PGSession 2026

nistration d’instances PostgreSQL avec CloudNa

0.2

Contents

1/ Introduction

1.1 OBJECTIF DE L'ATELIER

- Découverte et prise en main de l'opérateur CloudNativePG
- Configuration d'instances PostgreSQL
- Supervision et journaux de traces

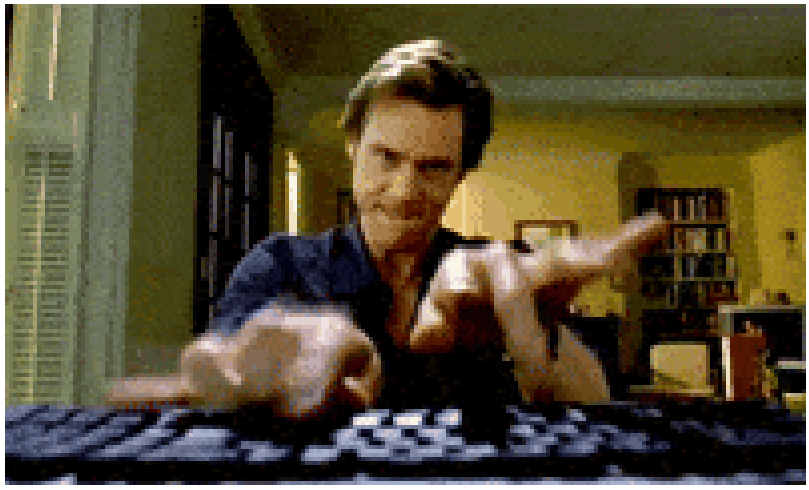
L'objectif de cet atelier est de découvrir l'opérateur CloudNativePG qui nous permet de déployer facilement PostgreSQL sur Kubernetes. Une présentation générale est faite pour évoquer certains aspects des opérateurs et de leur utilisation.

Le TP vous permettra de manipuler un `Cluster` PostgreSQL, déployé avec CloudNativePG, à travers sa configuration, la création d'une base, la supervision de celui-ci et la gestion des journaux de traces. D'autres notions pourront être abordés selon le temps restant.

1.2 DÉROULÉ DE L'ATELIER

Durée : ~ 3 heures
Pause : 15h30

1. Quelques mots sur PostgreSQL dans Kubernetes
2. L'opérateur CloudNativePG
3. **Travaux pratiques** : accompagné, avoir le même rythme



1.3

2/ Kubernetes

- Plateforme de déploiement de conteneurs libre et *Open Source*
- *Cloud Native Computing Foundation* (CNCF)
- Distributions : K3s, AKS, GKE, Rancher, Openshift, **kind**, …
- Déploiements applicatifs, *auto-scaling*, redéploiement automatique, *load balancing*, …

Kubernetes est de plus en plus présent dans les infrastructures techniques des départements/services IT. Grâce à ses fonctionnalités, comme l'*auto-scaling* ou le redéploiement automatique, il est plébiscité pour le déploiement des applications dites *stateless*.

De plus en plus d'outils liés à la donnée sont déployés dans Kubernetes. Les systèmes de gestion de bases de données "classiques" n'y coupent pas.

2.1 DES DONNÉES DANS KUBERNETES ?!

- Pas une évidence
- Habituellement du *stateless*
- Apparition des `StatefulSet`
- Couche supplémentaire (complexité ?)
- Effet de mode ? Pas que !

Choisissez surtout une solution qui correspond à vos besoins !

Historiquement, le déploiement d'instances PostgreSQL ou autre système de gestion de bases de données (SGBD) en général, dans Kubernetes, peut sembler tout sauf évident, voire même contre-intuitif. Cependant, l'évolution de Kubernetes, avec l'apparition des `StatefulSet`, et surtout des opérateurs, offrent aujourd'hui des solutions viables pour le déploiement de bases de données.

La simplification des déploiements et les fonctionnalités proposées vont de pair avec une couche d'abstraction et de complexité supplémentaire : ici les opérateurs Kubernetes pour PostgreSQL.

3/ L’opérateur CloudNativePG

CloudNativePG is a comprehensive platform designed to seamlessly manage PostgreSQL databases within Kubernetes environments, covering the entire operational lifecycle from initial deployment to ongoing maintenance.

3.1

- https://cloudnative-pg.io
- Débuté par 2ndQuadrant puis EDB. Enfin libéré en 2022
- Projet open source, licence Apache 2.0
- Mode de gouvernance similaire à PostgreSQL

Le projet a été initialement développé par EDB puis libéré en 2022 pour la communauté. La gouvernance du projet se rapproche de celle du projet PostgreSQL avec une *core team* et l'ouverture à la contribution.

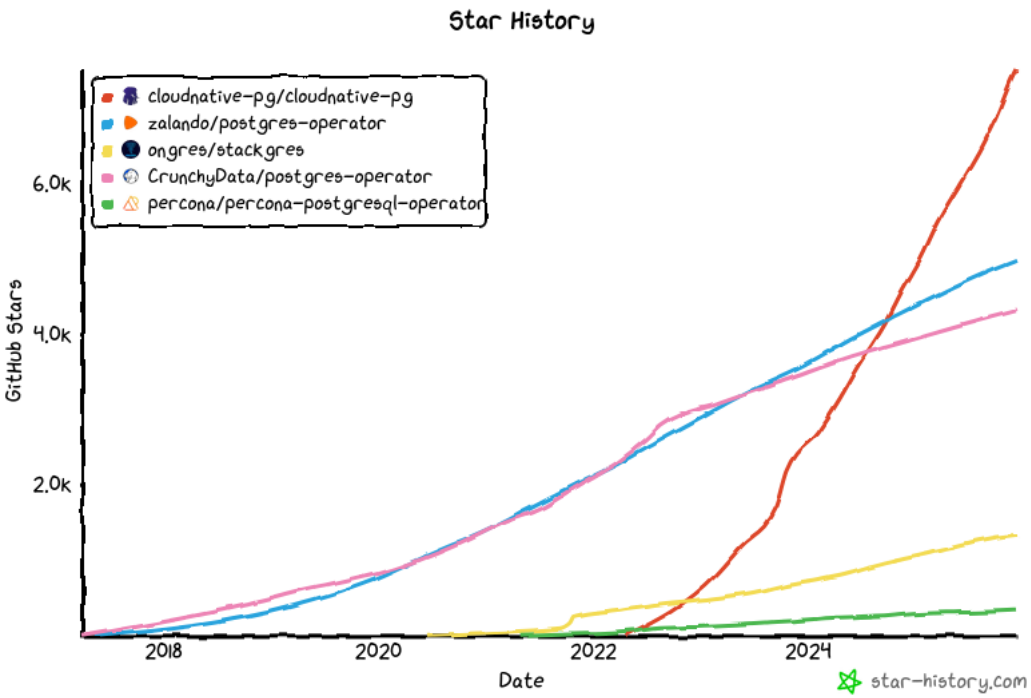
Le projet est bien engagé dans une démarche communautaire et open source avec une incubation au sein du projet CNCF.

3.2 L'ADOPTION

- De plus en plus cité, des présentations (KubeCon, PGConfEU)
- Intégré au programme *Sandbox* de la CNCF
- Grand attrait sur Github

> CloudNativePG was accepted to CNCF on January 21, 2025 at the Sandbox maturity level.

3.3 L'ADOPTION



Les *stars* ne sont pas un argument "d'autorité", mais montre en engouement certain.

3.4 CE QUE PERMET CLOUDNATIVEPG

- Déploiement de cluster d'instances PostgreSQL facilité
- Mise en place de réplication automatique
- Sauvegardes PITR, planifiées
  - Via *plugin*
- Bascule (automatique ou manuelle)
- Haute disponibilité

3.5 SUPERVISION

Une instance gérée via CloudNativePG :

- Exporte des métriques `Prometheus` prédéfinies
- Permet de définir des métriques maisons
- S'intègre facilement avec Grafana

La supervision des instances PostgreSQL déployées par CloudNativePG est possible via :

- L'export de métriques "Prometheus" pour chaque instance sur le port `9187` (*Endpoint* `/metrics`, ce *endpoint* expose aussi des informations à propos des sauvegardes) ;
- des métriques prédéfinies, ou possibilité de définir des métriques personnalisées ;
- Des requêtes atomiques exécutées avec le rôle PostgreSQL `pg_monitor` ;
- Grafana peut être utilisé pour visualiser les métriques et créer des Dashboards. Voir le dépôt Github grafana-dashboards[1].

[1] github.com/cloudnative-pg/grafana-dashboards

3.6 TRACE

CloudNativePG standardise la gestion des traces :

- JSON, envoyés vers la sortie standard (*stdout*) du Pod
- Aucune persistance locale (pas de fichier de log)
- Rétention & analyse nécessitent des outils spécifiques
- Différents niveaux : *error*, *warning*, `info` (défaut), *debug*, *trace*

Principes généraux :

- Tous les logs (opérateur & PostgreSQL) sont émis en JSON vers *stdout*
- Aucune persistance locale des logs (pas de fichiers)
- Facilite l'intégration avec des outils dédiés (ex. EFK, Loki, VictoriaLogs, …)
- La rétention long terme est gérée au niveau de l'infrastructure Kubernetes

Structure des logs :

- Champs communs : level, ts, logger, msg, record, logging_pod
- logger identifie la source qui a émis la trace (postgres, pgaudit, operator, wal, backup…)
- Niveaux de logs : error, warning, info (défaut), debug, trace

La configuration se fait :

- Pour les instances (instances PostgreSQL) via spec.logLevel
- Pour l'opérateur / manager via –log-level dans le Deployment

Le changement de niveau s'applique uniquement aux nouveaux `Pods`

Pour PGAudit :

- Support natif
- Activation via l'ajout de paramètres PostgreSQL
- Gestion automatique de `shared_preload_libraries` et des extensions
- Logs d'audit exposés en JSON (logger=pgaudit)

3.7 ATTENTION, TOUT N'EST PAS SI SIMPLE

- Connaissances Kubernetes
  - Même en tant que DBA
- Connaissances PostgreSQL
  - Même en tant qu'admin K8S
- Changements d'habitudes et d'outils
  - Plus de *SSH* sur la machine où se trouve l'instance
  - Configuration dans les ressources Kubernetes

Les échanges entre administrateurs Kubernetes et PostgreSQL sont à renforcer pour qu'ils puissent se comprendre. Laisser la gestion d'instances PostgreSQL aux administrateurs Kubernetes, sous prétexte que tout se fait via un opérateur, et donc sans impliquer un DBA, n'est pas une solution viable. PostgreSQL est très spécifique et demande une vraie connaissance de son fonctionnement. Un DBA saura comprendre ce que fait un opérateur et réagira correctement en cas de problème.

Une montée en compétences et connaissances est donc nécessaires pour les différents administrateurs.

Des changements d'habitudes de travail auront nécessairement lieu et impliquent également un accompagnement des équipes DBAs. L'exemple le plus parlant est l'absence d'accès SSH au serveur où est déployée l'instance, ou encore le fait de devoir passer par Kubernetes pour configurer l'instance. Certains opérateurs imposent des outils (notamment pour la partie sauvegarde) qui doivent être connus des DBAs.

3.8 ATTENTION, TOUT N'EST PAS SI SIMPLE

- Couche(s) d'abstraction(s) supplémentaire(s)
  - Debug plus long / compliqué
  - Avoir les bons outils
- Jongler avec les versions
  - Kubernetes : 3 supportées
  - CloudNativePG : 2 supportées
  - PostgreSQL : 5 supportées

Le déploiement de PostgreSQL dans Kubernetes a des conséquences qu'il est important d'avoir en tête. Celles-ci ne sont pas insurmontables, à condition d'en être conscient.

La première à citer est la couche d'abstraction supplémentaire apportée par Kubernetes et l'opérateur. L'empilement de couches rendra par exemple le débug probablement plus long sans les bons outils de monitoring.

La gestion des versions se voit complexifiée avec un "jonglage" à faire pour avoir un bon alignement des versions supportées de PostgreSQL, de l'opérateur et de Kubernetes. La fréquence des différentes *releases* est assez élevée.

4/ Travaux pratiques

Lien du TP :

https://dali.bo/ws_cnpg_2026

4.1 PRISE EN MAIN DE L'ENVIRONNEMENT KUBERNETES

But : Prendre en main le cluster Kubernetes et découvrir ce qui est installé.

Se connecter à la machine qui vous est dédiée.

```
ssh -p 22XX dalibo@A.B.C.D
```

Trouver la version de l'utilitaire `kubectl`.

```
kubectl version
```

```
Client Version: v1.35.0
Kustomize Version: v5.7.1
Server Version: v1.35.0
```

Lister les nœuds du *cluster* Kubernetes.

```
kubectl get nodes
```

```
NAME                 STATUS   ROLES           AGE     VERSION
kind-control-plane   Ready    control-plane   9m8s    v1.35.0
kind-worker          Ready    <none>          8m55s   v1.35.0
kind-worker2         Ready    <none>          8m55s   v1.35.0
```

Cet utilitaire sait avec quel *cluster* Kubernetes interagir grâce au fichier `~/.kube/config` qui se trouve dans le répertoire de l'utilisateur système.

L'opérateur CloudNativePG a été installé après la création du *cluster* Kubernetes. Assurez-vous que l'opérateur est bien présent dans le `Namespace cnpg-system`.

```
kubectl get pod -n cnpg-system
```

```
NAME                                       READY   STATUS    RESTARTS   AGE
cnpg-controller-manager-65bfdb64c9-wfbx7   1/1     Running   0          77s
```

Pour information, la version installée de l'opérateur est la version 1.27.2.

4.2 CRÉATION D'UN POSTGRESQL

But : Créer un *cluster* PostgreSQL avec deux instances.

Nous allons éditer un fichier YAML qui va nous servir de base pour notre *cluster* PostgreSQL.

La structure initiale est la suivante :

```
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-postgresql
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.6-standard-trixie
  instances: 2
  storage:
    size: 10Gi
  walStorage:
    size: 5Gi
```

Un *cluster* PostgreSQL avec deux instances PostgreSQL en version 17.6. Avec un espace de stockage de 10 Go pour les données et 5 Go pour les journaux de transactions.

> Le fichier est déjà présent dans le *home* du compte `dalibo`. Créer ce `Cluster` avec la commande :

```
kubectl apply -f cluster.yaml
```

> Attendre que les images soient téléchargées et que le `Cluster` soit créé par l'opérateur. Les deux `Pods` sont *up and running* quelques minutes après.

```
kubectl get pod
```

```
NAME                   READY   STATUS    RESTARTS   AGE
cluster-postgresql-1   1/1     Running   0          68s
cluster-postgresql-2   1/1     Running   0          85s
```

La réplication entre les deux instances est automatiquement créée par CloudNativePG.

4.3 CONFIGURATION RAM ET CPU

But : Allouer de la RAM et du CPU à notre `Cluster` PostgreSQL.

Dans notre exemple, rien n'a été alloué en terme de CPU et RAM à nos `Pods` PostgreSQL. Vous pouvez le voir en regardant les paramètres `Requests` et `Limits` d'un `Pod`. Il n'y a donc pas de limite définie.

> Retrouver les resources attribuées à un `Pod` avec la commande suivante :

```
kubectl get pod cluster-postgresql-1 -o json | jq '.spec.containers[].resources'
```

La réponse renvoyée est vide :

```
kubectl get pod cluster-postgresql-1 -o json | jq '.spec.containers[].resources'

{}
```

Par défaut, nos `Pods` ne sont soumis à aucune limite en terme de ressources. La seule limite qu'ils auraient est la capacité en RAM et CPU du nœud où ils se trouvent.

> Modifier la définition YAML pour allouer 2 Go de RAM et 1 CPU à votre `Cluster`.

L'ajout de ressources se fait dans la section `spec.resources`:

```yaml
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-postgresql
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.6-standard-trixie
  instances: 2
  storage:
    size: 10Gi
  walStorage:
    size: 5Gi
  resources:
    requests:
      memory: 2Gi
      cpu: 1
```

Comme vous le voyez, la définition est faite dans l'objet `Cluster`. Cela veut dire que les ressources attribuées seront les mêmes que ce soit pour l'instance primaire et l'instance secondaire.

> Appliquer la modification avec la commande et regarder ce qu'il se passe du côté des `Pods` :

```
kubectl apply -f cluster.yaml
```

Vous aurez remarqué que les `Pods` sont redémarrées pour que soient pris en compte les ressources demandées. C'est une limitation actuelle, qui devrait disparaître grâce à la dernière version de Kubernetes (1.35) qui permet la modification à chaud des ressources.

Les paramètres de `requests` permettent d'indiquer les quantités minimales de RAM et de CPU qui doivent être disponibles sur un nœud pour qu'elles soient allouées au `Pod`. Si de telles quantités ne sont pas disponibles alors les `Pods` ne seront tout simplement pas déployés. Vous pouvez faire le test !

> Rajoutons une section `limits`. Modifier la définition YAML pour limiter à 2 Go la RAM et à 1 CPU votre `Cluster`.

```yaml
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-postgresql
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.6-standard-trixie
  instances: 2
  storage:
    size: 10Gi
  walStorage:
    size: 5Gi
  resources:
    requests:
      memory: 2Gi
      cpu: 1
    limits:
      memory: 2Gi
      cpu: 1
```

> Appliquer la modification avec la commande :

```
kubectl apply -f cluster.yaml
```

Avec `limits` vous limiter les ressources RAM et CPU que vos `Pods` peuvent consommer sur le nœud. Comprenez que dans notre cas, il s'agit de la consommation de l'instance (i.e du `postmaster` et des `background process`) mais aussi de tous les `backend` (i.e clients) qui sont créés lorsque qu'une nouvelle session est créée sur l'instance.

> Si vous relancez la commande pour trouver les resources d'un Pod, le résultat aura changé.

```
kubectl get pod cluster-postgresql-1 -o json | jq '.spec.containers[].resources'
```

```
{
  "limits": {
    "cpu": "1",
    "memory": "2Gi"
  },
  "requests": {
    "cpu": "1",
    "memory": "2Gi"
  }
}
```

Cette dernière configuration, où les `limits` sont égales aux `requests`, permet d'attribuer la QoS `Guaranteed` à vos `Pods` PostgreSQL.

C'est une bonne pratique, car en cas de pression sur les nœuds Kubernetes, les `Pods` ayant une QoS `Guaranteed` seront les derniers à être évincés.

4.4 CONFIGURATION POSTGRESQL

But : Modifier des paramètres basiques de PostgreSQL

Notre `Cluster` est démarré et tout semble fonctionner pour le mieux. Nous avons configuré la RAM et le CPU utilisables par nos `Pods`. Attardons-nous maintenant à la configuration de PostgreSQL.

En tant que DBA PostgreSQL, je veux par exemple modifier le paramètre `shared_buffers` et le passer à 4 Go. La commande pour faire cela est la suivante :

```
ALTER SYSTEM SET shared_buffers = '4GB';
```

> Se connecter à l'instance et modifier le paramètre `shared_buffers`.

```
kubectl cnpg psql cluster-postgresql
```

```
psql (17.6 (Debian 17.6-2.pgdg13+1))
Type "help" for help.

postgres=# ALTER SYSTEM SET shared_buffers = '4GB';
ERROR:  ALTER SYSTEM is not allowed in this environment
```

Ce qui est normal à vrai dire. Le paramètre `show_alter_system` (disponible depuis la version 17 de PostgreSQL) empêche cela si il est passé à `off`.

> Vérifier la valeur du paramètre `allow_alter_system`.

```
postgres=# show allow_alter_system ;
 allow_alter_system
--------------------
 off
(1 row)
```

Il est bien à `off`.

Par défaut, l'opérateur CloudNativePG modifie certains paramètres de PostgreSQL et en fixe d'autres. Voir la documentation à ce sujet https://cloudnative-pg.io/docs/1.28/postgresql_conf#fixed-parameters.

Si `ALTER SYSTEM` n'est plus possible comment faire ?

Tout doit désormais se faire dans le fichier YAML et plus précisément dans la section `postgresql.parameters` de notre objet `Cluster`. Fini le temps où tout se faisait directement sur l'instance ou dans le fichiers de configuration.

D'ailleurs, il sera fort probable que vous utilisiez des outils comme ArgoCD, Jenkins ou autre pour déployer des ressources dans Kubernetes.

```
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-postgresql
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.6-standard-trixie
  instances: 2
  storage:
    size: 10Gi
  walStorage:
    size: 5Gi
  resources:
    requests:
      memory: 2Gi
      cpu: 1
    limits:
      memory: 2Gi
      cpu: 1
  postgresql:
    parameters:
      shared_buffers: 4GB
```

> Appliquer cette nouvelle définition et observer… un beau message d'erreur.

```
kubectl apply -f cluster.yaml
```

```
Kubectl command failed: The Cluster "cluster-postgresql" is invalid:
↪  spec.resources.requests: Invalid value: "2Gi": Memory request is lower than
↪  PostgreSQL `shared_buffers` value
```

Certaines vérifications ont été mises en place dans le code de l'opérateur pour éviter de vous retrouver avec des configurations incohérentes.

Reprenons la définition de notre `Cluster` et renseigner une valeur cohérente pour le paramètre `shared_buffers`.

```
  postgresql:
    parameters:
      shared_buffers: 512MB
```

> Appliquer cette nouvelle définition et observer que la prise en compte du paramètre déclenche un redémarrage des instances du `Cluster`

```
kubectl apply -f cluster.yaml
```

Par défaut l'opérateur CloudNativePG va redémarrer (ou recharger la configuration selon le paramètre) automatiquement les instances pour la prise en compte des nouvelles valeurs. Cette mise à jour peut se faire automatiquement ou de manière supervisée selon la valeur du paramètre `primaryUpdateStrategy` qui peut prendre deux valeurs :

- `unsupervised` : tout est fait automatiquement. C'est la valeur par défaut ;
- `supervised` : une une opération manuelle est demandée.

En mode `unsupervised`, le paramètre `primaryUpdateMethod` est pris en compte pour savoir comment procéder.

- `restart` : l'instance primaire est redémarrée. C'est la valeur par défaut ;
- `switchover` : une bascule sur un secondaire est effectuée et le primaire devient secondaire.

> Modifier le paramètre `work_mem` en le passant à 8 Mo. Observer que les instance sont rechargées mais pas redémarrées.

```
  parameters:
    shared_buffers: 512MB
    work_mem: 8MB
```

```
kubectl apply -f cluster.yaml
```

Vous pouvez vérifier que les instances ont été rechargées en regardant les traces d'une instance.

```
kubectl logs -f cluster-postgresql-1
```

Vous devriez trouver une ligne avec le message `"msg": "Requesting configuration reload"`.

4.5 CRÉER UNE BASE DE DONNÉES

But : Créer une base de données dans nos instances.

> Créer une base de données db dans le `Cluster` `cluster-postgresql`. Le propriétaire de cette base doit être le rôle `app`. Pour cela, créer un fichier `db.yaml` contenant la définition d'une ressource `Database`.

Voici la déclaration de la base de données.

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: db
spec:
  name: db
  owner: app
  ensure: present
  cluster:
    name: cluster-postgresql
```

> Créer la nouvelle ressource avec `kubectl apply -f ~/db.yaml`.

```
kubectl apply -f db.yaml
database.postgresql.cnpg.io/db created
```

> Vérifier la présence de cette base de données dans l'instance.

Plusieurs possibilités. En voici une :

```
kubectl cnpg psql cluster-postgresql -- -c "\l"
```

```
                                                    List of databases
   Name    |  Owner   | Encoding | Locale Provider | Collate | Ctype | Locale | ICU
↪  Rules |   Access privileges
-----------+----------+----------+-----------------+---------+-------+--------+-----
↪  ------+-----------------------
 app       | app      | UTF8     | libc            | C       | C     |        |
 db        | app      | UTF8     | libc            | C       | C     |        |
 postgres  | postgres | UTF8     | libc            | C       | C     |        |
 template0 | postgres | UTF8     | libc            | C       | C     |        |
↪  | =c/postgres          +
```

```
          |          |          |                 |         |       |        |
↪  | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | C       | C     |        |
↪  | =c/postgres          +
          |          |          |                 |         |       |        |
↪  | postgres=CTc/postgres
(5 rows)
```

4.6 SUPERVISER - AFFICHER ET AJOUTER DES MÉTRIQUES

Buts : Afficher et consulter les métriques de base puis ajouter une métrique personnalisée (fonctionnalité expérimentale).

Un *exporter* compatible Prometheus est présent dans les `Pods` de nos instances PostgreSQL. Il est accessible sur le port `9187` de chaque Pod.

Forwarder le port `9187` localement avec la commande :

```
kubectl port-forward pod/cluster-postgresql-1  9187:9187 &
[1] 433043
Forwarding from 127.0.0.1:9187 -> 9187
Forwarding from [::1]:9187 -> 9187
```

Récupérer la liste de toutes les métriques avec l'outil `curl` :

```
curl http://127.0.0.1:9187/metrics
```

Ce *endpoint* est donc compatible avec Prometheus. Vous pouvez donc intégrer la surveillance de vos instances avec votre *stack* Prometheus et, par exemple, Grafana pour obtenir des *dashboards*. Par exemple :

Figure 4/ .1: Dashboard Grafana

Regardons maintenant comment ajouter une métrique personnalisée.

Pour cela, il est nécessaire de créer une ressource`ConfigMap` qui sera ensuite référencée dans la configuration de notre `Cluster` PostgreSQL.

Pour notre TP, on propose d'ajouter une simulateur de lancé de dés :

```
SELECT floor(random() * 6 + 1)::int  AS roll;
```

> Créer le fichier `configmap.yaml` avec le contenu suivant :

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-monitoring
  labels:
    cnpg.io/reload: ""
data:
  custom-queries: |
    dice:
      query: "SELECT floor(random() * 6 + 1)::int AS roll;"
      metrics:
        - roll:
            usage: "GAUGE"
            description: "rolling dice"
```

> Créer la ressource `ConfigMap` :

```
kubectl apply -f configmap.yaml
```

> Modifier la définition du `Cluster` dans le fichier `cluster.yaml` en rajoutant la partie `spec.monitoring:`

```
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-postgresql
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.6-standard-trixie
  instances: 2
  storage:
    size: 10Gi
  walStorage:
    size: 5Gi
  resources:
    requests:
      memory: 2Gi
```

```
      cpu: 1
    limits:
      memory: 2Gi
      cpu: 1
  postgresql:
    parameters:
      shared_buffers: 512MB
  monitoring:
    customQueriesConfigMap:
      - name: example-monitoring # on référence l'objet ConfigMap créé un peu avant
        key: custom-queries
```

```
kubectl apply -f cluster.yaml
```

> Récupérer plusieurs fois la valeur de cette nouvelle métrique. Qu'observez vous ?

```
watch "curl http://127.0.0.1:9187/metrics 2>/dev/null | grep dice_roll"
```

Comme l'indique la documentation, il y a un mécanisme de cache pour les requêtes. C'est une nouveauté de la version 1.28.0 de l'opérateur.

> By default, the outputs of monitoring queries are cached for thirty seconds. This is done to enhance resource efficiency and to avoid PostgreSQL to run monitoring queries every time the prometheus endpoint is scraped.
>
> The cache itself can be observed by the cache_hits, cache_misses and last_update_timestamp metrics.

4.7 CONSULTER ET UTILISER LES TRACES

But : Récupérer les traces d'un Pod, activer pgAudit et modifier le niveau de trace.

Consulter les journaux du Pod de votre instance primaire.

La commande `kubectl get cluster` vous indique quel est l'instance primaire dans la colonne `PRIMARY`.

```
kubectl logs cluster-postgresql-1
```

```
{"level":"info","ts":"2026-01-
↪ 08T16:08:02.282580934Z","logger":"postgres","msg":"record","logging_pod":"cluster-
↪ postgresql-1","record":{"log_time":"2026-01-08 16:08:02.282
↪ UTC","process_id":"32","session_id":"695fd534.20","session_line_num":"2","session_start_time":
↪ 01-08 16:03:00
↪ UTC","transaction_id":"0","error_severity":"LOG","sql_state_code":"00000","message":"checkpoin
↪ complete: wrote 5 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled;
↪ write=0.576 s, sync=0.478 s, total=1.383 s; sync files=4, longest=0.461 s,
↪ average=0.120 s; distance=5 kB, estimate=5 kB; lsn=0/D000098, redo
↪ lsn=0/C0015F8","backend_type":"checkpointer","query_id":"0"}}
```

Assez verbeux tout cela … et pas franchement lisible.

Pour faciliter la lisibilité des traces, l'utilitaire `jq` s'avère utile

```
kubectl logs cluster-postgresql-1 | jq
```

Pour faciliter la lisibilité des traces, la commande `cnpg logs pretty` (du *plugin* `cnpg` pour `kubectl`) est également utile :

```
kubectl logs cluster-postgresql-1 | kubectl cnpg logs pretty
```

Consulter les journaux de la ressource `Cluster` entière (i.e. de toutes les instances déployées) :

```
kubectl cnpg logs cluster cluster-postgresql | kubectl cnpg logs pretty
```

4.8 AJOUTER L'EXTENSION PGAUDIT

But : Récupérer les traces d'un Pod, activer pgAudit et modifier le niveau de trace.

Certaines extensions sont directement disponibles car embarquées dans les images proposées par CloudNativePG. C'est le cas de pgAudit.

Avant de l'activer pgAudit, regardons le paramètre `shared_preload_libraries`.

```
kubectl cnpg psql cluster-postgresql
psql (18.1 (Debian 18.1-1.pgdg13+2))
Type "help" for help.

postgres=# show shared_preload_libraries;
 shared_preload_libraries
--------------------------

(1 row)
```

Ajouter cette extension dans l'instance avec un `ALTER SYSTEM ...`

```
app=# ALTER SYSTEM SET shared_preload_libraries = 'auto_explain';
ERROR:  ALTER SYSTEM is not allowed in this environment
```

Comme vous le voyez, il n'est pas possible d'utiliser ce genre de commande. Pourquoi ?

Regarder la valeur du paramaètre `allow_alter_system`.

```
postgres=# show allow_alter_system;
 allow_alter_system
--------------------
 off
(1 row)
```

Ce paramètre interdit l'utilisation de `ALTER SYSTEM`. La modification de la configuration PostgreSQL doit donc se faire dans le fichier YAML.

Modifier le manifeste du `Cluster` pour ajouter pgAudit. Il suffit pour cela d'ajouter des paramètres de configuration pgAudit dans la `section postgresql.parameters` :

```
[...]
      parameters:
        pgaudit.log: "all, -misc"
        pgaudit.log_catalog: "off"
        pgaudit.log_parameter: "on"
        pgaudit.log_relation: "on"
```

Un redémarrage des instances est nécessaire car une entrée dans `shared_preload_libraries` a été faite. L'opérateur le fait automatiquement pour nous.

```
kubectl cnpg psql cluster-postgresql
psql (18.1 (Debian 18.1-1.pgdg13+2))
Type "help" for help.

postgres=# show shared_preload_libraries;
 shared_preload_libraries
--------------------------
pgaudit

(1 row)
```

> Pour finir, nous pouvons consulter à nouveau les journaux et observer la différence.

```
kubectl logs cluster-postgresql-1 | kubectl cnpg logs pretty

{"level":"info","ts":"2026-01-08T14:18:02.533653068Z","logger":"pgaudit",....
```

Il est possible de modifier le niveau de trace d'un `Cluster` d'instances. Par défaut, c'est le niveau `info` qui est utilisé. Ici, nous allons passer le niveau de `info` à `trace` :

> Modifier le niveau de trace avec le commande `kubectl patch` suivante :

```
kubectl patch cluster cluster-postgresql \
--type=merge \
-p '{
  "spec": {
    "logLevel": "trace"
  }
}'
```

Cette opération déclenche une modification dans les `Pods` et nécessite la recréation de ceux-ci. On peut la voir avec la commande suivante.

```
kubectl get pod cluster-postgresql-1 --output=json \
  | jq '.spec.containers[] | {name: .name, command: .command}'

{
  "name": "postgres",
  "command": [
    "/controller/manager",
    "instance",
    "run",
    "--status-port-tls",
    "--log-level=trace"
  ]
}
```

L'option `--log-level=trace` a été ajouté à la `command` du Pod. D'où le redémarrage.

4.9 MISE À JOUR MINEURE

But : Mettre à jour notre `Cluster` PostgreSQL en version 17.7.

La mise à jour se fait automatiquement en mode *Rolling Update* par l'opérateur lorsque le tag de l'image est modifié. Les secondaires sont mis à jour puis l'instance primaire, si tout s'est bien déroulé.

Modifier le tag de l'image et appliquer la nouvelle définition de la ressource `Cluster`.

```
  imageName: ghcr.io/cloudnative-pg/postgresql:17.7-standard-trixie
```

```
kubectl apply -f cluster.yaml
```

Une mise à jour mineure revient à télécharger la nouvelle image de conteneur et redémarrer les instances avec cette nouvelle image.

Une reconnexion des applications sera nécessaire comme l'instance primaire sera redémarrée.

4.10 MISE À JOUR MAJEURE

But : Mettre à jour notre `Cluster` PostgreSQL en version 18.1.

Nous venons de voir comment faire une mise à jour mineure. Regardons comment faire une mise à jour majeure. Il est là aussi nécessaire de modifier le tag utilisé dans l'image.

Modifier le tag de l'image et appliquer la nouvelle définition de la ressource `Cluster`.

```
  imageName: ghcr.io/cloudnative-pg/postgresql:18.1-standard-trixie
```

```
kubectl apply -f cluster.yaml
```

À la différence d'une montée de version mineure, lors d'une montée de version majeure, tous les `Pod`s sont supprimés (mais pas les volumes). Un `Pod` dédié à la montée de version est créé (par exemple : `cluster-postgresql-2-major-upgrade-fbtzp`) et est en charge du passage de l'une à l'autre des versions. Lors de cette étape, l'outil `pg_upgrade` est notamment utilisé (avec l'option `--link` pour les connaisseurs).

Lorsque l'instance primaire a fini d'être recréée, la ou les instances secondaires sont à leur tour re-créés.

Une mise à jour majeure revient à télécharger la nouvelle image de conteneur, à recréer entièrement une instance et à recréer les instances secondaires à partir de celle-ci.

5/ Conclusion

- Un opérateur complet
- Le grand favoris des opérateurs
- Changement de paradigme pour les DBAs

6/ Conclusion

Plugin de sauvegarde pgBackRest :
https://github.com/dalibo/cnpg-plugin-pgbackrest
Articles de blog :

7/ Références

- CloudNativePG : https://cloudnative-pg.io
- Dalibo : https://dalibo.com

8/ Remerciements

- Alexandre Pereira pour le déploiement de l’infra et ses tests

Notes

Notes

Notes

Nos autres publications

FORMATIONS

- 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
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

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.

9/ DALIBO, L’Expertise PostgreSQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l’animation de la communauté francophone de PostgreSQL. La société est également à l’origine de nombreux outils libres de supervision, de migration, de sauvegarde et d’optimisation.

Le succès de PostgreSQL démontre que la transparence, l’ouverture et l’auto-gestion sont à la fois une source d’innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.