=1)
Output: f_max_balance()
Planning Time: 0.149 ms
Execution Time: 49.326 ms
```

Par défaut, auto\_explain ne va pas capturer plus d'information que la commande EXPLAIN ANALYZE. Le fichier log de l'instance capture le même plan lorsque la fonction est exécutée.

Il est cependant possible d'activer le paramètre log\_nested\_statements avant l'appel de la fonction, de préférence uniquement dans la ou les sessions concernées :

```
\c bench
SET auto_explain.log_nested_statements = true;
SELECT f_max_balance();
```

Le plan d'exécution de la requête SQL est alors visible dans les traces de l'instance :

```
2021-07-01 14:58:40.189 CEST [2202] LOG: duration: 58.938 ms plan:
Query Text: select max(abalance)
            from pgbench_accounts
Finalize Aggregate
(cost=22632.85..22632.86 rows=1 width=4)
(actual time=58.252..58.935 rows=1 loops=1)
    -> Gather
        (cost=22632.64..22632.85 rows=2 width=4)
        (actual time=57.856..58.928 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate
                (cost=21632.64..21632.65 rows=1 width=4)
                (actual time=51.846..51.847 rows=1 loops=3)
            -> Parallel Seq Scan on pgbench_accounts
                    (cost=0.00..20589.51 rows=417251 width=4)
                    (actual time=0.014..29.379 rows=333333 loops=3)
```

pgBadger est capable de lire les plans tracés par <u>auto\_explain</u>, de les intégrer à son rapport et d'inclure un lien vers explain.depesz.com<sup>5</sup> pour une version plus lisible.

<sup>&</sup>lt;sup>5</sup>https://explain.depesz.com/

## 3.5 PG\_BUFFERCACHE



Qu'y a-t'il dans le cache de PostgreSQL? Fournit une vue :

- Pour chaque bloc de l'instance
  - fichier (donc objet) associé, base
  - fork (0:table, 1:FSM, 2:VM)
  - numéro de bloc
  - isdirty
  - usagecount (0-5)
- Pour: utilisation du cache, hot blocks
- pg\_buffercache\_evict() (v17+)

pg\_buffercache est une extension faisant partie de PostgreSQL, à installer dans chaque base au besoin.

Pour chaque entrée (bloc, par défaut de 8 ko) du cache disque de PostgreSQL, une vue nous fournit les informations suivantes :

- le fichier d'où provient ce bloc (donc indirectement le nom de la table ou de l'index...);
- le numéro de bloc dans ce fichier;
- si ce bloc est synchronisé avec le disque (isdirty à false);
- s'il est « sale » (dirty, modifié en mémoire mais non synchronisé sur disque);
- et si ce bloc a été utilisé récemment, selon la valeur de usagecount, de 0 (« n'est plus utilisé dernièrement ») à 5 (« récemment et fréquemment utilisé ») (voir les notions de *buffer pin* et *clocksweep*<sup>6</sup>).

### Contenu du cache de PostgreSQL:

pg\_buffercache permet donc d'abord de voir ce qui est dans le cache de PostgreSQL. À titre d'exemple, cette requête affiche les dix plus gros objets de la base en cours en mémoire cache (dont, ici, un index):

<sup>6</sup>https://dali.bo/m3\_html#:~:text=Buffer%20pin

```
AND b.reldatabase IN (0,(SELECT oid FROM pg_database WHERE datname = current_database()))

GROUP BY c.relname, c.relkind

ORDER BY 3 DESC NULLS LAST

LIMIT 5;
```

relname	relkind	buffers '	taille_mem
pgbench_accounts ø pgbench_accounts_pkey pgbench_history pgbench_tellers	+   r   ø   i   r   r	625618	5663 MB   4888 MB   1710 MB   22 MB   1192 kB



Il faut être dans la bonne base de données pour que les noms des tables soient accessibles (d'où le LEFT OUTER JOIN et les noms à NULL, qui sont des blocs soit libres, soit utilisés par d'autres bases).

Il y a parfois des surprises. Par exemple, certains index sur des UUID v4 peuvent avoir tendance à monopoliser le cache<sup>7</sup>.

Rappelons que certains processus (VACUUM), ANALYZE, COPY, un parcours de table...) ont leur propre *ring buffer*<sup>8</sup>, et donc n'apparaîtront pas immédiatement dans pg\_buffercache.

#### Hot blocks:

pg\_buffercache permet aussi de déterminer les *hot blocks* de la base, ou d'avoir une idée un peu plus précise du bon dimensionnement du cache. Si aucun bloc n'atteint un usagecount de 5, le cache est manifestement trop petit : il n'est pas capable de détecter les pages devant impérativement rester en cache. Inversement, si vous avez énormément d'entrées à 0 et quelques pages avec des usagecount très élevés, toutes ces pages à 0 sont égales devant le mécanisme d'éviction du cache. Elles sont donc supprimées à peu près de la même façon que du cache du système d'exploitation. Le cache de PostgreSQL dans ce cas fait « double emploi » avec lui, et pourrait être réduit.

Attention toutefois avec les expérimentations sur les caches : il existe des effets de seuils. Un cache trop petit peut de la même façon qu'un cache trop grand avoir une grande fraction d'enregistrements avec un usagecount à 0. Par ailleurs, le cache bouge extrêmement rapidement par rapport à notre capacité d'analyse. Nous ne voyons qu'un instantané, qui peut ne pas refléter toute la réalité. Le cache peut aussi avoir un comportement complètement différent selon le moment de la journée.

isdirty indique si un buffer est synchronisé avec le disque ou pas. Il est intéressant de vérifier qu'une instance dispose en permanence d'un certain nombre de buffers pour lesquels isdirty vaut false et pour lesquels usagecount vaut 0. Ces blocs n'ont pas à être écrits sur disque, et ils n'ont pas été utilisés depuis un certain temps : ils peuvent être remplacés immédiatement par d'autres blocs. S'il n'y a pas assez de tels blocs, c'est le signe :

<sup>&</sup>lt;sup>7</sup>https://dali.bo/s9\_html#uuid-inconvénients

<sup>8</sup>https://dali.bo/m3\_html#:~:text=Ring%20buffer

- que shared\_buffers est probablement trop petit (il n'arrive pas à contenir les modifications);
- et/ou que le background\_writer n'est pas assez agressif.

On peut suivre la quantité de blocs *dirty* et l'*usagecount* avec une requête de ce genre, ici juste après une petite mise à jour de la table test :

```
relname,
   isdirty,
   usagecount,
   pinning_backends,
   count(bufferid)
FROM pg_buffercache b
INNER JOIN pg_class c ON b.relfilenode = pg_relation_filenode(c.oid)
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
       relname,
       isdirty,
       usagecount,
       pinning_backends
ORDER BY 1, 2, 3, 4;
             | isdirty | usagecount | pinning_backends | count
   relname
brin_btree_idx | f
                                 0 |
                                                   0 |
                                                           1
                                                   0 |
brin_btree_idx | f
                                1 |
                                                        7151
                                                   0 |
brin_btree_idx | f
                                 2
                                                       3103
                                                   0 | 10695
brin_btree_idx | f
                                 3
                                 4 |
brin_btree_idx | f
                                                   0 | 141078
                                 5
brin_btree_idx | f
                                                   0 |
                                                           2
                                                   0 |
brin_btree_idx | t
                                 1 |
                                 2 |
                                                   0 |
brin_btree_idx | t
                                                           1
brin_btree_idx | t
                                 5 |
                                                   0 |
                                                   0 | 12371
              | f
                                 0 |
              | f
                                                   0 |
test
                                 1 |
                                                        6009
              | f
test
                                 2
                                                   0 |
                                                        8466
              | f
                                 3 |
test
                                                   0 |
                                                         1682
              | f
                                 4 |
test
                                                   0 l
                                                         7393
              | f
test
                                 5 |
                                                   0 |
                                                         112
test
              Ιt
                                  1 |
                                                   0 I
                                                          1
              | t
                                  5
                                                   0 |
                                                         267
test
test_pkey
              l f
                                  1 |
                                                   0 |
                                                         173
              | f
                                                   0 | 27448
test_pkey
                                  2
                                 3 |
test_pkey
              | f
                                                   0 |
              | f
                                 4
test_pkey
                                                   0 |
                                                       10324
                                 5
              | f
                                                   0 |
test_pkey
                                                        3420
                                                   0 |
              | t
test_pkey
                                 1 |
                                                         57
                                                  0 |
                                 3 |
test_pkey
              | t
                                                          81
                                 4
                                                   0 |
test_pkey
                t
                                                         116
test_pkey
              | t
                                  5 |
                                                   0 |
                                                       15067
```

### Évincer des blocs du cache :

Depuis PostgreSQL 17, la fonction pg\_buffercache\_evict() permet de libérer un bloc du cache. Dans les versions précédentes, nettoyer le cache impose de redémarrer PostgreSQL, ou de charger

**SELECT** 

une très grosse table avec pg\_prewarm. Cette fonctionnalité est utile par exemple pour tester des requêtes ou du paramétrage de manière reproductible, sans être perturbé par l'effet de cache.

Il faut appeler pg\_buffercache\_evict() pour chaque bloc, avec l'identifiant du bloc en paramètre, . Le code retour de la fonction est un booléen, à true en général, sauf si le buffer pointé n'existe pas, était utilisé (pinned) ou s'il a été sali (dirtied).

Pour supprimer du cache de PostgreSQL tous les blocs de la relation pgbench\_accounts dans la base en cours :

```
SELECT c.relnamespace::regnamespace, c.relname,
      b.bufferid, pg_buffercache_evict(b.bufferid)
FROM pg_buffercache b
JOIN pg_class c
ON b.relfilenode = pg_relation_filenode(c.oid) AND
  b.reldatabase IN (0, (SELECT oid FROM pg_database
                       WHERE datname = current_database()))
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relname = 'pgbench_accounts';
public
            | pgbench_accounts | 1572860 | t
public
             | pgbench_accounts |
                                   1572861 | t
             | pgbench_accounts | 1572862 | t
public
public | pgbench_accounts | 1572863 | t
public | pgbench_accounts | 1572864 | t
(1564533 lignes)
ou de manière plus synthétique :
SELECT c.relnamespace::regnamespace, c.relname,
      pg_buffercache_evict(b.bufferid), count(b.bufferid)
FROM pg_buffercache b
JOIN pg_class c
ON b.relfilenode = pg_relation_filenode(c.oid) AND
  b.reldatabase IN (0, (SELECT oid FROM pg_database
                       WHERE datname = current_database()))
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relname = 'pgbench_accounts'
GROUP BY 1,2,3 ;
                  relname | pg_buffercache_evict | count
relnamespace |
| pgbench_accounts | f
public
            | pgbench_accounts | t
                                                       1292638
À l'appel suivant, la table devra être relue hors du cache de PostgreSQL (mention read) :
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM pgbench_accounts ;
                       QUERY PLAN
```

Seq Scan on pgbench\_accounts (actual time=0.199..27994.323 rows=100000000 loops=1)

Buffers: shared read=1646640 I/O Timings: shared read=18049.770

Planning Time: 0.035 ms Execution Time: 32228.828 ms

#### Cache de Linux :

Le cache de Linux est complémentaire de celui de PostgreSQL. Pour des tests, il ne faut pas oublier de le purger aussi :

echo 3 | sudo tee /proc/sys/vm/drop\_caches

Pour étudier le comportement du cache de Linux, il existe l'extension pg\_stat\_kcache<sup>9</sup>.

<sup>&</sup>lt;sup>9</sup>https://github.com/powa-team/pg\_stat\_kcache

## 3.6 PG PREWARM



- Charge des blocs en cache:

```
-- cache de PG

SELECT pg_prewarm ('pgbench_accounts', 'buffer');

-- cache de Linux (asynchrone)

SELECT pg_prewarm ('pgbench_accounts', 'prefetch');

-- cache (tous OS)

SELECT pg_prewarm ('pgbench_accounts', 'read');
```

- Ne pas oublier les index!
- N'interdit pas l'éviction
- Récupération du cache au redémarrage (v11)
  - avec un petit paramétrage

Grâce à l'extension pg\_prewarm, intégrée à PostgreSQL, il est possible de pré-charger une table ou d'autres objets dans la mémoire de PostgreSQL, ou celle du système d'exploitation, pour améliorer les performances par la suite.

```
Par exemple, on charge la table pgbench_accounts dans le cache de PostgreSQL ainsi, et on le vérifie avec pg_buffercache :
```

La valeur retournée correspond aux blocs chargés.

Il faut rappeler qu'une table ne se résume pas à ses données! Il est au moins aussi intéressant de récupérer les index de la table en question:

```
SELECT pg_prewarm ('pgbench_accounts_pkey','buffer');
```

Si le cache de PostgreSQL ne suffit pas, celui du système peut être aussi préchargé :

```
SELECT pg_prewarm ('pgbench_accounts_pkey','read');
SELECT pg_prewarm ('pgbench_accounts_pkey','prefetch'); -- à préférer sur Linux
```

Charger une table en cache ne veut pas dire qu'elle va y rester! Si les blocs chargés ne sont pas utilisés, ils seront évincés quand PostgreSQL aura besoin de faire de la place dans le cache, comme n'importe quels autres blocs.

#### **Automatisation:**

Cette extension peut sauvegarder le contenu du cache à intervalles réguliers ou lors de l'arrêt (propre) de PostgreSQL et le restaurer au redémarrage. Pour cela, paramétrer ceci :

```
shared_preload_libraries = 'pg_prewarm'
pg_prewarm.autoprewarm = on
pg_prewarm.autoprewarm_interval = '5min'
```

Les blocs concernés sont sauvés dans un fichier autoprewarm. blocks dans le répertoire PGDATA. Un worker nommé autoprewarm leader apparaîtra.

L'intérêt est de réduire énormément la phase de rechargement en cache des donnés actives après un redémarrage, accidentel ou non. En effet, une grosse base très active et aux disques un peu lents peut mettre longtemps à re-remplir son cache et à retrouver des performances acceptables. De plus, ne seront rechargées que les données en cache précédemment, donc à priori les parties de tables réellement actives.

### Autres possibilités :

La documentation<sup>10</sup> décrit également comment charger :

- d'autres parties de la table comme la visibility map;
- certains blocs précis.

<sup>&</sup>lt;sup>10</sup>https://docs.postgresql.fr/current/pgprewarm.html

# 3.7 LANGAGES PROCÉDURAUX



- Procédures & fonctions en différents langages
- Par défaut : SQL, C et PL/pgSQLExtensions officielles : Perl, PythonMais aussi Java, Ruby, Javascript...
- Intérêts : fonctionnalités, performances

Les langages officiellement supportés par le projet sont :

```
PL/pgSQL;
PL/Perl<sup>11</sup>;
PL/Python<sup>12</sup>;
PL/Tcl.
```

Voici une liste non exhaustive des langages procéduraux disponibles, à différents degrés de maturité :

```
PL/sh<sup>13</sup>;
PL/R<sup>14</sup>;
PL/Java<sup>15</sup>;
PL/lolcode;
PL/Scheme;
PL/PHP;
PL/Ruby;
PL/Lua<sup>16</sup>;
PL/pgPSM;
PL/v8<sup>17</sup> (Javascript).
```

Le wiki PostgreSQL contient un tableau des langages supportés<sup>18</sup>.

Pour qu'un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL. Les autres doivent être ajoutés à partir des paquets de la distribution ou du PGDG, ou compilés à la main, puis l'extension installée dans la base :

```
11https://docs.postgresql.fr/current/plperl.html
12https://docs.postgresql.fr/current/plpython.html
13https://github.com/petere/plsh
14https://github.com/postgres-plr/plr
15https://tada.github.io/pljava/
16https://github.com/pllua/pllua
17https://github.com/plv8/plv8
18https://wiki.postgresql.org/wiki/PL_Matrix
```

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
-- etc.
```

Ces fonctions peuvent être utilisées dans des index fonctionnels et des triggers comme toute fonction SQL ou PL/pgSQL.

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n'est pas performant quand il s'agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage. S'ajoute l'intérêt de ne pas avoir à réécrire en PL/pgSQL des fonctions existantes dans d'autres langages ou d'accéder à des modules bien établis de ces langages.

### 3.7.1 Avantages & inconvénients



- PL/pgSQL plus performant pour l'accès aux données
- Chaque langage a son point fort
- Performances:
  - latence (pas d'allers-retours)
  - accès aux données depuis les fonctions
  - bibliothèques de chaque langage
  - index
- Langages trusted / untrusted

Il est courant de considérer que la logique métier (les fonctions) doit être intégralement dans l'applicatif, et pas dans la base de données. Même si l'on adopte ce point de vue, il faut savoir faire des exceptions pour prendre en compte les performances : une fonction, en PL/pgSQL ou un autre langage, exécutée dans la base de données économisera des aller-retours entre la base et le serveur applicatif, ce qui peut avoir un impact énorme (latence due à de nombreux ordres, ou durée de transfert des résultats intermédiaires).

Une fonction en Perl ou Python complexe peut servir aussi de critère d'indexation, pour des gains parfois énormes.

Le PL/pgSQL<sup>19</sup> est le mieux intégré des langages (avec le C), D'autres langages peuvent subir une pénalité due à la communication avec l'interpréteur (car c'est bien celui présent sur le serveur qui est

<sup>&</sup>lt;sup>19</sup>https://www.postgresql.org/docs/current/plpgsql-overview.html

utilisé). Cependant, ils peuvent apporter des fonctionnalités qui manquent à PostgreSQL : PL/R, bibliothèques numériques NumPy et Scipy de Python...

Pour des raisons de sécurité, on distingue des langages *trusted* et *untrusted*. Un langage *trusted* est disponible pour tous les utilisateurs de la base, n'autorise pas l'accès à des données normalement inaccessibles à l'utilisateur, mais quelques fonctionnalités ont pu être supprimées (interaction avec l'environnement notamment). Un langage *untrusted* n'a pas ces limites et les fonctions ne peuvent être créées que par un super-utilisateur. PL/pgSQL est *trusted*. PL/Python n'existe qu'en *untrusted* (l'extension pour la version 3 se nomme plpython3u). PL/Perl existe dans les deux versions (extensions plperl et plperlu).

### 3.8 HLL



- COUNT(DISTINCT) est notoirement lent
- Principe d'HyperLogLog:
  - travail sur des hachages, avec perte
  - estimation statistique
  - non exact, mais beaucoup plus rapide
- Exemple:

```
SELECT mois, hll_cardinality(hll_add_agg(hll_hash_text( id )))
FROM voyages ;
```

- Type hll pour pré-agréger des données

Les décomptes de valeurs distinctes sont une opération assez courante dans certains domaines : décompte de visiteurs distincts d'un site web ou d'un lieu, de patients d'un hôpital, de voyageurs, etc. Or COUNT (DISTINCT) est notoirement lent quand on fait face à un grand nombre de valeurs distinctes, à cause de la déduplication des valeurs, du maintien d'un espace pour le décompte, du besoin fréquent de fichiers temporaires...

Le principe de HyperLogLog est de ne pas opérer de calculs exacts mais de compiler un hachage des données rencontrées, avec perte, et donc beaucoup de manière plus compacte ; puis d'étudier la répartition statistique des valeurs rencontrées, et d'en déduire la volumétrie approximative. En effet, dans beaucoup de contexte, il n'est pas forcément utile de connaître le nombre *exact* de clients, de passagers... Une approximation peut répondre à beaucoup de besoins. En fonction de l'imprécision acceptée, on peut économiser beaucoup de mémoire et de temps (un gain d'un facteur supérieur à 10 est fréquent).

Une extension dédiée existe, à présent maintenue par Citusdata. Le source est sur Github<sup>20</sup>, et on trouvera les paquets dans les dépôts communautaires habituels.

La bibliothèque doit être préchargée dans chaque session pour être exécuté par l'optimiseur pour influencer les plans générés :

```
shared_preload_libraries = 'hll'
```

Puis charger l'extension dans la base concernée :

```
CREATE EXTENSION hll;
```

On peut alors immédiatement remplacer un COUNT (DISTINCT id) par cet équivalent :

<sup>&</sup>lt;sup>20</sup>https://github.com/citusdata/postgresql-hll

```
SELECT mois, hll_cardinality(hll_add_agg(hll_hash_text( id )))
FROM matable ;
```

Concrètement, l'identifiant à trier est haché (il y a une fonction dédiée par type). Puis ces hachages sont agrégés en un ensemble par la fonction hll\_add\_agg(). Ensuite, la fonction hll\_cardinality() estime le nombre de valeurs distinctes originales à partir de cet ensemble.

Le paramétrage par défaut est déjà pertinent pour des cardinalités jusqu'au billion (10<sup>12</sup>) d'après la documentation<sup>21</sup>, avec une erreur de l'ordre du pour cent. La précision de l'estimation peut être ajustée de manière générale, ou bien comme paramètre à la fonction de création de l'ensemble, comme dans ces exemples (ici avec les valeurs par défaut) :

```
SELECT hll_set_defaults(11, 5, -1, 1);
SELECT hll_cardinality(hll_add_agg(hll_hash_text( id ), 11, 5, -1, 1 ))
FROM matable;
```

Les deux premiers paramètres sont les plus importants : le nombre de registres utilisés (de 4 à 31, chaque incrément de 1 doublant la taille mémoire requise), et la taille des registres en bits (de 1 à 8). Des valeurs trop grandes risquent de rendre l'estimation inutilisable (résultat NaN).

Dans le monde décisionnel, il est fréquent de créer des tables d'agrégat avec des résultats précalculés sur un jour ou un mois. Cela ne fonctionne que partiellement pour des COUNT(DISTINCT): par exemple, on ne peut sommer le nombre de voyageurs distincts de chaque mois pour calculer celui sur l'année, ce sont peut-être les mêmes clients toute l'année. L'extension apporte donc aussi un type hll destiné à stocker des résultats agrégés issus d'un appel à hll\_add\_agg(). On agrège le contenu de ces champs hll avec la fonction hll\_union\_agg(), et on peut procéder à l'estimation sur l'ensemble avec hll\_cardinality.

<sup>&</sup>lt;sup>21</sup>https://github.com/citusdata/postgresql-hll

# **3.9 QUIZ**



https://dali.bo/x2\_quiz

# 3.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/x2\_solutions.

### 3.10.1 Indexation de pattern avec les varchar\_patterns et pg\_trgm



**But**: Indexer des patterns avec les varchar\_patterns et pg\_trgm

Ces exercices nécessitent une base contenant une quantité de données importante.

On utilisera donc le contenu de livres issus du projet Gutenberg. La base **gutenberg** est disponible en deux versions : complète sur https://dali.bo/tp\_gutenberg (dump de 0,5 Go, table de 21 millions de lignes dans 3 Go) ou https://dali.bo/tp\_gutenberg10 pour un extrait d'un dizième. Le dump peut se restaurer par exemple dans une nouvelle base, et contient juste une table nommée textes.

```
curl -kL https://dali.bo/tp_gutenberg -o /tmp/gutenberg.dmp
createdb gutenberg
pg_restore -d gutenberg /tmp/gutenberg.dmp
# le message sur le schéma public exitant est normale
rm -- /tmp/gutenberg.dmp
```

Pour obtenir des plans plus lisibles, on désactive JIT et parallélisme :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
```

Créer un index simple sur la colonne contenu de la table.

Rechercher un enregistrement commençant par « comme disent » : l'index est-il utilisé ?

Créer un index utilisant la classe text\_pattern\_ops . Refaire le test.

On veut chercher les lignes finissant par « Et vivre ». Indexer reverse (contenu) et trouver les lignes.

Installer l'extension pg\_trgm, puis créer un index GIN spécialisé de recherche dans les chaînes. Rechercher toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis insensible.

Si vous avez des connaissances sur les expression rationnelles, utilisez aussi ces trigrammes pour des recherches plus avancées. Les opérateurs sont :

opérateur	fonction
~	correspondance sensible à la casse
~*	correspondance insensible à la casse
!~	non-correspondance sensible à la casse
!~*	non-correspondance insensible à la casse

Rechercher toutes les lignes contenant « Fantine » OU « Valjean » : on peut utiliser une expression rationnelle.

Rechercher toutes les lignes mentionnant à la fois « Fantine » ET « Valjean ». Une formulation d'expression rationnelle simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

### 3.10.2 auto\_explain



**But** : Capturer les plans d'exécutions automatiquement avec auto\_explain

Installer le module auto\_explain (documentation : https://docs.postgresql.fr/current/auto-explain.html).

Exécuter des requêtes sur n'importe quelle base de données, et inspecter les traces générées.

Passer le niveau de messages de sa session (client\_min\_messages) à log.

### 3.10.3 pg\_stat\_statements



**But**: Analyser les performances des requêtes avec pg\_stat\_statements

- pg\_stat\_statements nécessite une bibliothèque préchargée. La positionner dans le fichier postgresql.conf, redémarrer PostgreSQL et créer l'extension.
- Inspecter le contenu de l'extension pg\_stat\_statements (\dx et \dx+).
- Vérifier que le serveur est capable d'activer la mesure de la durée des entrées-sorties avec pg\_test\_timing. Puis l'activer (track\_io\_timing), sans oublier de redémarrer PostgreSQL.
- Depuis un autre terminal, créer une base **pgbench** (si pas déjà disponible), l'initialiser (même si elle existait), et lancer une activité dessus :

```
# en tant qu'utilisateur postgres
createdb -e pgbench
/usr/pgsql-16/bin/pgbench -i -s135 pgbench
/usr/pgsql-16/bin/pgbench -c5 -j1 pgbench -T 600 -P1
```

- Dans la vue pg\_stat\_statements , récupérer les 5 requêtes les plus gourmandes en temps cumulé sur l'instance et leur nombre de lignes.

Quelle est la requête générant le plus d'écritures directes sur disques (*written*) ? Et en temps d'écriture ?

Quel est le hit ratio des requêtes les plus fréquentes ?

### 3.10.4 PL/Python, import de page web et compression



**But**: Importer et stocker une page web au format compressé avec PL/Python

Sur la base du code suivant en python 3 utilisant un des modules standard (documentation: http s://docs.python.org/3/library/urllib.request.html), créer une fonction PL/Python récupérant le code HTML d'une page web avec un simple SELECT pageweb ('https://www.postgresql.org/'):

```
import urllib.request
f = urllib.request.urlopen('https://www.postgresql.org/')
print (f.read().decode('utf-8'))
```

Stocker le résultat dans une table.

Puis stocker cette page en compression maximale dans un champ bytea, en passant par une fonction python inspirée du code suivant (documentation : https://docs.python.org/3/library/bz2.html) :

```
import bz2
compressed_data = bz2.compress(data, compresslevel=9)
```

Écrire la fonction de décompression avec la fonction python bz2.decompress.

```
Utiliser ensuite convert_from( bytea, 'UTF8') pour récupérer un text.
```

### 3.10.5 PL/Perl et comparaison de performances



But : Exécuter un traitement performant avec PL/Perl

Ce TP s'inspire d'un billet de blog de Daniel Vérité<sup>22</sup>, qui a publié le code des fonctions sur le wiki PostgreSQL sous licence PostgreSQL. Le principe est d'implémenter un remplacement en masse de nombreuses chaînes de caractères par d'autres. Une fonction codée en PL/perl peut se révéler plus rapide qu'une autre en PL/pgSQL.

Il utilise la base de données contenant des livres issus du projet Gutenberg, dans sa version complète qui contient *Les Misérables* de Victor Hugo. La base **gutenberg** est disponible en deux versions : complète sur https://dali.bo/tp\_gutenberg (dump de 0,5 Go, table de 21 millions de lignes dans 3 Go) ou https://dali.bo/tp\_gutenberg10 pour un extrait d'un dizième. Le dump peut se restaurer par exemple dans une nouvelle base, et contient juste une table nommée textes.

```
curl -kL https://dali.bo/tp_gutenberg -o /tmp/gutenberg.dmp
createdb gutenberg
pg_restore -d gutenberg /tmp/gutenberg.dmp
# le message sur le schéma public exitant est normale
rm -- /tmp/gutenberg.dmp
```

Créer la fonction multi\_replace en PL/pgSQL à partir du wiki PostgreSQL : https://wiki.postgresql.org/wiki/Multi\_Replace\_plpgsql

Récupérer la fonction en PL/perl sur le même wiki : https://wiki.postgresql.org/wiki/Multi\_Replace\_Perl.

<sup>&</sup>lt;sup>22</sup>https://blog-postgresql.verite.pro/2020/01/22/multi-replace.html

Vérifier que les deux fonctions ont le même nom mais des types de paramètres différents.

Le test va consister à transposer tous les noms et lieux des *Misérables* de Victor Hugo dans une version américaine :

- charger la base du projet Gutenberg si elle n'est pas déjà en place.
- créer une table miserables reprenant tous les livres dont le titre commence par « Les misérables ».

```
Tester le bon fonctionnement avec ces requêtes :
```

```
SELECT multi_replace (contenu,'{"Valjean":"Valjohn", "Cosette":"Lucy"}'::jsonb)
FROM miserables
WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5;

SELECT multi_replace(contenu, '{Valjean,Cosette}', '{Valjohn, Lucy}')
FROM miserables
WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5;
```

Pour faciliter la modification, prévoir une table pour stocker les critères :

```
CREATE TABLE remplacement (j jsonb, old_t text[], new_t text[]);
```

Insérer par exemple les données suivantes :

```
INSERT INTO remplacement (j)
SELECT '{"Valjean":"Valjohn", "Jean Valjean":"John Valjohn",
"Cosette":"Lucy", "Fantine":"Fanny", "Javert":"Green",
"Thénardier":"Thenardy", "Éponine":"Sharon", "Azelma":"Azealia",
"Marius":"Marc", "Gavroche":"Garry", "Enjolras":"Joker",
"Notre-Dame":"Empire State Building", "Victor Hugo":"Victor Hugues",
"Hugo":"Hugues", "Fauchelevent":"Dropwind", "Bouchart":"Butcher",
"Célestine":"Celeste","Mabeuf":"Myoax", "Leblanc":"White",
"Combeferre":"Combiron", "Magloire":"Glory",
"Gillenormand":"Jillnorthman", "France":"États-Unis",
"Paris":"New York", "Louis Philippe":"Andrew Jackson" }'::jsonb ;
```

Copier le contenu sous forme de tableau de caractères dans les autres champs :

### Comparer la performance des deux fonctions suivantes :

#### 3.10.6 hll



**But**: Estimer le nombre de valeurs distinctes plus rapidement avec hll

- Installer l'extension hll dans la base de données de test :
- le paquet est hll\_17 ou postgresql-17-hll (adapter le numéro à votre version de PostgreSQL) selon la distribution;
- l'extension se nomme | hll |;
- elle nécessite d'être préalablement déclarée dans shared\_preload\_libraries.
- Créer un jeu de données simulant des voyages en transport en commun, par passager selon la date :

```
CREATE TABLE voyages
(voyage_id
               bigint GENERATED ALWAYS AS IDENTITY,
 passager_id
                 text,
 d
                 date
) ;
INSERT INTO voyages (passager_id, d)
SELECT sem+mod(i, sem+1) \mid \mid \cdot \mid \cdot \mid \mid mod(i,77777) AS passager_id, d
FROM generate_series (0,51) sem,
      LATERAL
         (SELECT i,
          '2019-01-01'::date + sem * interval '7 days' + i * interval '2s' AS d
      FROM generate_series (1,
        (case when sem in (31,32,33) then 0 else 22 end +abs(30-sem))*5000 ) i
      ) j
;
```

- Activer l'affichage du temps ( timing ).
- Désactiver JIT et le parallélisme.
- Passer la mémoire de tri à 1 Go.
- Précharger la table dans le cache de PostgreSQL.
- Calculer, par mois, le nombre exact de voyages et de passagers **distincts**.
- Dans le plan de la requête, chercher où est perdu le temps.
- Calculer, pour l'année, le nombre exact de voyages et de passagers distincts.
- Recompter les passagers dans les deux cas en remplaçant le COUNT(DISTINCT) par cette expression: hll\_cardinality(hll\_add\_agg(hll\_hash\_text(passager\_id)))::int
- Réexécuter les requêtes après modification du paramétrage de hll:

```
SELECT hll_set_defaults(17, 5, -1, 0);
```

- Créer une table d'agrégat par mois avec un champ d'agrégat hll et la remplir.

## À partir de cette table d'agrégat :

- calculer le nombre moyen mensuel de passagers distincts,
- recalculer le nombre de passagers distincts sur l'année à partir de cette table d'agrégat.

Avec une fonction de fenêtrage sur hll\_union\_agg, calculer une moyenne glissante sur 3 mois du nombre de passagers distincts.

# 3.11 TRAVAUX PRATIQUES (SOLUTIONS)

### 3.11.1 Indexation de pattern avec les varchar\_patterns et pg\_trgm

Créer un index simple sur la colonne contenu de la table.

```
CREATE INDEX ON textes(contenu);
```

Il y aura une erreur si la base **gutenberg** est dans sa version complète, un livre de Marcel Proust dépasse la taille indexable maximale :

```
ERROR: index row size 2968 exceeds maximum 2712 for index "textes_contenu_idx" ASTUCE: Values larger than 1/3 of a buffer page cannot be indexed. Consider a function index of an MD5 hash of the value, or use full text indexing.
```

Pour l'exercice, on supprime ce livre avant d'indexer la colonne :

```
DELETE FROM textes where livre = 'Les Demi-Vierges, Prévost, Marcel';
CREATE INDEX ON textes(contenu);
```

Rechercher un enregistrement commençant par « comme disent » : l'index est-il utilisé ?

Le plan exact peut dépendre de la version de PostgreSQL, du paramétrage exact, d'éventuelles modifications à la table. Dans beaucoup de cas, on obtiendra :

```
SET jit TO off;
SET max_parallel_workers_per_gather TO 0;
VACUUM ANALYZE textes;
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
                             QUERY PLAN
 Seq Scan on textes (cost=0.00..669657.38 rows=1668 width=124)
            (actual time=305.848..6275.845 rows=47 loops=1)
   Filter: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Filter: 20945503
 Planning Time: 1.033 ms
 Execution Time: 6275.957 ms
C'est un Seq Scan : l'index n'est pas utilisé!
Dans d'autres cas, on aura ceci (avec PostgreSQL 12 et la version complète de la base ici) :
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
                             QUERY PLAN
 Index Scan using textes_contenu_idx on textes (...)
   Index Cond: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Index Recheck: 110
   Buffers: shared hit=28 read=49279
   I/O Timings: read=311238.192
 Planning Time: 0.352 ms
```

Execution Time: 313481.602 ms

C'est un Index Scan mais il ne faut pas crier victoire : l'index est parcouru entièrement (50 000 blocs !). Il ne sert qu'à lire toutes les valeurs de contenu en lisant moins de blocs que par un Seq Scan de la table. Le choix de PostgreSQL entre lire cet index et lire la table dépend notamment du paramétrage et des tailles respectives.

Le problème est que l'index sur contenu utilise la collation C et non la collation par défaut de la base, généralement en\_US.UTF-8 ou fr\_FR.UTF-8. Pour contourner cette limitation, PostgreSQL fournit deux classes d'opérateurs : varchar\_pattern\_ops pour varchar et text\_pattern\_ops pour text.

Créer un index utilisant la classe text\_pattern\_ops . Refaire le test.

On constate que comme l'ordre choisi est l'ordre ASCII, l'optimiseur sait qu'après « comme disent », c'est « comme disenu » qui apparaît dans l'index.

Noter que Index Cond contient le filtre utilisé pour l'index (réexprimé sous forme d'inégalités en collation C) et Filter un filtrage des résultats de l'index.

On veut chercher les lignes finissant par « Et vivre ». Indexer reverse (contenu) et trouver les lignes.

Cette recherche n'est possible avec un index B-Tree qu'en utilisant un index sur fonction :

```
CREATE INDEX ON textes(reverse(contenu) text_pattern_ops);

Il faut ensuite utiliser ce reverse systématiquement dans les requêtes:

EXPLAIN (ANALYZE)

SELECT * FROM textes WHERE reverse(contenu) LIKE reverse('%Et vivre');

QUERY PLAN

Index Scan using textes_reverse_idx on textes
```

```
(cost=0.56..377770.76 rows=104728 width=123)
                           (actual time=0.083..0.098 rows=2 loops=1)
   Index Cond: ((reverse(contenu) ~>=~ 'erviv tE'::text)
             AND (reverse(contenu) ~<~ 'erviv tF'::text))</pre>
   Filter: (reverse(contenu) ~~ 'erviv tE%'::text)
 Planning Time: 1.903 ms
 Execution Time: 0.421 ms
On constate que le résultat de reverse (contenu) a été directement utilisé par l'optimiseur. La re-
quête est donc très rapide. On peut utiliser une méthode similaire pour la recherche insensible à la
casse, en utiliser lower() ou upper().
Toutefois, ces méthodes ne permettent de filtrer qu'au début ou à la fin de la chaîne, ne permettent
qu'une recherche sensible ou insensible à la casse, mais pas les deux simultanément, et imposent aux
développeurs de préciser reverse, lower, etc. partout.
  Installer l'extension pg_trgm, puis créer un index GIN spécialisé de recherche dans les chaînes.
  Rechercher toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis in-
  sensible.
Pour installer l'extension pg_trgm :
CREATE EXTENSION pg_trgm;
Pour créer un index GIN sur la colonne contenu :
CREATE INDEX idx_textes_trgm ON textes USING gin (contenu gin_trgm_ops);
Recherche des lignes contenant « Valjean » de façon sensible à la casse :
EXPLAIN (ANALYZE)
SELECT * FROM textes WHERE contenu LIKE '%Valjean%';
                                   QUERY PLAN
 Bitmap Heap Scan on textes (cost=77.01..6479.68 rows=1679 width=123)
                       (actual time=11.004..14.769 rows=1213 loops=1)
   Recheck Cond: (contenu ~~ '%Valjean%'::text)
   Rows Removed by Index Recheck: 1
   Heap Blocks: exact=353
   -> Bitmap Index Scan on idx_textes_trgm
                                 (cost=0.00..76.59 rows=1679 width=0)
                        (actual time=10.797..10.797 rows=1214 loops=1)
         Index Cond: (contenu ~~ '%Valjean%'::text)
 Planning Time: 0.815 ms
 Execution Time: 15.122 ms
Puis insensible à la casse :
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ILIKE '%Valjean%';
```

QUERY PLAN

Bitmap Heap Scan on textes (cost=77.01..6479.68 rows=1679 width=123)

On constate que l'index a été nettement plus long à créer, et que la recherche est plus lente. La contrepartie est évidemment que les trigrammes sont infiniment plus souples. On constate aussi que le LIKE a dû encore filtrer 1 enregistrement après le parcours de l'index : en effet l'index trigramme est insensible à la casse, il ramène donc trop d'enregistrements, et une ligne avec « VALJEAN » a dû être filtrée.

Rechercher toutes les lignes contenant « Fantine » OU « Valjean » : on peut utiliser une expression rationnelle.

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ~ 'Valjean|Fantine';
```

```
QUERY PLAN
```

Rechercher toutes les lignes mentionnant à la fois « Fantine » ET « Valjean ». Une formulation d'expression rationnelle simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

```
EXPLAIN ANALYZE SELECT * FROM textes
WHERE contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)';
```

#### QUERY PLAN

### 3.11.2 auto\_explain

Installer le module auto\_explain (documentation : https://docs.postgresql.fr/current/auto-explain.html).

Dans le fichier postgresql.conf, chargement du module et activation globale pour *toutes* les requêtes (ce qu'on évitera de faire en production) :

```
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = 0
```

Redémarrer PostgreSQL.

Exécuter des requêtes sur n'importe quelle base de données, et inspecter les traces générées.

Le plan de la moindre requête (même un \d+ ) doit apparaître dans la trace.

```
Passer le niveau de messages de sa session (client_min_messages) à log.
```

Il est possible de recevoir les messages directement dans sa session. Tous les messages de log sont marqués d'un niveau de priorité. Les messages produits par auto\_explain sont au niveau log . Il suffit donc de passer le paramètre client\_min\_messages au niveau log .

Positionner le paramètre de session comme ci-dessous, ré-exécuter la requête.

```
SET client_min_messages TO log;
SELECT...
```

### 3.11.3 pg\_stat\_statements

 pg\_stat\_statements nécessite une bibliothèque préchargée. La positionner dans le fichier postgresql.conf, redémarrer PostgreSQL et créer l'extension.

```
Si une autre extension (ici auto_explain ) est également présente, on peut les lister ainsi :
```

```
shared_preload_libraries = 'auto_explain,pg_stat_statements'
```

Redémarrer PostgreSQL.

Dans la base **postgres** (par exemple), créer l'extension :

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements ;
```

Inspecter le contenu de l'extension pg\_stat\_statements (\dx et \dx+).

 Vérifier que le serveur est capable d'activer la mesure de la durée des entrées-sorties avec pg\_test\_timing. Puis l'activer (track\_io\_timing), sans oublier de redémarrer PostgreSQL.

### pg\_test\_timing<sup>23</sup> est livré avec PostgreSQL :

```
/usr/pgsql-16/bin/pg_test_timing
```

Testing timing overhead for 3 seconds. Per loop time including overhead: 33.24 ns

Histogram of timing durations:

< us	% 01 totat	Count
1	97.25509	87770521
2	2.72390	2458258
4	0.00072	646
8	0.00244	2200
16	0.00984	8882
32	0.00328	2958
64	0.00298	2689
128	0.00099	892
256	0.00055	499
512	0.00016	141
1024	0.00006	53
2048	0.00000	1

<sup>&</sup>lt;sup>23</sup>https://docs.postgresql.fr/current/pgtesttiming.html

Si le temps de mesure n'est que de quelques dizaines de nanosecondes, c'est OK. (C'est le cas sur presque toutes les machines et systèmes d'exploitation actuels, mais il y a parfois des surprises.) Sinon, éviter de faire ce qui suit sur un serveur de production. Sur une machine de formation, ce n'est pas un problème.

```
Dans le fichier postgresql.conf , positionner:
track_io_timing = on
```

Changer ce paramètre nécessite de redémarrer PostgreSQL.

 Depuis un autre terminal, créer une base **pgbench** (si pas déjà disponible), l'initialiser (même si elle existait), et lancer une activité dessus :

```
# en tant qu'utilisateur postgres
  createdb -e pgbench
  /usr/pgsql-16/bin/pgbench -i -s135 pgbench
  /usr/pgsql-16/bin/pgbench -c5 -j1 pgbench -T 600 -P1
createdb -e pgbench
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE pgbench;
/usr/pgsql-16/bin/pgbench -i -s135 pgbench
creating tables...
generating data (client-side)...
13500000 of 13500000 tuples (100%) done (elapsed 10.99 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 14.97 s (drop tables 0.00 s, create tables 0.02 s, client-side generate 11.04
\rightarrow s, vacuum 0.34 s, primary keys 3.57 s).
/usr/pgsql-16/bin/pgbench -c5 -j1 pgbench -T 600 -P1
pgbench (16.2)
starting vacuum...end.
progress: 1.0 s, 2364.9 tps, lat 2.078 ms stddev 1.203, 0 failed
progress: 2.0 s, 2240.0 tps, lat 2.221 ms stddev 0.871, 0 failed
```

On a donc 5 clients qui vont mettre à jour la base à raison de 2000 transactions par seconde (valeur très dépendante des CPUs et des disques).

- Dans la vue pg\_stat\_statements , récupérer les 5 requêtes les plus gourmandes en temps cumulé sur l'instance et leur nombre de lignes.

```
SELECT calls, query, rows,
total_exec_time*interval '1ms' AS tps_total
FROM pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 5
\gx
```

Le résultat va dépendre de l'historique de votre instance, et du temps déroulé depuis le lancement de pgbench, mais c'est probablement proche de ceci :

```
-[ RECORD 1 ]------
calls | 879669
query
      | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
     | 879669
rows
tps_total | 00:02:56.184131
calls | 879664
query | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
rows
      879664
tps_total | 00:00:44.803628
-[ RECORD 3 ]------
calls | 879664
     UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2 | 879664
query
rows
tps_total | 00:00:12.196055
-[ RECORD 4 ]------
     | 1
calls
      | copy pgbench_accounts from stdin with (freeze on)
query
rows
       13500000
tps_total | 00:00:10.698976
-[ RECORD 5 ]-----
calls | 879664
query
      | SELECT abalance FROM pgbench_accounts WHERE aid = $1
       879664
rows
tps_total | 00:00:06.530169
```

Noter que l'unique COPY pour créer la base dure plus que les centaines de milliers d'occurences de la cinquième requête.

Quelle est la requête générant le plus d'écritures directes sur disques (*written*) ? Et en temps d'écriture ?

Pour les *written*, il faut tenir compte des trois sources : blocs du cache partagé, blocs des *backends*, fichiers temporaires.

```
SELECT calls,
      pg_size_pretty(8192::numeric
      * (shared_blks_written+local_blks_written+temp_blks_written)) AS written,
      pg_size_pretty(8192::numeric*shared_blks_written) AS shared_written,
      pg_size_pretty(8192::numeric*temp_blks_written)
AS temp_written,
      blk_write_time * interval '1ms'
AS blk_write_time,
      temp_blk_write_time * interval '1ms' AS temp_blk_write_time,
     query
FROM pg_stat_statements
ORDER BY shared_blks_written+local_blks_written+temp_blks_written DESC LIMIT 3 ;
-[ RECORD 1 ]-----+------
calls
                 2400667
written
                 | 15 GB
shared_written
                | 15 GB
temp_written | 0 bytes
```

```
blk_write_time | 00:00:11.840499
temp_blk_write_time | 00:00:00
                 UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE a.
query
                |.id = $2
-[ RECORD 2 ]-----+-----------
calls
                | 3442 MB
written
shared_written | 3442 MB
temp_written | 0 bytes
blk_write_time | 00:00:00
temp_blk_write_time | 00:00:00
           | copy pgbench_accounts from stdin with (freeze on)
query
calls
                | 1
written
                 | 516 MB
shared_written
                0 bytes
blk_write_time
                | 516 MB
                 00:00:00
temp_blk_write_time | 00:00:00
                | alter table pgbench_accounts add primary key (aid)
```

Il y a donc beaucoup d'écritures directes. C'est le signe que le cache en écriture de PostgreSQL est insuffisant (la base fait 2 Go, à peu près intégralement balayée, et le shared\_buffers par défaut ne fait que 128 Mo) ou que le background writer doit être modifié pour nettoyer plus souvent les blocs dirty.

On note que l'UPDATE et le COPY ont écrit des blocs qui auraient dû passer uniquement par le cache, alors le ALTER TABLE, lui, a essentiellement écrit un fichier temporaire (c'est logique lors d'une création d'index).

Avec des *shared buffers* plus importants, les <u>shared\_written</u> sont quasiment absents. Ils proviennent essentiellement d'ordres lourds comme <u>COPY</u>.

Quel est le hit ratio des requêtes les plus fréquentes ?

```
SELECT calls, total_exec_time,
      round(100.0*shared_blks_hit
      /nullif(shared_blks_hit+shared_blks_read, 0),2) AS "hit %",
      query
FROM pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 5 ;
             2400667
calls
total_exec_time | 464702.3557850064
              | 73.24
hit %
              | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =.
query
-[ RECORD 2 ]---+---
                        ______
             2400658
calls
total_exec_time | 141310.02034101041
hit %
              100.00
              | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =.
query
```

#### **DALIBO Formations**

```
-[ RECORD 3 ]---+-----
calls | 2400659
total_exec_time | 34201.65339700031
hit % | 100.00
          | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = .
query
          1.$2
-[ RECORD 4 ]---+-----
calls | 2400661
total_exec_time | 16494.857696000774
hit % | 100.00
query
          | SELECT abalance FROM pgbench_accounts WHERE aid = $1
-[ RECORD 5 ]---+-----
          2400656
total_exec_time | 11685.776115000388
hit % | 100.00
           | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES.
query
           |. ($1, $2, $3, $4, CURRENT_TIMESTAMP)
```

On constate que le *hit ratio* est parfait, sauf la première requête. C'est logique, car la table pgbench\_accounts ne tient pas dans le cache par défaut et elle est balayée à peu près entièrement par les requêtes de pgbench.

### 3.11.4 PL/Python, import de page web et compression

```
Sur la base du code suivant en python 3 utilisant un des modules standard (documentation: http s://docs.python.org/3/library/urllib.request.html), créer une fonction PL/Python récupérant le code HTML d'une page web avec un simple SELECT pageweb('https://www.postgresql.org/'):

import urllib.request
f = urllib.request.urlopen('https://www.postgresql.org/')
print (f.read().decode('utf-8'))

Il faut bien évidemment que PL/Python soit installé. D'abord le paquet, ici sous Rocky Linux 8 avec PostgreSQL 17 (le nom du paquet doit être adapté à la version de PostgreSQL):
```

```
# dnf install postgresql17-plpython3
Sur Debian, Ubuntu et dérivés ce sera:
# apt install postgresql-plpython3-17
Puis, dans la base de données concernée:
CREATE EXTENSION plpython3u;
La fonction PL/Python est:
CREATE OR REPLACE FUNCTION pageweb (url text)
    RETURNS text
AS $$
import urllib.request
f = urllib.request.urlopen(url)
return f.read().decode('utf-8')
$$ LANGUAGE plpython3u COST 100000;
```

Évidemment, il ne s'agit que d'un squelette ne gérant pas les erreurs, les redirections, etc.

Stocker le résultat dans une table.

```
On vérifie ainsi le bon fonctionnement :
```

```
CREATE TABLE pagesweb (url text, page text, pagebz2 bytea, page2 text ) ;
INSERT INTO pagesweb (url, page)
SELECT 'https://www.postgresql.org/', pageweb('https://www.postgresql.org/') ;
```

Puis stocker cette page en compression maximale dans un champ bytea, en passant par une fonction python inspirée du code suivant (documentation : https://docs.python.org/3/library/bz2.html) :

```
import bz2
compressed_data = bz2.compress(data, compresslevel=9)
```

```
import bz2
```

```
c=bz2.compress(data, compresslevel=9)
```

Même si la page récupérée est en texte, la fonction python exige du binaire, donc le champ en entrée sera du bytea :

```
-- version pour bytea
CREATE OR REPLACE FUNCTION bz2 (objet bytea)
  RETURNS bytea
AS $$
import bz2
return bz2.compress(objet, compresslevel=9)
$$ LANGUAGE plpython3u IMMUTABLE COST 1000000;
```

On peut faire la conversion depuis text à l'appel ou modifier la fonction pour qu'elle convertisse d'elle-même. Mais le plus confortable est de créer une fonction SQL de même nom qui se chargera de la conversion. Selon le type en paramètre, l'une ou l'autre fonction sera appelée.

```
-- fonction d'enrobage pour s'épargner une conversion explicite en bytea
CREATE OR REPLACE FUNCTION bz2 (objet text)
   RETURNS bytea
AS $$
SELECT bz2(objet::bytea) ;
$$ LANGUAGE sql IMMUTABLE ;

Compression de la page:
UPDATE pagesweb
SET pagebz2 = bz2 (page) ;
```



NB : PostgreSQL stocke déjà les textes longs sous forme compressée (mécanisme du TOAST).

Tout ceci n'a donc d'intérêt que pour gagner quelques octets supplémentaires, ou si le .bz2 doit être réutilisé directement. Noter que l'on utilise ici uniquement des fonctionnalités standards de PostgreSQL et python3, sans module extérieur à la fiabilité inconnue.

De plus, les données ne quittent pas le serveur, épargnant du trafic réseau.

Écrire la fonction de décompression avec la fonction python bz2.decompress.

```
CREATE OR REPLACE FUNCTION bz2d (objet bytea)
RETURNS bytea

AS $$
import bz2
return bz2.decompress(objet)
$$ LANGUAGE plpython3u IMMUTABLE COST 10000000;

Utiliser ensuite convert_from( bytea, 'UTF8') pour récupérer un text.

CREATE OR REPLACE FUNCTION bz2_to_text (objetbz2 bytea)
RETURNS text
```

### 3.11.5 PL/Perl et comparaison de performances

Créer la fonction multi\_replace en PL/pgSQL à partir du wiki PostgreSQL : https://wiki.postgresql.org/wiki/Multi\_Replace\_plpgsql

#### Le code sur le wiki est le suivant :

```
/* This function quotes characters that may be interpreted as special
   in a regular expression.
   It's used by the function below and declared separately for clarity. */
CREATE FUNCTION quote_meta(text) RETURNS text AS $$
  SELECT regexp_replace($1, '([\[\]\\\^\$\.\|\?\*\+\(\)])', '\\\1', 'g');
$$ LANGUAGE SQL strict immutable;
/* Substitute a set of substrings within a larger string.
  When several strings match, the longest wins.
   Similar to php's strtr(string $str, array $replace_pairs).
  Example:
  select multi_replace('foo and bar is not foobar',
             '{"bar":"foo", "foo":"bar", "foobar":"foobar"}'::jsonb);
  => 'bar and foo is not foobar'
CREATE FUNCTION multi_replace(str text, substitutions jsonb)
RETURNS text
AS $$
DECLARE
rx text;
s_left text;
s_tail text;
res text:='';
BEGIN
SELECT string_agg(quote_meta(term), '|')
FROM jsonb_object_keys(substitutions) AS x(term)
  WHERE term <> ''
INTO rx;
IF (COALESCE(rx, '') = '') THEN
   -- the loop on the RE can't work with an empty alternation
   RETURN str;
END IF;
rx := concat('^(.*?)(', rx, ')(.*)$'); -- match no more than 1 row
   s_tail := str;
   SELECT
       concat(matches[1], substitutions->>matches[2]),
       matches[3]
    FROM
      regexp_matches(str, rx, 'g') AS matches
    INTO s_left, str;
   exit WHEN s_left IS NULL;
   res := res || s_left;
```

```
END loop;
res := res || s_tail;
RETURN res;

END
$$ LANGUAGE plpgsql strict immutable;
```

Récupérer la fonction en PL/perl sur le même wiki : https://wiki.postgresql.org/wiki/Multi\_Rep lace\_Perl.

Évidemment, il faudra l'extension dédiée au langage Perl:

```
# dnf install postgresql14-plperl
CREATE EXTENSION plperl;
Le code de la fonction est :
CREATE FUNCTION multi_replace(string text, orig text[], repl text[])
RETURNS text
AS $BODY$
 my ($string, $orig, $repl) = @_;
 my %subs;
  if (@$orig != @$repl) {
     elog(ERROR, "array sizes mismatch");
  if (ref @$orig[0] eq 'ARRAY' || ref @$repl[0] eq 'ARRAY') {
     elog(ERROR, "array dimensions mismatch");
  @subs{@$orig} = @$repl;
  my $re = join "|", map quotemeta,
    sort { (length($b) <=> length($a)) } keys %subs;
  re = qr/(re)/;
  $string =~ s/$re/$subs{$1}/g;
  return $string;
$BODY$ language plperl strict immutable;
```

Vérifier que les deux fonctions ont le même nom mais des types de paramètres différents.

PostgreSQL sait quelle fonction appeler selon les paramètres fournis.

Le test va consister à transposer tous les noms et lieux des *Misérables* de Victor Hugo dans une version américaine :

Le test va consister à transposer tous les noms et lieux des *Misérables* de Victor Hugo dans une version américaine :

- charger la base du projet Gutenberg si elle n'est pas déjà en place.
- créer une table miserables reprenant tous les livres dont le titre commence par « Les misérables ».

```
CREATE TABLE miserables as select * from textes
    WHERE livre LIKE 'Les misérables%';
```

Cette table fait 68 000 lignes.

```
Tester le bon fonctionnement avec ces requêtes:

SELECT multi_replace (contenu,'{"Valjean":"Valjohn", "Cosette":"Lucy"}'::jsonb)
FROM miserables
WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5;

SELECT multi_replace(contenu, '{Valjean,Cosette}', '{Valjohn, Lucy}')
FROM miserables
WHERE contenu ~ '(Valjean|Cosette)' LIMIT 5;
```

Le texte affiché doit comporter « Jean Valjohn » et « Lucy ».

```
Pour faciliter la modification, prévoir une table pour stocker les critères : 

CREATE TABLE remplacement (j jsonb, old_t text[], new_t text[]) ;
```

Insérer par exemple les données suivantes :

```
INSERT INTO remplacement (j)
SELECT '{"Valjean":"Valjohn", "Jean Valjean":"John Valjohn",
"Cosette":"Lucy", "Fantine":"Fanny", "Javert":"Green",
"Thénardier":"Thenardy", "Éponine":"Sharon", "Azelma":"Azealia",
"Marius":"Marc", "Gavroche":"Garry", "Enjolras":"Joker",
"Notre-Dame":"Empire State Building", "Victor Hugo":"Victor Hugues",
"Hugo":"Hugues", "Fauchelevent":"Dropwind", "Bouchart":"Butcher",
"Célestine":"Celeste","Mabeuf":"Myoax", "Leblanc":"White",
"Combeferre":"Combiron", "Magloire":"Glory",
"Gillenormand":"Jillnorthman", "France":"États-Unis",
"Paris":"New York", "Louis Philippe":"Andrew Jackson" }'::jsonb ;
```

Copier le contenu sous forme de tableau de caractères dans les autres champs :

```
) j1
) j2;
```

On vérifie le contenu :

```
SELECT * FROM remplacement \gx
```

```
Comparer la performance des deux fonctions suivantes :
  \pset pager off
  -- fonction en PL/perl
  EXPLAIN (ANALYZE, BUFFERS)
  SELECT multi_replace (contenu, (SELECT j FROM remplacement))
  FROM miserables ;
  -- fonction en PL/pgSQL
  EXPLAIN (ANALYZE, BUFFERS)
  SELECT multi_replace (contenu,
                       (SELECT old_t FROM remplacement),
                      (SELECT new_t FROM remplacement) )
  FROM miserables ;
\pset pager off
EXPLAIN (ANALYZE, BUFFERS)
SELECT multi_replace (contenu, (SELECT j FROM remplacement))
FROM miserables ;
EXPLAIN (ANALYZE, BUFFERS)
SELECT multi_replace (contenu,
                      (SELECT old_t FROM remplacement),
                      (SELECT new_t FROM remplacement) )
FROM miserables;
```

Selon les performances de la machine, les résultats peuvent varier, mais la première (en PL/perl) est probablement plus rapide. La fonction en PL/perl montre son intérêt quand il y a beaucoup de substitutions.

#### 3.11.6 hll

- Installer l'extension hll dans la base de données de test :
- le paquet est hll\_17 ou postgresql-17-hll (adapter le numéro à votre version de PostgreSQL) selon la distribution ;
- l'extension se nomme hll;
- elle nécessite d'être préalablement déclarée dans shared\_preload\_libraries.

Sur Rocky Linux et autres dérivés Red Hat :

```
# dnf install hll_17
```

Sur Debian, Ubuntu et dérivés :

```
# apt install postgresql-17-hll
```

Modifier postgresql.conf ainsi afin que la bibliothèque soit préchargée dès le démarrage du serveur :

```
shared_preload_libraries = 'hll'
```

Redémarrer PostgreSQL.

Installer l'extension dans la base :

```
# CREATE EXTENSION hll;
```

- Créer un jeu de données simulant des voyages en transport en commun, par passager selon la date :

```
CREATE TABLE voyages
              bigint GENERATED ALWAYS AS IDENTITY,
(voyage_id
             text,
passager_id
                date
);
INSERT INTO voyages (passager_id, d)
SELECT sem+mod(i, sem+1) | | '- ' | | \mod(i,77777) AS passager_id, d
FROM generate_series (0,51) sem,
      LATERAL
        (SELECT i,
         '2019-01-01'::date + sem * interval '7 days' + i * interval '2s' AS d
      FROM generate_series (1,
        (case when sem in (31,32,33) then 0 else 22 end +abs(30-sem))*5000 ) i
      ) j
;
```

Cette table de 9 millions de voyages étalés de janvier à décembre 2019 pèse 442 Mo.

```
- Activer l'affichage du temps (timing).
```

- Désactiver JIT et le parallélisme.
- Passer la mémoire de tri à 1 Go.
- Précharger la table dans le cache de PostgreSQL.

```
\timing on
SET max_parallel_workers_per_gather TO 0;
SET jit TO off;
SET work_mem TO '1GB';
CREATE EXTENSION pg_prewarm;
SELECT pg_prewarm('voyages');
```

- Calculer, par mois, le nombre exact de voyages et de passagers **distincts**.
- Dans le plan de la requête, chercher où est perdu le temps.

```
SELECT
```

```
date_trunc('month', d)::date AS mois,
   COUNT(*) AS nb_voyages,
   count(DISTINCT passager_id) AS nb_d_passagers_mois
FROM voyages
GROUP BY 1 ORDER BY 1;
```

mois	nb_voyages	nb_d_passagers_mois
2019-01-01	1139599	573853
2019-02-01	930000	560840
2019-03-01	920401	670993
2019-04-01	793199	613376
2019-05-01	781801	655970
2019-06-01	570000	513439
2019-07-01	576399	518478
2019-08-01	183601	179913
2019-09-01	570000	527994
2019-10-01	779599	639944
2019-11-01	795401	728657
2019-12-01	830000	767419
(12 lignes)		

Durée : 57301,383 ms (00:57,301)

Le plan de cette même requête avec EXPLAIN (ANALYZE, BUFFERS) est:

```
QUERY PLAN
```

Le plan est visible sur https://explain.dalibo.com/plan/Hj (pour PostgreSQL 14). Il suppose que shared\_buffers est assez grand pour que tous tous les accès se fassent en mémoire (shared hits). Le work\_mem élevé permet que le tri des 765 Mo soit aussi en mémoire. Le cas est donc idéal. L'essentiel du temps est perdu en tri.

Pour donner une idée de la lourdeur d'un COUNT (DISTINCT) : un décompte non distinct (qui revient à calculer le nombre de voyages) prend sur la même machine 5 secondes, même moins si le parallélisme est utilisé, mais ce qu'un COUNT (DISTINCT) ne permet pas.

- Calculer, pour l'année, le nombre exact de voyages et de passagers distincts.

On a donc plusieurs millions de voyages chaque mois, répartis sur quelques centaines de milliers de passagers mensuels, qui ne totalisent que 4,7 millions de personnes distinctes. Il y a donc un fort turnover tout au long de l'année sans que ce soit un renouvellement complet d'un mois sur l'autre.

- Recompter les passagers dans les deux cas en remplaçant le COUNT(DISTINCT) par cette expression: hll\_cardinality(hll\_add\_agg(hll\_hash\_text(passager\_id)))::int

Les ID des passagers sont hachés, agrégés, et le calcul de cardinalité se fait sur l'ensemble complet.

#### **SELECT**

```
date_trunc('month', d)::date AS mois,
hll_cardinality(hll_add_agg(hll_hash_text(passager_id)))::int
    AS nb_d_passagers_mois

FROM voyages
GROUP BY 1 ORDER BY 1;
mois | nb_d_passagers_mois
```

IIIOTS	nb_d_passagers_mors
2019-01-01	   563372
2019-02-01	553182
2019-03-01	683411
2019-04-01	637927
2019-05-01	670292

```
2019-06-01 | 505151

2019-07-01 | 517140

2019-08-01 | 178431

2019-09-01 | 527655

2019-10-01 | 632810

2019-11-01 | 708418

2019-12-01 | 766208

(12 lignes)
```

Durée: 4556,646 ms (00:04,557)

L'accélération est foudroyante (facteur 10 ici). Les chiffres sont différents, mais très proches (écart souvent inférieur à 1 %, au maximum 2,8 %).

Le plan indique un parcours de table et un agrégat par hachage :

```
Sort (actual time=5374.025..5374.025 rows=12 loops=1)
   Sort Key: ((date_trunc('month'::text, (d)::timestamp with time zone))::date)
   Sort Method: quicksort Memory: 25kB
  Buffers: shared hit=56497
   -> HashAggregate (actual time=5373.793..5374.009 rows=12 loops=1)
         Group Key: (date_trunc('month'::text, (d)::timestamp with time zone))::date
         Buffers: shared hit=56497
         -> Seq Scan on voyages (actual time=0.020..3633.757 rows=8870000 loops=1)
              Buffers: shared hit=56497
Planning Time: 0.122 ms
Execution Time: 5374.062 ms
Pour l'année, on a un résultat similaire :
   hll_cardinality (hll_add_agg (hll_hash_text (passager_id)))::int
    AS nb_d_passagers_annee
FROM voyages;
nb_d_passagers_annee
             4645096
(1 ligne)
Durée: 1461,006 ms (00:01,461)
```

L'écart est de 1,8 % pour une durée réduite d'un facteur 40. Cet écart est-il acceptable pour les besoins applicatifs ? C'est un choix fonctionnel. On peut d'ailleurs agir dessus.

```
    Réexécuter les requêtes après modification du paramétrage de hll :
    SELECT hll_set_defaults(17, 5, -1, 0);
```

```
Les défauts sont: log2m=11, regwidth=5, expthresh=-1, sparseon=1.
```

La requête mensuelle dure à peine plus longtemps (environ 6 s sur la machine de test) pour un écart par rapport à la réalité de l'ordre de 0,02 à 0,6 %.

La requête sur l'année dure environ le même temps pour seulement 0,2 % d'erreur cette fois :

Selon les cas et après des tests soigneux, on testera donc l'intérêt de modifier ces paramètres tels que décrits sur le site du projet : https://github.com/citusdata/postgresql-hll

- Créer une table d'agrégat par mois avec un champ d'agrégat hll et la remplir.

Cette table d'agrégat n'a que 12 lignes mais contient un champ de type hll agrégeant les passager\_id de ce mois. Sa taille n'est que d'1 Mo:

hll=# \d+

Schéma	Nom	Liste des   Type	relations   Propriétaire +	Taille	
public	voyages voyages_mois voyages_voyage_id_seq	table   table   séquence	postgres   postgres   postgres	442 MB   1072 kB   8192 bytes	   

## À partir de cette table d'agrégat :

- calculer le nombre moyen mensuel de passagers distincts,
- recalculer le nombre de passagers distincts sur l'année à partir de cette table d'agrégat.

La fonction pour agréger des champs de type hll est hll\_union\_agg . La requête est donc :

L'extension HyperLogLog permet donc d'utiliser des tables d'agrégat pour un COUNT (DISTINCT). De manière presque instantanée, on retrouve la même estimation presque parfaite que ci-dessus. Il aurait été impossible de la recalculer depuis la table d'agrégat (au contraire de la moyenne par mois, ou d'une somme du nombre de voyages).

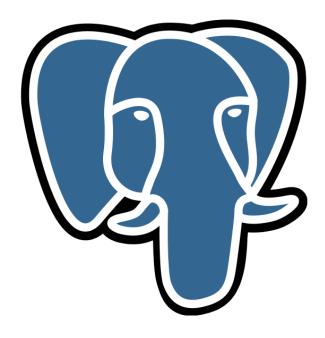
Avec une fonction de fenêtrage sur hll\_union\_agg, calculer une moyenne glissante sur 3 mois du nombre de passagers distincts.

```
SELECT mois,
       nb_exact_passagers_mois,
       CASE WHEN ROW_NUMBER() OVER() > 2 THEN
          hll_cardinality(hll_union_agg(passagers_hll)
                  OVER (ORDER BY mois ASC ROWS 2 PRECEDING) )::bigint
        ELSE null END AS nb_d_passagers_3_mois_glissants
FROM voyages_mois
   mois | nb_exact_passagers_mois | nb_d_passagers_3_mois_glissants
2019-01-01
                             573853
2019-02-01
                             560840
2019-03-01
                             670993
                                                             1463112
2019-04-01
                             613376
                                                             1439444
2019-05-01
                                                             1485437
                             655970
2019-06-01
                             513439
                                                             1368534
2019-07-01
                             518478
                                                             1333368
2019-08-01
                             179913
                                                             1018605
2019-09-01
                             527994
                                                             1057278
2019-10-01
                             639944
                                                             1165308
2019-11-01
                             728657
                                                             1579378
2019-12-01
                             767419
                                                             1741934
(12 lignes)
```

Temps: 78,522 ms

Un COUNT (DISTINCT) avec une fonction de fenêtrage n'est en pratique pas faisable, en tout cas pas aussi aisément, et bien plus lentement.

# 4/ Partitionnement sous PostgreSQL





- Ses principes et intérêts
- Historique
- Les différents types
  - applicatif
  - par héritage (historique)
  - déclaratif (à favoriser)

Le partitionnement est une technique consistant à scinder des grosses tables en plusieurs « partitions », d'abord pour des raisons de place et de maintenance. Dans l'idéal, la table partitionnée reste utilisable en tant que table unique, de manière plus ou moins transparente.

Un partitionnement entièrement géré par le moteur n'existe réellement que depuis la version 10 de PostgreSQL.

Jusqu'à la version 9.6, PostgreSQL ne connaissait que le partitionnement « par héritage », nettement moins flexible et performant. Bien sûr, il existe encore beaucoup d'applications qui « partitionnent » les données elles-mêmes, sans utiliser les mécanismes du moteur.

## 4.1 PRINCIPE & INTÉRÊTS DU PARTITIONNEMENT

## **4.1.1 Pourquoi partitionner?**



- Faciliter la maintenance de gros volumes
  - VACUUM (FULL), réindexation, déplacements, sauvegarde logique...
- Performances
  - parcours complet sur de plus petites tables
  - statistiques par partition plus précises
  - purge par partitions entières
  - pg\_dump parallélisable
  - tablespaces différents (données froides/chaudes)
- Attention à la maintenance sur le code

Maintenir de très grosses tables peut devenir fastidieux, voire impossible: VACUUM FULL trop long, espace disque insuffisant, autovacuum pas assez réactif, réindexation interminable... Il est aussi aberrant de conserver beaucoup de données d'archives dans des tables lourdement sollicitées pour les données récentes.

Le partitionnement consiste à séparer une même table en plusieurs sous-tables (partitions) manipulables en tant que tables à part entière.

#### Maintenance

La maintenance s'effectue sur les partitions plutôt que sur l'ensemble complet des données. En particulier, un VACUUM FULL ou une réindexation peuvent s'effectuer partition par partition, ce qui permet de limiter les interruptions en production, et lisser la charge. pg\_dump ne sait pas paralléliser la sauvegarde d'une table volumineuse et non partitionnée, mais parallélise celle de différentes partitions d'une même table.

C'est aussi un moyen de déplacer une partie des données dans un autre *tablespace* pour des raisons de place, ou pour déporter les parties les moins utilisées de la table vers des disques plus lents et moins chers.

## Parcours complet de partitions

Certaines requêtes (notamment décisionnelles) ramènent tant de lignes, ou ont des critères si complexes, qu'un parcours complet de la table est souvent privilégié par l'optimiseur.

Un partitionnement, souvent par date, permet de ne parcourir qu'une ou quelques partitions au lieu de l'ensemble des données. C'est le rôle de l'optimiseur de choisir la partition (partition pruning), par exemple celle de l'année en cours, ou des mois sélectionnés.

## **Suppression des partitions**

La suppression de données parmi un gros volume peut poser des problèmes d'accès concurrents ou de performance, par exemple dans le cas de purges. En configurant judicieusement les partitions, on peut résoudre cette problématique en supprimant une partition :

```
DROP TABLE nompartition ;
ou en la détachant :
ALTER TABLE table_partitionnee DETACH PARTITION nompartition ;
```

pour l'archiver (et la réattacher au besoin) ou la supprimer ultérieurement.

D'autres optimisations sont décrites dans ce billet de blog d'Adrien Nayrat<sup>1</sup> : statistiques plus précises au niveau d'une partition, réduction plus simple de la fragmentation des index, jointure par rapprochement des partitions...

La principale difficulté d'un système de partitionnement consiste à partitionner avec un impact minimal sur la maintenance du code par rapport à une table classique.

<sup>&</sup>lt;sup>1</sup>https://blog.anayrat.info/2021/09/01/cas-dusages-du-partitionnement-natif-dans-postgresql/

## 4.2 PARTITIONNEMENT APPLICATIF



...ou la réinvention de la roue

## 4.2.1 Partitionnement applicatif



- Gestion au niveau applicatif, table par table
  Complexité pour le développeur
  Intégrité des données ?

L'application peut gérer le partitionnement elle-même, par exemple en créant des tables numérotées par mois, année... Le moteur de PostgreSQL ne voit que des tables classiques et ne peut assurer l'intégrité entre ces données.

C'est au développeur de réinventer la roue : choix de la table, gestion des index... La lecture des données qui concerne plusieurs tables peut devenir délicate.

Ce modèle extrêmement fréquent est bien sûr à éviter.

## 4.3 PARTITIONNEMENT PAR HÉRITAGE



Historique, ou pour un cas très spécifique

## 4.3.1 Partitionnement par héritage



- Table mère:
  - définie normalement, peut contenir des lignes
- Tables filles:
  - héritent des propriétés de la table mère
  - ...mais pas des contraintes, index et droits
  - colonnes supplémentaires possibles
- Insertion applicative ou par trigger (lent!)

Cette technique était assez utilisée, faute de mieux, pour partitionner avant PostgreSQL 10. Ces applications peuvent encore se rencontrer.

#### Principe du partitionnement par héritage :

PostgreSQL permet de créer des tables qui héritent les unes des autres. L'héritage d'une table mère transmet les propriétés suivantes à la table fille :

- les colonnes, avec le type et les valeurs par défaut ;
- les contraintes CHECK.

Les tables filles peuvent ajouter leurs propres colonnes. Par exemple :

```
CREATE TABLE animaux (nom text PRIMARY KEY);
INSERT INTO animaux VALUES ('Éponge');
INSERT INTO animaux VALUES ('Ver de terre');

CREATE TABLE cephalopodes (nb_tentacules integer) INHERITS (animaux);
INSERT INTO cephalopodes VALUES ('Poulpe', 8);

CREATE TABLE vertebres (nb_membres integer) INHERITS (animaux);

CREATE TABLE tetrapodes () INHERITS (vertebres);
```

```
ALTER TABLE ONLY tetrapodes ALTER COLUMN nb_membres SET DEFAULT 4;

CREATE TABLE poissons (eau_douce boolean) INHERITS (tetrapodes);
INSERT INTO poissons (nom, eau_douce) VALUES ('Requin', false);
INSERT INTO poissons (nom, nb_membres, eau_douce) VALUES ('Anguille', 0, false);

La table poissons possède les champs des tables dont elle hérite:

\( \text{d+ poissons} \)

\[ \text{Column} \quad \text{Type} \quad \text{Collation} \quad \text{Nullable} \quad \text{Default} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \
```

On peut créer toute une hiérarchie avec des branches parallèles, chacune avec ses colonnes propres :

```
CREATE TABLE reptiles (venimeux boolean) INHERITS (tetrapodes);
INSERT INTO reptiles VALUES ('Crocodile', 4, false);
INSERT INTO reptiles VALUES ('Cobra', 0, true);
CREATE TABLE mammiferes () INHERITS (tetrapodes);
CREATE TABLE cetartiodactyles (
    cornes boolean,
    bosse boolean
) INHERITS (mammiferes);
INSERT INTO cetartiodactyles VALUES ('Girafe', 4, true, false);
INSERT INTO cetartiodactyles VALUES ('Chameau', 4, false, true);
CREATE TABLE primates (debout boolean) INHERITS (mammiferes);
INSERT INTO primates (nom, debout) VALUES ('Chimpanzé', false);
INSERT INTO primates (nom, debout) VALUES ('Homme', true);
\d+ primates
                   Table "public.primates"
  Column | Type | Collation | Nullable | Default | ...
______

        nom
        | text
        | not null | ...

        nb_membres
        | integer | ...
        | 4
        | ...
```

On remarquera que la clé primaire manque. En effet, l'héritage ne transmet pas :

- les contraintes d'unicité et référentielles ;
- les index:

debout | boolean |

Inherits: mammiferes
Access method: heap

- les droits.

Chaque table possède ses propres lignes :

```
SELECT * FROM poissons;
```

nom	nb_membres	eau_douce
Requin	   4	   f
Anguille	0	f

Par défaut une table affiche aussi le contenu de ses tables filles et les colonnes communes :

```
SELECT * FROM animaux ORDER BY 1;
```

```
nom
Anguille
Chameau
Chimpanzé
Cobra
Crocodile
Éponge
Girafe
Homme
Poulpe
Requin
Ver de terre
```

#### **SELECT** \* **FROM** tetrapodes **ORDER BY** 1 ;

nom	nb_membres
Anguille	0
Chameau	4
Chimpanzé	4
Cobra	0
Crocodile	4
Girafe	4
Homme	4
Requin	4

#### **EXPLAIN SELECT** \* **FROM** tetrapodes **ORDER BY** 1 ;

#### QUERY PLAN

-----

```
Sort (cost=420.67..433.12 rows=4982 width=36)
Sort Key: tetrapodes.nom
```

- -> Append (cost=0.00..114.71 rows=4982 width=36)
  - -> Seq Scan on tetrapodes (cost=0.00..0.00 rows=1 width=36)
  - -> Seq Scan on poissons (cost=0.00..22.50 rows=1250 width=36)
    -> Seq Scan on reptiles (cost=0.00..22.50 rows=1250 width=36)
  - -> Seq Scan on mammiferes (cost=0.00..0.00 rows=1 width=36)
  - -> Seq Scan on cetartiodactyles (cost=0.00..22.30 rows=1230 width=36)
  - -> Seq Scan on primates (cost=0.00..22.50 rows=1250 width=36)

Pour ne consulter que le contenu de la table sans ses filles :

```
SELECT * FROM ONLY animaux ;
```

nom

```
Éponge
Ver de terre
```

#### Limites et problèmes :

En conséquence, on a bien affaire à des tables indépendantes. Rien n'empêche d'avoir des doublons entre la table mère et la table fille. Cela empêche aussi bien sûr la mise en place de clé étrangère, puisqu'une clé étrangère s'appuie sur une contrainte d'unicité de la table référencée. Lors d'une insertion, voire d'une mise à jour, le choix de la table cible se fait par l'application ou un trigger sur la table mère.

Il faut être vigilant à bien recréer les contraintes et index manquants ainsi qu'à attribuer les droits sur les objets de manière adéquate. L'une des erreurs les plus fréquentes est d'oublier de créer les contraintes, index et droits qui n'ont pas été transmis.

Ce type de partitionnement est un héritage des débuts de PostgreSQL, à l'époque de la mode des « bases de donnée objet ». Dans la pratique, dans les versions antérieures à la version 10, l'héritage était utilisé pour mettre en place le partitionnement. La maintenance des index, des contraintes et la nécessité d'un trigger pour aiguiller les insertions vers la bonne table fille, ne facilitaient pas la maintenance. Les performances en écritures étaient bien en-deçà des tables classiques ou du nouveau partitionnement déclaratif.

## Table partitionnée en détournant le partitionnement par héritage :

```
CREATE TABLE t3 (c1 integer, c2 text);
CREATE TABLE t3_1 (CHECK (c1 BETWEEN
                                           0 AND 999999)) INHERITS (t3);
CREATE TABLE t3_2 (CHECK (c1 BETWEEN 1000000 AND 1999999)) INHERITS (t3);
CREATE TABLE t3_3 (CHECK (c1 BETWEEN 2000000 AND 2999999)) INHERITS (t3);
CREATE TABLE t3_4 (CHECK (c1 BETWEEN 3000000 AND 3999999)) INHERITS (t3);
CREATE TABLE t3 5 (CHECK (c1 BETWEEN 4000000 AND 4999999)) INHERITS (t3);
CREATE TABLE t3_6 (CHECK (c1 BETWEEN 5000000 AND 5999999)) INHERITS (t3);
CREATE TABLE t3_7 (CHECK (c1 BETWEEN 6000000 AND 6999999)) INHERITS (t3);
CREATE TABLE t3_8 (CHECK (c1 BETWEEN 7000000 AND 7999999)) INHERITS (t3);
CREATE TABLE t3_9 (CHECK (c1 BETWEEN 8000000 AND 8999999)) INHERITS (t3);
CREATE TABLE t3_0 (CHECK (c1 BETWEEN 9000000 AND 9999999)) INHERITS (t3);
-- Fonction et trigger de répartition pour les insertions :
CREATE OR REPLACE FUNCTION insert_into() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.c1
               BETWEEN
                             0 AND 999999 THEN
    INSERT INTO t3_1 VALUES (NEW.*);
  ELSIF NEW.cl BETWEEN 1000000 AND 1999999 THEN
    INSERT INTO t3_2 VALUES (NEW.*);
  ELSIF NEW.cl BETWEEN 2000000 AND 2999999 THEN
    INSERT INTO t3_3 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 3000000 AND 3999999 THEN
    INSERT INTO t3 4 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 4000000 AND 4999999 THEN
    INSERT INTO t3_5 VALUES (NEW.*);
  ELSIF NEW.cl BETWEEN 5000000 AND 5999999 THEN
    INSERT INTO t3_6 VALUES (NEW.*);
```

```
ELSIF NEW.c1 BETWEEN 6000000 AND 6999999 THEN
    INSERT INTO t3_7 VALUES (NEW.*);
ELSIF NEW.c1 BETWEEN 7000000 AND 7999999 THEN
    INSERT INTO t3_8 VALUES (NEW.*);
ELSIF NEW.c1 BETWEEN 8000000 AND 8999999 THEN
    INSERT INTO t3_9 VALUES (NEW.*);
ELSIF NEW.c1 BETWEEN 9000000 AND 9999999 THEN
    INSERT INTO t3_0 VALUES (NEW.*);
END IF;
RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_insert_t3
BEFORE INSERT ON t3 FOR EACH ROW
EXECUTE PROCEDURE insert_into();
```

Noter qu'il reste encore à gérer les mises à jour de lignes... À cause de ce trigger, le temps d'insertion peut être allègrement multiplié par huit ou dix par rapport à une insertion dans une table normale ou dans une table avec le partitionnement déclaratif moderne.

La même table en partitionnement déclaratif par liste est :

```
CREATE TABLE t2 (c1 integer, c2 text) PARTITION BY RANGE (c1);

CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM ( 0) TO ( 1000000);

CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (1000000) TO ( 2000000);

CREATE TABLE t2_3 PARTITION OF t2 FOR VALUES FROM (2000000) TO ( 3000000);

CREATE TABLE t2_4 PARTITION OF t2 FOR VALUES FROM (3000000) TO ( 4000000);

CREATE TABLE t2_5 PARTITION OF t2 FOR VALUES FROM (4000000) TO ( 5000000);

CREATE TABLE t2_6 PARTITION OF t2 FOR VALUES FROM (5000000) TO ( 6000000);

CREATE TABLE t2_7 PARTITION OF t2 FOR VALUES FROM (6000000) TO ( 70000000);

CREATE TABLE t2_8 PARTITION OF t2 FOR VALUES FROM (7000000) TO ( 80000000);

CREATE TABLE t2_9 PARTITION OF t2 FOR VALUES FROM (8000000) TO ( 90000000);

CREATE TABLE t2_9 PARTITION OF t2 FOR VALUES FROM (8000000) TO ( 100000000);

CREATE TABLE t2_0 PARTITION OF t2 FOR VALUES FROM (9000000) TO (100000000);
```



Si le partitionnement par héritage fonctionne toujours sur les versions récentes de PostgreSQL, il est déconseillé pour les nouveaux développements.

## 4.4 PARTITIONNEMENT DÉCLARATIF

#### 4.4.1 Partitionnement déclaratif



- Mise en place et administration simplifiées, intégrées au moteur
- Gestion automatique des lectures et écritures
  - et rapide
- Partitions
  - attacher/détacher une partition
  - contrainte implicite de partitionnement
  - expression possible pour la clé de partitionnement
  - sous-partitions possibles
  - partition par défaut

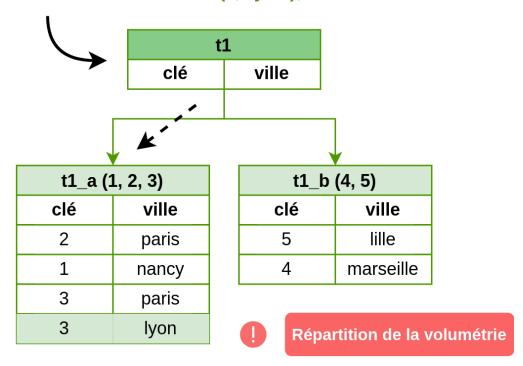
Le partitionnement déclaratif est le système à privilégier de nos jours. Apparu en version 10, il a été grandement amélioré dans les versions 11 à 13, en fonctionnalités comme en performances. Des améliorations continuent à arriver à chaque version : préférez les versions récentes de PostgreSQL. Son but est de permettre une mise en place et une administration simples des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL, comme CREATE TABLE et ALTER TABLE, pour attacher (ATTACH PARTITION) et détacher des partitions (DETACH PARTITION).

Grosse simplification par rapport à l'ancien partitionnement par héritage : il n'est pas nécessaire d'ajouter des *triggers* pour gérer les insertions et mises à jour. Le routage est géré de façon automatique en fonction de la définition des partitions, au besoin vers une partition par défaut, et sans pénalité notable en performances.

Contrairement au partitionnement par héritage, la table partitionnée ne contient pas elle-même de ligne, ce n'est qu'une coquille vide.

## 4.4.2 Partitionnement par liste

## INSERT INTO t1 VALUES (3, 'lyon');



## 4.4.3 Partitionnement par liste: implémentation



- Liste de valeurs par partition
  - statut, client, pays, année ou mois...
- Clé de partitionnement forcément mono-colonne
- Syntaxe:

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);

CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
...
```

Il est possible de partitionner une table par valeurs. Ce type de partitionnement fonctionne

uniquement avec une clé de partitionnement mono-colonne (on verra qu'il est possible de souspartitionner). Il faut que le nombre de valeurs soit assez faible pour être listé explicitement. Le partitionnement par liste est adapté par exemple au partitionnement par :

- type d'un objet;
- client final (si peu de clients);
- géographie : pays, entité commerciale, entrepôt...;
- statut d'un objet, pour isoler les données froides dans leur partition ;
- année ou mois (ne pas oublier de créer régulièrement de nouvelles partitions) pour faciliter les purges.

Voici un exemple de création d'une table partitionnée par liste et de ses partitions :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);

CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
```

Les noms des partitions sont à définir par l'utilisateur, il n'y a pas d'automatisme ni de convention particulière.

Lors de l'insertion, les données sont correctement redirigées vers leur partition, comme le montre cette requête :

Il est aussi possible d'interroger directement une partition (ici t1\_a et non t1):

```
SELECT * FROM t1_a ;
c1 | c2
---+---
1 |
2 |
```

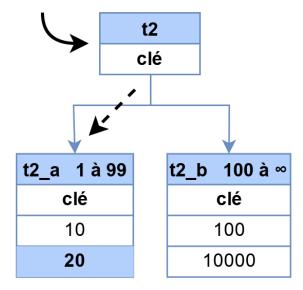
Si aucune partition correspondant à la clé insérée n'est trouvée et qu'aucune partition par défaut n'est déclarée, une erreur se produit.

```
INSERT INTO t1 VALUES (0);
ERROR: no PARTITION OF relation "t1" found for row
DETAIL: Partition key of the failing row contains (c1) = (0).
INSERT INTO t1 VALUES (6);
ERROR: no PARTITION OF relation "t1" found for row
DETAIL: Partition key of the failing row contains (c1) = (6).
```

Si la clé de partitionnement d'une ligne est modifiée par un UPDATE, la ligne change automatiquement de partition (sauf en version 10, où ce n'est pas implémenté, et l'on obtient une erreur).

## 4.4.4 Partitionnement par intervalle

## **INSERT INTO t2 VALUES (20);**



Pépartition de la volumétrie

## 4.4.5 Partitionnement par intervalle: implémentation



- Clé de partitionnement mono- ou multicolonne
  - dates, id...
- Bornes:
  - supérieure exclue
  - MINVALUE / MAXVALUE pour infinis

```
- Syntaxe:
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
```

Le partitionnement par intervalle est très courant quand il y a de nombreuses valeurs différentes de la clé de partitionnement, ou qu'elle doit être multicolonne, par exemple :

- selon une tranche de dates, notamment pour faciliter des purges ;
- selon des plages d'identifiants d'objets : client final, produit...

Voici un exemple de création de la table partitionnée et de deux partitions :

```
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2 1 PARTITION OF t2 FOR VALUES FROM (1) to (100);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
```

Le MAXVALUE indique la valeur maximale du type de données : t2\_2 acceptera donc tous les entiers supérieurs ou égaux à 100.



Noter que les bornes supérieures des partitions sont exclues! La valeur 100 ira donc dans la seconde partition.

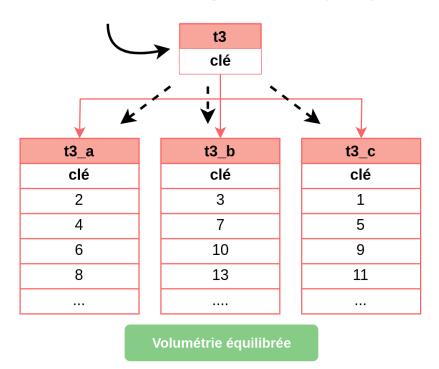
Lors de l'insertion, les données sont redirigées vers leur partition, s'il y en a une :

```
INSERT INTO t2 VALUES (0);
ERROR: no PARTITION OF relation "t2" found for row
DETAIL: Partition key of the failing row contains (c1) = (0).
INSERT INTO t2 VALUES (10, 'dix');
```

```
INSERT INTO t2 VALUES (100, 'cent');
INSERT INTO t2 VALUES (10000, 'dix mille');
SELECT * FROM t2;
 c1 | c2
  10 | dix
  100 | cent
10000 | dix mille
(3 lignes)
SELECT * FROM t2_2 ;
 c1 | c2
 100 | cent
10000 | dix mille
(2 lignes)
La colonne système tableoid permet de connaître la partition d'où provient une ligne :
SELECT ctid, tableoid::regclass, * FROM t2;
ctid | tableoid | c1 | c2
```

## 4.4.6 Partitionnement par hachage

## INSERT INTO t3 SELECT generate\_series(1, 100);



## 4.4.7 Partitionnement par hachage: principe



Pour une répartition uniforme des données :

- Hachage de valeurs par partition
  - indiquer un modulo et un reste
- Clé de partitionnement mono- ou multicolonnes
- Syntaxe:

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);

CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 0);

CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 1);

CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 2);
```

Si la clé de partitionnement n'est pas évidente et que le besoin est surtout de répartir la volumétrie

en partitions de tailles équivalentes, le partitionnement par hachage est adapté. Voici comment partitionner par hachage une table en trois partitions :

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);
CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 0);
CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 1);
CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 2);
```

Une grosse insertion de données répartira les données de manière équitable entre les différentes partitions :

## 4.4.8 Clé de partitionnement multicolonne



- Clé sur plusieurs colonnes :
  - si partitionnement par intervalle ou hash (pas par liste)
  - et si 1er champ toujours présent
- Syntaxe:

```
CREATE TABLE t1(c1 integer, c2 text, c3 date)
PARTITION BY RANGE (c1, c3);

CREATE TABLE t1_a PARTITION OF t1
FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11');
...
```

Avec le partitionnement par intervalle, il est possible de créer les partitions en utilisant plusieurs colonnes. On profitera de l'exemple ci-dessous pour montrer l'utilisation conjointe de tablespaces différents. Commençons par créer les tablespaces :

```
CREATE TABLESPACE ts0 LOCATION '/tablespaces/ts0';
CREATE TABLESPACE ts1 LOCATION '/tablespaces/ts1';
CREATE TABLESPACE ts2 LOCATION '/tablespaces/ts2';
CREATE TABLESPACE ts3 LOCATION '/tablespaces/ts3';
```

Créons maintenant la table partitionnée et deux partitions :

CREATE TABLE t2(c1 integer, c2 text, c3 date not null)

```
PARTITION BY RANGE (c1, c3);
CREATE TABLE t2_1 PARTITION OF t2
       FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11')
       TABLESPACE ts1;
CREATE TABLE t2_2 PARTITION OF t2
       FOR VALUES FROM (100, '2017-08-11') TO (200, '2017-08-12')
       TABLESPACE ts2;
La borne supérieure étant exclue, la valeur (100, '2017-08-11') fera donc partie de la seconde
partition. Si les valeurs sont bien comprises dans les bornes, tout va bien :
INSERT INTO t2 VALUES (1, 'test', '2017-08-10');
INSERT INTO t2 VALUES (150, 'test2', '2017-08-11');
Mais si la valeur pour c1 est trop petite :
INSERT INTO t2 VALUES (0, 'test', '2017-08-10');
ERROR: no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (0, 2017-08-10).
De même, si la valeur pour c3 (colonne de type date) est antérieure :
INSERT INTO t2 VALUES (1, 'test', '2017-08-09');
ERROR: no partition of relation "t2" found for row
DÉTAIL: Partition key of the failing row contains (c1, c3) = (1, 2017-08-09).
Les valeurs spéciales MINVALUE et MAXVALUE permettent de ne pas indiquer de valeur de seuil li-
mite. Les partitions t2_0 et t2_3 pourront par exemple être déclarées comme suit et permettront
d'insérer les lignes qui étaient ci-dessus en erreur.
CREATE TABLE t2 0 PARTITION OF t2
       FOR VALUES FROM (MINVALUE, MINVALUE) TO (1, '2017-08-10')
       TABLESPACE ts0;
CREATE TABLE t2_3 PARTITION OF t2
       FOR VALUES FROM (200, '2017-08-12') TO (MAXVALUE, MAXVALUE)
       TABLESPACE ts3;
Enfin, on peut consulter la table pg_class afin de vérifier la présence des différentes partitions :
ANALYZE t2;
SELECT relname, relispartition, relkind, reltuples
FROM pg_class WHERE relname LIKE 't2%';
relname | relispartition | relkind | reltuples
 t2 | f
                           | p
                                                0
 t2_0
t2_1
         | t
                           | r
                                                2
         l t
                           | r
                                                1
 t2_2
         | t
                           | r
         | t
 t2_3
```

#### Performances:

Si le premier champ de la clé de partitionnement n'est pas fourni, il semble que l'optimiseur ne sache pas cibler correctement les partitions. Il balaiera toutes les partitions. Ce peut être gênant si ce premier champ n'est pas systématiquement présent dans les requêtes.

Le sous-partitionnement est une alternative à étudier, également plus souple.

Il faut faire attention à ce que le nombre de combinaisons possibles ne mène pas à trop de partitions.

## 4.4.9 Sous-partitionnement



S'il y a deux chemins d'accès privilégiés :

```
CREATE TABLE objets (id int, statut int, annee int, t text)
PARTITION BY LIST (statut);

CREATE TABLE objets_123
PARTITION OF objets FOR VALUES IN (1, 2, 3)
PARTITION BY LIST (annee);

CREATE TABLE objets_123_2023
PARTITION OF objets_123 FOR VALUES IN (2023);
CREATE TABLE objets_123_2024
PARTITION OF objets_123 FOR VALUES IN (2024);

CREATE TABLE objets_45
PARTITION OF objets FOR VALUES IN (4,5);
```

- Plus souple que le partitionnement multicolonne

#### Principe:

Les partitions sont des tables à part entière, qui peuvent donc être elles-mêmes partitionnées. Ce peut être utile si les requêtes alternent entre deux schémas d'accès.

#### Exemple:

L'exemple ci-dessus crée deux partitions selon statut (objets\_123 et objets\_45). La première partition est elle-même sous-partitionnée par année (objets\_123\_2023 et objets\_123\_2024). Cela permet par exemple, de faciliter la purge des données ou d'accélérer le temps de traitement si l'on requête sur une année entière. Il n'a pas été jugé nécessaire de sous-partitionner la seconde partition objets\_45 (par exemple parce qu'elle est petite).

```
\dt objets*

Liste des relations

Schéma | Nom | Type | Propriétaire
```

Il n'est pas obligatoire de sous-partitionner avec la même technique (liste, intervalle, hachage...) que le partitionnement de plus haut niveau.

Il n'y a pas besoin de fournir la première clé de partitionnement pour que les sous-partitions soient directement accessibles :

Fournir uniquement la première clé de partitionnement entraînera le parcours de toutes les souspartitions concernées :

Bien sûr, l'idéal est de fournir les deux clés de partitionnement pour n'accéder qu'à une partition :

Cette fonctionnalité peut être ponctuellement utile, mais il ne faut pas en abuser en raison de la complexité supplémentaire. Toutes les clés de partitionnement devront se retrouver dans les clés primaires techniques des tables. Le nombre de partitions peut devenir très important.

## Comparaison avec le partitionnement multicolonne :

Le partitionnement multicolonne est conceptuellement plus simple. Pour un même besoin, le nombre de partitions est identique. Mais le sous-partitionnement est plus souple :

- mélange possible des types de partitionnement (par exemple un partitionnement par liste souspartitionné par intervalle de dates);
- type de sous-partitionnement différent possible selon les partitions.

## 4.4.10 Partition par défaut



- Pour le partitionnement par liste ou par intervalle
- Toutes les données n'allant pas dans les partitions définies iront dans la partition par défaut

```
CREATE TABLE t2_autres PARTITION OF t2 DEFAULT ;
```

- La conserver petite

Ajouter une partition par défaut permet de ne plus avoir d'erreur au cas où une partition n'est pas définie. Par exemple :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);
CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
INSERT INTO t1 VALUES (0);
ERROR: no PARTITION OF relation "t1" found for row
DETAIL: Partition key of the failing row contains (c1) = (0).
INSERT INTO t1 VALUES (6);
ERROR: no PARTITION OF relation "t1" found for row
DETAIL: Partition key of the failing row contains (c1) = (6).
-- partition par défaut
CREATE TABLE t1_defaut PARTITION OF t1 DEFAULT ;
-- on réessaie l'insertion
INSERT INTO t1 VALUES (0);
INSERT INTO t1 VALUES (6);
SELECT tableoid::regclass, * FROM t1;
 tableoid | c1 | c2
 t1_a | 1 |
          | 2 |
 t1_a
 t1_b | 5 |
 t1_defaut | 0 |
 t1_defaut | 6 |
```

Comme la partition par défaut risque d'être parcourue intégralement à chaque ajout d'une nouvelle partition, il vaut mieux la garder de petite taille.

Un partitionnement par hachage ne peut posséder de table par défaut puisque les données sont forcément aiguillées vers une partition ou une autre.

## 4.4.11 Attacher une partition



ALTER TABLE ... ATTACH PARTITION ... FOR VALUES ... ;

- La table doit préexister
- Vérification du respect de la contrainte par les données existantes
  - parcours complet de la table : lent!
  - ...sauf si contrainte CHECK identique déjà ajoutée
- Si la partition par défaut a des données qui iraient dans cette partition :
  - erreur à l'ajout de la nouvelle partition
  - détacher la partition par défaut
  - ajouter la nouvelle partition
  - déplacer les données de l'ancienne partition par défaut
  - ré-attacher la partition par défaut

Ajouter une table comme partition d'une table partitionnée est possible mais cela nécessite de vérifier que la contrainte de partitionnement est valide pour toute la table attachée, et que la partition par défaut ne contient pas de données qui devraient figurer dans cette nouvelle partition. Cela résulte en un parcours complet de la table attachée, et de la partition par défaut si elle existe, ce qui sera d'autant plus lent qu'elles sont volumineuses.

Ce peut être très coûteux en disque, mais le plus gros problème est la durée du verrou sur la table partitionnée, pendant toute cette opération. Il est donc conseillé d'ajouter une contrainte adéquate avant l'ATTACH: la durée du verrou sera raccourcie d'autant. Pour que cette contrainte soit prise en compte, il faut faire attention aux bornes exactes en partitionnement par intervalle, et préciser IS NOT NULL si le champ n'est pas NOT NULL.

Si des lignes pour cette nouvelle partition figurent déjà dans la partition par défaut, des opérations supplémentaires sont à réaliser pour les déplacer. Ce n'est pas automatique.

#### Exemple:

```
\set ECHO all \set timing off
```

```
DROP TABLE IF EXISTS t1;
-- Une table partitionnée avec partition par défaut
CREATE TABLE t1 (c1 integer, filler char (10))
PARTITION BY LIST (c1);
CREATE TABLE t1_123 PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_45 PARTITION OF t1 FOR VALUES IN (4, 5);
CREATE TABLE t1_default PARTITION OF t1 DEFAULT ;
-- Données d'origine
INSERT INTO t1 SELECT 1+mod(i,5) FROM generate_series (1,5000000) i;
-- Les données sont bien dans les partitions t1_123 et t1_45
SELECT tableoid::regclass, c1, count(*) FROM t1
GROUP BY 1,2 ORDER BY c1;
-- Création d'une table pour les valeurs 6 à attacher
CREATE TABLE t1_6 (LIKE t1 INCLUDING ALL) ;
INSERT INTO t1_6 SELECT 6 FROM generate_series (1,1000000);
\dt+ t1*
\timing on
 - on attache la table : elle est scannée, ce qui est long
ALTER TABLE t1 ATTACH PARTITION t1 6 FOR VALUES IN (6);
-- noter la nouvelle contrainte sur la table
\d+ t1_6
-- on la détache, la contrainte a disparu
ALTER TABLE t1 DETACH PARTITION t1_6;
\d+ t1_6
-- on remet manuellement la même contrainte que ci-dessus
-- (ce qui reste long mais ne pose pas de verrou sur t1)
ALTER TABLE t1_6 ADD CONSTRAINT t1_6_ck CHECK (c1 IS NOT NULL AND c1 = 6);
\d+ t1 6
 - l'ATTACH est cette fois presque instantané
ALTER TABLE t1 ATTACH PARTITION t1_6 FOR VALUES IN (6);
\timing off
-- On insère par erreur des valeurs 7 sans avoir fait la partition
-- (et sans avoir le droit de les enlever de t1 ensuite)
INSERT INTO t1 SELECT 7 FROM generate_series (1,100);
-- Plus tard, créer la partition pour les valeurs 7 échoue avec l'erreur
-- "constraint for default partition "t1_default" would be violated""
CREATE TABLE t1_7 PARTITION OF t1 FOR VALUES IN (7);
-- Pour corriger cela, au sein d'une transaction,
-- on transfére les données de la partition par défaut
-- vers une nouvelle table qui est ensuite attachée
CREATE TABLE t1_7 (LIKE t1 INCLUDING ALL) ;
ALTER TABLE t1_7 ADD CONSTRAINT t1_7_ck CHECK (c1 IS NOT NULL AND c1 = 7);
BEGIN;
    INSERT INTO t1_7
    SELECT * FROM t1_default WHERE c1=7 ;
    DELETE FROM t1_default WHERE c1=7 ;
```

```
ALTER TABLE t1 ATTACH PARTITION t1_7 FOR VALUES IN (7);
COMMIT;
```

### 4.4.12 Détacher une partition



```
ALTER TABLE ... DETACH PARTITION ...
```

- Simple et rapideMais nécessite un verrou exclusif
  - option CONCURRENTLY (v14+)

Détacher une partition est beaucoup plus rapide qu'en attacher une. En effet, il n'est pas nécessaire de procéder à des vérifications sur les données des partitions.

Cependant, il reste nécessaire d'acquérir un verrou exclusif sur la table partitionnée, ce qui peut prendre du temps si des transactions sont en cours d'exécution. L'option CONCURRENTLY (à partir de PostgreSQL 14) mitige le problème malgré quelques restrictions<sup>2</sup>, notamment : pas d'utilisation dans une transaction, incompatibilité avec la présence d'une partition par défaut, et nécessité d'une commande FINALIZE si l'ordre a échoué ou été interrompu.

La partition détachée devient alors une table tout à fait classique. Elle perd la contrainte de partitionnement, mais elle conserve les autres contraintes, les index, etc. hérités de la table partitionnée originale, ou directement posés sur la table quand elle était une partition, voire posés avant que la table soit attachée comme partition à la table partitionnée.

#### 4.4.13 Changer la définition d'une partition



- Si pas de conflit entre partitions :

```
ALTER TABLE ... DETACH PARTITION ... CONCURRENTLY ;
  ALTER TABLE ... ATTACH PARTITION ... FOR <nouveau critère> ;
END ;
```

- Sinon : déplacement de données manuel

<sup>&</sup>lt;sup>2</sup>https://docs.postgresql.fr/current/sql-altertable.html#SQL-ALTERTABLE-DETACH-PARTITION

Modifier la définition d'une partition n'est pas directement prévu. S'il s'agit d'élargir une partition (liste ou intervalle), sans avoir à gérer un recouvrement avec une autre partition, il est possible de détacher la partition, puis de la réattacher avec sa nouvelle définition, le tout au sein d'une partition. La re-vérification des données prendra cependant du temps. L'astuce du CHECK redondant avec la contrainte de partitionnement, posé avant le DETACH, fonctionne là aussi et réduit la durée du verrou.

S'il faut couper une partition en deux (split), ou fusionner des partitions (merge), il faudra migrer les données manuellement, si possible dans une seule transaction. (La fonctionnalité était prévue pour PostgreSQL 17 mais a été retirée peu avant la sortie.)

# 4.4.14 Supprimer une partition



Une partition étant une table, supprimer la table revient à supprimer la partition, en détruisant bien sûr les données qu'elle contient. Il n'y a pas besoin de détacher la table auparavant. L'opération est simple et rapide, mais demande un verrou exclusif.

Il est fréquent de partitionner par date pour profiter de cette facilité dans la purge des vieilles données. La durée en est énormément réduite, ainsi que les écritures de journaux puisqu'il n'y a plus de DELETE.

## 4.4.15 Fonctions de gestion et vues système



- Sous psql: \dP
   pg\_partition\_tree ('logs') : liste entière des partitions
   pg\_partition\_root ('logs\_2019') : racine d'une partition
   pg\_partition\_ancestors ('logs\_201901') : parents d'une partition

Voici le jeu de tests pour l'exemple qui suivra. Il illustre également l'utilisation de sous-partitions (ici sur la même clé, mais cela n'a rien d'obligatoire).

```
-- Table partitionnée
CREATE TABLE logs (dreception timestamptz, contenu text) PARTITION BY

¬ RANGE(dreception);

-- Partition 2018, elle-même partitionnée
```

```
CREATE TABLE logs_2018 PARTITION OF logs FOR VALUES FROM ('2018-01-01') TO

    ('2019-01-01')

                      PARTITION BY range(dreception);
-- Sous-partitions 2018
CREATE TABLE logs_201801 PARTITION OF logs_2018 FOR VALUES FROM ('2018-01-01') TO

    ('2018-02-01');

CREATE TABLE logs_201802 PARTITION OF logs_2018 FOR VALUES FROM ('2018-02-01') TO
-- Idem en 2019
CREATE TABLE logs_2019 PARTITION OF logs FOR VALUES FROM ('2019-01-01') TO

    ('2020-01-01')

                      PARTITION BY range(dreception);
CREATE TABLE logs_201901 PARTITION OF logs_2019 FOR VALUES FROM ('2019-01-01') TO

    ('2019-02-01');

Et voici le test des différentes fonctions :
SELECT pg_partition_root('logs_2019');
pg_partition_root
 logs
SELECT pg_partition_root('logs_201901');
pg_partition_root
______
logs
SELECT pg_partition_ancestors('logs_2018');
pg_partition_ancestors
 logs_2018
 logs
SELECT pg_partition_ancestors('logs_201901');
 pg_partition_ancestors
 logs_201901
 logs_2019
SELECT * FROM pg_partition_tree('logs');
   relid | parentrelid | isleaf | level
                         | f
 logs
                                       0
                         | f
 logs_2018 | logs
                                         1
 logs_2019 | logs
                         | f
                                         1
 logs_201801 | logs_2018 | t
                                         2
 logs_201802 | logs_2018 | t
 logs_201901 | logs_2019 | t
```

Noter les propriétés de « feuille » (leaf) et le niveau de profondeur dans le partitionnement.

Sous psql, \d affichera toutes les tables, partitions comprises, ce qui peut vite encombrer l'affichage. \dP affiche uniquement les tables et index partitionnés :

=# \dP

Schéma	iste des relations   Propriétaire   	•	Table
public public	 1 1 0 1	table partitionnée index partitionné	1

La table système pg\_partitioned\_table 3 permet des requêtes plus complexes. Le champ pg\_class.relpartbound 4 contient les définitions des clés de partitionnement.



Pour masquer les partitions dans certains outils, il peut être intéressant de déclarer les partitions dans un schéma différent de la table principale.

Dans un cadre « multitenant » avec de nombreux schémas, et des partitions de même noms chacune dans son schéma, positionner search\_path permet de sélectionner implicitement la partition, facilitant la vie au développeur ou permettant de « mentir » à l'application.

# 4.4.16 Clé primaire et clé de partitionnement



La clé primaire doit contenir toutes les colonnes de la clé de partitionnement.

- Idem pour une contrainte unique
- Pas un problème si on partitionne selon la cléPlus gênant dans d'autres cas (date, statut...)

Le partitionnement impose une contrainte importante sur la modélisation : la clé de partitionnement doit impérativement faire partie de la clé primaire (ainsi que des contraintes et index uniques). En effet, PostgreSQL ne maintient pas d'index global couvrant toutes les partitions. Il ne peut donc garantir l'unicité d'un champ qu'au sein de chaque partition.

Dans beaucoup de cas cela ne posera pas de problème, notamment si on partitionne justement sur tout ou partie de cette clé primaire. Dans d'autres cas, c'est plus gênant. Si la vraie clé primaire est un identifiant géré par la base à l'insertion (serial ou IDENTITY), le risque reste limité. Mais avec des identifiants générés côté applicatif, il y a un risque d'introduire un doublon. Dans le cas où les valeurs de la clé de partitionnement ne sont pas une simple constante (par exemple des dates au lieu d'une

<sup>&</sup>lt;sup>3</sup>https://www.postgresql.org/docs/current/catalog-pg-partitioned-table.html

<sup>&</sup>lt;sup>4</sup>https://www.postgresql.org/docs/current/catalog-pg-class.html

seule année), le problème peut être mitigé en ajoutant une contrainte unique directement sur chaque partition, garantissant l'unicité de la clé primaire réelle au moins au sein de la partition.

Une solution générale est de créer une autre table non partitionnée avec la clé primaire réelle, et une contrainte vers cette table depuis la table partitionnée. Conceptuellement, cela est équivalent à ne pas partitionner une grosse table mais à en « sortir » les données dans une sous-table partitionnée portant une contrainte.

#### Exemple:

```
-- On voudrait partitionner ainsi mais le moteur refuse
CREATE TABLE factures_p
                   (id bigint PRIMARY KEY,
                   d timestamptz,
                   id_client int NOT NULL,
                   montant_c int NOT NULL DEFAULT 0)
PARTITION BY RANGE (d);
ERROR: unique constraint on partitioned table must include all partitioning columns
DÉTAIL: PRIMARY KEY constraint on table "factures_p" lacks column "d" which is part
\,\,\,\,\,\,\,\,\,\,\,\,\,\, of the partition key.
-- On se rabat sur une clé primaire incluant la date
CREATE TABLE factures_p
                   (id bigint NOT NULL,
                   d timestamptz NOT NULL,
                   id_client int,
                   montant_c int,
                   PRIMARY KEY (id, d)
PARTITION BY RANGE (d);
CREATE TABLE factures_p_202310 PARTITION OF factures_p
  FOR VALUES FROM ('2023-10-01') TO ('2023-11-01');
CREATE TABLE factures_p_202311 PARTITION OF factures_p
  FOR VALUES FROM ('2023-11-01') TO ('2023-12-01');
ALTER TABLE factures_p_202310 ADD CONSTRAINT factures_p_202310_uq UNIQUE (id);
-- Ces contraintes sécurisent les clés primaire au niveau partition
ALTER TABLE factures_p_202311 ADD CONSTRAINT factures_p_202311_uq UNIQUE (id);
-- Ajout de quelques lignes de 1 à 5 sur les deux partitions
INSERT INTO factures_p (id, d, id_client)
SELECT i, '2023-10-26'::timestamptz+i*interval '2 days', 42 FROM generate_series
\hookrightarrow (1,5) i;
-- Ce doublon est accepté car les deux valeurs 3 ne sont pas dans la même partition
INSERT INTO factures_p (id, d, id_client)
SELECT 3, '2023-11-01'::timestamptz-interval '1s', 42;
-- Vérification que 3 est en double
SELECT tableoid::regclass, id, d FROM factures_p ORDER BY id ;
ROLLBACK;
-- Cette table permet de garantir l'unicité dans toutes les partitions
CREATE TABLE factures_ref (id bigint NOT NULL PRIMARY KEY,
                       d timestamptz NOT NULL,
                       UNIQUE (id,d) -- nécessaire pour la contrainte
```

```
) ;
INSERT INTO factures_ref SELECT id,d FROM factures_p ;
-- Contrainte depuis la table partitionnée
ALTER TABLE factures_p ADD CONSTRAINT factures_p_id_fk
FOREIGN KEY (id, d) REFERENCES factures_ref (id,d);
-- Par la suite, il faut insérer chaque nouvelle valeur de `id`
-- dans les deux tables
-- Ce doublon est à présent correctement rejeté :
WITH ins AS (
   INSERT INTO factures_p (id, d, id_client)
    SELECT 3, '2023-11-01'::timestamptz-interval '1s', 42
    RETURNING id,d )
INSERT INTO factures_ref
SELECT id, d FROM ins;
ERROR: duplicate key value violates unique constraint "factures_ref_pkey"
DÉTAIL: Key (id)=(3) already exists.
```

#### 4.4.17 Indexation



- Propagation automatique
- Index supplémentaires par partition possibles
- Clés étrangères entre tables partitionnées

Les index sont propagés de la table mère aux partitions : tout index créé sur la table partitionnée sera automatiquement créé sur les partitions existantes. Toute nouvelle partition disposera des index de la table partitionnée. La suppression d'un index se fait sur la table partitionnée et concernera toutes les partitions. Il n'est pas possible de supprimer un tel index d'une seule partition.

Gérer des index manuellement sur certaines partitions est possible. Par exemple, on peut n'avoir besoin de certains index que sur les partitions de données récentes, et ne pas les créer sur des partitions de données d'archives.

Une clé primaire ou unique peut exister sur une table partitionnée (mais elle devra contenir toutes les colonnes de la clé de partitionnement) ; ainsi qu'une clé étrangère d'une table partitionnée *vers* une table normale.

Depuis PostgreSQL 12, il est possible de créer une clé étrangère *vers* une table partitionnée de la même manière qu'entre deux tables normales. Par exemple, si les tables ventes et lignes\_ventes sont toutes deux partitionnées :

```
ALTER TABLE lignes_ventes
ADD CONSTRAINT lignes_ventes_ventes_fk
FOREIGN KEY (vente_id) REFERENCES ventes (vente_id);
```

Noter que les versions 10 et 11 possèdent des limites sur ces fonctionnalités, que l'on peut souvent contourner en créant index et contraintes manuellement sur chaque partition.

## 4.4.18 Planification & performances



- Mettre la clé dans la requête autant que possible
- ou : cibler les partitions directement
- Temps de planification
  - nombre de tables, d'index, leurs statistisques...
  - max ~ 100 partitions
- À activer?
  - enable\_partitionwise\_aggregate
  - enable\_partitionwise\_join

## Cibler la partition par la clé :

Si la clé de partitionnement n'est pas fournie, l'exécution concernera toutes les partitions, qu'elles soient accédées par *Seq Scan* ou *Index Scan*.



Autant que possible, le développeur doit fournir la clé de partitionnement dans chaque requête, et le plan ciblera directement la bonne partition.

Comme avec les index, il faut vérifier que la clé est bien claire pour PostgreSQL. Si ce n'est pas le cas toutes les partitions seront lues :

#### **Partition pruning:**

Dans le cas où la clé de partitionnement dépend du résultat d'un calcul, d'une sous-requête ou d'une jointure, PostgreSQL prévoit un plan concernant toutes les partitions, mais élaguera à l'exécution les appels aux partitions non concernées. Ci-dessous, seule pgbench\_accounts\_8 est interrogé (et ce peut être une autre partition si l'on répète la requête):

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT * FROM pgbench_accounts WHERE aid = (SELECT (random()*1000000)::int ) ;
                                   QUERY PLAN
Append (actual time=23.083..23.101 rows=1 loops=1)
  InitPlan 1 (returns $0)
    -> Result (actual time=0.001..0.002 rows=1 loops=1)
   -> Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 (never
⇔ executed)
         Index Cond: (aid = $0)
   -> Index Scan using pgbench_accounts_2_pkey on pgbench_accounts_2 (never
⇔ executed)
        Index Cond: (aid = $0)
   -> Index Scan using pgbench_accounts_3_pkey on pgbench_accounts_3 (never
   executed)
         Index Cond: (aid = $0)
   -> Index Scan using pgbench_accounts_4_pkey on pgbench_accounts_4 (never
⇔ executed)
        Index Cond: (aid = $0)
   -> Index Scan using pgbench_accounts_5_pkey on pgbench_accounts_5 (never
   executed)
         Index Cond: (aid = $0)
   -> Index Scan using pgbench_accounts_6_pkey on pgbench_accounts_6 (never
   executed)
         Index Cond: (aid = $0)
  -> Index Scan using pgbench_accounts_7_pkey on pgbench_accounts_7 (never
   executed)
         Index Cond: (aid = $0)
   -> Index Scan using pgbench_accounts_8_pkey on pgbench_accounts_8 (actual

    time=23.077..23.080 rows=1 loops=1)

        Index Cond: (aid = $0)
  -> Index Scan using pgbench_accounts_9_pkey on pgbench_accounts_9 (never
   executed)
        Index Cond: (aid = $0)
```

```
"
""
"
-> Index Scan using pgbench_accounts_100_pkey on pgbench_accounts_100 (never

    executed)
        Index Cond: (aid = $0)
Planning Time: 1.118 ms
Execution Time: 23.341 ms
```

#### Temps de planification:

Une limitation sérieuse du partitionnement tient au temps de planification qui augmente très vite avec le nombre de partitions, même petites. En effet, chaque partition ajoute ses statistiques et souvent plusieurs index aux tables système. Par exemple, dans le cas le plus défavorable d'une session qui démarre :

```
-- Base pabench de taille 100, non partitionnée
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM pgbench_accounts WHERE aid = 123 LIMIT 1 ;
                                   QUERY PLAN
Limit (actual time=0.021..0.022 rows=1 loops=1)
  Buffers: shared hit=4
  -> Index Scan using pgbench_accounts_pkey on pgbench_accounts (actual

    time=0.021..0.021 rows=1 loops=1)

         Index Cond: (aid = 123)
         Buffers: shared hit=4
Planning:
  Buffers: shared hit=70
Planning Time: 0.358 ms
Execution Time: 0.063 ms
-- Base pgbench de taille 100, partitionnée en 100 partitions
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM pgbench_accounts WHERE aid = 123 LIMIT 1;
                                   QUERY PLAN
Limit (actual time=0.015..0.016 rows=1 loops=1)
  Buffers: shared hit=3
  -> Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1
→ pgbench_accounts (actual time=0.015..0.015 rows=1 loops=1)
         Index Cond: (aid = 123)
         Buffers: shared hit=3
Planning:
  Buffers: shared hit=423
Planning Time: 1.030 ms
Execution Time: 0.061 ms
```

La section Planning indique le nombre de blocs qu'une session qui démarre doit mettre en cache, liés notamment aux tables systèmes et statistiques (ce phénomène est encore une raison d'éviter des sessions trop courtes). Dans cet exemple, sur la base partitionnée, il y a presque six fois plus de ces blocs, et on triple le temps de planification, qui reste raisonnable.



En général, on considère qu'il ne faut pas dépasser 100 partitions si l'on ne veut pas pénaliser les transactions courtes. Les dernières versions de PostgreSQL sont cependant meilleures sur ce point.

Ce problème de planification est moins gênant pour les requêtes longues (analytiques).

Pour contourner cette limite, il est possible d'utiliser directement les partitions, s'il est facile pour le développeur (ou le générateur de code...) de trouver leur nom, en plus de toujours fournir la clé. Interroger directement une partition est en effet aussi rapide à planifier qu'interroger une table monolithique:

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)

SELECT * FROM pgbench_accounts_1 WHERE aid = 123 LIMIT 1;

QUERY PLAN

Limit (actual time=0.006..0.007 rows=1 loops=1)

Buffers: shared hit=3

-> Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 (actual time=0.006..0.006 rows=1 loops=1)

Index Cond: (aid = 123)

Buffers: shared hit=3

Planning Time: 0.046 ms

Execution Time: 0.016 ms
```

Utiliser directement les partitions est particulièrement économe si leur nombre est grand, mais on perd alors le côté « transparent » du partitionnement, et on augmente la complexité du code applicatif.

#### Paramètres « partitionwise » :

Dans des cas plus complexes, notamment en cas de jointure entre tables partitionnées, le temps de planification peut exploser. Par exemple, pour la requête suivante où la table partitionnée est jointe à elle-même, le plan sur la table non partitionnée, cache de session chaud, renvoie:

Avec cent partitions, le temps de planification est ici multiplié par 50 :

```
Gather
 Workers Planned: 4
  -> Parallel Hash Join
        Hash Cond: (b.aid = a.aid)
        -> Parallel Append
              -> Parallel Seq Scan on pgbench_accounts_1 b_1
              -> Parallel Seq Scan on pgbench_accounts_2 b_2
              -> Parallel Seq Scan on pgbench_accounts_99 b_99
              -> Parallel Seq Scan on pgbench_accounts_100 b_100
        -> Parallel Hash
              -> Parallel Append
                    -> Parallel Seq Scan on pgbench_accounts_1 a_1
                          Filter: (bid = 55)
                       Parallel Seq Scan on pgbench_accounts_2 a_2
                          Filter: (bid = 55)
                    -> Parallel Seq Scan on pgbench_accounts_99 a_99
                          Filter: (bid = 55)
                    -> Parallel Seq Scan on pgbench_accounts_100 a_100
                          Filter: (bid = 55)
Planning Time: 5.513 ms
```

Ce plan est perfectible : il récupère tout pgbench\_accounts et le joint à toute la table. Il serait plus intelligent de travailler partition par partition puisque la clé de jointure est celle de partitionnement. Pour que PostgreSQL cherche à faire ce genre de chose, un paramètre doit être activé :

```
SET enable_partitionwise_join TO on ;
```

Les jointures se font alors entre partitions :

```
Gather
  Workers Planned: 4
   -> Parallel Append
         -> Parallel Hash Join
              Hash Cond: (a_55.aid = b_55.aid)
              -> Parallel Seq Scan on pgbench_accounts_55 a_55
                    Filter: (bid = 55)
                 Parallel Hash
                    -> Parallel Seq Scan on pgbench_accounts_55 b_55
         -> Nested Loop
              -> Parallel Seq Scan on pgbench_accounts_100 a_100
                    Filter: (bid = 55)
              -> Index Scan using pgbench_accounts_100_pkey on pgbench_accounts_100
→ b_100
                    Index Cond: (aid = a_100.aid)
Planning:
   Buffers: shared hit=1200
Planning Time: 12.449 ms
```

Le temps d'exécution passe de 1,2 à 0,2 s, ce qui justifie les quelques millisecondes perdues en plus en planification.

Un autre paramètre est à activer si des agrégations sur plusieurs partitions sont à faire :

#### SET enable\_partitionwise\_aggregate TO on ;

enable\_partitionwise\_aggregate et enable\_partitionwise\_join sont désactivés par défaut à cause de leur coût en planification sur les petites requêtes, mais les activer est souvent rentable. Avec SET, cela peut se décider requête par requête.

## 4.4.19 Opérations de maintenance



- Changement de tablespace
- autovacuum/analyze
  - sur les partitions comme sur toute table
- VACUUM, VACUUM FULL, ANALYZE
  - sur table mère : redescendent sur les partitions
- REINDEX
  - avant v14: uniquement par partition
- ANALYZE
  - prévoir aussi sur la table mère (manuellement...)

Les opérations de maintenance profitent grandement du fait de pouvoir scinder les opérations en autant d'étapes qu'il y a de partitions. Des données « froides » peuvent être déplacées dans un autre tablespace sur des disques moins chers, partition par partition, ce qui est impossible avec une table monolithique :

# ALTER TABLE pgbench\_accounts\_8 SET TABLESPACE hdd ;

L'autovacuum et l'autoanalyze fonctionnent normalement et indépendamment sur chaque partition, comme sur les tables classiques. Ainsi ils peuvent se déclencher plus souvent sur les partitions actives. Par rapport à une grosse table monolithique, il y a moins souvent besoin de régler l'autovacuum.

Les ordres ANALYZE et VACUUM peuvent être effectués sur une partition, mais aussi sur la table partitionnée, auquel cas l'ordre redescendra en cascade sur les partitions (l'option VERBOSE permet de le vérifier). Les statistiques seront calculées par partition, donc plus précises.

Reconstruire une table partitionnée avec VACUUM FULL se fera généralement partition par partition. Le partitionnement permet ainsi de résoudre les cas où le verrou sur une table monolithique serait trop long, ou l'espace disque total serait insuffisant.

Noter cependant ces spécificités sur les tables partitionnées :

#### **REINDEX:**

À partir de PostgreSQL 14, un REINDEX sur la table partitionnée réindexe toutes les partitions automatiquement. Dans les versions précédentes, il faut réindexer partition par partition.

#### **ANALYZE:**

L'autovacuum ne crée pas spontanément de statistiques sur les données pour la table partitionnée dans son ensemble, mais uniquement partition par partition. Pour obtenir des statistiques sur toute la table partitionnée, il faut exécuter manuellement :

**ANALYZE** table\_partitionnée ;

## 4.4.20 Sauvegardes



Sauvegarde physique : peu de différence Avec pg\_dump :

- --jobs : efficace
- --load-via-partition-root
- exclusion de partitions (v16+):
  - --table-and-children
  - --exclude-table-and-children
  - --exclude-table-data-and-children

Grâce au partitionnement, un export par pg\_dump --jobs devient efficace puisque plusieurs partitions peuvent être sauvegardées en parallèle.

La parallélisation peut être aussi un peu meilleure avec un outil de sauvegarde physique (comme pg-BackRest ou Barman), qui parallélise les copies de fichiers, mais les grosses tables non partitionnées étaient de toute façon déjà découpées en fichier de 1 Go.

pg\_dump a des options pour gérer l'export des tables partitionnées :

--load-via-partition-root permet de générer des ordres COPY ciblant la table mère et non la partition. Ce peut être pratique pour restaurer les données dans une base où la table est partitionnée séparément.

À partir de PostgreSQL 16, n'exporter qu'une table partitionnée se fait avec —table—and—children (et non —table / —t qui ne concernerait que la table mère). Exclure des tables partitionnées se fait avec —exclude—table—and—children (et non —exclude—table / —T ). Pour exclure uniquement les données d'une table partitionnée en gardant sa structure, on utilisera —exclude—table—data—and—children . Ces trois options acceptent un motif (par exemple : pgbench\_accounts\_\*) et peuvent être répétées dans la commande.

## 4.4.21 Limitations du partitionnement déclaratif et versions



- Pas de création automatique des partitions
  - ni fusion/scission facile
- Planification: 100 partitions max conseillé
- Pas d'héritage multiple, schéma fixe
  - sauf ajout d'index/contraintes sur chaque partition
- Partitions distantes : sans propagation d'index
- Clé de partitionnement calculée : pas vraiment possible
- (≤v16) IDENTITY: uniquement sur la table partitionnée
- PostgreSQL récent toujours conseillé

Certaines de ces limites ont été évoquées plus haut.

## Limitations générales :

Une table partitionnée ne peut être convertie en table classique, ni vice-versa. (Par contre, une table classique peut être attachée comme partition, ou une partition détachée).

Les partitions ont forcément le même schéma de données que leur partition mère et héritent des contraintes et index de celle-ci. Il est interdit d'ajouter ou de retrancher de champs. L'ajout d'index ou de contraintes supplémentaires directement sur une partition est par contre possible.

Les clés de partition ne doivent évidemment pas se recouvrir.

Les contraintes ne peuvent s'exercer qu'au sein d'une même partition : les clés d'unicité doivent donc inclure toute la clé de partitionnement. Les contraintes d'exclusion ne peuvent vérifier toutes les partitions.

Il n'y a pas de notion d'héritage multiple.

#### Limitations liées à la création des partitions :

La création des partitions n'est pas automatisée : il faut les créer par avance manuellement ou par script planifié, ou un outil externe comme pg\_partman ou pg\_cron. Éventuellement prévoir une partition par défaut pour les cas qui ont pu être oubliés.

Changer les limites d'une partition n'est pas directement prévu. Il faut détacher la partition et réattacher la partition.

Il n'est pas possible de fusionner ou scinder des partitions. Il faudra déplacer les données manuellement.

#### **Limitation sur les champs IDENTITY :**

Jusque PostgreSQL 16 inclus, un champ GENERATED ALWAYS/BY DEFAULT AS IDENTITY est utilisable uniquement en insérant dans la table partitionnée, pas en insérant dans la partition. Sinon PostgreSQL se plaint que le champ est NULL. PostgreSQL 17 lève la limite.

Cependant les valeurs d'un champ IDENTITY continuent d'être générées par la séquence indépendamment des partitions, les unes à la suite des autres. Si ce champ fait partie de la clé de partitionnement, il y a aura une erreur si la valeur générée ne respecte pas la contrainte de partitionnement.

#### Limitations sur les clés calculées :

Il serait parfois pratique de partitionner suivant un champ calculé, par exemple par année quand on dispose d'un champ jour. Malheureusement, le moteur a des limites sur ce point<sup>5</sup>:

- techniquement, une table avec PARTITION BY LIST ((extract ('year' from jour))) fonctionne, mais PostgreSQL ne voudra pas créer de clé primaire incluant ce champ;
- une colonne générée automatiquement (clause GENERATED ALWAYS AS ... STORED) ne peut faire partie de la clé de partitionnement ;
- un trigger BEFORE INSERT ne peut modifier la partition cible d'une ligne;
- un trigger INSTEAD OF sur une vue utilitaire peut fonctionner, mais complexifie l'interface et, comme tout trigger, pénalise les performances.

Des contournements sont possibles. La clé peut être calculée côté applicatif, même s'il s'agit juste d'appeler une fonction SQL. Le critère de partitionnement est souvent simple : un partitionnement différent peut convenir. Pour reprendre l'exemple du partitionnement par liste par année calculée à partir du jour, un partitionnement par intervalle de date serait tout aussi adapté.

#### Limitations liées aux performances :

Il faut éviter d'avoir trop de partitions, pour limiter les risques de dérapage du temps de planification. S'il y en a beaucoup, tenter de cibler directement les partitions qui intéressent une requête.

L'ordre CLUSTER, pour réécrire une table dans l'ordre d'un index donné, ne fonctionne pour les tables partitionnées qu'à partir de PostgreSQL 15. Il peut toutefois être exécuté manuellement table par table.

#### Limitations liées aux tables distantes :

Un TRUNCATE d'une table distante n'est pas possible avant PostgreSQL 14.

Il est possible d'attacher comme partitions des tables distantes, généralement déclarées avec postgres\_fdw, mais la propagation d'index ne fonctionnera pas sur ces tables. Il faudra les créer manuellement sur les instances distantes.

#### Limitations liées aux triggers :

À partir de la version 13, les triggers BEFORE UPDATE ... FOR EACH ROW sont possibles, mais il ne permettent toujours pas de modifier la partition de destination.

#### Limitations des versions non supportées :

Ces versions peuvent encore se rencontrer.

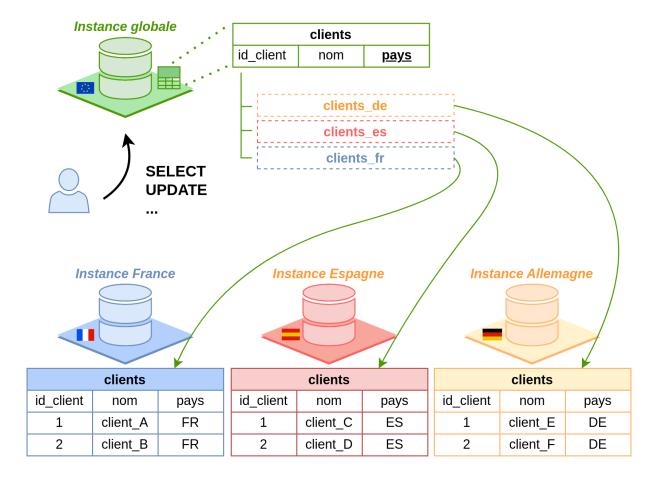
<sup>&</sup>lt;sup>5</sup>https://docs.postgresql.fr/current/ddl-partitioning.html#DDL-PARTITIONING-INHERITANCE-CAVEATS

- Les partitions par défaut n'existent pas en version 10.
- Les index, les clés primaires et les clés étrangères sont très limitées avant la version 12.
- Les triggers ont beaucoup plus de limitations avant la version 13.
- En version 10, un UPDATE ne peut modifier la clé de partitionnement, il faut faire un DELETE et un INSERT à la place, et une partition distante n'est accessible qu'en lecture via la table mère.



Beaucoup de ces limitations dépendent des versions de PostgreSQL. Heureusement, elles concernent surtout des versions de PostgreSQL qui ne sont plus supportées, et qui ne devraient plus être utilisées. Une version récente reste de toute manière préférable.

# 4.5 TABLES DISTANTES & SHARDING





- Tables distantes comme partitions: sharding
- (v14+) Interrogation simultanée asynchrone

Il est possible d'attacher comme partitions des tables distantes (situées sur d'autres serveurs), généralement déclarées avec le *Foreign Data Wrapper* postgres\_fdw .

NB: Dans le reste de ce chapitre, nous nommerons **table étrangère** (*foreign table* dans la documentation officielle) l'objet qui sert d'interface pour accéder, depuis l'instance locale, à la **table distante** (*remote table*), qui contient réellement les données.

Par exemple, si trois instances en France, Allemagne et Espagne possèdent chacune des tables clients et commandes ne contenant que les données de leur pays, on peut créer une autre instance utilisant des tables étrangères pour accéder aux trois tables, chaque table étrangère étant une partition d'une table partitionnée de cette instance européene.

Pour les instances nationales, cette instance européenne n'est qu'un client comme un autre, qui en-

voie des requêtes, et ouvre parfois des curseurs (fonctionnement normal de postgres\_fdw ). Si le pays est précisé dans une requête, la bonne partition est ciblée, et l'instance européenne n'interroge qu'une seule instance nationale.

La maquette suivante donne une idée du fonctionnement :

```
-- Maguette rapide sous psql de sharding
-- avec trois bases demosharding_fr , _de, _es
-- et une base globale pour le requêtage
\set timing off
\set ECHO all
\set ON_ERROR_STOP 1
\connect postgres postgres serveur1
DROP DATABASE IF EXISTS demosharding_fr ;
CREATE DATABASE demosharding_fr ;
ALTER DATABASE demosharding_fr SET log_min_duration_statement TO \odot ;
\connect postgres postgres serveur2
DROP DATABASE IF EXISTS demosharding_de ;
CREATE DATABASE demosharding_de ;
ALTER DATABASE demosharding_de SET log_min_duration_statement TO 0;
\connect postgres postgres serveur3
DROP DATABASE IF EXISTS demosharding_es ;
CREATE DATABASE demosharding_es ;
ALTER DATABASE demosharding_es SET log_min_duration_statement TO 0;
\connect postgres postgres serveur4
DROP DATABASE IF EXISTS demosharding_global ;
CREATE DATABASE demosharding_global ;
ALTER DATABASE demosharding_global SET log_min_duration_statement TO 0;
-- Tables identiques sur chaque serveur
\connect demosharding_fr postgres serveur1
CREATE TABLE clients
                        (id_client int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                                   text,
                        nom
                                   char (2) DEFAULT 'FR' CHECK (pays = 'FR')
                        pays
                        );
CREATE TABLE commandes (id_commande bigint GENERATED ALWAYS AS IDENTITY PRIMARY
char (2) DEFAULT 'FR' CHECK (pays = 'FR'),
                        pavs
                                    int REFERENCES clients ,
                        id_client
                        montant
                                     float
                        );
\connect demosharding_de postgres serveur2
CREATE TABLE clients
                        (id_client int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                                    text,
                        nom
                                    char (2) DEFAULT 'DE' CHECK (pays = 'DE')
                        pays
                        );
CREATE TABLE commandes (id_commande bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                                    char (2) DEFAULT 'DE' CHECK (pays = 'DE') ,
                        pays
                        id_client int REFERENCES clients,
```

```
montant
                                     float
                        );
\connect demosharding_es postgres serveur3
CREATE TABLE clients
                        (id_client int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                        nom
                                    char (2) DEFAULT 'ES' CHECK (pays = 'ES')
                        pays
                        );
CREATE TABLE commandes (id_commande bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                                    char (2) DEFAULT 'ES' CHECK (pays = 'ES'),
                        pays
                        id_client
                                    int REFERENCES clients ,
                        montant
                                    float
                        );
-- Tables partitionnées globales
\connect demosharding_global postgres serveur4
CREATE TABLE clients (id_client int, nom text, pays char(2))
PARTITION BY LIST (pays);
CREATE TABLE commandes (id_commande bigint, pays char(2),
                        id_client int, montant float)
PARTITION BY LIST (pays);
-- Serveurs distants (adapter les chaines de connexion)
-- NB: l'option async capable n existe pas avant PostareSOL 14
CREATE EXTENSION postgres_fdw ;
CREATE SERVER dist_fr
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'demosharding_fr', port '16001',
async_capable 'on', fetch_size '10000');
CREATE SERVER dist de
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'demosharding_de', port '16001',
async_capable 'on', fetch_size '10000');
CREATE SERVER dist_es
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'demosharding_es', port '16001',
async_capable 'on', fetch_size '10000');
CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER dist_fr ;
CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER dist_de ;
CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER dist_es ;
-- Les partitions distantes
CREATE FOREIGN TABLE clients_fr PARTITION OF clients FOR VALUES IN ('FR')
SERVER dist_fr OPTIONS (table_name 'clients');
CREATE FOREIGN TABLE clients_de PARTITION OF clients FOR VALUES IN ('DE')
SERVER dist_de OPTIONS (table_name 'clients') ;
CREATE FOREIGN TABLE clients_es PARTITION OF clients FOR VALUES IN ('ES')
SERVER dist_es OPTIONS (table_name 'clients') ;
CREATE FOREIGN TABLE commandes_fr PARTITION OF commandes FOR VALUES IN ('FR')
```

```
SERVER dist_fr OPTIONS (table_name 'commandes') ;
CREATE FOREIGN TABLE commandes_de PARTITION OF commandes FOR VALUES IN ('DE')
SERVER dist_de OPTIONS (table_name 'commandes') ;
CREATE FOREIGN TABLE commandes_es PARTITION OF commandes FOR VALUES IN ('ES')
SERVER dist es OPTIONS (table name 'commandes') ;
-- Alimentations des pays (séparément)
\connect demosharding_fr postgres serveur1
WITH ins_clients AS (
    INSERT INTO clients (nom)
    SELECT md5 (random()::text) FROM generate_series (1,10) i
    WHERE random()<0.8
RETURNING id_client
ins_commandes AS (
    INSERT INTO commandes (id_client, montant)
    SELECT c.id_client, random()*j::float
    FROM ins_clients c CROSS JOIN generate_series (1,100000) j
    WHERE random()<0.8
RETURNING *
SELECT count(*) FROM ins_commandes ;
\connect demosharding_de postgres serveur2
\g
\connect demosharding_es postgres serveur3
\g
\connect demosharding_global postgres serveur4
-- Les ANALYZE redescendent sur les partitions
ANALYZE (VERBOSE) clients, commandes;
-- Pour un plan optimal
SET enable_partitionwise_join TO on ;
SET enable_partitionwise_aggregate TO on ;
-- Requête globale : top 8 des clients
-- Plan disponible sur https://explain.dalibo.com/plan/27f964651518a65g
SELECT pays,
       count(DISTINCT id_commande) AS nb_commandes,
       avg(montant) AS montant_avg_commande,
       sum(montant) AS montant_sum
    commandes INNER JOIN clients USING (pays, id_client)
GROUP BY 1,2
ORDER BY montant_sum DESC
LIMIT 8 ;
```

Une nouveauté de PostgreSQL 14 est ici particulièrement intéressante : l'option async\_capable du serveur étranger (éventuellement de la table) peut être passée à on (le défaut est off). Les nœuds Foreign Scan typiques des accès distants sont alors remplacés par des nœuds Async Foreign Scan (asynchrones), et le serveur principal interroge alors simultanément les trois serveurs qui lui renvoient les données. Dans cet extrait des traces, les ordres FETCH sont entremêlés :

```
...user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG: duration: 0.384 ms execute <unnamed>: DECLARE c1 CURSOR FOR
        SELECT id_commande, pays, id_client, montant FROM public.commandes
...user=postgres,db=demosharding_es,app=postgres_fdw,client=::1
LOG: duration: 0.314 ms execute <unnamed>: DECLARE c2 CURSOR FOR
        SELECT id_commande, pays, id_client, montant FROM public.commandes
...user=postgres,db=demosharding_fr,app=postgres_fdw,client=::1
LOG: duration: 0.374 ms execute <unnamed>: DECLARE c3 CURSOR FOR
        SELECT id_commande, pays, id_client, montant FROM public.commandes
...user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG: duration: 6.081 ms statement: FETCH 10000 FROM c1
...user=postgres,db=demosharding_es,app=postgres_fdw,client=::1
LOG: duration: 5.878 ms statement: FETCH 10000 FROM c2
...user=postgres,db=demosharding_fr,app=postgres_fdw,client=::1
LOG: duration: 6.263 ms statement: FETCH 10000 FROM c3
...user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG: duration: 2.418 ms statement: FETCH 10000 FROM c1
...user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG: duration: 2.397 ms statement: FETCH 10000 FROM c1
...user=postgres,db=demosharding_es,app=postgres_fdw,client=::1
LOG: duration: 2.423 ms statement: FETCH 10000 FROM c2
...user=postgres,db=demosharding_fr,app=postgres_fdw,client=::1
LOG: duration: 4.381 ms statement: FETCH 10000 FROM c3
```

Les paramètres enable\_partitionwise\_join et enable\_partitionwise\_aggregate doivent impérativement être activés dans cette configuration. L'optimiseur cherche alors à « pousser » les jointures et les agrégations au niveau du nœud (et à les calculer directement sur les serveurs distants) comme dans le plan suivant (voir la version complète<sup>6</sup>) :

<sup>&</sup>lt;sup>6</sup>https://explain.dalibo.com/plan/27f964651518a65g#raw

```
Output: commandes_1.pays, clients_1.nom, (count(DISTINCT
commandes_1.id_commande)), (avg(commandes_1.montant)),
 (sum(commandes_1.montant))
               Relations: Aggregate on ((public.commandes_es commandes_1) INNER
 JOIN (public.clients es clients 1))
              Remote SQL: SELECT r5.pays, r8.nom, count(DISTINCT r5.id_commande),
avg(r5.montant), sum(r5.montant) FROM (public.commandes r5 INNER JOIN
 public.clients r8 ON (((r5.pays = r8.pays)) AND ((r5.id_client =
r8.id_client)))) GROUP BY 1, 2
         -> Async Foreign Scan (cost=9102.09..53300.65 rows=9 width=60) (actual
 time=2.189..2.190 rows=9 loops=1)
               Output: commandes_2.pays, clients_2.nom, (count(DISTINCT
commandes_2.id_commande)), (avg(commandes_2.montant)),
 (sum(commandes_2.montant))
               Relations: Aggregate on ((public.commandes_fr commandes_2) INNER
 JOIN (public.clients_fr clients_2))
              Remote SQL: SELECT r6.pays, r9.nom, count(DISTINCT r6.id_commande),
avg(r6.montant), sum(r6.montant) FROM (public.commandes r6 INNER JOIN
public.clients r9 ON (((r6.pays = r9.pays)) AND ((r6.id_client =
r9.id_client)))) GROUP BY 1, 2
```



Quand on utilise les tables étrangères, il est conseillée d'utiliser EXPLAIN (VERBOSE), pour afficher les requêtes envoyées aux serveurs distants et vérifier que le minimum de volumétrie transite sur le réseau.

Cet exemple est une version un peu primitive de *sharding*, à réserver aux cas où les données sont clairement séparées. L'administration d'une configuration « multimaître » peut devenir compliquée : cohérence des différents schémas et des contraintes sur chaque instance, copie des tables de référence communes, risques de recouvrement des clés primaires entre bases, gestion des indisponibilités, sauvegardes cohérentes...

Noter que l'utilisation de partitions distantes rend impossible notamment la gestion automatique des index, il faut retourner à une manipulation table par table.

## 4.6 EXTENSIONS & OUTILS



- Extension pg partman
  - automatisation
- Extensions dédiées à un domaine :
  - timescaledb
  - citus

L'extension pg\_partman<sup>7</sup>, de Crunchy Data, est un complément aux systèmes de partitionnement de PostgreSQL. Elle est apparue d'abord pour automatiser le partitionnement par héritage. Elle peut être utile avec le partitionnement déclaratif, pour simplifier la maintenance d'un partitionnement sur une échelle temporelle ou de valeurs (par *range*).

PostgresPro proposait un outil nommé pg\_pathman<sup>8</sup>, à présent déprécié en faveur du partitionnement déclaratif intégré à PostgreSQL.

**timescaledb** est une extension spécialisée dans les séries temporelles. Basée sur le partitionnement par héritage, elle vaut surtout pour sa technique de compression et ses utilitaires. La version communautaire sur Github<sup>9</sup> ne comprend pas tout ce qu'offre la version commerciale.

**citus**<sup>10</sup> est une autre extension commerciale. Le principe est de partitionner agressivement les tables sur plusieurs instances, et d'utiliser simultanément les processeurs, disques de toutes ces instances (*sharding*). Citus gère la distribution des requêtes, mais pas la maintenance des instances PostgreSQL supplémentaires. L'éditeur Citusdata a été racheté par Microsoft, qui le propose à présent dans Azure. En 2022, l'entièreté du code est passée sous licence libre<sup>11</sup>. Le gain de performance peut être impressionnant, mais attention : certaines requêtes se prêtent très mal au *sharding*.

<sup>&</sup>lt;sup>7</sup>https://github.com/pgpartman/pg\_partman

<sup>&</sup>lt;sup>8</sup>https://github.com/postgrespro/pg\_pathman

<sup>&</sup>lt;sup>9</sup>https://github.com/timescale/timescaledb

<sup>&</sup>lt;sup>10</sup>https://github.com/citusdata/citus

<sup>&</sup>lt;sup>11</sup>https://www.citusdata.com/blog/2022/06/17/citus-11-goes-fully-open-source/

# 4.7 CONCLUSION



Le partitionnement déclaratif a de gros avantages pour le DBA - ...mais les développeurs doivent savoir l'utiliser
- Préférer une version récente de PostgreSQL

Le partitionnement par héritage n'a plus d'utilité pour la plupart des applications.

Le partitionnement déclaratif apparu en version 10 est mûr dans les dernières versions. Il introduit une complexité supplémentaire, que les développeurs doivent maîtriser, mais peut rendre de grands services quand la volumétrie augmente.

# **4.8 QUIZ**



# **4.9 TRAVAUX PRATIQUES**

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/v1\_solutions.

#### 4.9.1 Partitionnement



**But** : Mettre en place le partitionnement déclaratif

Nous travaillons sur la base **cave**. La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

Nous allons partitionner la table stock sur l'année.

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans stock (cela nous évitera la création de 50 partitions) :

```
-- Création de lignes en 2001-2005

INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)

FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
-- purge des lignes prédédentes

DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer stock en stock old.
- Créer une table partitionnée stock vide, sans index pour le moment.
- Créer les partitions de stock, avec la contrainte d'année : stock\_2001 à stock\_2005.
- Insérer tous les enregistrements venant de l'ancienne table stock.
- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

- Vérifier la présence d'enregistrements dans stock\_2001 (syntaxe SELECT ONLY).
- Vérifier qu'il n'y en a aucun dans stock.
- Vérifier qu'une requête sur stock sur 2002 ne parcourt qu'une seule partition.
- Remettre en place les index présents dans la table stock originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).
- Quel est le plan pour la récupération du stock des bouteilles du vin\_id 1725, année 2003
   ?
- Essayer de changer l'année de ce même enregistrement de stock (la même que la précédente). Pourquoi cela échoue-t-il ?
- Supprimer les enregistrements de 2004 pour vin\_id = 1725.
- Retenter la mise à jour.
- Pour vider complètement le stock de 2001, supprimer la partition stock\_2001.
- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il?
- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.
- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il?
- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés;
- ré-attacher la partition par défaut.

## 4.9.2 Partitionner pendant l'activité



But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

## 4.9.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

NB : Pour le TP, la base sera d'échelle 10 (environ 168 Mo). Des échelles 100 ou 1000 seraient plus réalistes.

Dans une fenêtre en arrière-plan, laisser tourner un processus pgbench avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

#### 4.9.2.2 Partitionnement par hash

Pour partitionner la table pgbench\_accounts par hash sur la colonne aid sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis pgbench\_accounts ;
- la substitution de la table partitionnée à la table originale.

Tester et exécuter.

Supprimer l'ancienne table pgbench\_accounts\_old.

#### 4.9.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

La table pgbench\_history se remplit avec le temps. Elle doit être partitionnée par date (champ mtime). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.

- Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.

NB : Pour éviter de coder des dates en dur, il est possible, avec psql , d'utiliser une variable :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;
```

Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

#### 4.9.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

## 4.9.2.5 Contraintes entre tables partitionnées

Ajouter une clé étrangère entre pgbench\_accounts et pgbench\_history. Voir les contraintes créées.

Si vous n'avez pas déjà eu un problème à cause du statement\_timeout, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner?

## 4.9.2.6 Index global

On veut créer un index sur pgbench\_history (aid).

Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

# **4.10 TRAVAUX PRATIQUES (SOLUTIONS)**

#### 4.10.1 Partitionnement



But : Mettre en place le partitionnement déclaratif

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans stock (cela nous évitera la création de 50 partitions).

```
INSERT INTO stock
SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock
GROUP BY vin_id, contenant_id, 2001 + annee % 5;
DELETE FROM stock WHERE annee<2001;</pre>
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer stock en stock\_old.
- Créer une table partitionnée stock vide, sans index pour le moment.

```
ALTER TABLE stock RENAME TO stock_old;
CREATE TABLE stock(LIKE stock_old) PARTITION BY LIST (annee);
```

- Créer les partitions de stock , avec la contrainte d'année : stock\_2001 à stock\_2005 .

```
CREATE TABLE stock_2001 PARTITION of stock FOR VALUES IN (2001);
CREATE TABLE stock_2002 PARTITION of stock FOR VALUES IN (2002);
CREATE TABLE stock_2003 PARTITION of stock FOR VALUES IN (2003);
CREATE TABLE stock_2004 PARTITION of stock FOR VALUES IN (2004);
CREATE TABLE stock_2005 PARTITION of stock FOR VALUES IN (2005);
```

- Insérer tous les enregistrements venant de l'ancienne table stock.

```
INSERT INTO stock SELECT * FROM stock_old;
```

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

#### ANALYZE;

- Vérifier la présence d'enregistrements dans stock\_2001 (syntaxe SELECT ONLY ).
- Vérifier qu'il n'y en a aucun dans stock.

```
SELECT count(*) FROM stock_2001;
SELECT count(*) FROM ONLY stock;
```

- Vérifier qu'une requête sur stock sur 2002 ne parcourt qu'une seule partition.

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE annee=2002;
```

```
QUERY PLAN
```

Append (cost=0.00..417.36 rows=18192 width=16) (...)

-> Seq Scan on stock\_2002 (cost=0.00..326.40 rows=18192 width=16) (...)

Filter: (annee = 2002)

Planning Time: 0.912 ms Execution Time: 21.518 ms

- Remettre en place les index présents dans la table stock originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

```
CREATE UNIQUE INDEX ON stock (vin_id,contenant_id,annee);
```

Les autres index ne servent à rien sur les partitions : idx\_stock\_annee est évidemment inutile, mais idx\_stock\_vin\_annee aussi, puisqu'il est inclus dans l'index unique que nous venons de créer.

Quel est le plan pour la récupération du stock des bouteilles du vin\_id 1725, année 2003
 ?

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE vin_id=1725 AND annee=2003 ;

Append (cost=0.29..4.36 rows=3 width=16) (...)
   -> Index Scan using stock_2003_vin_id_contenant_id_annee_idx on stock_2003 (...)
        Index Cond: ((vin_id = 1725) AND (annee = 2003))

Planning Time: 1.634 ms
Execution Time: 0.166 ms
```

- Essayer de changer l'année de ce même enregistrement de stock (la même que la précédente). Pourquoi cela échoue-t-il ?

C'est une violation de contrainte unique, qui est une erreur normale : nous avons déjà un enregistrement de stock pour ce vin pour l'année 2004.

- Supprimer les enregistrements de 2004 pour vin\_id = 1725.
- Retenter la mise à jour.

```
DELETE FROM stock WHERE annee=2004 and vin_id=1725;
UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

Pour vider complètement le stock de 2001, supprimer la partition stock\_2001.

```
DROP TABLE stock_2001;
```

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il?

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100);
ERROR: no partition of relation "stock" found for row
DETAIL: Partition key of the failing row contains (annee) = (2006).
```

Il n'existe pas de partition définie pour l'année 2006, cela échoue donc.

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

```
CREATE TABLE stock_default PARTITION OF stock DEFAULT ;
INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;
```

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

```
CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;
ERROR: updated partition constraint for default partition "stock_default"
    would be violated by some row
```

Cela échoue car des enregistrements présents dans la partition par défaut répondent à cette nouvelle contrainte de partitionnement.

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés;
- ré-attacher la partition par défaut.

```
BEGIN;
```

```
ALTER TABLE stock DETACH PARTITION stock_default;

CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006);

INSERT INTO stock SELECT * FROM stock_default WHERE annee = 2006;

DELETE FROM stock_default WHERE annee = 2006;

ALTER TABLE stock ATTACH PARTITION stock_default DEFAULT;

COMMIT;
```

## 4.10.2 Partitionner pendant l'activité



But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

## 4.10.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 10 pgbench
```

Dans une fenêtre en arrière-plan, laisser tourner un processus pgbench avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

```
$ /usr/pgsql-14/bin/pgbench -n -T3600 -c20 -j2 --debug pgbench
```

L'activité est à ajuster en fonction de la puissance de la machine. Laisser l'affichage défiler dans une fenêtre pour bien voir les blocages.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

Un verrou en attente peut bloquer les opérations d'autres transactions venant après. On peut annuler l'opération à partir d'un certain seuil pour éviter ce phénomène :

```
pgbench=# SET lock_timeout TO '10s';
```

Cela ne concerne cependant pas les opérations une fois que les verrous sont acquis. On peut garantir qu'un ordre donné ne durera pas plus d'une certaine durée :

```
SET statement_timeout TO '10s';
```

En fonction de la rapidité de la machine et des données à déplacer, cette interruption peut être tolérable ou non.

## 4.10.2.2 Partitionnement par hash

Pour partitionner la table pgbench\_accounts par hash sur la colonne aid sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis pgbench\_accounts ;
- la substitution de la table partitionnée à la table originale.

# Tester et exécuter.

Le champ aid n'a pas de signification, un partitionnement par hash est adéquat. Le script peut être le suivant : \timing on \set ON\_ERROR\_STOP 1 SET lock\_timeout TO '10s'; SET statement\_timeout TO '10s' ; **BEGIN**; -- Nouvelle table partitionnée CREATE TABLE pgbench\_accounts\_part (LIKE pgbench\_accounts INCLUDING ALL) PARTITION BY HASH (aid); CREATE TABLE pgbench\_accounts\_1 PARTITION OF pgbench\_accounts\_part FOR VALUES WITH (MODULUS 3, REMAINDER 0); CREATE TABLE pgbench\_accounts\_2 PARTITION OF pgbench\_accounts\_part FOR VALUES WITH (MODULUS 3, REMAINDER 1 ); CREATE TABLE pgbench\_accounts\_3 PARTITION OF pgbench\_accounts\_part FOR VALUES WITH (MODULUS 3, REMAINDER 2 ); -- Transfert des données -- Bloquer les accès à la table le temps du transfert -- (sinon risque de perte de données !) LOCK TABLE pgbench\_accounts; -- Copie des données INSERT INTO pgbench\_accounts\_part SELECT \* FROM pgbench\_accounts ; -- Substitution par renommage ALTER TABLE pgbench\_accounts RENAME TO pgbench\_accounts\_old; ALTER TABLE pgbench\_accounts\_part RENAME TO pgbench\_accounts ; -- Contrôle \d+ -- On ne validera qu'après contrôle -- (pendant ce temps les sessions concurrentes restent bloquées !) COMMIT; À la moindre erreur, la transaction tombe en erreur. Il faudra demander manuellement ROLLBACK. Si la durée fixée par statement\_timeout est dépassée, on aura cette erreur :

ERROR: canceling statement due to statement timeout Time: 10115.506 ms (00:10.116)

Surtout, le traitement pgbench reprend en arrière-plan. On peut alors relancer le script corrigé plus tard.

Si tout se passe bien, un \d+ renvoie ceci:

Liste des relations					
Schéma	Nom .	Type	Propriétaire	Taille	
public public public public public public public	<pre>pgbench_accounts pgbench_accounts_1 pgbench_accounts_2 pgbench_accounts_3 pgbench_accounts_old pgbench_branches</pre>	table partitionnée   table   table   table   table   table   table	postgres   postgres   postgres   postgres   postgres   postgres	+   0 bytes   43 MB   43 MB   43 MB   130 MB   136 kB	+         
public	pgbench_history	table	postgres	5168 kB	İ
public	pgbench_tellers	table	postgres	216 kB	

On peut vérifier rapidement que les valeurs de aid sont bien réparties entre les 3 partitions :

```
select aid from pgbench_accounts_1 Limit 3;
aid
-----
2    6
8

select aid from pgbench_accounts_2 Limit 3;
aid
----
3    7
10

select aid from pgbench_accounts_3 Limit 3;
aid
-----
1    9
11
```

Après la validation du script, on voit apparaître les lignes dans les nouvelles partitions :

333240

pgbench\_accounts\_3

```
Supprimer l'ancienne table pgbench_accounts_old.
```

```
DROP TABLE pgbench_accounts_old ;
```

### 4.10.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

La table pgbench\_history se remplit avec le temps. Elle doit être partitionnée par date (champ mtime). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.

- Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.

NB : Pour éviter de coder des dates en dur, il est possible, avec psql, d'utiliser une variable :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;
```

La « date frontière » doit être dans le futur (proche). En effet, pgbench va modifier les tables en permanence, on ne sait pas exactement à quel moment la transition aura lieu (et de toute façon on ne maîtrise pas les valeurs de mtime): il continuera donc à écrire dans l'ancienne table, devenue partition, pendant encore quelques secondes.

Cette date est arbitrairement à 1 minute dans le futur, pour dérouler le script manuellement :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
Et on peut réutiliser cette variable ainsi;
SELECT :'date_frontiere'::timestamptz ;
Le script peut être celui-ci:
\timing on \set ON_ERROR_STOP 1

SET lock_timeout TO '10s';
SET statement_timeout TO '10s';
SELECT ( now()+ interval '60s') AS date_frontiere \gset SELECT :'date_frontiere'::timestamptz;
BEGIN ;
-- Nouvelle table partitionnée
CREATE TABLE pgbench_history_part (LIKE pgbench_history INCLUDING ALL)
PARTITION BY RANGE (mtime);
```

```
-- Des partitions pour les prochaines minutes
CREATE TABLE pgbench_history_1
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz )
             TO (:'date_frontiere'::timestamptz + interval '1min');
CREATE TABLE pgbench_history_2
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz + interval '1min' )
             TO (:'date_frontiere'::timestamptz + interval '2min' );
-- Au cas où le service perdure au-delà des partitions prévues,
-- on débordera dans cette table
CREATE TABLE pgbench_history_default
PARTITION OF pgbench_history_part DEFAULT ;
-- Jusqu'ici pgbench continue de tourner en arrière plan
-- La table devient une simple partition
-- Ce renommage pose un verrou, les sessions pgbench sont bloquées
ALTER TABLE pgbench_history RENAME TO pgbench_history_orig ;
ALTER TABLE pgbench_history_part
ATTACH PARTITION pgbench_history_orig
FOR VALUES FROM (MINVALUE) TO (:'date_frontiere'::timestamptz) ;
-- Contrôle
\dP
-- Substitution de la table partitionnée à celle d'origine.
ALTER TABLE pgbench_history_part RENAME TO pgbench_history;
-- Contrôle
\d+ pgbench_history
COMMIT :
  Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.
Pour surveiller le contenu des tables jusqu'à la transition :
SELECT relname, n_live_tup, now()
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_history%';
\watch 3
Un \d+ doit renvoyer ceci:
                                    Liste des relations
 Schéma |
                                                        | Propriétaire | Taille | ...
 public | pgbench_accounts
                                   | table partitionnée | postgres
                                                                       | 0 bytes |
```

<pre>public   pgbench_accounts_1</pre>	table	postgres	44 MB	
public   pgbench_accounts_2	table	postgres	44 MB	
<pre>public   pgbench_accounts_3</pre>	table	postgres	44 MB	
<pre>public   pgbench_branches</pre>	table	postgres	136 kB	
<pre>public   pgbench_history</pre>	table partitionnée	postgres	0 bytes	
<pre>public   pgbench_history_1</pre>	table	postgres	672 kB	
<pre>public   pgbench_history_2</pre>	table	postgres	0 bytes	
<pre>public   pgbench_history_default</pre>	table	postgres	0 bytes	
<pre>public   pgbench_history_orig</pre>	table	postgres	8736 kB	
<pre>public   pgbench_tellers</pre>	table	postgres	216 kB	

## 4.10.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

```
ALTER TABLE pgbench_history
DETACH PARTITION pgbench_history_orig;
-- On pourrait faire le DROP directement
DROP TABLE pgbench_history_orig;
```

# 4.10.2.5 Contraintes entre tables partitionnées

Ajouter une clé étrangère entre pgbench\_accounts et pgbench\_history. Voir les contraintes créées.

NB: les clés étrangères entre tables partitionnées ne sont pas disponibles avant PostgreSQL 12.

```
SET lock_timeout T0 '3s';
SET statement_timeout T0 '10s';

CREATE INDEX ON pgbench_history (aid);

ALTER TABLE pgbench_history
ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid) REFERENCES

pgbench_accounts;
```

On voit que chaque partition porte un index comme la table mère. La contrainte est portée par chaque partition.

```
pgbench=# \d+ pgbench_history
   Table partitionnée « public.pgbench_history »
...
Clé de partition : RANGE (mtime)
Index :
   "pgbench_history_aid_idx" btree (aid)
Contraintes de clés étrangères :
   "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Partitions: pgbench_history_1 FOR VALUES FROM ('2020-02-14 17:41:08.298445')
```

```
TO ('2020-02-14 17:42:08.298445'),
            pgbench_history_2 FOR VALUES FROM ('2020-02-14 17:42:08.298445')
                               TO ('2020-02-14 17:43:08.298445'),
            pgbench_history_default DEFAULT
pgbench=# \d+ pgbench_history_1
   Table « public.pgbench_history_1 »
Partition de : pgbench_history FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                                           TO ('2020-02-14 17:42:08.298445')
Contrainte de partition : ((mtime IS NOT NULL)
    AND(mtime >= '2020-02-14 17:41:08.298445'::timestamp without time zone)
    AND (mtime < '2020-02-14 17:42:08.298445'::timestamp without time zone))
Index :
    "pgbench_history_1_aid_idx" btree (aid)
Contraintes de clés étrangères :
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                             FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Méthode d'accès : heap
  Si vous n'avez pas déjà eu un problème à cause du statement_timeout, dropper la contrainte
  et recommencer avec une valeur plus basse. Comment contourner?
Le statement_timeout peut être un problème :
SET
pgbench=# ALTER TABLE pgbench_history
          ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid)
          REFERENCES pgbench_accounts;
ERROR: canceling statement due to statement timeout
On peut créer les contraintes séparément sur les tables. Cela permet de ne poser un verrou sur la
partition active (sans doute pgbench_history_default) que pendant le strict minimum de temps
(les autres partitions de pgbench_history ne sont pas utilisées).
SET statement_timeout to '1s' ;
ALTER TABLE pgbench_history_1 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_2 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_default ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
La contrainte au niveau global sera alors posée presque instantanément :
ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

# 4.10.2.6 Index global

On veut créer un index sur pgbench\_history (aid).

Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

Construire un index de manière concurrente (clause CONCURRENTLY) permet de ne pas bloquer la table en écriture pendant la création de l'index, qui peut être très longue. Mais il n'est pas possible de le faire sur la table partitionnée :

Comme lors de toute création concurrente, il faut vérifier que les index sont bien valides : la requête suivante ne doit rien retourner.

```
SELECT indexrelid::regclass FROM pg_index WHERE NOT indisvalid ;
```

WHERE relname like 'pgbench\_history%' AND relispartition \gexec

Enfin on crée l'index au niveau de la table partitionnée : il réutilise les index existants et sera donc créé presque instantanément :

```
CREATE INDEX ON pgbench_history(aid) ;
pgbench=# \d+ pgbench_history
..
Partition key: RANGE (mtime)
Indexes:
    "pgbench_history_aid_idx" btree (aid)
```

# 5/ Types avancés



PostgreSQL offre des types avancés :

- UUID
- Tableaux
- Composés:
  - hstore - JSON: json, jsonb - XML
- Pour les objets binaires :

  - byteaLarge Objects

# **5.1 UUID**

# 5.1.1 UUID: principe



- Ex: d67572bf-5d8c-47a7-9457-a9ddce259f05

Un identifiant universellement unique sur 128 bits
Type: uuid
Avec des inconvénients...

Les UUID (pour Universally Unique IDentifier) sont nés d'un besoin d'avoir des identifiants uniques au niveau mondial pour divers objets, avec un risque de collision théoriquement négligeable. Ce sont des identifiants sur 128 bits.

Le standard propose plusieurs versions à cause d'un historique déjà long depuis les années 1980, et de différents algorithmes de création ou d'utilisation dans des bases de données. Il existe aussi des versions dérivées liées à certains éditeurs.

Dans une base, les clés primaires « techniques » (surrogate), servent à identifier de manière unique une ligne, sans posséder de sens propre : les UUID peuvent donc parfaitement remplacer les numéros de séquence traditionnels. Ce n'est pas toujours une bonne idée.

#### Références:

- Pages Wikipédia francophone<sup>1</sup> et anglophone<sup>2</sup>;
- RFC 4122<sup>3</sup>, ISO/IEC 9834-8:2005 (juillet 2005), définissant les UUID versions 1 à 5 et leurs utilisa-
- RFC 9562<sup>4</sup> (mai 2024), définissant les UUID versions 1 à 8;
- Un excellent article de Victor Adossi sur le choix de différents types de clés primaires<sup>5</sup>, les différents types d'UUID, et des benchmarks de génération d'UUID en masse (septembre 2022).

<sup>&</sup>lt;sup>1</sup>https://fr.wikipedia.org/wiki/Universally\_unique\_identifier

<sup>&</sup>lt;sup>2</sup>https://fr.wikipedia.org/wiki/Universally unique identifier

<sup>&</sup>lt;sup>3</sup>https://datatracker.ietf.org/doc/html/rfc4122

<sup>4</sup>https://datatracker.ietf.org/doc/html/rfc9562

<sup>&</sup>lt;sup>5</sup>https://supabase.com/blog/choosing-a-postgres-primary-key

## 5.1.2 UUID: avantages



- Faciles à générer
  - gen\_random\_uuid() et d'autres
- Pouvoir désigner des entités arbitrairement
- Pouvoir exporter des données de sources différentes
  - au contraire des séquences traditionnelles
- Pouvoir fusionner des bases
- Pour toutes les clés primaires ?

Sous PostgreSQL, nous verrons que de simples fonctions comme <code>gen\_random\_uuid()</code>, ou celles de l'extension standard <code>uuid-ossp</code>, permettent de générer des UUID aussi facilement que des numéros de séquences. Il est bien sûr possible que ces UUID soient fournis par des applications extérieures.

Généralement, les clés primaires des tables proviennent d'entiers générés successivement (séquences), généralement en partant de 1. L'unicité des identifiants est ainsi facilement garantie. Cela ne pose aucun souci jusqu'au jour où les données sont à rapprocher de données d'une autre base. Il y a de bonnes chances que les deux bases utilisent les mêmes identifiants pour des choses différentes. Souvent, une clé fonctionnelle (unique aussi) permet de faire le lien (commande DALIBO-CRA-1234, personne de numéro 25502123123 ...) mais ce n'est pas toujours le cas et des erreurs de génération sont possibles. Un UUID arbitraire et unique est une solution facile pour nommer n'importe quelle entité logique ou physique sans risque de collision avec les identifiants d'un autre système.

Les UUID sont parfaits s'il y a des cas où il faut fusionner des bases de données issues de plusieurs bases de même structure. Cela peut arriver dans certains contextes distribués ou multitenants.

Hormis ce cas particulier, ils sont surtout utiles pour identifier un ensemble de données échangés entre deux systèmes (que ce soit en JSON, CSV ou un autre moyen) : l'UUID devient une sorte de clé primaire publique. En interne, les deux bases peuvent continuer à utiliser des séquences classiques pour leurs jointures.

Il est techniquement possible de pousser la logique jusqu'au bout et de décider que chaque clé primaire d'une ligne sera un UUID et non un entier, et de joindre sur ces UUID.

Des numéros de séquence consécutifs peuvent se deviner (dans l'URL d'un site web par exemple), ce qui peut être une faille de sécurité. Des UUID (apparemment) aléatoires ne présentent pas ce problème... si l'on a bien choisi la version d'UUID (voir plus loin).

# 5.1.3 UUID: inconvénients



- Lisibilité
- Temps de génération (mineur)
- Taille
  - 16 octets...
- Surtout: fragmentation des index
  - mauvais pour le cache
  - sauf UUID v7 (tout récent)

#### Lisibilité :

Le premier inconvénient n'est pas technique mais humain : il est plus aisé de lire et retenir des valeurs courtes comme un ticket 10023, une commande 2024-67 ou une immatriculation AT-389-RC que « d67572bf-5d8c-47a7-9457-a9ddce259f05 ». Les UUID sont donc à réserver aux clés techniques. De même, un développeur qui consulte une base retiendra et discernera plus facilement des valeurs entre 1 000 et 100 000 que des UUID à première vue aléatoires, et surtout à rallonge.

Pour la base de données, il y a d'autres inconvénients :

# Taille:

Le type uuid de PostgreSQL prend 128 bits, donc 16 octets. Les types numériques entiers de PostgreSQL<sup>6</sup> utilisent 2 octets pour un smallint (int2, de -32768 à +32767), 4 pour un integer (de -2 à +2 milliards environ), 8 pour un bigint (int8, de -9.10<sup>18</sup> à +9.10<sup>18</sup> environ). Ces types entiers suffisent généralement à combler les besoins, tout en permettant de choisir le type le plus petit possible. On a donc une différence de 8 octets par ligne entre uuid et bigint, à multiplier par autant de lignes, parfois des milliards.

Cette différence s'amplifie tout le long de l'utilisation de la clé :

- index plus gros (puisque ces champs sont toujours indexés);
- clés étrangères plus grosses, avec leurs index;
- jointures plus gourmandes en CPU, mémoire, voire disque;
- avec un impact sur le cache.

Ce n'est pas forcément bloquant si votre utilisation le nécessite.

<sup>&</sup>lt;sup>6</sup>https://docs.postgresql.fr/current/datatype-numeric.html



Le pire est le stockage d'UUID dans un champ varchar : la taille passe à 36, les jointures sont bien plus lourdes, et la garantie d'avoir un véritable UUID disparaît!

## Temps de génération :

Selon l'algorithme de génération utilisé, la création d'un UUID peut être plusieurs fois plus lente que celle d'un numéro de séquence. Mais ce n'est pas vraiment un souci avec les processeurs modernes, qui sont capables de générer des dizaines, voire des centaines de milliers d'UUID, aléatoires ou pas, par seconde.

# Fragmentation des index:

Le plus gros problème des UUID vient de leur apparence aléatoire. Cela a un impact sur la fragmentation des index et leur utilisation du cache.

Parlons d'abord de l'insertion de nouvelles lignes. Par défaut, les UUID sont générés en utilisant la version 4. Elle repose sur un algorithme générant des nombres aléatoires. Par conséquent, les UUID produits sont imprévisibles. Cela peut entraîner de fréquents *splits* des pages d'index (division d'une page pleine qui doit accueillir de nouvelles entrées). Les conséquences directes sont la fragmentation de l'index, une augmentation de sa taille (avec un effet négatif sur le cache), et l'augmentation du nombre d'accès disques (en lecture et écriture).

De plus, toujours avec des UUID version 4, comme les mises à jour sont réparties sur toute la longueur des index, ceux-ci tendent à rester entièrement dans le cache de PostgreSQL. Si celui-ci est insuffisant, des accès disques aléatoires fréquents peuvent devenir gênants.

À l'inverse, une séquence génère des valeurs strictement croissantes, donc toujours insérées à la fin de l'index. Non seulement la fragmentation reste basse, mais la partie utile de l'index, en cache, reste petite.

Évidemment, tout cela devient plus complexe quand on modifie ensuite les lignes. Mais beaucoup d'applications ont tendance à modifier surtout les lignes récentes, et délaissent les blocs d'index des lignes anciennes.

Pour un index qui reste petit, donc une table statique ou dont les anciennes lignes sont vite supprimées, ce n'est pas vraiment un problème. Mais un modèle où chaque clé de table et chaque clé étrangère est un index a intérêt à pouvoir garder tous ces index en mémoire.

Récemment, une solution standardisée est apparue avec les UUID version 7 (standardisés dans la RFC 9562<sup>7</sup> en 2024) : ces UUID utilisent l'heure de génération et sont donc triés. Le souci de pollution du cache disparaît donc.

<sup>&</sup>lt;sup>7</sup>https://datatracker.ietf.org/doc/html/rfc9562

# 5.1.4 UUID: utilisation sous PostgreSQL



- uuid:
  - type simple (16 octets)
  - garantit le format
- Génération:
  - gen\_random\_uuid (v4, aléatoire)
  - extension uuid-ossp (v1,3,4,5)
  - extension (pg\_idkit ...) ou fonction en SQL (v7)

Le type uuid est connu de PostgreSQL, c'est un champ simple de taille fixe. Si l'UUID provient de l'extérieur, le type garantit qu'il s'agit d'un UUID valide.

# gen\_random\_uuid():

Générer un UUID depuis le SQL est très simple avec la fonction gen\_random\_uuid() 8:

```
SELECT gen_random_uuid() FROM generate_series (1,4) ;
```

Les UUID générés sont de version 4, c'est-à-dire totalement aléatoires, avec tous les inconvénients vus ci-dessus.

## uuid-ossp:

La fonction <code>gen\_random\_uuid()</code> n'est disponible directement que depuis PostgreSQL 13. Auparavant, il fallait forcément utiliser une extension : soit <code>pgcrypto</code> <sup>9</sup>, qui fournissait cette fonction, soit <code>uuid-ossp</code> <sup>10</sup>, toutes deux livrées avec PostgreSQL. <code>uuid-ossp</code> reste utile car elle fournit plusieurs algorithmes de génération d'UUID avec les fonctions suivantes.

Avec uuid\_generate\_v1(), l'UUID généré est lié à l'adresse MAC de la machine et à l'heure.

<sup>8</sup>https://www.postgresql.org/docs/current/functions-uuid.html

<sup>&</sup>lt;sup>9</sup>https://www.postgresql.org/docs/current/pgcrypto.html

<sup>&</sup>lt;sup>10</sup>https://www.postgresql.org/docs/current/uuid-ossp.html



Cette propriété peut faciliter la prédiction de futures valeurs d'UUID. Les UUID v1 peuvent donc être considérés comme une faille de sécurité dans certains contextes.

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
SELECT uuid_generate_v1() from generate_series(1,5) ;
          uuid_generate_v1
82e94192-45e3-11ef-92e5-04cf4b21f39a
82e94193-45e3-11ef-92e5-04cf4b21f39a
82e94194-45e3-11ef-92e5-04cf4b21f39a
 82e94195-45e3-11ef-92e5-04cf4b21f39a
Sur une autre machine:
SELECT uuid_generate_v1(),pg_sleep(5) from generate_series(1,5);
          uuid_generate_v1
                                  | pg_sleep
____________
ef5078b4-45e3-11ef-a2d4-67bc5acec5f2 |
 f24c2982-45e3-11ef-a2d4-67bc5acec5f2
 f547b552-45e3-11ef-a2d4-67bc5acec5f2
 f84345aa-45e3-11ef-a2d4-67bc5acec5f2
 fb3ed120-45e3-11ef-a2d4-67bc5acec5f2 |
```

Noter que le problème de fragmentation des index se pose déjà.

Il existe une version uuid\_generate\_v1mc() un peu plus sécurisée.

uuid\_generate\_v3() et uuid\_generate\_v5() génèrent des valeurs reproductibles en fonction des paramètres. La version 5 utilise un algorithme plus sûr.

uuid\_generate\_v4 génère un UUID totalement aléatoire, comme gen\_random\_uuid().

#### **UUID** version 7:

PostgreSQL ne sait pas encore générer d'UUID en version 7. Il existe cependant plusieurs extensions dédiées, avec les soucis habituels de disponibilité de paquets, maintenance des versions, confiance dans le mainteneur et disponibilité dans un PostgreSQL en SaaS. Par exemple, Supabase propose pg\_idkit (versions Rust<sup>11</sup>, et PL/pgSQL<sup>12</sup>).

Le plus simple est sans doute d'utiliser la fonction SQL suivante, de Kyle Hubert, modifiée par Daniel Vérité<sup>13</sup>. Elle est sans doute suffisamment rapide pour la plupart des besoins.

```
CREATE FUNCTION uuidv7() RETURNS uuid
AS $$
```

```
-- Replace the first 48 bits of a uuidv4 with the current
-- number of milliseconds since 1970-01-01 UTC
-- and set the "ver" field to 7 by setting additional bits
```

<sup>11</sup>https://github.com/VADOSWARE/pg\_idkit

<sup>&</sup>lt;sup>12</sup>https://github.com/kiwicopple/pg-extensions/tree/main/pg\_idkit

<sup>&</sup>lt;sup>13</sup>https://postgresql.verite.pro/blog/2024/07/15/uuid-v7-pure-sql.html

Il existe une version plus lente avec une précision inférieure à la milliseconde. Le même billet de blog offre une fonction retrouvant l'heure de création d'un UUID v7 :

```
CREATE FUNCTION uuidv7_extract_timestamp(uuid) RETURNS timestamptz
SELECT to_timestamp(
   right(substring(uuid_send($1) FROM 1 for 6)::text, -1)::bit(48)::int8
    /1000.0);
$$ LANGUAGE sql IMMUTABLE STRICT;
-- 10 UUID v 7 espacés de 3 secondes
WITH us AS (SELECT uuidv7() AS u, pg_sleep(3)
            FROM generate_series (1,10))
SELECT u, uuidv7_extract_timestamp(u)
FROM us;
                                    | uuidv7_extract_timestamp
_____+
0190cbaf-7879-7a4c-9ee3-8d383157b5cc | 2024-07-19 17:49:52.889+02
0190cbaf-8435-7bb8-8417-30376a2e7251 | 2024-07-19 17:49:55.893+02
0190cbaf-8fef-7535-8fd6-ab7316259338 | 2024-07-19 17:49:58.895+02
0190cbaf-9baa-74f3-aa9e-bf2d2fa84e68 | 2024-07-19 17:50:01.898+02
0190cbaf-a766-7ef6-871d-2f25e217a6ea | 2024-07-19 17:50:04.902+02
0190cbaf-b321-717b-8d42-5969de7e7c1e | 2024-07-19 17:50:07.905+02
0190cbaf-bedb-79c1-b67d-0034d51ac1ad | 2024-07-19 17:50:10.907+02
0190cbaf-ca95-7d70-a8c0-f4daa60cbe21 | 2024-07-19 17:50:13.909+02
0190cbaf-d64f-7ffe-89cd-987377b2cc07 | 2024-07-19 17:50:16.911+02
0190cbaf-e20a-7260-95d6-32fec0a7e472 | 2024-07-19 17:50:19.914+02
```

Ils sont classés à la suite dans l'index, ce qui est tout l'intérêt de la version 7.

Noter que cette fonction économise les 8 octets par ligne d'un champ creation\_date, que beaucoup de développeurs ajoutent.

# Création de table :

```
Utilisez une clause DEFAULT pour générer l'UUID à la volée :
```

## 5.1.5 UUID: une indexation facile



CREATE UNIQUE INDEX ON nomtable USING btree (champ\_uuid);

L'index B-tree classique convient parfaitement pour trier des UUID. En général on le veut UNIQUE (plus pour parer à des erreurs humaines qu'à de très improbables collisions dans l'algorithme de génération).

## 5.1.6 UUID: résumé



- Si vous avez besoin des UUID, préférez la version 7.
  Les séquences habituelles restent recommandables en interne.

Si des données doivent être échangées avec d'autres systèmes, les UUID sont un excellent moyen de garantir l'unicité d'identifiants universels.

Si vous les générez vous-mêmes, préférez les UUID version 7. Des UUID v4 (totalement aléatoires) restent sinon recommandables, avec les soucis potentiels de cache et de fragmentation évoqués cidessus.

Pour les jointures internes à l'applicatif, conservez les séquences habituelles, (notamment avec GENERATED ALWAYS AS IDENTITY), ne serait-ce que pour leur simplicité.

# **5.2 TYPES TABLEAUX**

# 5.2.1 Tableaux: principe



- Types integer[], text[], etc.

- Collection ordonnée d'un même type

```
SELECT ARRAY [1,2,3];
SELECT '{1,2,3}'::integer[];
-- lignes vers tableau
SELECT array_agg (i) FROM generate_series (1,3) i;
-- tableau vers lignes
SELECT unnest ( '{1,2,3}'::integer[] );
CREATE TABLE demotab ( id int, liste integer[] );
```

Une tableau est un ensemble d'objets d'un même type. Ce type de base est généralement un numérique ou une chaîne, mais ce peut être un type structuré (géométrique, JSON, type personnalisé...), voire un type tableau. Les tableaux peuvent être multidimensionnels.

Un tableau se crée par exemple avec le constructeur ARRAY, avec la syntaxe {...}::type[], ou en agrégeant des lignes existantes avec array\_agg . À l'inverse, on peut transformer un tableau en lignes grâce à la fonction unnest . Les syntaxes [numéro] et [début:fin] permettent d'extraire un élément ou une partie d'un tableau. Deux tableaux se concatènent avec [].



Les tableaux sont ordonnés, ce ne sont pas des ensembles. Deux tableaux avec les mêmes données dans un ordre différent ne sont pas identiques.

#### Références :

- Documentation officielle:
  - syntaxe et exemples<sup>14</sup>;
  - constructeur de tableau ARRAY<sup>15</sup>;

<sup>&</sup>lt;sup>14</sup>https://docs.postgresql.fr/current/arrays.html

 $<sup>^{15}</sup> https://docs.postgresql.fr/current/sql-expressions.html \# SQL-SYNTAX-ARRAY-CONSTRUCTORS$ 

- fonctions et opérateurs tableau<sup>16</sup> : array\_to\_string, string\_to\_array, unnest, array\_agg, array\_length, array\_cat, array\_append, array\_prepend, cardinality, array\_position / array\_positions, array\_fill, array\_remove, array\_shuffle, trim\_array ...

#### 5.2.2 Tableaux: recherche de valeurs



```
-- Recherche de valeurs (toutes)
           SELECT * FROM demotab
           WHERE liste @> ARRAY[15,11] ;
            -- Au moins 1 élément commun ?
            SELECT * FROM demotab
           WHERE liste && ARRAY[11,55] ;
            -- 1 tableau exact
           SELECT * FROM demotab
            WHERE liste = '{11,55}'::int[] ;
CREATE TABLE demotab ( id int, liste int[] );
INSERT INTO demotab (id, liste)
SELECT i, array_agg (j)
FROM generate_series (1,5) i,
LATERAL generate_series (i*10, i*10+5) j
GROUP BY i
TABLE demotab ;
        liste
 id |
  1 | {10,11,12,13,14,15}
  3 | {30,31,32,33,34,35}
  5 | {50,51,52,53,54,55}
  4 | {40,41,42,43,44,45}
  2 | {20,21,22,23,24,25}
Recherchons des lignes contenant certaines valeurs :
-- Ceci échoue car 11 n'est le PREMIER élément sur aucune ligne
SELECT * FROM demotab
WHERE liste[1] = 11 ;
-- Recherche de la ligne qui contient 11
SELECT * FROM demotab
WHERE liste @> ARRAY[11] ;
```

<sup>16</sup>https://docs.postgresql.fr/current/arrays.html

## 5.2.3 Tableaux: performances



Quel intérêt par rapport à 1 valeur par ligne?

- Allège certains modèles en réduisant les jointures
  - ex : champ avec des n°s de téléphone
  - si pas de contraintes
- Grosse économie en place (time series)
  - 24 octets par ligne
  - et parfois mécanisme TOAST
- Mais...
  - mise à jour compliquée/lente
  - indexation moins simple

Une bonne modélisation aboutit en général à des valeurs uniques, chacune dans son champ sur sa ligne. Les tableaux stockent plusieurs valeurs dans un même champ d'une ligne, et vont donc à l'encontre de cette bonne pratique.

Cependant les tableaux peuvent être très pratiques pour alléger la modélisation sans tomber dans de mauvaises pratiques. Typiquement, on remplacera :

```
-- Mauvaise pratique : champs identiques séparés à nombre fixe CREATE TABLE personnes ( ...
```

```
telephone1 text,
telephone2 text ... );
par:

CREATE TABLE personnes ( ...
telephones text[] );
```

Une table des numéros de téléphone serait *stricto censu* plus propre et flexible, mais induirait des jointures supplémentaires. De plus, il est impossible de poser des contraintes de validation (CHECK) sur des éléments de tableau sans créer un type intermédiaire. (Dans des cas plus complexes où il faut typer le numéro, on peut utiliser un tableau d'un type structuré, ou basculer vers un type JSON, qui peut lui-même contenir des tableaux, mais a un maniement un peu moins évident. L'intérêt du type structuré sur un champ JSON ou hstore est qu'il est plus compact, mais évidemment sans aucune flexibilité.)

Quand il y a beaucoup de lignes et peu de valeurs sur celles-ci, il faut se rappeler que chaque ligne d'une base de données PostgreSQL a un coût d'au moins 24 octets de données « administratives », même si l'on ne stocke qu'un entier par ligne, par exemple. Agréger des valeurs dans un tableau permet de réduire mécaniquement le nombre de lignes et la volumétrie sur le disque et celles des écritures. Par contre, le développeur devra savoir comment utiliser ces tableaux, comment retrouver une valeur donnée à l'intérieur d'un champ multicolonne, et comment le faire efficacement.

De plus, si le champ concaténé est assez gros (typiquement 2 ko), le mécanisme TOAST<sup>17</sup> peut s'activer et procéder à la compression du champ tableau, ou à son déport dans une table système de manière transparente.

Les tableaux peuvent donc permettre un gros gain de volumétrie. De plus, les données d'un même tableau, souvent utilisées ensemble, restent forcément dans le même bloc. L'effet sur le cache est donc extrêmement intéressant.

Par exemple, ces deux tables contiennent 6,3 millions de valeurs réparties sur 366 jours :

```
-- Table avec 1 valeur/liane
CREATE TABLE serieparligne (d timestamptz PRIMARY KEY,
                           valeur int );
INSERT INTO serieparligne
SELECT d, extract (hour from d)
FROM generate_series ('2020-01-01'::timestamptz,
                      '2020-12-31'::timestamptz, interval '5 s') d;
SET default_toast_compression TO lz4 ; -- conseillé pour PG >= 14
-- Table avec les 17280 valeurs du jour sur 366 lignes :
CREATE TABLE serieparjour (d
                                    date PRIMARY KEY,
                       valeurs int[]
                       );
INSERT INTO serieparjour
SELECT d::date, array_agg ( extract (hour from d) )
FROM generate_series ('2020-01-01'::timestamptz,
```

<sup>&</sup>lt;sup>17</sup>https://dali.bo/m4\_html#m%C3%A9canisme-toast

```
'2020-12-31'::timestamptz, interval '5 s') d
GROUP BY d::date ;
La différence de taille est d'un facteur 1000 :
ANALYZE serieparjour, serieparligne;
SELECT
   c.relnamespace::regnamespace || '.' || relname AS TABLE,
   reltuples AS nb_lignes_estimees,
   pg_size_pretty(pg_table_size(c.oid)) AS " Table (dont TOAST)",
   pg_size_pretty(pg_relation_size(c.oid)) AS " Heap",
   pg_size_pretty(pg_relation_size(reltoastrelid)) AS "
   pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
   pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM pg_class c
WHERE relkind = 'r'
AND relname like 'seriepar%';
-[ RECORD 1 ]-----
                   | public.serieparjour
nb_lignes_estimees | 366
Table (dont TOAST) | 200 kB
                   | 168 kB
 Toast
                   | 0 bytes
Index
                  | 16 kB
                  | 216 kB
Total
-[ RECORD 2 ]-----
table
                   | public.serieparligne
nb_lignes_estimees | 6.3072e+06
Table (dont TOAST) | 266 MB
 Heap
                     266 MB
 Toast
                    Ø
Index
                   | 135 MB
Total
                   | 402 MB
```

Ce cas est certes extrême (beaucoup de valeurs par ligne et peu de valeurs distinctes).

Les cas réels sont plus complexes, avec un horodatage moins régulier. Par exemple, l'outil OPM stocke plutôt des tableaux d'un type composé d'une date et de la valeur relevée<sup>18</sup>, et non la valeur seule.

Dans beaucoup de cas, cette technique assez simple évite de recourir à des extensions spécialisées payantes comme TimescaleDB<sup>19</sup>, voire à des bases de données spécialisées.

<sup>18</sup>https://github.com/OPMDG/opm-core/blob/ae89f025407ab144e1e30abd7d6580f258945d61/pg/opm\_core-2.6.sql#L41

<sup>&</sup>lt;sup>19</sup>https://www.timescale.com/products



Évidemment, le code devient moins simple. Selon les besoins, il peut y avoir besoin de stockage temporaire, de fonctions de compactage périodique...

Il devient plus compliqué de retrouver une valeur précise. Ce n'est pas trop un souci dans les cas pour une recherche ou pré-sélection à partir d'un autre critère (ici la date, indexée). Pour la recherche dans les tableaux, voir plus bas.

Les mises à jour des données à l'intérieur d'un tableau deviennent moins faciles et peuvent être lourdes en CPU avec de trop gros tableaux.

#### 5.2.4 Tableaux: indexation



- B-tree inutilisable
- GIN plus adapté pour recherche d'une valeur
  - opérateurs && , @>
  - mais plus lourd

Indexer un champ tableau est techniquement possible avec un index B-tree, le type d'index par défaut. Mais cet index est en pratique peu performant. De toute façon il ne permet que de chercher un tableau entier (ordonné) comme critère.

Dans les exemples précédents, les index B-tree sont plutôt à placer sur un autre champ (la date, l'ID), qui ramène une ligne entière.

D'autres cas nécessitent de chercher une valeur parmi celles des tableaux (par exemple dans des listes de propriétés). Dans ce cas, un index GIN sera plus adapté, même si cet index est un peu lourd. Les opérateurs | | et @> sont utilisables. La valeur recherchée peut être n'importe où dans les tableaux.

```
TRUNCATE TABLE demotab;
-- 500 000 lignes

INSERT INTO demotab (id, liste)

SELECT i, array_agg (j)

FROM generate_series (1,500000) i,

LATERAL generate_series (mod(i*10,100000), mod(i*10,100000)+5) j

GROUP BY i;

CREATE INDEX demotab_gin ON demotab USING gin (liste);

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demotab

WHERE liste @> ARRAY[45];

QUERY PLAN

Bitmap Heap Scan on demotab (cost=183.17..2377.32 rows=2732 width=49) (actual

time=0.958..1.014 rows=50 loops=1)
```

Là encore, on récupère les tableaux entiers qui contiennent la valeur demandée. Selon le besoin, il faudra peut-être reparcourir les éléments récupérés, ce qui coûtera un peu de CPU:

```
EXPLAIN (ANALYZE)
SELECT id,
    (SELECT count(*) FROM unnest (liste) e WHERE e=45) AS nb_occurences_45
FROM demotab
WHERE liste @> ARRAY[45];
                            QUERY PLAN
_____
Bitmap Heap Scan on demotab (cost=23.37..2417.62 rows=2500 width=12) (actual

    time=0.067..0.325 rows=50 loops=1)

  Recheck Cond: (liste @> '{45}'::integer[])
  Heap Blocks: exact=50
  -> Bitmap Index Scan on demotab_gin (cost=0.00..22.75 rows=2500 width=0) (actual

    time=0.024..0.024 rows=50 loops=1)

         Index Cond: (liste @> '{45}'::integer[])
   SubPlan 1
    -> Aggregate (cost=0.13..0.14 rows=1 width=8) (actual time=0.003..0.003 rows=1
   loops=50)
              Function Scan on unnest e (cost=0.00..0.13 rows=1 width=0) (actual

    time=0.002..0.002 rows=1 loops=50)

                Filter: (e = 45)
                Rows Removed by Filter: 5
Planning Time: 0.240 ms
 Execution Time: 0.388 ms
```

Quant aux recherches sur une plage de valeurs dans les tableaux, elles ne sont pas directement indexables par un index GIN.

Pour les détails sur les index GIN, voir le module J5<sup>20</sup>.

<sup>&</sup>lt;sup>20</sup>https://dali.bo/j5\_html#gin-et-les-tableaux

# **5.3 TYPES COMPOSÉS**

# 5.3.1 Types composés: généralités



- Un champ = plusieurs attributs
- De loin préférable à une table Entité/Attribut/Valeur
- Uniquement si le modèle relationnel n'est pas assez souple
- 3 types dans PostgreSQL:
  - hstore : clé/valeur
  - json: JSON, stockage texte, validation syntaxique, fonctions d'extraction
  - jsonb: JSON, stockage binaire, accès rapide, fonctions d'extraction, de requêtage, indexation avancée

Ces types sont utilisés quand le modèle relationnel n'est pas assez souple, donc s'il est nécessaire d'ajouter dynamiquement des colonnes à la table suivant les besoins du client, ou si le détail des attributs d'une entité n'est pas connu (modélisation géographique par exemple), etc.

La solution traditionnelle est de créer des tables entité/attribut de ce format :

```
CREATE TABLE attributs_sup (entite int, attribut text, valeur text);
```

On y stocke dans entite la clé de l'enregistrement de la table principale, dans attribut la colonne supplémentaire, et dans valeur la valeur de cet attribut. Ce modèle présente l'avantage évident de résoudre le problème. Les défauts sont par contre nombreux :

- Les attributs d'une ligne peuvent être totalement éparpillés dans la table attributs\_sup : récupérer n'importe quelle information demandera donc des accès à de nombreux blocs différents.
- Il faudra plusieurs requêtes (au moins deux) pour récupérer le détail d'un enregistrement, avec du code plus lourd côté client pour fusionner le résultat des deux requêtes, ou bien une requête effectuant des jointures (autant que d'attributs, sachant que le nombre de jointures complexifie énormément le travail de l'optimiseur SQL) pour retourner directement l'enregistrement complet.

Toute recherche complexe est très inefficace: une recherche multicritère sur ce schéma va être extrêmement peu performante. Les statistiques sur les valeurs d'un attribut deviennent nettement moins faciles à estimer pour PostgreSQL. Quant aux contraintes d'intégrité entre valeurs, elles deviennent pour le moins complexes à gérer.

Les types hstore, json et jsonb permettent de résoudre le problème autrement. Ils stockent les différentes entités dans un seul champ pour chaque ligne de l'entité. L'accès aux attributs se fait par une syntaxe ou des fonctions spécifiques.

# **DALIBO Formations**

Il n'y a même pas besoin de créer une table des attributs séparée : le mécanisme du TOAST<sup>21</sup> permet de déporter les champs volumineux (texte, JSON, hstore ...) dans une table séparée gérée par PostgreSQL, éventuellement en les compressant, et cela de manière totalement transparente. On y gagne donc en simplicité de développement.

<sup>&</sup>lt;sup>21</sup>https://dali.bo/m4\_html#m%C3%A9canisme-toast

# **5.4 HSTORE**

# 5.4.1 hstore: principe



Stocker des données non structurées

- Extension
- Stockage Clé/Valeur, uniquement texte
- Binaire
- Indexable
- Plusieurs opérateurs disponibles

**hstore** est une extension, fournie en « contrib ». Elle est donc systématiquement disponible. L'installer permet d'utiliser le type de même nom. On peut ainsi stocker un ensemble de clés/valeurs, exclusivement textes, dans un unique champ.

Ces champs sont indexables et peuvent recevoir des contraintes d'intégrité (unicité, non recouvrement...).

Les <u>hstore</u> ne permettent par contre qu'un modèle « plat ». Il s'agit d'un pur stockage clé-valeur. Si vous avez besoin de stocker des informations davantage orientées document, vous devrez vous tourner vers un type JSON.

Ce type perd donc de son intérêt depuis que PostgreSQL 9.4 a apporté le type jsonb. Il lui reste sa simplicité d'utilisation.

## 5.4.2 hstore: exemple



Les ordres précédents installent l'extension, créent une table avec un champ de type hstore, insèrent trois lignes, avec des attributs variant sur chacune, indexent l'ensemble avec un index GiST, et enfin recherchent les lignes où l'attribut carnivore possède la valeur t.

Les différentes fonctions disponibles sont bien sûr dans la documentation<sup>22</sup>.

## Par exemple:

L'indexation de ces champs peut se faire avec divers types d'index. Un index unique n'est possible qu'avec un index B-tree classique. Les index GIN ou GiST sont utiles pour rechercher des valeurs d'un attribut. Les index hash ne sont utiles que pour des recherches d'égalité d'un champ entier; par contre ils sont très compacts.

# **5.5 JSON**

# 5.5.1 JSON & PostgreSQL



```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "age": 27,
  "address": {
     "streetAddress": "43 rue du Faubourg Montmartre",
     "city": "Paris",
     "postalCode": "75009"
  }
}

- json : format texte
- jsonb : format binaire, à préférer
- jsonpath : SQL/JSON paths (requêtage)
```

Le format JSON<sup>23</sup> est devenu extrêmement populaire. Au-delà d'un simple stockage clé/valeur, il permet de stocker des tableaux, ou des hiérarchies, de manière plus simple et lisible qu'en XML. Par exemple, pour décrire une personne, on peut utiliser cette structure :

```
"firstName": "Jean",
"lastName": "Dupont",
"isAlive": true,
"age": 27,
"address": {
  "streetAddress": "43 rue du Faubourg Montmartre",
  "city": "Paris",
  "state": "".
  "postalCode": "75002"
},
"phoneNumbers": [
    "type": "personnel",
    "number": "06 12 34 56 78"
  },
    "type": "bureau",
    "number": "07 89 10 11 12"
  }
"children": [],
```

<sup>&</sup>lt;sup>23</sup>https://fr.wikipedia.org/wiki/JavaScript\_Object\_Notation

```
"spouse": null
}
```

Historiquement, le JSON est apparu dans PostgreSQL 9.2, mais n'est vraiment utilisable qu'avec l'arrivée du type j sonb (binaire) dans PostgreSQL 9.4. Ce dernier est le type à utiliser.

Les opérateurs SQL/JSON path ont été ajoutés dans PostgreSQL 12<sup>24</sup>, suite à l'introduction du JSON dans le standard SQL:2016.

# 5.5.2 Type json



- Type texte
  avec validation du format
  Réservé au stockage à l'identique
  Préférer jsonb

Le type natif json, dans PostgreSQL, n'est rien d'autre qu'un habillage autour du type texte. Il valide à chaque insertion/modification que la donnée fournie est une syntaxe JSON valide. Le stockage est exactement le même qu'une chaîne de texte, et utilise le mécanisme du TOAST<sup>25</sup>, qui compresse les grands champs au besoin, de manière transparente pour l'utilisateur. Le fait que la donnée soit validée comme du JSON permet d'utiliser des fonctions de manipulation, comme l'extraction d'un attribut, la conversion d'un JSON en enregistrement, de façon systématique sur un champ sans craindre d'erreur.

Mais on préférera généralement le type binaire j sonb pour les performances, et ses fonctionnalités supplémentaires. Le seul intérêt du type j son texte est de conserver un objet JSON sous sa forme originale, y compris l'ordre des clés, les espaces inutiles compris, et les clés dupliquées (la dernière étant celle prise en compte) :

```
SELECT '{"cle2": 0, "cle1": 6, "cle2": 4, "cle3": 17}'::json;
                     json
{"cle2": 0, "cle1": 6, "cle2": 4, "cle3": 17}
SELECT '{"cle2": 0, "cle1": 6, "cle2": 4, "cle3": 17}'::jsonb;
             jsonb
{"cle1": 6, "cle2": 4, "cle3": 17}
```

<sup>&</sup>lt;sup>24</sup>https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

<sup>&</sup>lt;sup>25</sup>https://dali.bo/m4\_html#mécanisme-toast

Une partie des exemples suivants avec le type jsonb est aussi applicable au json. Beaucoup de fonctions existent sous les deux formes (par exemple json\_build\_object et jsonb\_build\_object), mais beaucoup d'autres sont propres au type jsonb.

# 5.5.3 Type jsonb



- JSON au format BinaireIndexation GINLangage JSONPath

Le type jsonb permet de stocker les données dans un format binaire optimisé. Ainsi, il n'est plus nécessaire de désérialiser l'intégralité du document pour accéder à une propriété.

Les gains en performance sont importants. Par exemple une requête simple comme celle-ci :

```
SELECT personne_nom->'id' FROM json.personnes;
```

passe de 5 à 1,5 s sur une machine en convertissant le champ de json à jsonb (pour ½ million de champs JSON totalisant 900 Mo environ pour chaque version, ici sans TOAST notable et avec une table intégralement en cache).

Encore plus intéressant : jsonb supporte les index GIN, et la syntaxe JSONPath pour le requêtage et l'extraction d'attributs. j sonb est donc le type le plus intéressant pour stocker du JSON dans PostgreSQL.

# 5.5.4 Validation du format JSON



```
SELECT '{"auteur": "JRR Tolkien","titre":"Le Hobbit"}' IS JSON;
SELECT '{"auteur": "JRR Tolkien","titre":"Le Hobbit}' IS JSON;
 -- false
   - Aussi : IS JSON WITH/WITHOUT UNIQUE KEYS,
                                                        IS JSON ARRAY,
     IS JSON OBJECT, IS JSON SCALAR

    PostgreSQL 16+
```

À partir de PostgreSQL 16 existe le prédicat IS JSON. Il peut être appliqué sur des champs text ou bytea et évidemment sur des champs json et jsonb. Il permet de repérer notamment une faute de syntaxe comme dans le deuxième exemple ci-dessus.

#### Existent aussi:

- l'opérateur IS JSON WITH UNIQUE KEYS pour garantir l'absence de clé en doublon :

```
SELECT '{"auteur": "JRR", "auteur": "Tolkien", "titre": "Le Hobbit"}'
IS JSON WITH UNIQUE KEYS AS valid;
 valid
 f
SELECT '{"prenom": "JRR", "nom": "Tolkien", "titre": "Le Hobbit"}'
IS JSON WITH UNIQUE KEYS AS valid;
 valid
 t
   - l'opérateur IS JSON WITHOUT UNIQUE KEYS pour garantir l'absence de clé unique;
   - l'opérateur IS JSON ARRAY pour le bon formatage des tableaux :
SELECT
$$[{"auteur": "JRR Tolkien", "titre": "La confrérie de l'anneau"},
    {"auteur": "JRR Tolkien", "titre": "Les deux tours"},
    {"auteur": "JRR Tolkien", "titre": "Le retour du roi"}]$$
IS JSON ARRAY AS valid;
 valid
_____
 t
   - des opérateurs IS JSON SCALAR et IS JSON OBJECT pour valider par exemple le contenu de
      fragments d'un objet JSON.
-- NB : l'opérateur ->> renvoie un texte
SELECT '{"nom": "production", "version":"1.1"}'::json ->> 'version'
IS JSON SCALAR AS est_nombre ;
est_nombre
_____
 t
```

Noter que la RFC impose qu'un JSON soit en UTF-8, qui est l'encodage recommandé, mais pas obligatoire, d'une base PostgreSQL.

# 5.5.5 JSON: Exemple d'utilisation



```
CREATE TABLE personnes (datas jsonb );
INSERT INTO personnes (datas) VALUES ('
 "firstName": "Jean",
 "lastName": "Valjean",
 "address": {
   "streetAddress": "43 rue du Faubourg Montmartre",
   "city": "Paris",
   "postalCode": "75002"
 },
  "phoneNumbers": [
   { "number": "06 12 34 56 78" },
    { "type": "bureau", "number": "07 89 10 11 12"}
  "children": ["Cosette"]
} '), ('{
  "firstName": "Georges",
  "lastName": "Durand",
  "address": {
   "streetAddress": "27 rue des Moulins",
    "city": "Châteauneuf",
   "postalCode": "45990"
  "phoneNumbers": [
    { "number": "06 21 34 56 78" },
    { "type": "bureau", "number": "07 98 10 11 12"}
 "children": []
} '), ('{
 "firstName": "Jacques",
 "lastName": "Dupont",
 "isAlive": true,
 "age": 27,
  "address": {
   "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
   "state": "",
   "postalCode": "75002"
  "phoneNumbers": [
   {
      "type": "personnel",
      "number": "+33 1 23 45 67 89"
      "type": "bureau",
     "number": "07 00 00 01 23"
   }
 ],
 "children": [],
 "spouse": "Martine Durand"
} ');
```

Un champ de type jsonb (ou json) accepte tout champ JSON directement.

## 5.5.6 JSON: construction



```
SELECT '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb;
SELECT jsonb_build_object ('nom', 'Lagaffe', 'prenom', 'Gaston');
```

Pour construire un petit JSON, le transtypage d'une chaîne peut suffire dans les cas simples. jsonb\_build\_object permet de limiter les erreurs de syntaxe.

Dans un JSON, l'ordre n'a pas d'importance.

# **5.5.7 JSON: Affichage des attributs**



Le type json dispose de nombreuses fonctions et opérateurs de manipulation et d'extraction.

Attention au type en retour, qui peut être du texte ou du JSON. Les opérateurs ->> et -> renvoient respectivement une valeur au format texte, et au format JSON :

```
Column | Type
 prenom | text
 addr | jsonb
Pour l'affichage, la fonction jsonb_pretty améliore la lisibilité:
SELECT datas->>'firstName' AS prenom,
       jsonb_pretty (datas->'address') AS addr
FROM personnes ;
 prenom
               "city": "Paris",
                "postalCode": "75002",
                "streetAddress": "43 rue du Faubourg Montmartre"+
 Georges
                "city": "Châteauneuf",
                "postalCode": "45990",
                "streetAddress": "27 rue des Moulins"
 Jacques
                "city": "Paris",
                "state": "",
                "postalCode": "75002",
                "streetAddress": "43 rue du Faubourg Montmartre"+
L'équivalent existe avec des chemins, avec #> et #>> :
SELECT datas #>> '{address,city}' AS villes FROM personnes ;
   villes
 Paris
 Châteauneuf
Depuis la version 14, une autre syntaxe plus claire est disponible, plus simple, et qui renvoie du
  SELECT datas['address']['city'] AS villes FROM personnes ;
    villes
 "Paris"
 "Châteauneuf"
 "Paris"
Avec cette syntaxe, une petite astuce permet de convertir en texte sans utiliser ->> ['city'] (en toute
rigueur, ->>0 renverra le premier élément d'un tableau):
SELECT datas['address']['city']->>0 AS villes FROM personnes ;
```

```
villes
-----
Paris
Châteauneuf
paris
```



PostgreSQL ne contrôle absolument pas que les clés JSON (comme ici firstName ou city) sont valides ou pas. La moindre faute de frappe (ou de casse!) entraînera une valeur NULL en retour. C'est la conséquence de l'absence de schéma dans un JSON, contrepartie de sa souplesse.

#### 5.5.8 Modifier un JSON



L'opérateur || concatène deux jsonb pour donner un autre jsonb :

```
SELECT '{"nom": "Durand"}'::jsonb ||
   '{"address" : {"city": "Paris", "postalcode": "75002"}}'::jsonb ;

{"nom": "Durand", "address": {"city": "Paris", "postalcode": "75002"}}
```

Comme d'habitude, le résultat est NULL si l'un des JSON est NULL. Dans le doute, on peut utiliser comme élément neutre :

```
SELECT '{"nom": "Durand"}'::jsonb || coalesce (NULL::jsonb, '{}');
```

```
{"nom": "Durand"}
Pour supprimer un attribut d'un jsonb, il suffit de l'opérateur – et d'un champ texte indiquant
l'attribut à supprimer. Il existe une variante avec text[] pour supprimer plusieurs attributs :
SELECT '{"nom": "Durand", "prenom": "Georges",
         "address": {"city": "Paris"}}'::jsonb
       - '{nom, prenom}'::text[] ;
 {"address": {"city": "Paris"}}
ainsi que l'opérateur pour supprimer un sous-attribut :
SELECT '{"nom": "Durand",
         "address": {"city": "Paris", "postalcode": "75002"}}'::jsonb
        #- '{address,postalcode}';
 {"nom": "Durand", "address": {"city": "Paris"}}
La fonction jsonb_set | modifie l'attribut indiqué dans un jsonb |:
SELECT jsonb_set ('{"nom": "Durand", "address": {"city": "Paris"}}'::jsonb,
       '{address}',
      '{"ville": "Lyon" }'::jsonb) ;
 {"nom": "Durand", "address": {"ville": "Lyon"}}
Attention, le sous-attribut est intégralement remplacé, et non fusionné. Dans cet exemple, le code
postal disparaît :
SELECT jsonb_set ('{"nom": "Durand",
                   "address": {"postalcode": 69001, "city": "Paris"}}'::jsonb,
       '{address}',
      '{"ville": "Lyon" }'::jsonb) ;
 {"nom": "Durand", "address": {"ville": "Lyon"}}
Il vaut mieux indiquer le chemin complet en second paramètre :
```

"address": {"postalcode": 69001, "city": "Paris"}}'::jsonb,

{"nom": "Durand", "address": {"city": "Lyon", "postalcode": 69001}}

SELECT jsonb\_set ('{"nom": "Durand",

'{address, city}',
'"Lyon"'::jsonb);

## 5.5.9 JSON, tableaux et agrégation



- Manipuler des tableaux :

```
- jsonb_array_elements(), jsonb_array_elements_text()
- jsonb_agg() (de JSON ou de scalaires)
- jsonb_agg()_strict (sans les NULL)

SELECT jsonb_array_elements (datas->'phoneNumbers')->>'number'
FROM personnes;
```

Un JSON peut contenir un tableau de texte, nombre, ou autres JSON. Il est possible de déstructurer ces tableaux, mais il est compliqué de requêter sur leur contenu.

```
jsonb_array_elements permet de parcourir ces tableaux :
```

Avec la syntaxe JSONPath, le résultat est le même :

Si l'on veut retrouver un type tableau, il faut réagréger, car j sonb\_path\_query et j sonb\_array\_elements renvoient un ensemble de lignes. On peut utiliser une clause LATERAL, qui sera appelée pour chaque ligne (ce qui est lent) puis réagréger :

```
Georges | ["06 21 34 56 78", "07 98 10 11 12"]
 Jacques | ["+33 1 23 45 67 89", "07 00 00 01 23"]
On voit que la fonction jsonb_agg envoie un tableau de JSON.
Si l'on veut un tableau de textes :
SELECT datas->>'firstName' AS prenom,
        array_agg ( n->>'number' ) AS telephones
                                                                  -- text[]
FROM personnes,
LATERAL (SELECT jsonb_path_query (datas, '$.phoneNumbers[*]') ) AS nums(n)
GROUP BY prenom ;
 prenom
                       telephones
 Jean | {"06 12 34 56 78", "07 89 10 11 12"}
 Georges | {"06 21 34 56 78", "07 98 10 11 12"}
 Jacques | {"+33 1 23 45 67 89", "07 00 00 01 23"}
Noter que la clause LATERAL supprime les personnes sans téléphone. On peut utiliser LEFT OUTER JOIN LATERAL,
ou l'exemple suivant. On suppose aussi que le prénom est une clé de regroupement suffisante ; un
identifiant quelconque serait plus pertinent.
Cette autre variante avec un sous- SELECT sera plus performante avec de nombreuses lignes, car elle
évite le regroupement :
SELECT datas->>'firstName' AS prenom,
        (SELECT array_agg (nt) FROM (
          SELECT (jsonb_path_query (datas, '$.phoneNumbers[*]'))->>'number'
        ) AS nums(nt) ) AS telephones
                                                      -- text[]
FROM
       personnes;
Il existe une autre fonction d'agrégation des JSON plus pratique, nommée jsonb_agg_strict(), qui
supprime les valeurs à null de l'agrégat (mais pas un attribut à null). Pour d'autres cas, il existe
aussi jsonb_strip_nulls() pour nettoyer un JSON de toutes les valeurs null si une clé est asso-
ciée (pas dans un tableau):
SELECT jsonb_agg (usename) AS u1,
       jsonb_strip_nulls (jsonb_agg (usename)) AS u1b,
       jsonb_agg_strict (usename) AS u2
```

FROM (

) **AS** rq ;

FROM pg\_stat\_activity \gx

-[ RECORD 1 ]-----

u1 | ["postgres", null, "postgres", null, null, null]
u1b | ["postgres", null, "postgres", null, null, null]
u2 | ["postgres", "postgres"]

jsonb\_agg\_strict ( to\_jsonb(rq) ) AS a2,

jsonb\_agg ( jsonb\_strip\_nulls (to\_jsonb(rq) ) ) AS a3, jsonb\_strip\_nulls(jsonb\_agg ( to\_jsonb(rq) ) ) AS a4

SELECT jsonb\_agg ( to\_jsonb(rq) ) AS a1,

SELECT pid, usename FROM pg\_stat\_activity

## 5.5.10 Conversions jsonb / relationnel (1)



- Construire un ensemble de tuples depuis un objet JSON :
  - jsonb\_each()
- Avec des types préexistants :
  - jsonb\_populate\_record() , jsonb\_populate\_recordset
- Types définis à la volée :
  - jsonb\_to\_record() , jsonb\_to\_recordset()
- Construire un JSON depuis une requête :
  - to\_jsonb (<requête>)
- Attention aux types
  - jsonb\_typeof()

Plusieurs fonctions permettant de construire du jsonb, ou de le manipuler de manière ensembliste. (En version 17 ou supérieure, le plus simple et pratique sera souvent la fonction JSON\_TABLE présentée plus bas.)

#### jsonb\_each:

jsonb\_each décompose les clés et retourne une ligne par clé. Là encore, on multiplie le nombre de lignes.

```
select
  j.key,     -- text
  j.value     -- jsonb
FROM personnes p CROSS JOIN jsonb_each(p.datas) j;
```

```
key
                                                         value
address | {"city": "Paris", "postalCode": "75002", "streetAddress":
               "43 rue du Faubourg Montmartre"}
children | ["Cosette"]
lastName | "Valjean"
firstName
             | "Jean"
phoneNumbers | [{"number": "06 12 34 56 78"}, {"type": "bureau", "number":
              "07 89 10 11 12"}]
           | {"city": "Châteauneuf", "postalCode": "45990", "streetAddress":
address
               "27 rue des Moulins"}
children
            | []
              "Durand"
lastName
firstName
              "Georges"
phoneNumbers | [{"number": "06 21 34 56 78"}, {"type": "bureau", "number":
               "07 98 10 11 12"}]
             | 27
age
             | "Martine Durand"
spouse
```

#### jsonb\_populate\_record/jsonb\_populate\_recordset:

Si les noms des attributs et champs sont bien identiques (casse comprise!) entre JSON et table cible, jsonb\_populate\_record peut être pratique:

```
CREATE TABLE nom_prenom_age (
"firstName" text,
"lastName" text,
age int,
present boolean) ;
-- Ceci renvoie un RECORD, peu pratique :
SELECT jsonb_populate_record (null::nom_prenom_age, datas) FROM personnes;
-- Cette version renvoie des lignes
SELECT np.*
FROM personnes,
   LATERAL jsonb_populate_record (null::nom_prenom_age, datas) np ;
firstName | lastName | age | present
      | Valjean | |
Jean
Georges | Durand |
Jacques | Dupont | 27 |
```

Les attributs du JSON non récupérés sont ignorés, les valeurs absentes du JSON sont à NULL. Il existe une possibilité de créer des valeurs par défaut :

jsonb\_populate\_recordset sert dans le cas des tableaux de JSON.

Définir un type au lieu d'une table fonctionne aussi.

#### jsonb\_to\_record/jsonb\_to\_recordset:

JSON, donc bien typés. Pour fonctionner, la fonction exige une clause AS avec les attributs voulus et leur bon type. (Là encore, attention à la casse exacte des noms d'attributs sous peine de se retrouver avec des valeurs à NULL .)

Les autres attributs de notre exemple peuvent être extraits également, ou re-convertis en enregistrements avec une autre clause LATERAL. Si ces attributs sont des tableaux, on peut générer une ligne par élément de tableau avec json\_to\_recordset :

À l'inverse, transformer le résultat d'une requête en JSON est très facile avec to\_jsonb:

```
{"pid": 2425, "datname": null, "application_name": ""}
{"pid": 2424, "datname": null, "application_name": ""}
{"pid": 2426, "datname": null, "application_name": ""}
```

#### jsonb\_typeof:

Pour connaître le type d'un attribut JSON :

# 5.5.11 Conversions jsonb / relationnel (2): JSON\_TABLE



À partir de PostgreSQL 17, la fonction JSON\_TABLE() permet de présenter des données JSON sous forme de vue. C'est sans doute la manière la plus pratique.

Le premier paramètre que prend la fonction, appelé context\_item, est une source JSON sur laquelle sera évaluée l'expression JSON. Il est possible de récupérer des données présentes dans une structure imbriquée, avec la clause NESTED PATH.

#### Exemple:

Prenons l'exemple suivant d'une table créée à partir d'un fichier JSON. Le fichier d'exemple généré à l'aide de mockaroo.com peut être récupéré sur https://dali.bo/999people.

```
# Récupération du fichier d'exemple curl -L https://dali.bo/999people -o /tmp/999people.json
```

```
-- Récupération du JSON, par exemple ainsi :
-- Table de travail
CREATE TEMP TABLE t0 (coll jsonb);
-- Chargement du JSON en modifiant les délimiteurs du CSV
\copy t0 (coll) FROM '/tmp/999people.json' CSV quote e'\x01' delimiter e'\x02';
 - Les 999 lignes sont converties en autant de lignes
CREATE TABLE t1 (uid int GENERATED ALWAYS AS IDENTITY, utilisateur jsonb);
INSERT INTO t1 (utilisateur)
SELECT jsonb_array_elements (coll) AS utilisateur from t0;
-- Insertion d'une nouvelle ligne
INSERT INTO t1 (utilisateur)
SELECT '{
  "guid": "c39c493b-6759-47b8-9c3f-10b1b59ba8ef",
  "isActive": false,
  "balance": "$3,363.44",
  "age": 23,
  "name": "Claudine Howell",
  "gender": "female",
  "email": "claudinehowell@aguasseur.com",
  "address": "482 Fiske Place, Ogema, Pennsylvania, 1753",
  "registered": "2023-12-22T05:16:27 -01:00",
  "latitude": -88.430942,
  "longitude": 34.196733
}'::jsonb ;
Voici un exemple d'une ligne contenue dans la table t1:
SELECT utilisateur FROM t1 WHERE uid=1000 ;
 {"age": 23, "guid": "c39c493b-6759-47b8-9c3f-10b1b59ba8ef", "name": "Claudine 

→ Howell", "email": "claudinehowell@aquasseur.com", "gender": "female",
    "address": "482 Fiske Place, Ogema, Pennsylvania, 1753", "balance":
     "$3,363.44", "isActive": false, "latitude": -88.430942, "longitude": 34.196733,
    "registered": "2023-12-22T05:16:27 -01:00"}
Si l'on souhaite récupérer une vue avec uniquement les informations de l'âge et du nom présents dans
les lignes de la table t1, la commande suivante peut être utilisée :
SELECT resultats.*
FROM t1,
  JSON_TABLE(utilisateur,
             '$' COLUMNS (age int PATH '$.age', nom text PATH '$.name'))
  AS resultats;
 age
               nom
  40 | Selma Tippell
  39 | Leonardo Coupland
  83 | Jessica Riggott
  81 | Smith Caroll
  59 | Marthe MacCaig
  66 | Hurley Cosgra
  23 | Claudine Howell
(1000 lignes)
```

- JSON\_TABLE(utilisateur, '\$' indique que toute la source JSON (le champ de nom utilisateur) doit être utilisée lors de l'évaluation. Cette fonction sera appelée une fois pour chaque ligne de t1.
- COLUMNS (age int PATH '\$.age', name text PATH '\$.name')) est la clause qui permet d'indiquer quelles colonnes seront présentes dans la vue résultante et où retrouver les données associées.
- La clause PATH indique l'emplacement de la donnée dans la donnée JSON.

Le résultat peut être manipulé comme tout résultat de requête : consultation, export, vue...



Les champs sont correctement typés (suivant ceux indiqués dans COLUMNS ). En cas d'incohérence, il n'y aura pas d'erreur, mais une valeur NULL .

#### Filtrage des valeurs :

La clause '\$' peut être plus complexe pour ne renvoyer que certaines lignes :

La syntaxe utilisée pour les filtrages et sélections de champs est le JSON path, qui sera détaillée plus loin.

## Nested path:

La clause NESTED PATH permet d'exploiter des tableaux à l'intérieur d'un champ JSON tel que celuici, ce qui multiplie encore les lignes. Ils peuvent être numérotés avec la clause FOR ORDINALITY.

L'expression d'évaluation est légèrement plus complexe :

```
SELECT t1.uid, resultats.*
FROM t1,
     JSON_TABLE(t1.utilisateur,
           '$' COLUMNS (
           nom text PATH '$.name',
           NESTED PATH '$.friends[*]'
             COLUMNS (
                numero FOR ORDINALITY,
                 ami text PATH '$.name' ))
                ) AS resultats
WHERE t1.uid >= 1000 ;
                     numero
            nom
                                       ami
(4 lignes)
Il existe une clause PASSING pour transmettre des critères de recherche proprement. Par exemple,
pour connaître les amis de Roach Crosby (paramètre filter1), qui ne commencent pas par « Pitts »
(paramètre filter2), et pour calculer un critère booléen sur un critère (filter2):
SELECT t1.uid, resultats.*
FROM t1,
    JSON_TABLE(utilisateur,
           '$ ? (@.name == $filter1)'
           PASSING 'Roach Crosby' AS filter1,
                   'Pitts' AS filter2,
                   0 AS filter3
           COLUMNS (
               nom text PATH '$.name',
               NESTED PATH '$.friends[*] ? (!(@.name starts with $filter2))'
                 COLUMNS (
                     numero FOR ORDINALITY,
                     ami text PATH '$.name',
                     meilleurami bool EXISTS PATH '$.id ? (@ == $filter3)'))
                    ) AS resultats
         nom | numero | ami | meilleurami
1 | Deloris Wilkerson | t
1001 | Roach Crosby | 2 | Alicia Mucro
_____+
Le plan d'exécution d'une telle requête (ici avec VERBOSE ) indique un appel de fonction pour chaque
ligne (loops=1001), sans détail sur l'exécution de cette fonction :
                          OUERY PLAN
Nested Loop (actual time=1.795..1.799 rows=2 loops=1)
```

Output: t1.uid, resultats.nom, resultats.numero, resultats.ami,

→ resultats.meilleurami Buffers: shared hit=48

#### **Documentation:**

- Documentation officielle: JSON TABLE<sup>26</sup>

#### **5.5.12 JSON: performances**



Inconvénients par rapport à un modèle normalisé :

- Perte d'intégrité (types, contraintes, FK...)
- Complexité du code
- Pas de statistiques sur les clés JSON!
- Pas forcément plus léger en disque
  - clés répétées
- Lire 1 attribut = lire tout le JSON
  - voire accès table TOAST
- Mise à jour : tout ou rien
- Indexation délicate

Les attributs JSON sont très pratiques quand le schéma est peu structuré. Mais la complexité supplémentaire de code nuit à la lisibilité des requêtes. En termes de performances, ils sont coûteux, pour les raisons que nous allons voir.

Les contraintes d'intégrité sur les types, les tailles, les clés étrangères... ne sont pas disponibles. Rien ne vous interdit d'utiliser un attribut country au lieu de pays, avec une valeur FR au lieu de France. Les contraintes protègent de nombreux bugs, mais elles sont aussi une aide précieuse pour l'optimiseur.

 $<sup>^{26}</sup> https://www.postgresql.org/docs/current/functions-json.html \#FUNCTIONS-SQLJSON-TABLE$ 

Chaque JSON récupéré l'est en bloc. Si un seul attribut est récupéré, PostgreSQL devra charger tout le JSON et le décomposer. Cela peut même coûter un accès supplémentaire à une table TOAST pour les gros JSON. Rappelons que le mécanisme du TOAST permet à PostgreSQL de compresser à la volée un grand champ texte, binaire, JSON... et/ou de le déporter dans une table annexe interne, le tout étant totalement transparent pour l'utilisateur. Pour les détails, voir cet extrait de la formation DBA2<sup>27</sup>.

Il n'y a pas de mise à jour partielle: modifier un attribut implique de décomposer tout le JSON pour le réécrire entièrement (et parfois en le *détoastant/retoastant*). Si le JSON est trop gros, modifier ses sous-attributs par plusieurs processus différents peut poser des problèmes de verrouillage. Pour citer la documentation<sup>28</sup>:



« Les données JSON sont sujettes aux mêmes considérations de contrôle de concurrence que pour n'importe quel autre type de données quand elles sont stockées en table. Même si stocker de gros documents est prévisible, il faut garder à l'esprit que chaque mise à jour acquiert un verrou de niveau ligne sur toute la ligne. Il faut envisager de limiter les documents JSON à une taille gérable pour réduire les contentions sur verrou lors des transactions en mise à jour. Idéalement, les documents JSON devraient chacun représenter une donnée atomique, que les règles métiers imposent de ne pas pouvoir subdiviser en données plus petites qui pourraient être modifiées séparément. »

Un gros point noir est l'absence de statistiques propres aux clés du JSON. Le planificateur va avoir beaucoup de mal à estimer les cardinalités des critères. Nous allons voir des contournements possibles.

Suivant le modèle, il peut y avoir une perte de place, puisque les clés sont répétées entre chaque attribut JSON, et non normalisées dans des tables séparées.

Enfin, nous allons voir que l'indexation est possible, mais moins triviale qu'à l'habitude.

Ces inconvénients sont à mettre en balance avec les intérêts du JSON (surtout : éviter des lignes avec trop d'attributs toujours à NULL, si même on les connaît), les fréquences de lecture et mises à jour des JSON, et les modalités d'utilisation des attributs.

Certaines de ces limites peuvent être réduites par les techniques ci-dessous.

<sup>&</sup>lt;sup>27</sup>https://dali.bo/m4\_html#mécanisme-toast

<sup>&</sup>lt;sup>28</sup>https://docs.postgresql.fr/current/datatype-json.html#JSON-INDEXING

## 5.5.13 Recherche dans un champ JSON (1)



```
Sans JSONPath:
```

```
SELECT * FROM personnes ...

WHERE datas->>'lastName' = 'Durand'; -- textes

WHERE datas->'lastName' = '"Durand"'::jsonb; -- JSON

WHERE datas @> '{"nom": "Durand"}'::jsonb; -- contient

WHERE datas ? 'spouse'; -- attibut existe ?

WHERE datas ? '{spouse,children}'::text[]; -- un des champs ?

WHERE datas ?& '{spouse,children}'::text[]; -- tous les champs ?

Mais comment indexer?
```

Pour chercher les lignes avec un champ JSON possédant un attribut d'une valeur donnée, il existe plusieurs opérateurs (au sens syntaxique). Les comparaisons directes de textes ou de JSON sont possibles, mais nous verrons qu'elles ne sont pas simplement indexables.

L'opérateur (« contient ») est généralement plus adapté, mais il faut fournir un JSON avec le critère de recherche.

L'opérateur ? permet de tester l'existence d'un attribut dans le JSON (même vide). Plusieurs attributs peuvent être testés avec ? | (« ou » logique) ou ?& (« et » logique).

### 5.5.14 Recherche dans un champ JSON (2): SQL/JSON et JSONPath



- SQL:2016 introduit SQL/JSON et le langage JSONPath
- ISONPath
  - langage de recherche pour JSON
  - concis, flexible, plus rapide
  - depuis PostgreSQL 12, étendu à chaque version

JSONPath<sup>29</sup> est un langage de requêtage permettant de spécifier des parties d'un champ JSON, même

<sup>&</sup>lt;sup>29</sup>https://en.wikipedia.org/wiki/JSONPath

complexe. Il a été implémenté dans de nombreux langages, et a donc une syntaxe différente de celle du SQL, mais souvent déjà familière aux développeurs. Il évite de parcourir manuellement les nœuds et tableaux du JSON, ce qui est vite fastidieux en SQL.

Le standard SQL:2016 intègre le SQL/JSON<sup>30</sup>. PostgreSQL 12 contient déjà l'essentiel des fonctionnalités SQL/JSON, y compris JSONPath, mais elles sont complétées dans les versions suivantes.

## 5.5.15 Recherche dans un champ JSON (3): Exemples SQL/JSON & JSONPath



Par exemple, une recherche peut se faire ainsi, et elle profitera d'un index GIN :

```
SELECT datas->>'firstName' AS prenom
FROM personnes
WHERE datas @@ '$.lastName == "Durand"';
prenom
------
Georges
```

Les opérateurs @@ et @? sont liés à la recherche et au filtrage. La différence entre les deux est liée à la syntaxe à utiliser. Ces deux exemples renvoient la même ligne :

```
SELECT * FROM personnes
WHERE datas @? '$.lastName ? (@ == "Valjean")';

SELECT * FROM personnes
WHERE datas @@ '$.lastName == "Valjean"';

Il existe des fonctions équivalentes, jsonb_path_exists et jsonb_path_match :
```

<sup>&</sup>lt;sup>30</sup>https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016#json-path

(Pour les détails sur ces opérateurs et fonctions, et des exemples sur des filtres plus complexes (inégalités par exemple), voir par exemple : https://justatheory.com/2023/10/sql-jsonpath-operators/.)

Un autre intérêt est la fonction jsonb\_path\_query , qui permet d'extraire facilement des parties d'un tableau :

Ici, jsonb\_path\_query génère une ligne par élément du tableau phoneNumbers inclus dans le JSON.

L'appel suivant effectue un filtrage sur la ville :

```
SELECT jsonb_path_query (datas, '$.address ? (@.city == "Paris")')
FROM personnes;
```

Cependant, pour que l'indexation GIN fonctionne, il faudra l'opérateur @? :

```
SELECT datas->>'lastName',
FROM personnes
WHERE personne @? '$.address ? (@.city == "Paris")';
```

Au final, le code JSONPath est souvent plus lisible que celui utilisant de nombreuses fonctions j sonb spécifiques à PostgreSQL. Un développeur le manipule déjà souvent dans un autre langage.

On trouvera d'autres exemples dans la présentation de Postgres Pro dédié à la fonctionnalité lors la parution de PostgreSQL 12<sup>31</sup>, ou dans un billet de Michael Paquier<sup>32</sup>.

<sup>31</sup> https://www.postgresql.eu/events/pgconfeu2019/sessions/session/2555/slides/221/jsonpath-pgconfeu-2019.pdf

<sup>32</sup> https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

#### **5.5.16** jsonb: indexation (1/2)



- Index fonctionnel sur un attribut précis
  - bonus: statistiques

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
ANALYZE personnes ;
```

- Colonne générée (dénormalisée) :
  - champ normal, indexable, facile à utiliser, rapide à lire
  - statistiques

```
ALTER TABLE personnes
ADD COLUMN lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED;
```

- Préférer @> et @? que des fonctions

#### Index fonctionnel:

L'extraction d'une partie d'un JSON est en fait une fonction immutable, donc indexable. Un index fonctionnel permet d'accéder directement à certaines propriétés, par exemple :

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
```

Mais il ne fonctionnera que s'il y a une clause WHERE avec cette expression exacte. Pour un attribut fréquemment utilisé pour des recherches, c'est le plus efficace.



On n'oubliera pas de lancer un ANALYZE pour calculer les statistiques après création de l'index fonctionnel. Même si l'index est peu discriminant, on obtient ainsi de bonnes statistiques sur son critère.

#### Colonne générée :

Une autre possibilité est de dénormaliser l'attribut JSON intéressant dans un champ séparé de la table, et indexable :

```
ALTER TABLE personnes
ADD COLUMN lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED;
ANALYZE personnes;
CREATE INDEX ON personnes (lastname);
```

Cette colonne générée est mise à jour quand le JSON est modifié, et n'est pas modifiable autrement. C'est à part cela un champ simple, indexable avec un B-tree, et avec ses statistiques propres.

Ce champ coûte certes un peu d'espace disque supplémentaire, mais il améliore la lisibilité du code, et facilite l'usage avec certains outils ou pour certains utilisateurs. Dans le cas des gros JSON, il peut aussi éviter quelques allers-retours vers la table TOAST. Même sans utilisation d'un index, un champ normal est beaucoup plus rapide à lire dans la ligne qu'un attribut extrait d'un JSON.

#### **5.5.17** jsonb: indexation (2/2)



- Indexation « schemaless » grâce au GIN :
  - attention au choix de l'opérateur

```
-- opérateur par défaut
-- pour critères : ?, ?|, ?& , @> , @?, @@)

CREATE INDEX idx_prs ON personnes USING gin(datas) ;
-- index plus performant
-- mais que @> , @?, @@

CREATE INDEX idx_prs ON personnes USING gin(datas jsonb_path_ops) ;
```

#### Index GIN:

Les champs jsonb peuvent tirer parti de fonctionnalités avancées de PostgreSQL, notamment les index GIN, et ce via deux classes d'opérateurs.

L'opérateur par défaut de GIN pour jsonb est jsonb\_ops . Mais il est souvent plus efficace de choisir l'opérateur jsonb\_path\_ops . Ce dernier donne des index plus petits et performants sur des clés fréquentes, bien qu'il ne supporte que certains opérateurs de recherche (@>, @? et @@) (voir les détails<sup>33</sup>), ce qui suffit généralement.

<sup>33</sup>https://docs.postgresql.fr/current/datatype-json.html#JSON-INDEXING

Un index GIN est moins efficace qu'un index fonctionnel B-tree classique, mais il est idéal quand la clé de recherche n'est pas connue, et que n'importe quel attribut du JSON peut être un critère.

Un index GIN ne permet cependant pas d'Index Only Scan.

Surtout, un index GIN ne permet pas de recherches sur des opérateurs B-tree classiques ( < , <= , > , > , > ), ou sur le contenu de tableaux. On est obligé pour cela de revenir au monde relationnel, ou se rabattre sur les index fonctionnels ou colonnes générées vus plus haut. Attention, des fonctions comme jsonb\_path\_query ne savent pas utiliser les index. Il est donc préférable d'utiliser les opérateurs spécifiques, comme « contient » ( @> ) ou « existe » en JSONPath ( @? ).

## 5.5.18 Pour aller plus loin...



Lire la documentation

- 8.14 Types JSON:
  - https://docs.postgresql.fr/current/datatype-json.html
- 9.16 Fonctions & opérateurs JSON:
  - https://docs.postgresql.fr/current/functions-json.html

Les fonctions et opérateurs indiqués ici ne représentent qu'une partie de ce qui existe. Certaines fonctions sont très spécialisées, ou existent en plusieurs variantes voisines. Il est conseillé de lire ces deux chapitres de documentation lors de tout travail avec les JSON. Attention à la version de la page : des fonctionnalités sont ajoutées à chaque version de PostgreSQL.

### 5.6 XML

# 5.6.1 XML: présentation



- Type xml
  - stocke un document XML
  - valide sa structure
- Quelques fonctions et opérateurs disponibles :
  - XMLPARSE, XMLSERIALIZE, query\_to\_xml, xmlagg
  - xpath (XPath 1.0 uniquement)

Le type xml, inclus de base, vérifie que le XML inséré est un document « bien formé », ou constitue des fragments de contenu (« content »). L'encodage UTF-8 est impératif. Il y a quelques limitations par rapport aux dernières versions du standard, XPath et XQuery<sup>34</sup>. Le stockage se fait en texte, donc bénéficie du mécanisme de compression TOAST.

Il existe quelques opérateurs et fonctions de validation et de manipulations, décrites dans la documentation du type xml<sup>35</sup> ou celle des fonctions<sup>36</sup>. Par contre, une simple comparaison est impossible et l'indexation est donc impossible directement. Il faudra passer par une expression XPath.

À titre d'exemple : XMLPARSE convertit une chaîne en document XML, XMLSERIALIZE procède à l'opération inverse.

<sup>&</sup>lt;sup>34</sup>https://docs.postgresql.fr/current/xml-limits-conformance.html

<sup>&</sup>lt;sup>35</sup>https://docs.postgresql.fr/current/datatype-xml.html

<sup>&</sup>lt;sup>36</sup>https://docs.postgresql.fr/current/functions-xml.html

```
<YEAR>1964</YEAR>
  </CD>
    <TITLE>Olympia 1961</TITLE>
    <ARTIST>Jacques Brel</ARTIST>
    <COUNTRY>France</COUNTRY>
    <YEAR>1962</YEAR>
  </CD>
</CATALOG> $$ ) ;
--- Noter le $$ pour délimiter une chaîne contenant une apostrophe
SELECT XMLSERIALIZE (DOCUMENT catalogue AS text) FROM liste_cd;
                  xmlserialize
 <?xml version="1.0" encoding="UTF-8"?>
 <CATALOG>
   <CD>
     <TITLE>The Times They Are a-Changin'</TITLE>+
     <ARTIST>Bob Dylan</ARTIST>
     <COUNTRY>USA</COUNTRY>
     <YEAR>1964</YEAR>
   </CD>
   <CD>
     <TITLE>Olympia 1961</TITLE>
     <ARTIST>Jacques Brel</ARTIST>
     <COUNTRY>France</COUNTRY>
     <YEAR>1962</YEAR>
   </CD>
 </CATALOG>
(1 ligne)
```

Il existe aussi query\_to\_xml pour convertir un résultat de requête en XML, xmlagg pour agréger des champs XML, ou xpath pour extraire des nœuds suivant une expression XPath 1.0.

NB: l'extension xml2<sup>37</sup> est dépréciée et ne doit pas être utilisée dans les nouveaux projets.

302

<sup>&</sup>lt;sup>37</sup>https://docs.postgresql.fr/current/xml2.html

### **5.7 OBJETS BINAIRES**

TOAST<sup>38</sup>).

## 5.7.1 Objets binaires: les types



- Souvent une mauvaise idée...
  2 méthodes
  bytea: type binaire
  Large Objects: manipulation comme un fichier

PostgreSQL permet de stocker des données au format binaire, potentiellement de n'importe quel type, par exemple des images ou des PDF.

Il faut vraiment se demander si des binaires ont leur place dans une base de données relationnelle. Ils sont généralement beaucoup plus gros que les données classiques. La volumétrie peut donc devenir énorme, et encore plus si les binaires sont modifiés, car le mode de fonctionnement de PostgreSQL aura tendance à les dupliquer. Cela aura un impact sur la fragmentation, la quantité de journaux, la taille des sauvegardes, et toutes les opérations de maintenance. Ce qui est intéressant à conserver dans une base sont des données qu'il faudra rechercher, et l'on recherche rarement au sein d'un gros binaire. En général, l'essentiel des données binaires que l'on voudrait confier à une base peut se contenter d'un stockage classique, PostgreSQL ne contenant qu'un chemin ou une URL vers le fichier réel.

PostgreSQL donne le choix entre deux méthodes pour gérer les données binaires :

- bytea: un type comme un autre;
- Large Object : des objets séparés, à gérer indépendamment des tables.

<sup>&</sup>lt;sup>38</sup>https://dali.bo/m4\_html#m%C3%A9canisme-toast

#### **5.7.2** bytea



- Un type comme les autres
  - bytea : tableau d'octets
  - en texte: bytea\_output = hex ou escape
- Récupération intégralement en mémoire!
- Toute modification entraîne la réécriture complète du bytea
- Maxi 1 Go (à éviter)
  - en pratique intéressant pour quelques Mo
  - compressé/déporté (TOAST)
- Import:

```
SELECT pg_read_binary_file ('/chemin/fichier');
```

#### Voici un exemple:

Nous avons inséré la chaîne de caractère « bonjour » dans le champ bytea, en fait sa représentation binaire dans l'encodage courant (UTF-8). Si nous interrogeons la table, nous voyons la représentation textuelle du champ bytea. Elle commence par \( \times \) pour indiquer un encodage de type \( \text{hex} \). Ensuite, chaque paire de valeurs hexadécimales représente un octet.

Un second format d'affichage est disponible : escape :

```
SET bytea_output = escape ;
SELECT * FROM demo_bytea ;

a
-----
bonjour

INSERT INTO demo_bytea VALUES ('journée'::bytea);
SELECT * FROM demo_bytea ;

a
------
bonjour
journ\303\251e
```

Le format de sortie escape ne protège donc que les valeurs qui ne sont pas représentables en ASCII 7 bits. Ce format peut être plus compact pour des données textuelles essentiellement en alphabet latin sans accent, où le plus gros des caractères n'aura pas besoin d'être protégé.

Cependant, le format hex est bien plus efficace à convertir, et est le défaut depuis PostgreSQL 9.0.



Avec les vieilles applications, ou celles restées avec cette configuration, il faudra peutêtre forcer bytea\_output à escape, sous peine de corruption.)

Pour charger directement un fichier, on peut notamment utiliser la fonction pg\_read\_binary\_file , exécutée par le serveur PostreSQL :

```
INSERT INTO demo_bytea (a)
SELECT pg_read_binary_file ('/chemin/fichier');
```

En théorie, un bytea peut contenir 1 Go. En pratique, on se limitera à nettement moins, ne serait-ce que parce pg\_dump tombe en erreur quand il doit exporter des bytea de plus de 500 Mo environ (le décodage double le nombre d'octets et dépasse cette limite de 1 Go).

Les bytea profitent du mécanisme TOAST<sup>39</sup> et sont donc généralement compressés et/ou déplacés dans une table associé à la table principale. Encore plus que d'habitude, il faut proscrire les SELECT \* dans votre code pour éviter de lire cette table quand ce n'est pas nécessaire.

La documentation officielle<sup>40</sup> liste les fonctions pour encoder, décoder, extraire, hacher... les bytea.

<sup>&</sup>lt;sup>39</sup>https://dali.bo/m4\_html#m%C3%A9canisme-toast

<sup>&</sup>lt;sup>40</sup>https://docs.postgresql.fr/current/functions-binarystring.html

## 5.7.3 Large Object



- À éviter...
  - préférer bytea
- Maxi 4 To (éviter...)
- Objet indépendant des tables
  - OID à stocker dans les tables
  - se compresse mal
- Suppression manuelle!
  - triggerlo\_unlink & vacuumlo
- Fonction de manipulation, modification
  - lo\_createlo\_importlo\_seeklo\_openlo\_readlo\_write

Un *large object* est un objet totalement décorrélé des tables. Le code doit donc gérer cet objet séparément :

- créer le *large object* et stocker ce qu'on souhaite dedans ;
- stocker la référence à ce *large object* dans une table (avec le type lob);
- interroger l'objet séparément de la table ;
- le supprimer explicitement quand il n'est plus référencé : il ne disparaîtra pas automatiquement !

Le large object nécessite donc un plus gros investissement au niveau du code.

En contrepartie, il a les avantages suivant :

- une taille jusqu'à 4 To, ce qui n'est tout de même pas conseillé;
- la possibilité d'accéder à une partie directement (par exemple les octets de 152 000 à 153 020),
   ce qui permet de le transférer par parties sans le charger en mémoire (notamment, le driver JDBC de PostgreSQL fournit une classe LargeObject<sup>41</sup>);
- de ne modifier que cette partie sans tout réécrire.

Cependant, nous déconseillons son utilisation autant que possible :

le stockage des large objects ne prévoit pas vraiment de compression : des bytea prendront moins de place ;

<sup>&</sup>lt;sup>41</sup>https://jdbc.postgresql.org/documentation/binary-data/

- il est facile et fréquent d'oublier de purger des large objects supprimés ;
- une sauvegarde logique partielle les oublie facilement (voir l'option --large-objects de pg\_dump 42);
- pg\_dump n'est pas optimisé pour sauver de nombreux *large objects* : la sauvegarde de la table pg\_largeobject ne peut être parallélisée et peut consommer transitoirement énormément de mémoire s'il y a trop d'objets.

Il y a plusieurs méthodes pour nettoyer les large objects devenu inutiles :

- appeler la fonction lo\_unlink dans le code client au risque d'oublier;
- utiliser la fonction trigger lo\_manage fournie par le module contrib lo : (voir documentation<sup>43</sup>, si les *large objects* ne sont jamais référencés plus d'une fois ;
- appeler régulièrement le programme vacuumlo (là encore un contrib<sup>44</sup>) : il liste tous les *large* objects référencés dans la base, puis supprime les autres. Ce traitement est bien sûr un peu lourd.

Techniquement, un *large object* est stocké dans la table système pg\_largeobject sous forme de pages de 2 ko. Voir la documentation<sup>45</sup> pour les détails.

<sup>&</sup>lt;sup>42</sup>https://docs.postgresql.fr/current/app-pgdump.html

<sup>&</sup>lt;sup>43</sup>https://docs.postgresql.fr/current/lo.html

<sup>44</sup>https://docs.postgresql.fr/current/vacuumlo.html

<sup>&</sup>lt;sup>45</sup>https://docs.postgresql.fr/current/largeobjects.html

# **5.8 QUIZ**



https://dali.bo/s9\_quiz

# **5.9 TRAVAUX PRATIQUES**

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s9\_solutions.

# 5.9.1 **UUID**



**But**: Constater l'impact en cache des types d'UUID avec pg\_buffercache.s

Ce TP étant est purement descriptif, allez voir direment la solution.

## 5.9.2 jsonb



La base **personnes\_et\_dossiers** pèse en version complète 613 Mo, pour 2 Go sur disque au final. Elle peut être installée comme suit :

```
# Dump complet
curl -kL https://dali.bo/tp_personnes -o /tmp/personnes.dump
# Taille 40%
# curl -kL https://dali.bo/tp_personnes_200k -o /tmp/personnes.dump
# Taille 16%
# curl -kL https://dali.bo/tp_personnes_fr -o /tmp/personnes.dump

createdb --echo personnes
# L'erreur sur un schéma 'public' existant est normale
pg_restore -v -d personnes /tmp/personnes.dump
rm -- /tmp/personnes.dump
```

La base personnes contient alors deux schémas json et eav avec les mêmes données sous deux formes différentes.

Chercher la ville et le numéro de téléphones (sous-attribut ville de l'attribut adresse du champ JSON personne) de Gaston Lagaffe, grâce aux attributs prenom et nom. Effectuer de préférence la recherche en cherchant un JSON avec (« contient ») (Ne pas chercher encore à utiliser JSONPath).

Créer une requête qui renvoie les attributs nom, prenom, date\_naissance (comme type date) de toutes les personnes avec le nom « Lagaffe ». Utiliser la fonction jsonb\_to\_record() et LATERAL. Rajouter ville et pays ensuite de la même manière.

En supprimant le filtre, comparer le temps d'exécution de la requête précédente avec cette requête plus simple qui récupère les champs plus manuellement :

```
SELECT personne->>'nom',
    personne->>'prenom',
    (personne->>'date_naissance')::date,
    personne#>>'{adresse,ville}',
    personne#>>'{adresse,pays}'
FROM json.personnes;
```

#### Créer un index GIN ainsi:

```
CREATE INDEX personnes_gin ON json.personnes
USING gin(personne jsonb_path_ops);
```

Quelle taille fait-il?

Retenter les requêtes précédentes. Lesquelles utilisent l'index?

Récupérer les numéros de téléphone de Léon Prunelle avec ces trois syntaxes. Quelles sont les différences ?

Afficher les noms et prénoms de Prunelles, et un tableau de champs texte contenant ses téléphones (utiliser jsonb\_array\_elements\_text).

Comparer le résultat et les performances de ces deux requêtes, qui récupèrent aussi les numéros de téléphone de Prunelle :

Chercher qui possède le numéro de téléphone 0650041821 avec la syntaxe JSONPath.

Compter le nombre de personnes habitant à Paris ou Bruxelles avec :

- la syntaxe @> et un OR ;
- une syntaxe JSONPath @? et un « ou logique ( | | );
- une syntaxe JSONPath @? et une regex @.ville like\_regex "^(Paris|Bruxelles)\$".

Le compte du nombre de personne par pays doit être optimisé au maximum. Ajouter un index fonctionnel sur l'attribut pays. Tester l'efficacité sur une recherche, et un décompte de toute les personnes par pays.

Ajouter un champ généré dans json.personne, correspondant à l'attribut pays.

Comparer les temps d'exécution du décompte des pays par l'attribut, et par cette colonne générée.

Créer un index B-tree sur la colonne générée pays. Consulter les statistiques dans pg\_stats. Cet index est-il utilisable pour des filtres et le décompte par pays ?

#### **DALIBO Formations**

(Optionnel) Créer des colonnes générées sur nom , prenom , date\_naissance , et ville (en un seul ordre). Reprendre la requête plus haut qui les affiche tous et comparer les performances.

Ajouter l'attribut animaux à Gaston Lagaffe, avec la valeur 18. Vérifier en relisant la ligne.

Ajouter l'attribut animaux à 2% des individus au hasard, avec une valeur 1 ou 2.

Compter le nombre de personnes avec des animaux (avec ou sans JSONPath). Proposer un index qui pourait convenir à d'autres futurs nouveaux attributs peu fréquents.

# **5.9.3 Large Objects**



**But**: Utilisation de Large Objects

- Créer une table fichiers avec un texte et une colonne permettant de référencer des *Large Objects*.
- Importer un fichier local à l'aide de psql dans un large object.
- Noter l' oid retourné.
- Importer un fichier du serveur à l'aide de psql dans un large object.
- Afficher le contenu de ces différents fichiers à l'aide de psql.
- Les sauvegarder dans des fichiers locaux.

# **5.10 TRAVAUX PRATIQUES (SOLUTIONS)**

### 5.10.1 UUID

Tout ce qui suit doit se dérouler dans la même base, par exemple :

```
CREATE DATABASE capteurs;
```

Ce TP est prévu pour un shared\_buffers de 128 Mo (celui par défaut). Si le vôtre est plus gros, le TP devra peut-être durer plus longtemps :

```
SHOW shared_buffers;
```

Utilisez au moins une fenêtre pour les ordres shell et une pour les ordres SQL.

Créer avec le script suivants les deux versions d'un petit modèle avec des capteurs, et les données horodatées qu'ils renvoient ; ainsi que les deux procédures pour remplir ces tables ligne à ligne :

#### \c capteurs

```
-- Modèle : une table 'capteurs' et ses 'donnees' horodatées
-- liées par une contrainte
-- Deux versions : avec ID et une séquence, et avec UUID
DROP TABLE IF EXISTS donnees1, donnees2, capteurs1, capteurs2;
-- Avec identifiants bigint
CREATE TABLE capteurs1 (id_capteur bigint PRIMARY KEY,
                                   char (50) UNIQUE,
                       nom
                                   char (50) default
                        filler
CREATE TABLE donnees1 (id_donnee
                                  bigserial PRIMARY KEY,
                       id_capteur int NOT NULL REFERENCES capteurs1,
                       horodatage timestamp with time zone,
                       valeur1 int,
                       valeur2
                                  int,
                                 float
                       valeur3
                      );
CREATE INDEX ON donnees1 (horodatage) ;
-- Version avec les UUID
CREATE TABLE capteurs2 (id_capteur uuid PRIMARY KEY,
                       nom char (50) UNIQUE, filler char (50) default ''
                       );
CREATE TABLE donnees2 (id_donnee uuid PRIMARY KEY,
                       id_capteur uuid NOT NULL REFERENCES capteurs2,
                       horodatage timestamp with time zone,
                       valeur1
                                  int,
                       valeur2
                                 int,
                       valeur3
                                 float
CREATE INDEX ON donnees2 (horodatage) ;
```

```
-- 1000 capteurs identiques
INSERT INTO capteurs1 (id_capteur, nom)
SELECT i,
        'M-'||md5(i::text)
FROM generate_series (1,1000) i
ORDER BY random();
INSERT INTO capteurs2 (id_capteur, nom)
SELECT gen_random_uuid(), nom FROM capteurs1;
-- 2 procédures d'insertion de données identiques sur quelques capteurs au hasard
-- insertion dans donnees1 avec une séquence
CREATE OR REPLACE PROCEDURE insere_donnees_1 ()
AS $$
    SET synchronous_commit TO off ; -- accélère
    INSERT INTO donnees1 (id_donnee, id_capteur, horodatage, valeur1, valeur2,

    valeur3)

    SELECT nextval('donnees1_id_donnee_seq'::regclass), -- clé primaire des données
                                                           -- clé étrangère
            m.id_capteur,
            now(), (random()*1000)::int,(random()*1000)::int,random()
    FROM capteurs1 m TABLESAMPLE BERNOULLI (1); -- 1% des lignes
$$ LANGUAGE sql;
 -- insertion dans donnees2 avec un UUID v7
CREATE OR REPLACE PROCEDURE insere_donnees_2 ()
    SET synchronous commit TO off; -- accélère
    INSERT INTO donnees2 (id_donnee, id_capteur, horodatage, valeur1, valeur2,

    valeur3)

    SELECT gen_random_uuid(), -- clé primaire des données, UUID v4 m.id_capteur, -- clé étrangère
            now(), (random()*1000)::int,(random()*1000)::int,random()
    FROM capteurs2 m TABLESAMPLE BERNOULLI (1); -- 1% des lignes
$$ LANGUAGE sql;
Vous devez obtenir ces tables et une séquence :
capteurs=# \d+
                                                  Liste des relations
 Schéma | Nom | Type | ... | ... | Taille | Description

      public | capteurs1 | table | ... | ... | ... | 168 kB |

      public | capteurs2 | table | ... | ... | ... | 176 kB |

      public | donnees1 | table | ... | ... | ... | 0 bytes |

      public | donnees1_id_donnee_seq | séquence | ... | ... | 8192 bytes |

 public | donnees2 | table | ... | ... | 0 bytes |
(5 lignes)
et ces index:
capteurs=# \di
                        Liste des relations
 Schéma |
                   Nom | Type | Propriétaire | Table
```

Créer deux fichiers SQL contenants juste les appels de fonctions, qui serviront pour pgbench :

```
echo "CALL insere_donnees_1 ()" > /tmp/insere1.sql
echo "CALL insere_donnees_2 ()" > /tmp/insere2.sql
```

Dans la même base que les table ci-dessus, installer l'extension pg\_buffercache<sup>a</sup> qui va nous permettre de voir ce qu'il y a dans le cache de PostgreSQL :

```
<sup>a</sup>https://dali.bo/x2 html
```

```
CREATE EXTENSION IF NOT EXISTS pg_buffercache ;
```

La vue du même nom contient une ligne par bloc. La requête suivante permet de voir lesquelles de nos tables utilisent le cache :

```
SELECT CASE WHEN datname = current_database()
        AND relname NOT LIKE 'pg%'
        THEN relname ELSE '*AUTRES*' END AS objet,
        count(*),
        pg_size_pretty(count(bufferid)*8192) as Taille_Mo
FROM pg_buffercache b
LEFT OUTER JOIN pg_class c ON c.relfilenode = b.relfilenode
LEFT OUTER JOIN pg_database d ON (d.oid = b.reldatabase)
GROUP BY objet
ORDER BY count(bufferid) DESC;
```

Cette version semi-graphique est peut-être plus parlante :

Dans une fenêtre, lancer l'une de ces requêtes (dans la bonne base !), puis la répéter toutes les secondes ainsi :

```
-- sous psql
\watch 1
```

Dans une autre fenêtre, lancer pgbench avec deux clients, et le script pour remplir la table donnees1 :

```
# Sous Rocky Linux/Almalinux...
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql
```

Le nombre de transactions dépend fortement de la machine, mais peut atteindre plusieurs milliers à la seconde.



Les tables peuvent rapidement atteindre plusieurs gigaoctets. N'hésitez pas à les vider ensemble de temps à autre.

TRUNCATE donnees1, donnees2;

# Quelle est la répartition des données dans le cache?

Après peu de temps, la répartition doit ressembler à peu près à ceci :

objet	taille_mo	taille
donnees1_pkey	28 MB	   ########
donnees1_id_donnee_seq	8192 bytes	
donnees1_horodatage_idx	12 MB	####
donnees1	86 MB	###########################
capteurs1_pkey	48 kB	
capteurs1	144 kB	
*AUTRES*	2296 kB	#

Et ce, même si la table et ses index ne tiennent plus intégralement dans le cache.

La table donnees1 représente la majorité du cache.

```
Interrompre pgbench et le relancer pour remplir donnees2:
```

```
# Sous Rocky Linux/Almalinux...
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere2.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere2.sql
```

Noter que le débit en transaction est du même ordre de grandeur : les UUID ne sont pas spécialement lourds à générer.

# Que devient la répartition des données dans le cache?

donnees1 et ses index est chassé du cache par les nouvelles données, ce qui est logique.

Surtout, on constate que la clé primaire de donnnes2 finit par remplir presque tout le cache. Dans ce petit cache, il n'y a plus de place même pour les données de données !

objet	taille_mo	taille
donnees2_pkey donnees2_horodatage_idx	120 MB   728 kB	
donnees2	6464 kB	##
capteurs2_pkey capteurs2	48 kB   152 kB	
*AUTRES*	132 KB   408 kB	[ 

Interrompre pgbench, purger les tables et lancer les deux scripts d'alimentation en même temps.

```
TRUNCATE donnees1, donnees2;
```

```
# Sous Rocky Linux/Almalinux...
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
```

On constate le même phénomène de monopolisation du cache par données , bien que les deux tables de données aient le même nombre de lignes :

objet	taille_mo	taille
donnees2_pkey donnees2_horodatage_idx	115 MB 624 kB	
donnees2	5568 kB	##
donnees1_pkey	1504 kB	
<pre>donnees1_id_donnee_seq donnees1_horodatage_idx</pre>	8192 bytes 632 kB	
donnees1	4544 kB	I   #
capteurs2_pkey	48 kB	
capteurs2	152 kB	
capteurs1_pkey	48 kB	
capteurs1	144 kB	
*AUTRES*	408 kB	
(12 lignes)		

Avez-vous remarqué une différence de vitesse entre les deux traitements?

Ce ne peut être rigoureusement établi ici. Les volumétries sont trop faibles par rapport à la taille des mémoires et il faut tester sur la durée. Le nombre de clients doit être étudié pour utiliser au mieux les

capacités de la machine sans monter jusqu'à ce que la contention devienne un problème. Les checkpoints font également varier les débits.

Cependant, si vous laissez le test tourner très longtemps avec des tailles de tables de plusieurs Go, les effets de cache seront très différents :

- dans donnees1, le débit en insertion devrait rester correct, car seuls les derniers blocs en cache sont utiles :
- dans donnees2, le débit en insertion doit progressivement baisser: chaque insertion a besoin d'un bloc de l'index de clé primaire différent, qui a de moins en moins de chance de se trouver dans le cache de PostgreSQL, puis dans le cache de Linux.

L'impact sur les I/O augmente donc, et pas seulement à cause de la volumétrie supérieure des tables avec UUID. À titre d'exemple, sur une petite base de formation avec 3 Go de RAM :

```
# avec ID numériques, pendant des insertions dans donnees1 uniquement,
# qui atteint 1,5 Go
# débit des requêtes : environ 3000 tps
$ iostat -h 1
avg-cpu: %user
                 %nice %system %iowait %steal
                                                 %idle
                0,0% 10,7% 0,0% 0,0%
         88,6%
                                                  0,7%
            kB_read/s
     tps
                         kB_wrtn/s
                                      kB_read
                                                 kB_wrtn Device
                0,0k
                         17,0M
                                                 17,0M vda
   86,00
                                         0,0k
    0,00
                 0,0k
                             0,0k
                                         0,0k
                                                   0,0k scd0
# avec UUID v4, pendant des insertions dans donnees2 uniquement,
# qui atteint 1,5 Go
# débit des requêtes : environ 700 tps
$ iostat -h 1
         %user %nice %system %iowait %steal 41,2% 0,0% 17,3% 25,9% 0,7%
                                                 %idle
avg-cpu: %user
                                                15,0%
     tps
            kB_read/s
                         kB_wrtn/s
                                      kB_read
                                                 kB_wrtn Device
                                                 63,0M vda
 2379,00
                 0,0k
                           63,0M
                                         0,0k
    0,00
                 0,0k
                              0,0k
                                         0,0k
                                                   0,0k scd0
```

Comparer les tailles des tables et index avant et après un VACUUM FULL. Où était la fragmentation?

VACUUM FULL reconstruit complètement les tables et aussi les index.

Tables avant le VACUUM FULL:

capteurs=# \d+

	Liste	des relation	ons				
Schéma	Nom	Type				Taille	
+		+	+ – – – -	+	+	· 	+
public	capteurs1	table				168 kB	
public	capteurs2	table				176 kB	
public	donnees1	table	ļ	<b></b>	<b></b>	2180 MB	ĺ
public	donnees1_id_donnee_seq	séquence				8192 bytes	
public	donnees2	table				2227 MB	

#### Après:

capteurs=# \d+

	Liste	des relati	ons			
Schéma		1 71			Taille	
		+	+	+	 ·	+
public	capteurs1	table			 144 kB	
public	capteurs2	table			 152 kB	
public	donnees1	table		<b></b>	 2180 MB	İ
public	donnees1_id_donnee_seq	séquence		<b></b>	8192 bytes	İ
public	donnees2	table	j	j	 2227 MB	İ
public	pg_buffercache	vue		<b></b>	0 bytes	İ
(6 lignes	5)	-	-	-		

Les tailles des tables donnees1 et donnees2 ne bougent pas. C'est normal, il n'y a eu que des insertions à chaque fois en fin de table, et ni modification ni suppression de données.

# Index avant le VACUUM FULL:

capteurs=# \di+

	Liste	des rela	tions	;				
Schéma	Nom	Type		Table		Méth.	Taille	
+		+	++		++	++		+
public	capteurs1_nom_key	index		capteurs1		btree	120 kB	
public	capteurs1_pkey	index		capteurs1		btree	56 kB	
public	capteurs2_nom_key	index		capteurs2		btree	120 kB	
public	capteurs2_pkey	index		capteurs2		btree	56 kB	
public	donnees1_horodatage_idx	index		donnees1		btree	298 MB	
public	donnees1_pkey	index		donnees1		btree	717 MB	
public	donnees2_horodatage_idx	index		donnees2		btree	245 MB	
public	donnees2_pkey	index		donnees2		btree	1166 MB	
(8 lignes	5)							

### Index après:

capteurs=# \di+

Liste des relations								
Schéma	Nom	Type		Table		Méth.	Taille	
		+	++		++	+	+	+
public	capteurs1_nom_key	index		capteurs1		btree	96 kB	
public	capteurs1_pkey	index		capteurs1		btree	40 kB	
public	capteurs2_nom_key	index		capteurs2		btree	96 kB	
public	capteurs2_pkey	index		capteurs2		btree	48 kB	
public	donnees1_horodatage_idx	index		donnees1		btree	296 MB	
public	donnees1_pkey	index		donnees1	<b></b>	btree	717 MB	ĺ
public	donnees2_horodatage_idx	index		donnees2		btree	245 MB	
public	donnees2_pkey	index		donnees2		btree	832 MB	
(8 lignes	s)							

Les index d'horodatage gardent la même taille qu'avant (la différence entre eux est dû à des nombres de lignes différents dans cet exemple). L'index sur la clé primaire de donnees1 (bigint) n'était pas fragmenté. Par contre, donnees2\_pkey se réduit de 29%! Les index UUID (v4) ont effectivement tendance à se fragmenter.

Les UUID générés avec gen\_random\_uuid sont de version 4. Créer la fonction suivante pour générer des UUID version 7, l'utiliser dans la fonction d'alimentation de donnees2, et relancer les deux alimentations :

```
-- Source: https://postgresql.verite.pro/blog/2024/07/15/uuid-v7-pure-sql.html
-- Daniel Vérité d'après Kyle Hubert
CREATE OR REPLACE FUNCTION uuidv7() RETURNS uuid
  -- Replace the first 48 bits of a uuidv4 with the current
  -- number of milliseconds since 1970-01-01 UTC
  -- and set the "ver" field to 7 by setting additional bits
  select encode(
    set_bit(
      set_bit(
        overlay(uuid_send(gen_random_uuid()) placing
        substring(int8send((extract(epoch from clock_timestamp())*1000)::bigint)
            from 3)
        from 1 for 6),
        52, 1),
      53, 1), 'hex')::uuid;
$$ LANGUAGE sql volatile ;
-- insertion dans donnees2 avec un UUID v7
CREATE OR REPLACE PROCEDURE insere_donnees_2 ()
AS $$
    SET synchronous_commit TO off ; -- accélère
    INSERT INTO donnees2 (id_donnee, id_capteur, horodatage, valeur1, valeur2,

    valeur3)

    SELECT uuidv7(),
                      -- clé primaire des données, UUID v7
            m.id_capteur, -- clé étrangère
            now(), (random()*1000)::int,(random()*1000)::int,random()
    FROM capteurs2 m TABLESAMPLE BERNOULLI (1); -- 1% des capteurs
$$ LANGUAGE sql;
# Sous Rocky Linux/Almalinux...
/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
# Sous Debian/Ubuntu
pgbench capteurs -c2 -j1 -n -T1800 -P1 \
-f /tmp/insere1.sql -f /tmp/insere2.sql
```

- Après quelques dizaines de secondes :
  - les deux tables doivent être présentes dans le cache de manière similaire ;
  - les index primaires doivent être présents de manière anecdotique ;
  - donnees2 occupe une taille un peu supérieure à cause de la taille double des UUID par rapport
     aux bigint de donnees1 :

objet	taille_mo +	taille
donnees2_pkey donnees2_horodatage_idx donnees2	18 MB   5392 kB   48 MB	#####   ##   #############

```
donnees1_pkey
                        | 13 MB
                                      | ####
donnees1_id_donnee_seq
                        | 8192 bytes |
donnees1_horodatage_idx | 5376 kB
                                       ###########
donnees1
                          38 MB
                          48 kB
capteurs2_pkey
capteurs2
                          152 kB
capteurs1_pkey
                          48 kB
capteurs1
                          144 kB
*AUTRES*
                          648 kB
(12 lignes)
```

```
Relancez pgbench pour charger donnees2, alternez entre les deux versions de la fonction insere_donnees_2.
```

/usr/pgsql-16/bin/pgbench capteurs -c2 -j1 -n -T1800 -P1 -f /tmp/insere2.sql

```
[fonction avec gen_random_uuid (UUID v4) ]
progress: 202.0 s, 781.2 tps, lat 2.546 ms stddev 6.631, 0 failed
progress: 203.0 s, 597.6 tps, lat 3.229 ms stddev 10.497, 0 failed progress: 204.0 s, 521.7 tps, lat 3.995 ms stddev 20.001, 0 failed progress: 205.0 s, 837.0 tps, lat 2.307 ms stddev 7.743, 0 failed progress: 206.0 s, 1112.1 tps, lat 1.856 ms stddev 7.602, 0 failed
progress: 207.0 s, 1722.8 tps, lat 1.097 ms stddev 0.469, 0 failed
progress: 208.0 s, 894.4 tps, lat 2.352 ms stddev 12.725, 0 failed
progress: 209.0 s, 1045.6 tps, lat 1.911 ms stddev 5.631, 0 failed
progress: 210.0 s, 1040.0 tps, lat 1.921 ms stddev 8.009, 0 failed
progress: 211.0 s, 734.6 tps, lat 2.259 ms stddev 9.833, 0 failed
progress: 212.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 213.0 s, 266.3 tps, lat 16.299 ms stddev 165.541, 0 failed
progress: 214.0 s, 1548.9 tps, lat 1.290 ms stddev 1.970, 0 failed
progress: 215.0 s, 896.0 tps, lat 2.163 ms stddev 5.404, 0 failed
progress: 216.0 s, 1113.0 tps, lat 1.798 ms stddev 4.115, 0 failed
progress: 217.0 s, 886.9 tps, lat 1.990 ms stddev 4.609, 0 failed
progress: 218.0 s, 771.1 tps, lat 2.965 ms stddev 9.767, 0 failed
      [modification avec uuidv7 (UUID v7) ]
progress: 219.0 s, 1952.1 tps, lat 1.022 ms stddev 2.513, 0 failed
progress: 220.0 s, 2241.1 tps, lat 0.890 ms stddev 0.431, 0 failed
progress: 221.0 s, 2184.0 tps, lat 0.914 ms stddev 0.853, 0 failed
progress: 222.0 s, 2191.1 tps, lat 0.911 ms stddev 0.373, 0 failed
progress: 223.0 s, 2355.8 tps, lat 0.847 ms stddev 0.332, 0 failed
progress: 224.0 s, 2267.0 tps, lat 0.880 ms stddev 0.857, 0 failed progress: 225.0 s, 2308.0 tps, lat 0.864 ms stddev 0.396, 0 failed progress: 226.0 s, 2230.9 tps, lat 0.894 ms stddev 0.441, 0 failed progress: 227.0 s, 2225.1 tps, lat 0.897 ms stddev 1.284, 0 failed progress: 228.0 s, 2250.2 tps, lat 0.886 ms stddev 0.408, 0 failed progress: 229.0 s, 2325.1 tps, lat 0.858 ms stddev 0.327, 0 failed
progress: 230.0 s, 2172.1 tps, lat 0.919 ms stddev 0.442, 0 failed
progress: 231.0 s, 2209.8 tps, lat 0.903 ms stddev 0.373, 0 failed
progress: 232.0 s, 2379.0 tps, lat 0.839 ms stddev 0.342, 0 failed
progress: 233.0 s, 2349.1 tps, lat 0.849 ms stddev 0.506, 0 failed
progress: 234.0 s, 2274.9 tps, lat 0.877 ms stddev 0.350, 0 failed
progress: 235.0 s, 2245.0 tps, lat 0.889 ms stddev 0.351, 0 failed
progress: 236.0 s, 2155.9 tps, lat 0.925 ms stddev 0.344, 0 failed
progress: 237.0 s, 2299.2 tps, lat 0.869 ms stddev 0.343, 0 failed
      [nouvelle modification, retour à gen_random_uuid ]
```

#### **DALIBO Formations**

```
progress: 238.0 s, 1296.9 tps, lat 1.540 ms stddev 2.092, 0 failed progress: 239.0 s, 1370.1 tps, lat 1.457 ms stddev 2.794, 0 failed progress: 240.0 s, 1089.9 tps, lat 1.832 ms stddev 4.234, 0 failed progress: 241.0 s, 770.0 tps, lat 2.594 ms stddev 13.761, 0 failed progress: 242.0 s, 412.0 tps, lat 4.736 ms stddev 28.332, 0 failed progress: 243.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed progress: 244.0 s, 632.6 tps, lat 6.403 ms stddev 65.839, 0 failed progress: 245.0 s, 1183.0 tps, lat 1.655 ms stddev 3.732, 0 failed progress: 246.0 s, 869.0 tps, lat 2.287 ms stddev 5.968, 0 failed progress: 247.0 s, 967.0 tps, lat 2.118 ms stddev 4.860, 0 failed progress: 249.0 s, 759.3 tps, lat 2.088 ms stddev 3.967, 0 failed progress: 250.0 s, 787.0 tps, lat 2.395 ms stddev 10.382, 0 failed progress: 251.0 s, 744.0 tps, lat 2.518 ms stddev 10.636, 0 failed progress: 252.0 s, 815.1 tps, lat 2.744 ms stddev 11.983, 0 failed progress: 253.0 s, 931.2 tps, lat 1.998 ms stddev 7.886, 0 failed progress: 254.0 s, 665.0 tps, lat 2.946 ms stddev 13.315, 0 failed progress: 255.0 s, 537.1 tps, lat 3.970 ms stddev 19.232, 0 failed progress: 255.0 s, 683.9 tps, lat 2.757 ms stddev 10.356, 0 failed progress: 256.0 s, 683.9 tps, lat 2.757 ms stddev 10.356, 0 failed
```

Le débit en transactions varie ici d'un facteur 2. Noter que la durée des transactions est aussi beaucoup plus stable (stddev).

# 5.10.2 jsonb: lecture de champs

La base **personnes\_et\_dossiers** pèse en version complète 613 Mo, pour 2 Go sur disque au final. Elle peut être installée comme suit :

```
# Dump complet
curl -kL https://dali.bo/tp_personnes -o /tmp/personnes.dump
# Taille 40%
# curl -kL https://dali.bo/tp_personnes_200k -o /tmp/personnes.dump
# Taille 16%
# curl -kL https://dali.bo/tp_personnes_fr -o /tmp/personnes.dump

createdb --echo personnes
# L'erreur sur un schéma 'public' existant est normale
pg_restore -v -d personnes /tmp/personnes.dump
rm -- /tmp/personnes.dump
```

La base personnes contient alors deux schémas json et eav avec les mêmes données sous deux formes différentes.

La table json. personnes contient une ligne par personne, un identifiant et un champ JSON avec de nombreux attributs. Elle n'est pas encore indexée :

Chercher la ville et le numéro de téléphone (sous-attribut ville de l'attribut adresse du champ JSON personne) de Gaston Lagaffe, grâce aux attributs prenom et nom. Effectuer de préférence la recherche en cherchant un JSON avec (« contient ») (Ne pas chercher encore à utiliser JSONPath).

La recherche peut s'effectuer en convertissant tous les attributs en texte :

```
SELECT personne->'adresse'->>'ville'
FROM     json.personnes p
WHERE     personne->>'nom' = 'Lagaffe'
AND     personne->>'prenom' = 'Gaston' ;
```

On obtient « Bruxelles ».

Avec la syntaxe en version 14:

```
SELECT personne['adresse']['ville']->>0 AS ville
FROM     json.personnes p
WHERE     personne['nom'] = '"Lagaffe"'::jsonb
AND     personne['prenom'] = '"Gaston"'::jsonb ;
```

Il est plus propre de rechercher grâce à une de ces syntaxes, notamment parce qu'elles seront indexables plus tard :

```
SELECT personne->'adresse'->>'ville'
FROM json.personnes p
WHERE personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb ;
ou:
SELECT personne->'adresse'->>'ville'
FROM json.personnes p
WHERE personne @> jsonb_build_object ('nom', 'Lagaffe', 'prenom', 'Gaston');
  Créer une requête qui renvoie les attributs nom , prenom , date_naissance (comme type date)
  de toutes les personnes avec le nom « Lagaffe ». Utiliser la fonction jsonb_to_record() et
  LATERAL . Rajouter ville et pays ensuite de la même manière.
jsonb_to_record exige que l'on fournisse le nom de l'attribut et son type :
SELECT r.*
FROM json.personnes,
LATERAL jsonb_to_record (personne) AS r (nom text, prenom text, date_naissance date)
WHERE personne @> '{"nom": "Lagaffe"}'::jsonb;
  nom | prenom | date_naissance
 Lagaffe | Gaston | 1938-09-22
 Lagaffe | Jeanne | 1940-02-14
Avec la ville, qui est dans un sous-attribut, il faut rajouter une clause LATERAL :
SELECT r1.*, r2.*
FROM json.personnes,
LATERAL jsonb_to_record (personne)
        AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
        AS r2 (ville text, pays text)
WHERE personne @> '{"nom": "Lagaffe"}'::jsonb;
   nom | prenom | date_naissance | ville | pays
  ______
 Lagaffe | Gaston | 1938-09-22 | Bruxelles | Belgique
Lagaffe | Jeanne | 1940-02-14 | Bruxelles | Belgique
  En supprimant le filtre, comparer le temps d'exécution de la requête précédente avec cette re-
  quête plus simple qui récupère les champs plus manuellement :
  SELECT personne->>'nom',
      personne->>'prenom',
      (personne->>'date_naissance')::date,
      personne#>>'{adresse,ville}',
      personne#>>'{adresse,pays}'
  FROM json.personnes;
```

Cette dernière requête est nettement plus lente que l'utilisation de jsonb\_to\_record, même si les I/O sont plus réduites :

```
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS)
SELECT personne->>'nom',
    personne->>'prenom'.
     (personne->>'date')::date,
    personne#>>'{adresse,ville}',
    personne#>>'{adresse,pays}'
FROM json.personnes;
                        QUERY PLAN
Seq Scan on personnes (cost=0.00..71383.74 rows=532645 width=132) (actual

    time=0.079..6009.601 rows=532645 loops=1)

   Buffers: shared hit=3825357 read=122949
Planning Time: 0.078 ms
Execution Time: 6022.738 ms
EXPLAIN (ANALYZE, BUFFERS)
SELECT r1.*, r2.* FROM json.personnes,
LATERAL jsonb_to_record (personne)
      AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
      AS r2 (ville text, pays text);
                       QUERY PLAN
Nested Loop (cost=0.01..83368.26 rows=532645 width=132) (actual

    time=0.064..3820.847 rows=532645 loops=1)

   Buffers: shared hit=1490408 read=122956
   -> Nested Loop (cost=0.00..72715.35 rows=532645 width=832) (actual

    time=0.059..2247.303 rows=532645 loops=1)

         Buffers: shared hit=712094 read=122956
         -> Seq Scan on personnes (cost=0.00..62062.45 rows=532645 width=764)
Buffers: shared read=56736
         -> Function Scan on jsonb_to_record r1 (cost=0.00..0.01 rows=1 width=68)
   (actual time=0.004..0.004 rows=1 loops=532645)
              Buffers: shared hit=712094 read=66220
  -> Function Scan on jsonb_to_record r2 (cost=0.01..0.01 rows=1 width=64) (actual

→ time=0.003..0.003 rows=1 loops=532645)

        Buffers: shared hit=778314
Planning Time: 0.103 ms
 Execution Time: 3953.137 ms
```

La maintenabilité plaide pour la seconde version. Quant à la lisibilité entre les deux versions de la requête, c'est un choix personnel.

# 5.10.3 jsonb:index GIN jsonb\_path\_ops

```
Créer un index GIN ainsi:
    CREATE INDEX personnes_gin ON json.personnes
    USING gin(personne jsonb_path_ops);
Quelle taille fait-il?
```

L'index peut être un peu long à construire (plusieurs dizaines de secondes) et est assez gros :

```
\di+ json.personnes_gin

Liste des relations

Schéma | Nom | ... | Table | ... | Méthode d'accès | Taille | ...

json | personnes_gin | ... | personnes | ... | gin | 230 MB |
```

Retenter les requêtes précédentes. Lesquelles utilisent l'index?

Les requêtes utilisant les égalités (que ce soit sur du texte ou en JSON) n'utilisent pas l'index :

```
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS)
SELECT personne->'adresse'->>'ville'
FROM json.personnes p
WHERE personne->>'nom' = 'Lagaffe'
 AND personne->>'prenom' = 'Gaston' ;
                        OUERY PLAN
Gather (actual time=0.427..566.202 rows=1 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=712208 read=122962
  -> Parallel Seq Scan on personnes p (actual time=372.989..561.152 rows=0 loops=3)
         Filter: (((personne ->> 'nom'::text) = 'Lagaffe'::text) AND ((personne ->>
  'prenom'::text) = 'Gaston'::text))
         Rows Removed by Filter: 177548
        Buffers: shared hit=712208 read=122962
Planning Time: 0.110 ms
Execution Time: 566.228 ms
```

Par contre, la syntaxe (« contient ») utilise l'index, quelle que soit la manière dont on construit le JSON critère. Le gain en temps et en I/O (et en CPU) grâce à l'index est assez foudroyant. Et ceci, quelle que soit la manière dont on récupère les champs, puisqu'il n'y a plus qu'une poignée de lignes à analyser:

```
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS)
SELECT personne->'adresse'->>'ville'
FROM json.personnes p
WHERE personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb;
                             QUERY PLAN
Bitmap Heap Scan on personnes p (actual time=0.047..0.049 rows=1 loops=1)
   Recheck Cond: (personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb)
  Heap Blocks: exact=1
  Buffers: shared hit=8
  -> Bitmap Index Scan on personnes_gin (actual time=0.026..0.027 rows=1 loops=1)
         Index Cond: (personne @> '{"nom": "Lagaffe", "prenom": "Gaston"}'::jsonb)
         Buffers: shared hit=7
Planning:
   Buffers: shared hit=1
Planning Time: 0.408 ms
Execution Time: 0.081 ms
```

```
EXPLAIN (ANALYZE, VERBOSE)
SELECT r1.*, r2.*
FROM json.personnes,
LATERAL jsonb_to_record (personne)
       AS r1 (nom text, prenom text, date_naissance date),
LATERAL jsonb_to_record (personne->'adresse')
        AS r2 (ville text, pays text)
WHERE personne @> '{"nom": "Lagaffe"}'::jsonb;
                            QUERY PLAN
Nested Loop (cost=25.90..235.79 rows=53 width=132) (actual time=0.051..0.063
→ rows=2 loops=1)
  Output: r1.nom, r1.prenom, r1.date_naissance, r2.ville, r2.pays
   -> Nested Loop (cost=25.90..234.73 rows=53 width=821) (actual time=0.047..0.056
→ rows=2 loops=1)
         Output: personnes.personne, r1.nom, r1.prenom, r1.date_naissance
         -> Bitmap Heap Scan on json.personnes (cost=25.90..233.66 rows=53
→ width=753) (actual time=0.029..0.034 rows=2 loops=1)
               Output: personnes.id_personne, personnes.personne
               Recheck Cond: (personnes.personne @> '{"nom": "Lagaffe"}'::jsonb)
               Heap Blocks: exact=2
               -> Bitmap Index Scan on personnes_gin (cost=0.00..25.88 rows=53
   width=0) (actual time=0.017..0.018 rows=2 loops=1)
                     Index Cond: (personnes.personne @> '{"nom": "Lagaffe"}'::jsonb)
         -> Function Scan on pg_catalog.jsonb_to_record r1 (cost=0.00..0.01 rows=1
   width=68) (actual time=0.009..0.009 rows=1 loops=2)
               Output: r1.nom, r1.prenom, r1.date_naissance
               Function Call: jsonb_to_record(personnes.personne)
   -> Function Scan on pg_catalog.jsonb_to_record r2 (cost=0.01..0.01 rows=1
   width=64) (actual time=0.002..0.003 rows=1 loops=2)
         Output: r2.ville, r2.pays
         Function Call: jsonb_to_record((personnes.personne -> 'adresse'::text))
Planning Time: 0.259 ms
 Execution Time: 0.098 ms
```

Les requêtes sans filtre n'utilisent pas l'index, bien sûr.

# 5.10.4 jsonb et tableaux

Récupérer les numéros de téléphone de Léon Prunelle avec ces trois syntaxes. Quelles sont les différences ?

```
-[RECORD 1]------
personne | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | ["0129951489", "0678327400"]
?column? | 0129951489
```

Les 4 premières lignes renvoient un j sonb , les trois dernières sa conversion en texte :

\gdesc Column	Type
personne ?column? ?column? ?column? ?column? ?column?	jsonb   jsonb   jsonb   jsonb   text   text   text

Afficher les noms et prénoms de Prunelles, et un tableau de champs texte contenant ses numéros de téléphone (utiliser jsonb\_array\_elements\_text).

Il vaut mieux ne pas « bricoler » avec des conversions manuelles du JSON en texte puis en tableau. La fonction dédiée est <code>jsonb\_array\_elements\_text</code>.

Cependant on multiplie les lignes par le nombre de numéros de téléphone, et il faut réagréger :

```
nom | text
tels | text[]
```

La version suivante fonctionnerait aussi dans ce cas précis (cependant elle sera moins performante s'il y a beaucoup de lignes, car PostgreSQL voudra faire un agrégat global au lieu d'un simple parcours ; il faudra aussi vérifier que la clé d'agrégation tient compte d'homonymes).

#### 5.10.5 Accès JSONPath

Comparer le résultat et les performances de ces deux requêtes, qui récupèrent aussi les numéros de téléphone de Prunelle :

```
SELECT jsonb_path_query (personne,
'$.adresse.telephones[*] ? ($.nom == "Prunelle" && $.prenom == "Léon")' ) #>>'{}' AS

   tel
FROM json.personnes ;

SELECT jsonb_path_query (personne, '$.adresse.telephones[*]')#>>'{}'
   AS tel
FROM json.personnes
WHERE personne @@ '$.nom == "Prunelle" && $.prenom == "Léon"' ;
```

Le résultat est le même dans les deux cas :

```
tel
-----
0129951489
0678327400
```

Par contre, le plan et les temps d'exécutions sont totalement différents. La clause j sonb\_path\_query unique parcourt complètement la table :

```
-> Result (actual time=1113.807..1269.568 rows=1 loops=3)
Buffers: shared hit=729122 read=123788
-> ProjectSet (actual time=1113.803..1269.564 rows=1 loops=3)
Buffers: shared hit=729122 read=123788
-> Parallel Seq Scan on personnes (actual time=0.240..304.804
→ rows=177548 loops=3)
Buffers: shared read=55888

Planning Time: 0.134 ms
Execution Time: 1293.548 ms
```

Tandis que la séparation du filtrage et de l'affichage permet à PostgreSQL de sélectionner les lignes, et donc de passer par un index avant de procéder à l'affichage.

```
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS)
SELECT jsonb_path_query (personne, '$.adresse.telephones[*]')#>>'{}' AS tel
FROM json.personnes
WHERE personne @@ '$.nom == "Prunelle" && $.prenom == "Léon"' ;
                             QUERY PLAN
 Result (actual time=2.196..2.207 rows=2 loops=1)
   Buffers: shared hit=2 read=6
   -> ProjectSet (actual time=2.186..2.194 rows=2 loops=1)
         Buffers: shared hit=2 read=6
         -> Bitmap Heap Scan on personnes (actual time=2.167..2.170 rows=1 loops=1)
               Recheck Cond: (personne @@ '($."nom" == "Prunelle" && $."prenom" ==

    "Léon")'::jsonpath)

               Heap Blocks: exact=1
               Buffers: shared hit=2 read=6
               -> Bitmap Index Scan on personnes_gin (actual time=2.113..2.114
→ rows=1 loops=1)
                   Index Cond: (personne @@ '($."nom" == "Prunelle" && $."prenom" ==
  "Léon")'::jsonpath)
                     Buffers: shared hit=2 read=5
Planning:
  Buffers: shared read=4
Planning Time: 2.316 ms
Execution Time: 2.269 ms
```

(À la place de @@, la syntaxe classique @> avec un JSON comme critère, est aussi performante dans ce cas simple.)

Chercher qui possède le numéro de téléphone 0650041821 avec la syntaxe JSONPath.

Ces deux syntaxes sont équivalentes :

```
SELECT personne->>'nom', personne->>'prenom'
FROM     json.personnes
WHERE     personne @@ '$.adresse.telephones[*] == "0650041821" ' ;

SELECT personne->>'nom', personne->>'prenom'
FROM     json.personnes
WHERE     personne @? '$.adresse.telephones[*] ? (@ == "0650041821")' ;
```

```
?column? | ?column?
-----
Delacroix | Justine
```

Dans les deux cas, EXPLAIN montre que l'index GIN est bien utilisé.

Compter le nombre de personnes habitant à Paris ou Bruxelles avec :

```
la syntaxe @> et un OR;
une syntaxe JSONPath @? et un « ou » logique ( | | );
une syntaxe JSONPath @? et une regex @.ville like_regex "^(Paris|Bruxelles)$".
```

Vous devez trouver 63 personnes avec la version complète de la base.

Cet appel va utiliser l'index GIN :

```
EXPLAIN SELECT count(*) FROM json.personnes
WHERE personne @> '{"adresse": {"ville": "Paris"}}'::jsonb
     personne @> '{"adresse": {"ville": "Bruxelles"}}'::jsonb ;
                             QUERY PLAN
Aggregate (cost=467.65..467.66 rows=1 width=8)
   -> Bitmap Heap Scan on personnes (cost=51.82..467.38 rows=107 width=0)
Recheck Cond: ((personne @> '{"adresse": {"ville": "Paris"}}\::jsonb) OR
   (personne @> '{"adresse": {"ville": "Bruxelles"}}'::jsonb))
         -> BitmapOr (cost=51.82..51.82 rows=107 width=0)
               -> Bitmap Index Scan on personnes_gin (cost=0.00..25.88 rows=53
   width=0)
                   Index Cond: (personne @> '{"adresse": {"ville": "Paris"}}'::jsonb)
               -> Bitmap Index Scan on personnes_gin (cost=0.00..25.88 rows=53
   width=0)
                     Index Cond: (personne @> '{"adresse": {"ville":
   "Bruxelles"}}'::jsonb)
Cet appel aussi:
EXPLAIN SELECT count(*) FROM json.personnes
WHERE personne @? '$.adresse ? ( @.ville == "Paris" || @.ville == "Bruxelles") ';
                             QUERY PLAN
Aggregate (cost=2020.86..2020.87 rows=1 width=8)
   -> Bitmap Heap Scan on personnes (cost=48.13..2019.53 rows=533 width=0)
         Recheck Cond: (personne @? '$."adresse"?(@."ville" == "Paris" || @."ville"
   == "Bruxelles")'::jsonpath)
         -> Bitmap Index Scan on personnes_gin (cost=0.00..47.99 rows=533 width=0)
               Index Cond: (personne @? '$."adresse"?(@."ville" == "Paris" ||
```

Par contre, l'index GIN est inutilisable si l'on demande une expression régulière (aussi simple soitelle):

```
EXPLAIN SELECT count(*) FROM json.personnes
WHERE personne @? '$.adresse ? ( @.ville like_regex "^(Paris|Bruxelles)$" ) ';
```

#### QUERY PLAN

```
Finalize Aggregate (cost=56899.96..56899.97 rows=1 width=8)

-> Gather (cost=56899.75..56899.96 rows=2 width=8)

Workers Planned: 2

-> Partial Aggregate (cost=55899.75..55899.76 rows=1 width=8)

-> Parallel Seq Scan on personnes (cost=0.00..55899.19 rows=222

→ width=0)

Filter: (personne @? '$."adresse"?(@."ville" like_regex

→ "^(Paris|Bruxelles)$")'::jsonpath)
```

# 5.10.6 Index fonctionnel, colonne générée et JSON

Le compte du nombre de personne par pays doit être optimisé au maximum. Ajouter un index fonctionnel sur l'attribut pays. Tester l'efficacité sur une recherche, et un décompte de toutes les personnes par pays.

Suivant la syntaxe préférée, l'index peut être par exemple ceci :

```
CREATE INDEX personnes_pays_idx ON json.personnes
USING btree ( (personne->'adresse'->>'pays'));
VACUUM ANALYZE json.personnes ;
```

L'index contient peu de valeurs et fait au plus 3 Mo (beaucoup plus sur une version antérieure à Post-greSQL 13).

Cet index est utilisable pour une recherche à condition que la syntaxe de l'expression soit rigoureusement identique, ce qui limite les cas d'usage.

```
EXPLAIN (ANALYZE, BUFFERS) SELECT count(*) FROM json.personnes
WHERE personne->'adresse'->>'pays' ='Belgique';
                           QUERY PLAN
Aggregate (cost=8.38..8.39 rows=1 width=8) (actual time=0.045..0.046 rows=1
→ loops=1)
  Buffers: shared hit=6
   -> Index Scan using personnes_pays_idx on personnes (cost=0.42..8.38 rows=1

    width=0) (actual time=0.032..0.037 rows=3 loops=1)

        Index Cond: (((personne -> 'adresse'::text) ->> 'pays'::text) =
→ 'Belgique'::text)
        Buffers: shared hit=6
Planning:
  Buffers: shared hit=1
Planning Time: 0.154 ms
 Execution Time: 0.078 ms
Par contre, pour le décompte complet, il n'a aucun intérêt :
EXPLAIN SELECT personne->'adresse'->>'pays', count(*)
FROM json.personnes GROUP BY 1;
                           QUERY PLAN
  ______
```

```
Finalize GroupAggregate (cost=61309.88..61312.72 rows=11 width=40)
Group Key: (((personne → 'adresse'::text) →> 'pays'::text))
→ Gather Merge (cost=61309.88..61312.45 rows=22 width=40)
Workers Planned: 2
→ Sort (cost=60309.86..60309.88 rows=11 width=40)
Sort Key: (((personne → 'adresse'::text) →> 'pays'::text))
→ Partial HashAggregate (cost=60309.50..60309.67 rows=11 width=40)
Group Key: ((personne → 'adresse'::text) →> 'pays'::text)
→ Parallel Seq Scan on personnes (cost=0.00..59199.38

→ rows=222025 width=32)
```

En effet, un index fonctionnel ne permet pas un *Index Only Scan*. Pourtant, il pourrait être très intéressant ici.

```
Ajouter un champ généré dans json.personne, correspondant à l'attribut pays.
```

Attention, l'ordre va réécrire la table, ce qui peut être long (de l'ordre de la minute, suivant le matériel) :

```
ALTER TABLE json.personnes ADD COLUMN pays text

GENERATED ALWAYS AS ( personne->'adresse'->>'pays' ) STORED ;

VACUUM ANALYZE json.personnes ;
```

Comparer les temps d'exécution du décompte des pays par l'attribut, et par cette colonne générée.

\timing on

SELECT personne->'adresse'->>'pays', count(\*) FROM json.personnes GROUP BY 1;

?column?	count
België Belgique Denmark España France Italia Lietuva Poland Portugal United Kingdom United States of America	39597   3   21818   79899   82936   33997   6606   91099   17850   64926

Temps : 601,815 ms

Par contre, la lecture directe du champ est nettement plus rapide :

```
SELECT pays, count(*) FROM json.personnes GROUP BY 1 ;
...
Temps : 58,811 ms
```

Le plan est pourtant le même : un *Seq Scan*, faute de clause de filtrage et d'index, suivi d'un agrégat parallélisé n'utilisant que quelques kilooctets de mémoire.

```
QUERY PLAN
Finalize GroupAggregate (cost=59529.88..59532.67 rows=11 width=19) (actual

    time=61.211..64.244 rows=11 loops=1)

 Group Key: pays
 Buffers: shared hit=55219
 -> Gather Merge (cost=59529.88..59532.45 rows=22 width=19) (actual

    time=61.204..64.235 rows=33 loops=1)

       Workers Planned: 2
       Workers Launched: 2
       Buffers: shared hit=55219
       -> Sort (cost=58529.85..58529.88 rows=11 width=19) (actual
  time=45.186..45.188 rows=11 loops=3)
             Sort Key: pays
             Sort Method: quicksort Memory: 25kB
             Worker 0: Sort Method: quicksort Memory: 25kB
             Worker 1: Sort Method: quicksort Memory: 25kB
             Buffers: shared hit=55219
             -> Partial HashAggregate (cost=58529.55..58529.66 rows=11 width=19)
Group Key: pays
                   Buffers: shared hit=55203
                   -> Parallel Seq Scan on personnes (cost=0.00..57420.70
  rows=221770 width=11) (actual time=0.005..13.678 rows=177548 loops=3)
                         Buffers: shared hit=55203
Planning Time: 0.105 ms
```

Le champ généré a donc un premier intérêt en terme de rapidité de lecture des champs, surtout avec des JSON importants comme ici.

Créer un index B-tree sur la colonne générée pays. Consulter les statistiques dans pg\_stats. Cet index est-il utilisable pour des filtres et le décompte par pays ?

```
CREATE INDEX personnes_g_pays_btree ON json.personnes (pays);
VACUUM ANALYZE json.personnes;
Ces deux ordres ne durent qu'1 ou 2 secondes.
EXPLAIN (ANALYZE, BUFFERS)
SELECT p.pays, count(*)
FROM json.personnes p
GROUP BY 1 ;
                             QUERY PLAN
 Finalize GroupAggregate (cost=1000.45..8885.35 rows=10 width=19) (actual

    time=7.629..49.349 rows=11 loops=1)

   Group Key: pays
   Buffers: shared hit=477
   -> Gather Merge (cost=1000.45..8885.15 rows=20 width=19) (actual

    time=7.625..49.340 rows=11 loops=1)

         Workers Planned: 2
         Workers Launched: 0
         Buffers: shared hit=477
```

Execution Time: 64.297 ms

```
-> Partial GroupAggregate (cost=0.42..7882.82 rows=10 width=19) (actual

time=7.371..49.034 rows=11 loops=1)

Group Key: pays

Buffers: shared hit=477

-> Parallel Index Only Scan using personnes_g_pays_btree on personnes

p (cost=0.42..6771.79 rows=222186 width=11) (actual time=0.023..22.578

rows=532645 loops=1)

Heap Fetches: 0

Buffers: shared hit=477

Planning Time: 0.114 ms

Execution Time: 49.391 ms
```

Le gain en temps est appréciable. Mais l'intérêt principal réside ici dans le nombre de blocs lus divisé par 100! Le nouvel index ne fait que 3 Mo.

#### \di+ json.personnes\*

	Liste (	des relat <sup>.</sup>	ions			
Schéma	Nom	Type		Méthode d'accès	Taille	<b></b>
	+	++-	+		+	+
json	personnes_g_pays_btree	index		btree	3664 kB	
json	personnes_gin	index		gin	230 MB	
json	personnes_pays_idx	index		btree	3664 kB	

(Optionnel) Créer des colonnes générées sur nom, prenom, date\_naissance, et ville (en un seul ordre). Reprendre la requête plus haut qui les affiche tous et comparer les performances.

Un champ va poser problème : la date de naissance. En effet, la date est stockée au format texte, il faudra soi-même faire la conversion. De plus, un simple opérateur ::date ne peut être utilisé dans une expression de GENERATED car il n'est pas « immutable » (pour des raisons techniques<sup>46</sup>).

Un contournement pas très performant est celui-ci:

```
ALTER TABLE json.personnes

ADD COLUMN nom text GENERATED ALWAYS AS (personne->>'prenom') STORED,

ADD COLUMN prenom text GENERATED ALWAYS AS (personne->>'nom') STORED,

ADD COLUMN date_naissance date

GENERATED ALWAYS AS (

make_date (left(personne->>'date_naissance',4)::int,

substring(personne->>'date_naissance',6,2)::int,

left(personne->>'date_naissance',2)::int))

STORED,

ADD COLUMN ville text GENERATED ALWAYS AS ( personne->'adresse'->>'ville') STORED;

VACUUM ANALYZE json.personnes;
```

Une autre possibilité plus performante est d'enrober to\_date() dans une fonction immutable, puisqu'il n'y a, dans ce cas précis, pas d'ambiguïté sur le format ISO :

```
CREATE OR REPLACE FUNCTION to_date_immutable (text)
RETURNS date
-- Cette fonction requiert que les dates soient bien
-- stockées au format ci-dessous
```

<sup>&</sup>lt;sup>46</sup>https://www.postgresql.org/message-id/flat/201111290311.pAT3Bf318279%40momjian.us

```
-- et ne fait aucune gestion d'erreur sinon

LANGUAGE sql

IMMUTABLE PARALLEL SAFE

AS $body$

SELECT to_date($1, 'YYYY-MM-DD');

$body$;

et l'ordre devient:

ALTER TABLE json.personnes
...

ADD COLUMN date_naissance date

GENERATED ALWAYS AS (to_date_immutable (personne->>'date_naissance')) STORED,
...;
```

Les conversions de texte vers des dates sont des sources fréquentes de problèmes. Le conseil habituel est de toujours stocker une date dans un champ de type date ou timestamp / timestamptz. Mais si elle provient d'un JSON, il faudra gérer soi-même la conversion.

Quelle que soit la méthode, la requête suivante :

elle-même plus rapide que les extractions manuelles des attributs un à un, comme vu plus haut.

Certes, la table est un peu plus grosse, mais le coût d'insertion des colonnes générées est donc souvent rentable pour les champs fréquemment utilisés.

#### 5.10.7 jsonb et mise à jour

Ajouter l'attribut animaux à Gaston Lagaffe, avec la valeur 18. Vérifier en relisant la ligne.

Ajouter l'attribut animaux à 2% des individus au hasard, avec une valeur 1 ou 2.

On utilise ici la fonction <code>jsonb\_build\_object()</code>, plus adaptée à la construction d'un JSON qui n'est pas une constante. Le choix des individus peut se faire de plusieurs manières, par exemple avec <code>random()</code>, <code>mod()</code>...

```
UPDATE json.personnes
SET personne = personne ||
    jsonb_build_object ('animaux', 1+mod ((personne->>'numgen')::int, 50))
WHERE mod((personne->>'numgen')::int,50) = 0;
UPDATE 10653
-- Conseillé après chaque mise à jour importante
VACUUM ANALYZE json.personnes;
```

Compter le nombre de personnes avec des animaux (avec ou sans JSONPath). Proposer un index qui pourrait convenir à d'autres futurs nouveaux attributs peu fréquents.

Ces requêtes renvoient 10654, mais effectuent toutes un *Seq Scan* avec une durée d'exécution aux alentours de la seconde :

```
SELECT count(*) FROM json.personnes
WHERE (personne->>'animaux')::int > 0 ;

SELECT count(*) FROM json.personnes
WHERE personne ? 'animaux' ;

SELECT count(*) FROM json.personnes
WHERE personne @@ '$.animaux > 0' ;

SELECT count(*) FROM json.personnes
WHERE personne @? '$.animaux ? (@ > 0) ' ;
```

(Remarquer que les deux dernières requêtes utiliseraient l'index GIN pour des égalités comme (@ == 0) ou (@ == 18), et seraient presque instantanées. Là encore, c'est une limite des index GIN.)

On pourrait indexer (personne->>'animaux')::int, ce qui serait excellent pour la première requête, mais ne conviendrait pas à d'autres critères.

L'opérateur ? ne sait pas utiliser l'index GIN jsonb\_path\_ops existant. Par contre, il peut profiter de l'opérateur GIN par défaut :

```
CREATE INDEX personnes_gin_df ON json.personnes USING gin (personne);

EXPLAIN

SELECT count(*) FROM json.personnes

WHERE personne ? 'animaux';

QUERY PLAN

Aggregate (cost=42263.34..42263.35 rows=1 width=8)
```

Cet index avec l'opérateur jsonb\_ops a par contre le gros inconvénient d'être encore plus gros que l'index GIN avec jsonb\_path\_ops (303 Mo contre 235 Mo), et d'alourdir encore les mises à jour. Il peut cependant remplacer ce dernier, de manière un peu moins performante. Il faut aviser selon les requêtes, la place, les écritures...

# 5.10.8 Large Objects

- Créer une table fichiers avec un texte et une colonne permettant de référencer des *Large Objects*.

```
CREATE TABLE fichiers (nom text PRIMARY KEY, data OID);
```

- Importer un fichier local à l'aide de psql dans un large object.
- Noter l'oid retourné.

```
psql -c "\lo_import '/etc/passwd'"
lo_import 6821285
INSERT INTO fichiers VALUES ('/etc/passwd',6821285);
```

- Importer un fichier du serveur à l'aide de psql dans un large object.

```
INSERT INTO fichiers SELECT 'postgresql.conf',
lo_import('/var/lib/pgsql/15/data/postgresql.conf');
```

- Afficher le contenu de ces différents fichiers à l'aide de psql.

- Les sauvegarder dans des fichiers locaux.

```
psql -c "\lo_export loid_retourné '/home/dalibo/passwd_serveur';"
```

# 6/ Fonctionnalités avancées pour la performance

# **6.1 PRÉAMBULE**



Quelques fonctionnalités, généralement liées aux performances.

# **6.1.1** Au menu



- Tables temporaires
  Tables non journalisées
  Colonnes générées
  JIT
  Recherche Full Text

#### **6.2 TABLES TEMPORAIRES**

# 6.2.1 Tables temporaires : utilité



- CREATE TEMP TABLE travail (...);

   N'existent que pendant la session
   Non journalisées : rapides !

#### Principe:

Sous PostgreSQL, les tables temporaires sont créées dans une session, et disparaissent à la déconnexion. Elles ne sont pas visibles par les autres sessions. Elles ne sont pas journalisées, ce qui est très intéressant pour les performances. Elles s'utilisent comme les autres tables, y compris pour l'indexation, les triggers, etc.

Les tables temporaires semblent donc idéales pour des tables de travail temporaires et « jetables ».

# 6.2.2 Tables temporaires : limites et paramétrage



- Ne pas en abuser!
  Ignorées par autovacuum
  ANALYZE et VACUUM manuels!
  Paramétrage:
  - - temp\_buffers : cache disque pour les objets temporaires, par session, à augmenter?



Cependant, il est déconseillé d'abuser des tables temporaires. En effet, leur création/destruction permanente entraîne une fragmentation importante des tables systèmes (en premier lieu pg\_catalog.pg\_class , pg\_catalog.pg\_attribute ...), qui peuvent devenir énormes. Ce n'est jamais bon pour les performances, et peut nécessiter un VACUUM FULL des tables système!



Le démon autovacuum ne voit pas les tables temporaires! Les statistiques devront donc être mises à jour manuellement avec ANALYZE, et il faudra penser à lancer VACUUM explicitement après de grosses modifications.

#### Aspect technique:

Les tables temporaires sont créées dans un schéma temporaire pg\_temp\_..., ce qui explique qu'elles ne sont pas visibles dans le schéma public.

Physiquement, par défaut, elles sont stockées sur le disque avec les autres données de la base, et non dans base/pgsql\_tmp comme les fichiers temporaires. Il est possible de définir des tablespaces dédiés aux objets temporaires (fichiers temporaires et données des tables temporaires) à l'aide du paramètre temp\_tablespaces, à condition de donner des droits CREATE dessus aux utilisateurs. Le nom du fichier d'une table temporaire est reconnaissable car il commence par t. Les éventuels index de la table suivent les même règles.

#### Exemple:

```
CREATE TEMP TABLE travail (x int PRIMARY KEY) ;
EXPLAIN (COSTS OFF, ANALYZE, BUFFERS, WAL)
INSERT INTO travail SELECT i FROM generate_series (1,1000000) i;
                              QUERY PLAN
Insert on travail (actual time=1025.752..1025.753 rows=0 loops=1)
  Buffers: shared hit=13, local hit=2172174 read=4 dirtied=7170 written=10246
  I/O Timings: read=0.012
   -> Function Scan on generate_series i (actual time=77.112..135.624 rows=1000000
→ loops=1)
Planning Time: 0.028 ms
Execution Time: 1034.984 ms
SELECT pg_relation_filepath ('travail') ;
pg_relation_filepath
base/13746/t7_5148873
\d pg_temp_7.travail
                Table « pg_temp_7.travail »
Colonne | Type | Collationnement | NULL-able | Par défaut
    | integer |
                           | not null |
   "travail_pkey" PRIMARY KEY, btree (x)
```

#### Cache:

Dans les plans d'exécution avec BUFFERS, l'habituelle mention shared est remplacée par local pour les tables temporaires.

```
CREATE TEMP TABLE demotemp AS SELECT * FROM pg_class;

EXPLAIN (ANALYZE, BUFFERS, COSTS OFF) SELECT * FROM demotemp;

QUERY PLAN

Seq Scan on demotemp (actual time=0.013..0.109 rows=602 loops=1)
Buffers: local hit=16
Planning:
Buffers: shared hit=66 read=1
I/0 Timings: shared read=0.009
Planning Time: 0.250 ms
Execution Time: 0.170 ms

(Les shared hit concernent des accès aux tables systèmes pour la planification.)
```

En effet, leur cache disque dédié est au niveau de la session, non des *shared buffers*. Ce cache est défini par le paramètre temp\_buffers (exprimé par session, et à 8 Mo par défaut). Ce paramètre peut être augmenté dans la session, avant la création de la table. Bien sûr, on risque de saturer la RAM en cas d'abus ou s'il y a trop de sessions, tout comme avec work\_mem. Et comme la mémoire de temp\_buffers n'est pas rendue avant la fin de la session, il faut éviter de maintenir inutilement ouvertes des sessions ayant créé des tables temporaires : elles bloquent de la mémoire.

À noter : ce cache n'empêche pas l'écriture des petites tables temporaires sur le disque.

Pour éviter de recréer perpétuellement la même table temporaire, une table *unlogged* (voir plus bas) sera sans doute plus indiquée. Le contenu de cette dernière sera aussi visible des autres sessions, ce qui est pratique pour suivre la progression d'un traitement, faciliter le travail de l'autovacuum, ou déboguer. Sinon, il est fréquent de pouvoir remplacer une table temporaire par une CTE (clause with) ou un tableau en mémoire.

L'extension pgtt<sup>1</sup> émule un autre type de table temporaire dite « globale » pour la compatibilité avec d'autres SGBD.

<sup>&</sup>lt;sup>1</sup>https://github.com/darold/pgtt

# 6.3 TABLES NON JOURNALISÉES (UNLOGGED)

#### 6.3.1 Tables non journalisées : utilité



- La durabilité est parfois accessoire :
  - tables temporaires et de travail
  - caches...
- Tables non journalisées
  - non répliquées, non restaurées
  - remises à zéro en cas de crash
- Respecter les contraintes

Une table *unlogged* est une table non journalisée. Comme la journalisation est responsable de la durabilité, une table non journalisée n'a pas cette garantie.



La table est systématiquement remise à zéro au redémarrage après un arrêt brutal. En effet, tout arrêt d'urgence peut entraîner une corruption des fichiers de la table ; et sans journalisation, il ne serait pas possible de la corriger au redémarrage et de garantir l'intégrité.

La non-journalisation de la table implique aussi que ses données ne sont pas répliquées vers des serveurs secondaires, et que les tables ne peuvent figurer dans une publication (réplication logique). En effet, les modes de réplication natifs de PostgreSQL utilisent les journaux de transactions. Pour la même raison, une restauration de sauvegarde PITR ne restaurera pas le contenu de la table. Le bon côté est qu'on allège la charge sur la sauvegarde et la réplication.

Les contraintes doivent être respectées même si la table *unlogged* est vidée : une table normale ne peut donc avoir de clé étrangère pointant vers une table *unlogged*. La contrainte inverse est possible, tout comme une contrainte entre deux tables *unlogged*.

À part ces limitations, les tables *unlogged* se comportent exactement comme les autres. Leur intérêt principal est d'être en moyenne 5 fois plus rapides à la mise à jour. Elles sont donc à réserver à des cas d'utilisation particuliers, comme :

- table de spooling/staging;
- table de cache/session applicative;
- table de travail partagée entre sessions ;
- table de travail systématiquement reconstruite avant utilisation dans le flux applicatif;

- et de manière générale toute table contenant des données dont on peut accepter la perte sans impact opérationnel ou dont on peut regénérer aisément les données.

Les tables unlogged ne doivent pas être confondues avec les tables temporaires (non journalisées et visibles uniquement dans la session qui les a créées). Les tables unlogged ne sont pas ignorées par l'autovacuum (les tables temporaires le sont). Abuser des tables temporaires a tendance à générer de la fragmentation dans les tables système, alors que les tables unlogged sont en général créées une fois pour toutes.

# 6.3.2 Tables non journalisées: mise en place



```
CREATE UNLOGGED TABLE ma_table (col1 int ...);
```

Une table unlogged se crée exactement comme une table journalisée classique, excepté qu'on rajoute le mot UNLOGGED dans la création.

# 6.3.3 Bascule d'une table en/depuis unlogged



```
ALTER TABLE table_normale SET UNLOGGED;
- réécriture
ALTER TABLE table_unlogged SET LOGGED;
- passage du contenu dans les WAL!
```

Il est possible de basculer une table à volonté de normale à unlogged et vice-versa.

Quand une table devient unlogged, on pourrait imaginer que PostgreSQL n'a rien besoin d'écrire. Malheureusement, pour des raisons techniques, la table doit tout de même être réécrite. Elle est défragmentée au passage, comme lors d'un VACUUM FULL. Ce peut être long pour une grosse table, et il faudra voir si le gain par la suite le justifie.

Les écritures dans les journaux à ce moment sont théoriquement inutiles, mais là encore des optimisations manquent et il se peut que de nombreux journaux soient écrits si les sommes de contrôles ou wal\_log\_hints sont activés. Par contre il n'y aura plus d'écritures dans les journaux lors des modifications de cette table, ce qui reste l'intérêt majeur.

#### **DALIBO Formations**

Quand une table *unlogged* devient *logged* (journalisée), la réécriture a aussi lieu, et tout le contenu de la table est journalisé (c'est indispensable pour la sauvegarde PITR et pour la réplication notamment), ce qui génère énormément de journaux et peut prendre du temps.

Par exemple, une table modifiée de manière répétée pendant un batch, peut être définie *unlogged* pour des raisons de performance, puis basculée en *logged* en fin de traitement pour pérenniser son contenu.

# **6.4 COLONNES GÉNÉRÉES**

# 6.4.1 Colonnes générées : principe



# 6.4.2 Colonnes générées vs DEFAULT



- DEFAULT
  - expressions très simples, modifiables
- GENERATED
  - fonctions « immutables », ne dépendant **que** de la ligne
  - danger sinon (ex : pas pour dates de mises à jour)
  - (avant v17) difficilement modifiables
  - peuvent porter des contraintes

Les valeurs par défaut sont très connues mais limitées. PostgreSQL connaît aussi les colonnes générées (ou calculées).

#### La syntaxe est:

```
nomchamp <type> GENERATED ALWAYS AS ( <expression> ) STORED ;
```

Les colonnes générées sont recalculées à chaque fois que les champs sur lesquels elles sont basées changent, donc aussi lors d'un UPDATE Ces champs calculés sont impérativement marqués ALWAYS,

c'est-à-dire obligatoires et non modifiables, et STORED, c'est-à-dire stockés sur le disque (et non re-calculés à la volée comme dans une vue). Ils ne doivent pas se baser sur d'autres champs calculés.

Un intérêt est que les champs calculés peuvent porter des contraintes, par exemple la clause CHECK ci-dessous, mais encore des clés étrangères ou unique.

#### Exemple:

```
CREATE TABLE paquet (
                         PRIMARY KEY,
           text
  reception timestamptz DEFAULT now(),
 livraison timestamptz DEFAULT now() + interval '3d',
 largeur int, longueur int, profondeur int,
  volume
           int
     GENERATED ALWAYS AS ( largeur * longueur * profondeur )
     STORED CHECK (volume > 0.0)
  ) ;
INSERT INTO paquet (code, largeur, longueur, profondeur)
VALUES ('ZZ1', 3, 5, 10);
\x on
TABLE paquet;
-[ RECORD 1 ]-----
          | ZZ1
reception | 2024-04-19 18:02:41.021444+02
livraison | 2024-04-22 18:02:41.021444+02
largeur
          | 3
longueur | 5
profondeur | 10
         | 150
-- Les champs DEFAULT sont modifiables
-- Changer la largeur va modifier le volume
UPDATE paquet
SET largeur=4,
   livraison = '2024-07-14'::timestamptz,
   reception = '2024-04-20'::timestamptz
WHERE code='ZZ1' ;
TABLE paquet;
-[ RECORD 1 ]-----
code
         ZZ1
reception | 2024-04-20 00:00:00+02
livraison | 2024-07-14 00:00:00+02
         | 4
| 5
largeur
longueur
profondeur | 10
         200
-- Le volume ne peut être modifié
UPDATE paquet
SET volume = 250
WHERE code = 'ZZ1' ;
```

```
ERROR: column "volume" can only be updated to DEFAULT DETAIL: Column "volume" is a generated column.
```

#### **Expression immutable:**

Avec GENERATED, l'expression du calcul doit être « immutable », c'est-à-dire ne dépendre **que** des autres champs de la même ligne, n'utiliser que des fonctions elles-mêmes immutables, et rien d'autre. Il n'est donc pas possible d'utiliser des fonctions comme now(), ni des fonctions de conversion de date dépendant du fuseau horaire, ou du paramètre de formatage de la session en cours (toutes choses autorisées avec DEFAULT), ni des appels à d'autres lignes ou tables...

La colonne calculée peut être convertie en colonne « normale » :

```
ALTER TABLE paquet ALTER COLUMN volume DROP EXPRESSION;
```

Modifier l'expression n'est pas possible avant PostgreSQL 17, sauf à supprimer la colonne générée et en créer une nouvelle. Il faut alors recalculer toutes les lignes et réécrire toute la table, ce qui peut être très lourd.

À partir de PostgreSQL 17, l'expression est modifiable avec cette syntaxe :

```
ALTER TABLE paquet ALTER COLUMN volume

SET EXPRESSION AS ( largeur * longueur * profondeur + 1 ) ;
```

Attention, la table est totalement bloquée le temps de la réécriture (verrou AccessExclusiveLock ).

#### Utilisation d'une fonction :

Il est possible de créer sa propre fonction pour l'expression, qui doit aussi être immutable :

```
CREATE OR REPLACE FUNCTION volume (l int, h int, p int)
RETURNS int
AS $$
SELECT l * h * p ;
$$
LANGUAGE sql
-- cette fonction dépend uniquement des données de la ligne donc :
PARALLEL SAFE
IMMUTABLE ;
-- Changement à partir de PostgreSQL v17
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS (volume (largeur, longueur, profondeur));
-- Changement avant PostgreSQL 16
ALTER TABLE paquet DROP COLUMN volume ;
ALTER TABLE paquet ADD COLUMN volume int
  GENERATED ALWAYS AS ( volume (largeur, longueur, profondeur) )
  STORED;
TABLE paquet;
-[ RECORD 1 ]-----
code
         ZZ1
reception | 2024-04-20 00:00:00+02
livraison | 2024-07-14 00:00:00+02
```

largeur | 4 longueur | 5 profondeur | 10 volume | 200



Attention : modifier la fonction ne réécrit pas spontanément la table, il faut forcer la réécriture avec par exemple :

```
UPDATE paquet SET longueur = longueur ;
```

et ceci dans la même transaction que la modification de fonction. On pourrait imaginer de négliger cet UPDATE pour garder les valeurs déjà présentes qui suivraient d'anciennes règles... mais ce serait une erreur. En effet, les valeurs calculées ne figurent pas dans une sauvegarde logique, et en cas de restauration, tous les champs sont recalculés avec la dernière formule !

On préférera donc gérer l'expression dans la définition de la table dans les cas simples.



Un autre piège : il faut résister à la tentation de déclarer une fonction comme immutable sans la certitude qu'elle l'est bien (penser aux paramètres de session, aux fuseaux horaires...), sous peine d'incohérences dans les données.

### Cas d'usage :

Les colonnes générées économisent la création de triggers, ou de vues de « présentation ». Elles facilitent la dénormalisation de données calculées dans une même table tout en garantissant l'intégrité.

Un cas d'usage courant est la dénormalisation d'attributs JSON pour les manipuler comme des champs de table classiques :

```
ALTER TABLE personnes
ADD COLUMN lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED;
```

L'accès au champ est notablement plus rapide que l'analyse systématique du champ JSON.

Par contre, les colonnes GENERATED ne sont **pas** un bon moyen pour créer des champs portant la dernière mise à jour. Certes, PostgreSQL ne vous empêchera pas de déclarer une fonction (abusivement) immutable utilisant now() ou une variante. Mais ces informations seront perdues en cas de restauration logique. Dans ce cas, les triggers restent une option plus complexe mais plus propre.

# 6.5 JIT

# 6.5.1 JIT: la compilation à la volée



- Compilation *Just In Time* des requêtes
- Utilise le compilateur LLVM
- Vérifier que l'installation est fonctionnelle
- Activé par défaut
  - sauf en v11; et absent auparavant

Une des nouveautés les plus visibles et techniquement pointues de la v11 est la « compilation à la volée » (*Just In Time compilation*, ou JIT) de certaines expressions dans les requêtes SQL. Le JIT n'est activé par défaut qu'à partir de la version 12.

Dans certaines requêtes, l'essentiel du temps est passé à décoder des enregistrements (*tuple deforming*), à analyser des clauses WHERE, à effectuer des calculs. En conséquence, l'idée du JIT est de transformer tout ou partie de la requête en un programme natif directement exécuté par le processeur.

Cette compilation est une opération lourde qui ne sera effectuée que pour des requêtes qui en valent le coup, donc qui dépassent un certain coût. Au contraire de la parallélisation, ce coût n'est pas pris en compte par le planificateur. La décision d'utiliser le JIT ou pas se fait une fois le plan décidé, si le coût calculé de la requête dépasse un certain seuil.

Le JIT de PostgreSQL s'appuie actuellement sur la chaîne de compilation LLVM, choisie pour sa flexibilité. L'utilisation nécessite un PostgreSQL compilé avec l'option ——with—llvm et l'installation des bibliothèques de LLVM.

Sur Debian, avec les paquets du PGDG, les dépendances sont en place dès l'installation.

Sur Rocky Linux/Red Hat 8 et 9, l'installation du paquet dédié suffit :

```
# dnf install postgresql14-llvmjit
```

Sur CentOS/Red Hat 7, ce paquet supplémentaire nécessite lui-même des paquets du dépôt EPEL :

```
# yum install epel-release
# yum install postgresql14-llvmjit
```

Les systèmes CentOS/Red Hat 6 ne permettent pas d'utiliser le JIT.

Si PostgreSQL ne trouve pas les bibliothèques nécessaires, il ne renvoie pas d'erreur et continue sans tenter de JIT. Pour tester si le JIT est fonctionnel sur votre machine, il faut le chercher dans un plan quand on force son utilisation ainsi :

```
SET jit=on;
SET jit_above_cost TO 0;
EXPLAIN (ANALYZE) SELECT 1;

QUERY PLAN

Result (cost=0.00..0.01 rows=1 width=4) (... rows=1 loops=1)
Planning Time: 0.069 ms

JIT:
Functions: 1
Options: Inlining false, Optimization false, Expressions true,
Deforming true

Timing: Generation 0.123 ms, Inlining 0.000 ms, Optimization 0.187 ms,
Emission 2.778 ms, Total 3.088 ms

Execution Time: 3.952 ms
```

La documentation officielle est assez accessible: https://doc.postgresql.fr/current/jit.html

# 6.5.2 JIT: qu'est-ce qui est compilé?



- Tuple deforming
- Évaluation d'expressions :
  - WHERE
  - agrégats, GROUP BY
- Appels de fonctions (inlining)
- Mais pas les jointures

Le JIT ne peut pas encore compiler toute une requête. La version actuelle se concentre sur des goulots d'étranglement classiques :

- le décodage des enregistrements (tuple deforming) pour en extraire les champs intéressants ;
- les évaluations d'expressions, notamment dans les clauses WHERE pour filtrer les lignes ;
- les agrégats, les GROUP BY ...

Les jointures ne sont pas (encore?) concernées par le JIT.

Le code résultant est utilisable plus efficacement avec les processeurs actuels qui utilisent les pipelines et les prédictions de branchement.

Pour les détails, on peut consulter notamment cette conférence très technique au FOSDEM 2018<sup>2</sup> par l'auteur principal du JIT, Andres Freund.

<sup>&</sup>lt;sup>2</sup>https://archive.fosdem.org/2018/schedule/event/jiting\_postgresql\_using\_llvm/

## 6.5.3 JIT: algorithme « naïf »



```
jit (défaut: on )
jit_above_cost (défaut: 100 000)
jit_inline_above_cost (défaut: 500 000)
jit_optimize_above_cost (défaut: 500 000)
À comparer au coût de la requête... I/O comprises
Seuils arbitraires!
```

De l'avis même de son auteur, l'algorithme de déclenchement du JIT est « naïf ». Quatre paramètres existent (hors débogage).

jit = on (défaut à partir de la v12) active le JIT **si** l'environnement technique évoqué plus haut le permet.

La compilation n'a cependant lieu que pour un coût de requête calculé d'au moins jit\_above\_cost (par défaut 100 000, une valeur élevée). Puis, si le coût atteint jit\_inline\_above\_cost (500 000), certaines fonctions utilisées par la requête et supportées par le JIT sont intégrées dans la compilation. Si jit\_optimize\_above\_cost (500 000) est atteint, une optimisation du code compilé est également effectuée. Ces deux dernières opérations étant longues, elles ne le sont que pour des coûts assez importants.

Ces seuils sont à comparer avec les coûts des requêtes, qui incluent les entrées-sorties, donc pas seulement le coût CPU. Ces seuils sont un peu arbitraires et nécessiteront sans doute un certain tuning en fonction de vos requêtes et de vos processeurs.

Des contre-performances dues au JIT ont déjà été observées, menant à monter les seuils. Le JIT est trop jeune pour que les développeurs de PostgreSQL eux-mêmes aient des règles d'ajustement des valeurs des différents paramètres. Il est fréquent de le désactiver ou de monter radicalement les seuils de déclenchement.

Un exemple de plan d'exécution sur une grosse table donne :

Rows Removed by Filter: 62500

Planning Time: 0.047 ms

JIT:

Functions: 42

Options: Inlining true, Optimization true, Expressions true, Deforming true Timing: Generation 5.611 ms, Inlining 422.019 ms, Optimization 229.956 ms,

Emission 125.768 ms, Total 783.354 ms

Execution Time: 11785.276 ms

Le plan d'exécution est complété, à la fin, des informations suivantes :

- le nombre de fonctions concernées;

- les temps de génération, d'inclusion des fonctions, d'optimisation du code compilé...

Dans l'exemple ci-dessus, on peut constater que ces coûts ne sont pas négligeables par rapport au temps total. Il reste à voir si ce temps perdu est récupéré sur le temps d'exécution de la requête... ce qui en pratique n'a rien d'évident.

Sans JIT, la durée de cette requête était d'environ 17 s. Ici le JIT est rentable.

# 6.5.4 Quand le JIT est-il utile?



- Goulot d'étranglement au niveau CPU (pas I/O)
- Requêtes complexes (calculs, agrégats, appels de fonctions...)
- Beaucoup de lignes, filtres
- Assez longues pour « rentabiliser » le JIT
- Analytiques, pas ERP

Vu son coût élevé, le JIT n'a d'intérêt que pour les requêtes utilisant beaucoup le CPU et où il est le facteur limitant.

Ce seront donc surtout des requêtes analytiques agrégeant beaucoup de lignes, comprenant beaucoup de calculs et filtres, et non les petites requêtes d'un ERP.

Il n'y a pas non plus de mise en cache du code compilé.

Si gain il y a, il est relativement modeste en deçà de quelques millions de lignes, et devient de plus en plus important au fur et à mesure que la volumétrie augmente, à condition bien sûr que d'autres limites n'apparaissent pas (bande passante...).

Documentation officielle: https://docs.postgresql.fr/current/jit-decision.html

# **6.6 RECHERCHE PLEIN TEXTE**



## 6.6.1 Full Text Search: principe



- Recherche « à la Google » ; fonctions dédiées
- On n'indexe plus une chaîne de caractère mais
  - les mots (« lexèmes ») qui la composent
  - on peut rechercher sur chaque lexème indépendamment
- Les lexèmes sont soumis à des règles spécifiques à chaque langue
  - notamment termes courants
  - permettent une normalisation, des synonymes...

L'indexation FTS est un des cas les plus fréquents d'utilisation non-relationnelle d'une base de données : les utilisateurs ont souvent besoin de pouvoir rechercher une information qu'ils ne connaissent pas parfaitement, d'une façon floue :

- recherche d'un produit/article par rapport à sa description;
- recherche dans le contenu de livres/documents...

PostgreSQL doit donc permettre de rechercher de façon efficace dans un champ texte. L'avantage de cette solution est d'être intégrée au SGBD. Le moteur de recherche est donc toujours parfaitement à jour avec le contenu de la base, puisqu'il est intégré avec le reste des transactions.

Le principe est de décomposer le texte en « lexèmes » propres à chaque langue. Cela implique donc une certaine forme de normalisation, et permettent aussi de tenir compte de dictionnaires de synonymes. Le dictionnaire inclue aussi les termes courants inutiles à indexer (*stop words*) propres à la langue (le, la, et, the, and, der, daß…).

Décomposition et recherche en plein texte utilisent des fonctions et opérateurs dédiés, ce qui nécessite donc une adaptation du code. Ce qui suit n'est qu'un premier aperçu. La recherche plein texte est un chapitre entier de la documentation officielle<sup>3</sup>.

Adrien Nayrat a donné une excellente conférence sur le sujet au PGDay France 2017 à Toulouse<sup>4</sup> (slides<sup>5</sup>).

<sup>&</sup>lt;sup>3</sup>https://docs.postgresql.fr/current/textsearch.html

<sup>4</sup>https://www.youtube.com/embed/9S5dBqMbw8A

<sup>&</sup>lt;sup>5</sup>https://2017.pgday.fr/slides/nayrat\_Le\_Full\_Text\_Search\_dans\_PostgreSQL.pdf

# 6.6.2 Full Text Search: exemple



- Décomposition :

to\_tsvector analyse un texte et le décompose en lexèmes, et non en mots. Les chiffres indiquent ici les positions et ouvrent la possibilité à des scores de proximité. Mais des indications de poids sont possibles.

Autre exemple de décomposition d'une phrase :

Les mots courts et le verbe « être » sont repérés comme termes trop courants, la casse est ignorée, même l'URL est décomposée en protocole et hôte. On peut voir en détail comment la FTS a procédé :

```
SELECT description, token, dictionary, lexemes
FROM ts_debug('La documentation de PostgreSQL est sur https://www.postgresql.org/');
```

description	token	dictionary	lexemes
Word, all ASCII Space symbols Word, all ASCII Space symbols		french_stem ¤ french_stem ¤	¤

```
Word, all ASCII | de
                         | french_stem | {}
Space symbols
Word, all ASCII | PostgreSQL
                        | french_stem | {postgresql}
Space symbols |
                        | m | m
Word, all ASCII | est
                        | french_stem | {}
Space symbols |
                        | ¤
Word, all ASCII | sur
                        | french_stem | {}
Space symbols
                        | m | m
Protocol head | https:// | ¤
                                  | ¤
          Host
Space symbols | /
```

Si l'on se trompe de langue, les termes courants sont mal repérés (et la recherche sera inefficace) :

```
SELECT to_tsvector ('english',
'La documentation de PostgreSQL est sur https://www.postgresql.org/');
                                    to_tsvector
 'de':3 'document':2 'est':5 'la':1 'postgresql':4 'sur':6 'www.postgresql.org':7
Pour construire un critère de recherche, to_tsquery est nécessaire :
SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('Valjean & Cosette');
Les termes à chercher peuvent être combinés par &, | (ou), ! (négation), <-> (mots successifs),
(séparés par N lexèmes). @ est l'opérateur de correspondance. Il y en a d'autres<sup>6</sup>.
Il existe une fonction phraseto_tsquery pour donner une phrase entière comme critère, laquelle
sera décomposée en lexèmes :
SELECT livre, contenu FROM textes
      livre ILIKE 'Les Misérables Tome V%'
AND ( to_tsvector ('french',contenu)
       @@ phraseto_tsquery('c''est la fautes de Voltaire')
  OR to_tsvector ('french',contenu)
        @@ phraseto_tsquery('nous sommes tombés à terre')
    );
                      livre
Les misérables Tome V Jean Valjean, Hugo, Victor | Je suis tombé par terre,
Les misérables Tome V Jean Valjean, Hugo, Victor | C'est la faute à Voltaire,
```

<sup>&</sup>lt;sup>6</sup>https://docs.postgresql.fr/current/functions-textsearch.html

### 6.6.3 Full Text Search: dictionnaires



- Configurations liées à la langue
  - basées sur des dictionnaires (parfois fournis)
  - dictionnaires filtrants ( unaccent )
  - synonymes
- Extensible grâce à des sources extérieures
- Configuration par défaut: default\_text\_search\_config

Les lexèmes, les termes courants, la manière de décomposer un terme... sont fortement liés à la langue.

Des configurations toutes prêtes sont fournies par PostgreSQL pour certaines langues :

#### #\dF

Liste des configurations de la recherche de texte					
Schéma	Nom	Description			
pg_catalog	arabic	configuration for arabic language			
pg_catalog	danish	configuration for danish language			
pg_catalog	dutch	configuration for dutch language			
pg_catalog	english	configuration for english language			
pg_catalog	finnish	configuration for finnish language			
pg_catalog	french	configuration for french language			
pg_catalog	german	configuration for german language			
pg_catalog	hungarian	configuration for hungarian language			
pg_catalog	indonesian	configuration for indonesian language			
pg_catalog	irish	configuration for irish language			
pg_catalog	italian	configuration for italian language			
pg_catalog	lithuanian	configuration for lithuanian language			
pg_catalog	nepali	configuration for nepali language			
pg_catalog	norwegian	configuration for norwegian language			
pg_catalog	portuguese	configuration for portuguese language			
pg_catalog	romanian	configuration for romanian language			
pg_catalog	russian	configuration for russian language			
pg_catalog	simple	simple configuration			
pg_catalog	spanish	configuration for spanish language			
pg_catalog	swedish	configuration for swedish language			
pg_catalog	tamil	configuration for tamil language			
pg_catalog	turkish	configuration for turkish language			

La recherche plein texte est donc directement utilisable pour le français ou l'anglais et beaucoup d'autres langues européennes. La configuration par défaut dépend du paramètre default\_text\_search\_config , même s'il est conseillé de toujours passer explicitement la configuration aux fonctions. Ce paramètre peut être modifié globalement, par session ou par un ALTER DATABASE SET.

En demandant le détail de la configuration french, on peut voir qu'elle se base sur des « dic-

tionnaires » pour chaque type d'élément qui peut être rencontré : mots, phrases mais aussi URL, entiers...

```
# \dF+ french
Configuration « pg_catalog.french » de la recherche de texte
Analyseur : « pg_catalog.default »
     Jeton | Dictionnaires
asciihword | french_stem
asciiword
               | french_stem
email
               | simple
file
               | simple
 float
              | simple
host
              | simple
         | french_stem
hword
hword_asciipart | french_stem
hword_numpart | simple
hword_part | french_stem
int
              | simple
numhword
.
              | simple
numword
              | simple
sfloat
              | simple
uint
               | simple
               | simple
url
url_path
version
              | simple
               | simple
word
               | french_stem
```

### On peut lister ces dictionnaires:

Ces dictionnaires sont de type « Snowball<sup>7</sup> », incluant notamment des algorithmes différents pour chaque langue. Le dictionnaire simple n'est pas lié à une langue et correspond à une simple décomposition après passage en minuscule et recherche de termes courants anglais : c'est suffisant pour des éléments comme les URL.

D'autres dictionnaires peuvent être combinés aux existants pour créer une nouvelle configuration. Le principe est que les dictionnaires reconnaissent certains éléments, et transmettent aux suivants ce qu'ils n'ont pas reconnu. Les dictionnaires précédents, de type Snowball, reconnaissent tout et doivent donc être placés en fin de liste.

<sup>&</sup>lt;sup>7</sup>https://snowballstem.org/

Par exemple, la contrib unaccent permet de faire une configuration négligeant les accents<sup>8</sup>. La contrib dict\_int fournit un dictionnaire qui réduit la précision des nombres<sup>9</sup> pour réduire la taille de l'index. La contrib dict\_xsyn permet de créer un dictionnaire pour gérer une liste de synonymes<sup>10</sup>. Mais les dictionnaires de synonymes peuvent être gérés manuellement<sup>11</sup>. Les fichiers adéquats sont déjà présents ou à ajouter dans \$SHAREDIR/tsearch\_data/ (par exemple

/usr/pgsql-14/share/tsearch\_data surRed Hat/CentOS ou /usr/share/postgresql/14/tsearch\_data sur Debian).

Par exemple, en utilisant le fichier d'exemple \$SHAREDIR/tsearch\_data/synonym\_sample.syn , dont le contenu est :

```
postgresql
postgre pgsql
gogle googl
indices index*
```

on peut définir un dictionnaire de synonymes, créer une nouvelle configuration reprenant et y insérer le nouveau dictionnaire en premier élément :

Les trois versions de « PostgreSQL » ont été reconnues.

Pour une analyse plus fine, on peut ajouter d'autres dictionnaires linguistiques depuis des sources extérieures (Ispell, OpenOffice...). Ce n'est pas intégré par défaut à PostgreSQL mais la procédure est dans la documentation<sup>12</sup>.

Des « thesaurus » peuvent être même être créés pour remplacer des expressions par des synonymes (et identifier par exemple « le meilleur SGBD » et « PostgreSQL »).

<sup>&</sup>lt;sup>8</sup>https://docs.postgresql.fr/current/unaccent.html

<sup>&</sup>lt;sup>9</sup>https://docs.postgresql.fr/current/dict-int.html

<sup>&</sup>lt;sup>10</sup>https://docs.postgresql.fr/current/dict-xsyn.html

<sup>11</sup> https://docs.postgresql.fr/current/textsearch-dictionaries.html#textsearch-synonym-dictionary

<sup>&</sup>lt;sup>12</sup>https://docs.postgresql.fr/current/textsearch-dictionaries.html

# 6.6.4 Full Text Search: stockage & indexation



- Stocker to\_tsvector (champtexte)
  - colonne mise à jour par trigger
  - ou colonne générée (v12)
- Indexation GIN ou GiST

#### Principe:

Sans indexation, une recherche FTS fonctionne, mais parcourra entièrement la table. L'indexation est possible, avec GIN ou GiST. On peut stocker le vecteur résultat de to\_tsvector dans une autre colonne de la table, et c'est elle qui sera indexée. Jusque PostgreSQL 11, il est nécessaire de le faire manuellement, ou d'écrire un trigger pour cela. À partir de PostgreSQL 12, on peut utiliser une colonne générée (il est nécessaire de préciser la configuration FTS), qui sera stockée sur le disque :

```
-- Attention, ceci réécrit la table
ALTER TABLE textes
ADD COLUMN vecteur tsvector
GENERATED ALWAYS AS (to_tsvector ('french', contenu)) STORED;
Les critères de recherche porteront sur la colonne vecteur :

SELECT * FROM textes
WHERE vecteur @@ to_tsquery ('french','Roméo <2> Juliette');
```

Cette colonne sera ensuite indexée par GIN pour avoir des temps d'accès corrects :

```
CREATE INDEX on textes USING gin (vecteur) ;
```

# **Alternative: index fonctionnel**

Plus simplement, il peut suffire de créer juste un index fonctionnel sur to\_tsvector ('french', contenu). On épargne ainsi l'espace du champ calculé dans la table.

Par contre, l'index devra porter sur le critère de recherche exact, sinon il ne sera pas utilisable. Cela n'est donc pertinent que si la majorité des recherches porte sur un nombre très restreint de critères, et il faudra un index par critère.

```
CREATE INDEX idx_fts ON public.textes
USING gin (to_tsvector('french'::regconfig, contenu))

SELECT * FROM textes
WHERE to_tsvector ('french', contenu) @@ to_tsquery ('french', 'Roméo <2> Juliette');
```

### Exemple complet de mise en place de FTS :

 Création d'une configuration de dictionnaire dédiée avec dictionnaire français, sans accent, dans une table de dépêches :

CREATE TEXT SEARCH CONFIGURATION depeches (COPY= french);

```
CREATE EXTENSION unaccent;
ALTER TEXT SEARCH CONFIGURATION depeches ALTER MAPPING FOR
hword, hword_part, word WITH unaccent, french_stem;
   - Ajout d'une colonne vectorisée à la table depeche, avec des poids différents pour le titre et le
     texte, ici gérée manuellement avec un trigger.
CREATE TABLE depeche (id int, titre text, texte text) ;
ALTER TABLE depeche ADD vect_depeche tsvector;
UPDATE depeche
SET vect_depeche =
(setweight(to_tsvector('depeches',coalesce(titre,'')), 'A') ||
setweight(to_tsvector('depeches',coalesce(texte,'')), 'C'));
CREATE FUNCTION to_vectdepeche( )
 RETURNS trigger
 LANGUAGE plpgsql
 -- common options: IMMUTABLE STABLE STRICT SECURITY DEFINER
AS $function$
BEGIN
  NEW.vect_depeche :=
    setweight(to_tsvector('depeches',coalesce(NEW.titre,'')), 'A') ||
    setweight(to_tsvector('depeches',coalesce(NEW.texte,'')), 'C');
  return NEW;
END
$function$;
CREATE TRIGGER trg_depeche before INSERT OR update ON depeche
FOR EACH ROW execute procedure to_vectdepeche();
   - Création de l'index associé au vecteur :
 CREATE INDEX idx_gin_texte ON depeche USING gin(vect_depeche);

    Collecte des statistiques sur la table :

ANALYZE depeche;
   - Utilisation basique:
SELECT titre, texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle');
SELECT titre, texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle & médecin');
   - Tri par pertinenence:
```

```
SELECT titre,texte
FROM depeche
WHERE vect_depeche @@ to_tsquery('depeches','varicelle & médecin')
ORDER BY ts_rank_cd(vect_depeche, to_tsquery('depeches','varicelle & médecin'));
    - Cette requête peut s'écrire aussi ainsi:

SELECT titre,ts_rank_cd(vect_depeche,query) AS rank
FROM depeche, to_tsquery('depeches','varicelle & médecin') query
WHERE query@@vect_depeche
ORDER BY rank DESC;
```

#### 6.6.5 Full Text Search sur du JSON



- Vectorisation possible des JSON

Une recherche FTS est directement possible sur des champs JSON. Voici un exemple :

```
CREATE TABLE commandes (info jsonb);

INSERT INTO commandes (info)
VALUES

(
    '{ "client": "Jean Dupont",
        "articles": {"produit": "Enveloppes A4","qté": 24}}'
),
 (
    '{ "client": "Jeanne Durand",
        "articles": {"produit": "Imprimante","qté": 1}}'
),
 (
    '{ "client": "Benoît Delaporte",
        "items": {"produit": "Rame papier normal A4","qté": 5}}'
),
 (
    '{ "client": "Lucie Dumoulin",
        "items": {"produit": "Pochette Papier dessin A3","qté": 5}}'
);
```

# La décomposition par FTS donne :

```
to_tsvector

'a4':5 'dupont':2 'envelopp':4 'jean':1
'durand':2 'imprim':4 'jeann':1
'a4':4 'benoît':6 'delaport':7 'normal':3 'papi':2 'ram':1
'a3':4 'dessin':3 'dumoulin':7 'luc':6 'papi':2 'pochet':1

Une recherche sur « papier » donne:

SELECT info FROM commandes c
WHERE to_tsvector ('french', c.info) @@ to_tsquery('papier') ;

info

{"items": {"qté": 5, "produit": "Rame papier normal A4"}, "client": "Benoît

Delaporte"}
{"items": {"qté": 5, "produit": "Pochette Papier dessin A3"}, "client": "Lucie

Dumoulin"}
```

Plus d'information chez Depesz: Full Text Search support for json and jsonb<sup>13</sup>.

<sup>&</sup>lt;sup>13</sup>https://www.depesz.com/2017/04/04/waiting-for-postgresql-10-full-text-search-support-for-json-and-jsonb/

# **6.7 QUIZ**



# **6.8 TRAVAUX PRATIQUES**

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/t1\_solutions.

# 6.8.1 Tables non journalisées



**But** : Tester les tables non journalisées

Afficher le nom du journal de transaction courant.

Créer une base **pgbench** vierge, de taille 80 (environ 1,2 Go). Les tables doivent être en mode **unlogged**.

Afficher la liste des objets **unlogged** dans la base **pgbench**.

Afficher le nom du journal de transaction courant. Que s'est-il passé?

Passer l'ensemble des tables de la base **pgbench** en mode **logged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé?

Repasser toutes les tables de la base **pgbench** en mode **unlogged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé?

Réinitialiser la base **pgbench** toujours avec une taille 80 mais avec les tables en mode **logged**. Que constate-t-on?

Réinitialiser la base **pgbench** mais avec une taille de 10. Les tables doivent être en mode **unlog-ged**.

Compter le nombre de lignes dans la table pgbench\_accounts.

Simuler un crash de l'instance PostgreSQL.

Redémarrer l'instance PostgreSQL.

Compter le nombre de lignes dans la table pgbench\_accounts . Que constate-t-on ?

### 6.8.2 Indexation Full Text



**But**: Tester l'indexation Full Text

Vous aurez besoin de la base **textes**. La base **gutenberg** est disponible en deux versions : complète sur https://dali.bo/tp\_gutenberg (dump de 0,5 Go, table de 21 millions de lignes dans 3 Go) ou https://dali.bo/tp\_gutenberg10 pour un extrait d'un dizième. Le dump peut se restaurer par exemple dans une nouvelle base, et contient juste une table nommée textes.

```
curl -kL https://dali.bo/tp_gutenberg -o /tmp/gutenberg.dmp
createdb gutenberg
pg_restore -d gutenberg /tmp/gutenberg.dmp
# le message sur le schéma public exitant est normale
rm -- /tmp/gutenberg.dmp
```

Ce TP utilise la version complète de la base **textes** basée sur le projet Gutenberg. Un index GIN va permettre d'utiliser la *Full Text Search* sur la table **textes**.

Créer un index GIN sur le vecteur du champ contenu (fonction to\_tsvector).

Quelle est la taille de cet index?

Quelle performance pour trouver « Fantine » (personnage des *Misérables* de Victor Hugo) dans la table ? Le résultat contient-il bien « Fantine » ?

Trouver les lignes qui contiennent à la fois les mots « affaire » et « couteau » et voir le plan.

# **6.9 TRAVAUX PRATIQUES (SOLUTIONS)**

# 6.9.1 Tables non journalisées

Afficher le nom du journal de transaction courant.

Créer une base **pgbench** vierge, de taille 80 (environ 1,2 Go). Les tables doivent être en mode **unlogged**.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 80 --unlogged-tables pgbench
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
8000000 of 80000000 tuples (100%) done (elapsed 4.93 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 8.84 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 5.02 s,
vacuum 1.79 s, primary keys 2.02 s).
```

Afficher la liste des objets unlogged dans la base pgbench.

```
SELECT relname FROM pg_class
WHERE relpersistence = 'u' ;

relname
-----
pgbench_accounts
pgbench_branches
pgbench_history
pgbench_tellers
pgbench_branches_pkey
pgbench_tellers_pkey
pgbench_accounts_pkey
```

Les 3 objets avec le suffixe **pkey** correspondent aux clés primaires des tables créées par **pgbench**. Comme elles dépendent des tables, elles sont également en mode **unlogged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé?

```
SELECT pg_walfile_name(pg_current_wal_lsn()) ;
```

```
pg_walfile_name
-----0000001000000100000024
```

Comme l'initialisation de **pgbench** a été réalisée en mode **unlogged**, aucune information concernant les tables et les données qu'elles contiennent n'a été inscrite dans les journaux de transaction. Donc le journal de transaction est toujours le même.

Passer l'ensemble des tables de la base **pgbench** en mode **logged**.

```
ALTER TABLE pgbench_accounts SET LOGGED;
ALTER TABLE pgbench_branches SET LOGGED;
ALTER TABLE pgbench_history SET LOGGED;
ALTER TABLE pgbench_tellers SET LOGGED;
```

Afficher le nom du journal de transaction courant. Que s'est-il passé?

Comme toutes les tables de la base **pgbench** ont été passées en mode **logged**, une réécriture de cellesci a eu lieu (comme pour un VACUUM FULL). Cette réécriture additionnée au mode **logged** a entraîné une forte écriture dans les journaux de transaction. Dans notre cas, 83 journaux de transaction ont été consommés, soit approximativement 1,3 Go d'utilisé sur disque.

Il faut donc faire particulièrement attention à la quantité de journaux de transaction qui peut être générée lors du passage d'une table du mode **unlogged** à **logged**.

Repasser toutes les tables de la base **pgbench** en mode **unlogged**.

```
ALTER TABLE pgbench_accounts SET UNLOGGED;
ALTER TABLE pgbench_branches SET UNLOGGED;
ALTER TABLE pgbench_history SET UNLOGGED;
ALTER TABLE pgbench_tellers SET UNLOGGED;
```

Afficher le nom du journal de transaction courant. Que s'est-il passé?

Le processus est le même que précedemment, mais, lors de la réécriture des tables, aucune information n'est stockée dans les journaux de transaction.

Réinitialiser la base **pgbench** toujours avec une taille 80 mais avec les tables en mode **logged**. Que constate-t-on?

```
$ /usr/pgsql-14/bin/pgbench -i -s 80 -d pgbench
```

```
dropping old tables...
creating tables...
generating data (client-side)...
8000000 of 80000000 tuples (100%) done (elapsed 9.96 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 16.60 s (drop tables 0.11 s, create tables 0.00 s, client-side generate 10.12

s,
vacuum 2.87 s, primary keys 3.49 s).
```

On constate que le temps mis par **pgbench** pour initialiser sa base est beaucoup plus long en mode **logged** que **unlogged**. On passe de 8,84 secondes en **unlogged** à 16,60 secondes en mode **logged**. Cette augmentation du temps de traitement est due à l'écriture dans les journaux de transaction.

Réinitialiser la base **pgbench** mais avec une taille de 10. Les tables doivent être en mode **unlog-ged**.

```
$ /usr/pgsql-14/bin/pgbench -i -s 10 -d pgbench --unlogged-tables
dropping old tables...
creating tables...
generating data (client-side)...
1000000 of 10000000 tuples (100%) done (elapsed 0.60 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 1.24 s (drop tables 0.02 s, create tables 0.02 s, client-side generate 0.62 s,
vacuum 0.27 s, primary keys 0.31 s).
```

Compter le nombre de lignes dans la table pgbench\_accounts .

```
SELECT count(*) FROM pgbench_accounts ;
  count
-----
1000000
```

Simuler un crash de l'instance PostgreSQL.

```
$ ps -ef | grep postmaster
postgres 697 1 0 14:32 ? 00:00:00 /usr/pgsql-14/bin/postmaster -D ...
$ kill -9 697
```



Ne faites jamais un kill -9 sur un processus de l'instance PostgreSQL en production, bien sûr!

Redémarrer l'instance PostgreSQL.

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data start
```

Compter le nombre de lignes dans la table pgbench\_accounts . Que constate-t-on ?

```
SELECT count(*) FROM pgbench_accounts ;
count
-----
0
```

Lors d'un crash, PostgreSQL remet tous les objets unlogged à zéro.

#### 6.9.2 Indexation Full Text

```
Créer un index GIN sur le vecteur du champ contenu (fonction to_tsvector ).
```

```
textes=# CREATE INDEX idx_fts ON textes
USING gin (to_tsvector('french',contenu));
CREATE INDEX
```

Quelle est la taille de cet index?

La table « pèse » 3 Go (même si on pourrait la stocker de manière beaucoup plus efficace). L'index GIN est lui-même assez lourd dans la configuration par défaut :

```
textes=# SELECT pg_size_pretty(pg_relation_size('idx_fts'));
  pg_size_pretty
-----
593 MB
(1 ligne)
```

Quelle performance pour trouver « Fantine » (personnage des *Misérables* de Victor Hugo) dans la table ? Le résultat contient-il bien « Fantine » ?

```
textes=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');
                                   QUERY PLAN
 Bitmap Heap Scan on textes (cost=107.94..36936.16 rows=9799 width=123)
                          (actual time=0.423..1.149 rows=326 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu)
                  @@ '''fantin'''::tsquery)
  Heap Blocks: exact=155
  Buffers: shared hit=159
  -> Bitmap Index Scan on idx_fts (cost=0.00..105.49 rows=9799 width=0)
                              (actual time=0.210..0.211 rows=326 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu)
                      @@ '''fantin'''::tsquery)
         Buffers: shared hit=4
 Planning Time: 1.248 ms
 Execution Time: 1.298 ms
```

On constate donc que le *Full Text Search* est très efficace du moins pour le *Full Text Search* + GIN : trouver 1 mot parmi plus de 100 millions avec 300 enregistrements correspondants dure 1,5 ms (cache chaud).

Si l'on compare avec une recherche par trigramme (extension pg\_trgm et index GIN), c'est bien meilleur. À l'inverse, les trigrammes permettent des recherches floues (orthographe approximative), des recherches sur autre chose que des mots, et ne nécessitent pas de modification de code.

Par contre, la recherche n'est pas exacte, « Fantin » est fréquemment trouvé. En fait, le plan montre que c'est le vrai critère retourné par to\_tsquery('french','fantine') et transformé en 'fantin'::tsquery. Si l'on tient à ce critère précis il faudra ajouter une clause plus classique contenu LIKE '%Fantine%' pour filtrer le résultat après que le FTS ait « dégrossi » la recherche.

Trouver les lignes qui contiennent à la fois les mots « affaire » et « couteau » et voir le plan.

10 lignes sont ramenées en quelques millisecondes :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','affaire & couteau')
                                  QUERY PLAN
 Bitmap Heap Scan on textes (cost=36.22..154.87 rows=28 width=123)
                       (actual time=6.642..6.672 rows=10 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu)
                 @@ '''affair'' & ''couteau'''::tsquery)
  Heap Blocks: exact=10
   Buffers: shared hit=53
   -> Bitmap Index Scan on idx_fts (cost=0.00..36.21 rows=28 width=0)
                             (actual time=6.624..6.624 rows=10 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu)
                      @@ '''affair'' & ''couteau'''::tsquery)
         Buffers: shared hit=43
 Planning Time: 0.519 ms
 Execution Time: 6.761 ms
```

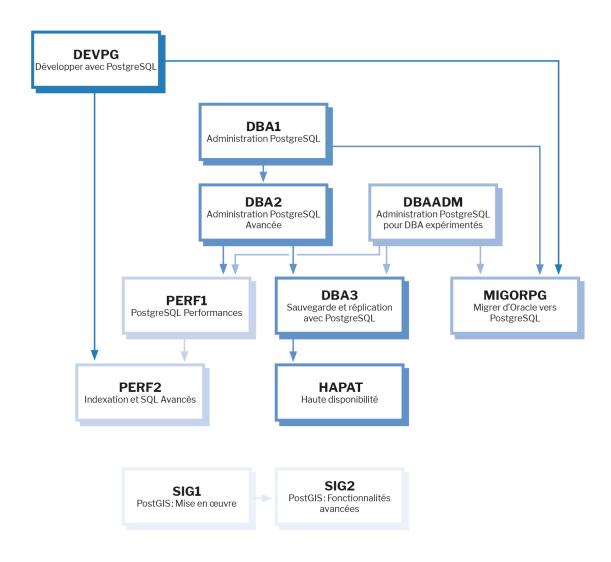
Noter que les pluriels « couteaux » et « affaires » figurent parmi les résultats puisque la recherche porte sur les léxèmes 'affair'' & ''couteau'.

# **Les formations Dalibo**

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

# **Cursus des formations**



#### Retrouvez nos formations dans leur dernière version:

- DBA1: Administration PostgreSQL https://dali.bo/dba1
- DBA2: Administration PostgreSQL avancé https://dali.bo/dba2
- DBA3: Sauvegarde et réplication avec PostgreSQL https://dali.bo/dba3
- DEVPG: Développer avec PostgreSQL https://dali.bo/devpg
- PERF1: PostgreSQL Performances https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés https://dali.bo/perf2
- MIGORPG: Migrer d'Oracle à PostgreSQL https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL https://dali.bo/hapat

### Les livres blancs

- Migrer d'Oracle à PostgreSQL https://dali.bo/dlb01
- Industrialiser PostgreSQL https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL https://dali.bo/dlb05

# **Téléchargement gratuit**

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

