Przetwarzanie rozproszone w SZBD Postgres

Celem zajęć jest zapoznanie się z własnościami SZBD Postgres umożliwiającymi przetwarzanie rozproszone. Własności te obejmują:

- Transparenty dostęp do zdalnych danych.
- Rozproszone transakcje.
- Obsługa rozproszonych zakleszczeń.
- Podstawowa replikacja danych.

1. Przygotowanie środowiska

Do ilustracji przetwarzania rozproszonego będziemy korzystać ze środowiska Kubernetes (w skrócie K8s). Środowisko to służy do orkiestracji skonteneryzowanych, rozproszonych aplikacji o architekturze mikro-usługowej. Głównymi zadaniami orkiestracji są: wdrożenie skonteneryzowanych składników aplikacji, dynamiczne skalowanie aplikacji na żądanie, automatyczne odtwarzanie składników aplikacji, uaktualnianie i wycofywanie aktualizacji bezstanowych składników aplikacji z zachowaniem ciągłości ich pracy. Konteneryzacja aplikacji polega na uruchomieniu jej w kontenerach, które są lekką odmianą mechanizmu wirtualizacji umożliwiającą separację różnych składników aplikacji i kontrolę przydziału do nich zasobów (pamięć, procesor, dysk, sieć). Najpopularniejszym środowiskiem konteneryzacji jest Docker. Architektura mikro-usługowa jest sposobem do projektowania dużych systemów aplikacji, które wykorzystuje podejście podziału systemu na wiele składników, nazywanych mikro-usługami, komunikujących się ze sobą najczęściej asynchronicznie z wykorzystaniem systemów kolejkowych (np. Kafka, RabbitMQ), które w celu uzyskania skalowalności rozprasza się na wiele maszyn. W ramach zajęć skupimy się składnikach aplikacji, które służą do trwałego składowania rozproszonych danych, czyli na rozproszonych bazach danych.

Kubernetes jest otwarto-źródłowym systemem rozwijanym przez Google. Jego usługi są oferowane przez największych dostawców rozwiązań chmurowych, np.: Google Cloud, Amazon AWS, Microsoft Azure, Alibaba Cloud oraz przez mniejszych dostawców np. Hosted Rancher. Kubernetes można też wdrożyć na własnych serwerach. W celu uniezależnienia się od infrastruktury chmurowej i sprzętowej wykorzystamy jedną z dystrybucji deweloperskich Kubernetes o nazwie *k3d*, którą można uruchomić na sprzęcie klasy desktop. *K3d* w odróżnieniu od innych dystrybucji deweloperskich Kubernetes np. *k3s* lub *Minikube* symuluje wiele węzłów klastra co umożliwia zwiększenie realizmu. Pliki manifestów opisujące sposób wdrożenia aplikacji przygotowane dla *k3d* po nieznacznych uzupełnieniach można wykorzystać w realnym środowisku chmurowym. *K3d* ma architekturę matrioszki, jest uruchamiany jako zbiór kontenerów Docker, każdy węzeł klastra *k3d* posiada własny kontener. Wewnątrz tych kontenerów są uruchamiane kontenery zawierające *k3s*, z kolei w tych kontenerach są uruchamiane kontenery udostępniające funkcjonalność Kubernetes i kontenery użytkownika.

Przygotowanie środowiska rozpoczniemy od zainstalowania Docker i *k3d*.

1. Zaloguj się do maszyny wirtualnej jako użytkownik rbd używając hasła RBD#7102.
2. Otwórz okno terminala, który nazwiemy terminalem pomocniczym.
3. Zainstaluj Docker wykonując w terminalu pomocniczym poniższe polecenia:
   ```
   sudo yum install -y yum-utils
   sudo yum-config-manager \
       --add-repo \
       https://download.docker.com/linux/centos/docker-ce.repo
   sudo yum -y install docker-ce-20.10.8 docker-ce-cli-20.10.8 containerd.io
   ```
4. Uruchom Docker oraz skonfiguruj jego automatyczne uruchamianie wraz ze startem systemu operacyjnego. Wykorzystaj w tym celu poniższe polecenia.
   ```
   sudo systemctl start docker
   sudo systemctl enable docker
   ```
5. Dodaj użytkownika rbd do grupy docker wykonując polecenie:
   ```
   sudo usermod -aG docker rbd
   ```
6. Wyloguj się i zaloguj ponownie w celu zaaplikowania zmian wprowadzonych w poprzednim punkcie. Uruchom ponownie terminal pomocniczy.
7. Zainstaluj k3d wykonując w terminalu pomocniczym poniższe polecenie:

```
curl -s https://raw.githubusercontent.com/rancher/k3d/main/install.sh | TAG=v4.4.8 bash
```

8. Skonfiguruj repozytorium Kubernetes, wykonaj w tym celu poniższe polecenie:

```
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo >/dev/null
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.29/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.29/rpm/repodata/repomd.xml.key
EOF
```

9. Zainstaluj narzędzie kubectl, które umożliwi zarządzanie Kubernetes działającym w *k3d*. W tym celu uruchom poniższe polecenie:
   ```
   sudo yum install -y kubectl
   ```
10. W kolejnym kroku utwórz klaster węzłów *k3d*, wykorzystaj poniższe polecenie:
    ```
    k3d cluster create RBDcluster --servers 1 --agents 2  --image
    rancher/k3s:v1.22.2-k3s1 \
            --port "5432-5433:5432-5433@loadbalancer"
    ```
    RBDcluster jest nazwą klastra. Klaster Kubernetes składa się z dwóch rodzajów węzłów: *control plane* (*server* w nazewnictwie *k3d*), *worker node* (*agent* w nazewnictwie *k3d*). Agenty posiadają infrastrukturę przeznaczoną do uruchamiania kontenerów użytkownika. Zwiększanie liczby agentów głównie służy do zwiększanie skalowalności klastra przez rozpraszanie ich na wiele maszyn. Serwery zlecają zadania agentom, wykrywają i reagują

na zdarzenia w klastrze (np. awarię agenta) oraz przechowują metadane opisujące stan klastra. Zwiększanie liczby serwerów służy głównie do zwiększania niezawodności działania funkcji systemowych klastra. Dodatkowym elementem klastra w dystrybucji *k3d* jest węzeł równoważenia obciążenia (*load balancer*), który zazwyczaj jest elementem infrastruktury dostawcy usług chmurowych. Węzeł ten odpowiedzialny za równomierne rozpraszanie żądań połączeń do replik usług użytkownika. Przełącznik *servers* służy do określenia liczby serwerów, przełącznik *agents* określa liczbę agentów, przełącznik *image* wskazuje na obraz kontenera zawierającego komponent *k3s*, przełącznik *port* umożliwia przekazywanie portów (ang. port forwarding) z węzłów klastra na adresy maszyny, na której jest uruchomiony *k3d*. W naszym przypadku przekazywane są 2 porty, 5432 i 5433 z węzła loadbalancer. Zabieg ten umożliwia wykorzystanie adresu *localhost* do komunikacji z loadbalancer. Chociaż adres loadbalancer jest dostępny na maszynie gospodarza, to jego wartość nie jest z góry znana i jest dynamicznie ustalana podczas tworzenia klastra. Na potrzeby ćwiczeń łatwiej będzie posługiwać się adresem *localhost* maszyny gospodarza.

11. Uruchom poniższe polecenie aby sprawdzić jakie klastry zostały uruchomione w *k3d*:
    `k3d cluster list`
12. Wyświetl listę węzłów klastrów *k3d*, uruchom poniższe polecenie;
    `k3d node list`
13. Teraz przystąpimy do uruchomienia w klastrze pierwszej bazy danych Postgres. W pierwszym kroku pobierz plik manifestów za pomocą poniższego polecenia:
    `wget www.cs.put.poznan.pl/jjezierski/RBDv2/rbd1.yaml`
14. Otwórz plik manifestów w celu jego przeglądnięcia za pomocą polecenia:
    `less rbd1.yaml`
    Plik manifestów jest opisany za pomocą języka YAML. Zwiera on deklarowany opis komponentów aplikacji wdrażanej w Kubernetes. W uproszeniu, język YAML jest hierarchicznym zestawem par klucz-wartość zwanych węzłami. Wartością może być skalar (tekst, liczba, wartość logiczna), uporządkowana sekwencja węzłów lub nieuporządkowany zbiór węzłów zwany mapą. Hierarchia węzłów jest wyznaczana przez wcięcia wierszy, analogicznie jak w języku Python. Elementy sekwencji zaznacza się znakiem myślnika. Plik rbd1.yaml zawiera 2 manifesty oddzielone trzema myślnikami i rozpoczynające się od klucza *apiVersion*, który określa wersję API Kubernetes wg której zastał zapisany manifest.
    Podstawową jednostką wdrożenia i skalowania składnika aplikacji w Kubernetes jest *Pod*. Pod zawiera najczęściej jeden kontener. W pojedynczym Pod można umieścić wiele kontenerów jeżeli współdzielą ten sam zasób (np.: dysk), w takich sytuacjach często kontener udostępniający główną usługę jest wspomagany przez inne kontenery, które pełnią rolę pomocniczą, np.: stanowią adapter z innymi usługami lub inicjują główny kontener. Istnieje możliwość bezpośredniego zainstalowania Pod w klastrze, jednakże tak zainstalowany Pod nie jest kontrolowany przez

Control Plane i jest pozbawiony funkcji automatycznego odtwarzania, skalowania oraz wersjonowanego aktualizowania i wycofywania zmian. W celu wyposażenia Pod w wyżej wymienione własności należy wdrożyć Pod za pomocą kontrolera. Kubernetes oferuje 3 rodzaje kontrolerów: *Deployment*, *StatefulSet* oraz *DeamonSet.* Pod wdrożony za pomocą *Deployment* jest całkowicie bezstanowy. W wyniku jego aktualizacji, skalowania, odtwarzania lub restartu Kubernetes może go fizycznie usunąć i utworzyć jego nową instancję. Jest to podejście bardzo elastyczne zapewniające dużą skalowalność i niezawodność, jednakże nie nadaje się dla składników aplikacji, które posiadają własne dane zwłaszcza dla baz danych. Do składowania danych istnieje możliwość zamontowania trwałego woluminu (PersistentVolume) w Pod przez kontroler Deployment, jednakże wszystkie repliki Poda współdzieliłyby ten wolumin co nie jest dozwolone dla baz danych typu *share nothing*, którego przedstawicielem jest Postgres. Kontroler *DeamonSet* uruchamia po jednej replice Pod na każdym węźle klastra. Ten kontroler najczęściej jest wykorzystywany dla usług kolekcjonujących dzienniki oraz monitorujących inne Pod. Wszystkie Pod kontrolowane za pomocą DeamonSet również współdzielą ten sam trwały wolumin i z tego powody ten rodzaj kontrolera nie jest przydatny do naszych celów. Kontroler *StatefulSet* jest przeznaczony do sterowania stanowymi Pod i z tego względu wykorzystuje się go uruchamiania baz danych i z tego powodu wykorzystamy go w dalszej części tutorialu.
Wracając do pliku manifestów. Pierwszy manifest opisuje Pod kontrolowany za pomocą StatefulSet zawiera jeden kontener z obrazem systemu Postgres. Klucz *kind* wskazuje na rodzaj użytego kontrolera, w tym przypadku StatefulSet. Klucz *metadata.name* został użyty do nazwania tego kontrolera jako pgsql-rbd1, nazwa ta musi być poprawną nazwą DNS ponieważ wchodzi w skład nazw replik Pod, które są rejestrowane w wewnętrznej usłudze klastra Kubernetes. Wszystkie węzły występujące bezpośrednio pod kluczem *spec* odnoszą się do specyfikacji kontrolera. Wartość klucza *spec.template* jest szablonem replik Pod, który jest wykorzystywany przez kontroler do tworzenia tych replik. W naszym przypadku szablon definiuje Pod składający się z jednego kontenera o nazwie *pgsql-rbd1*, który ma być utworzony z wykorzystaniem obrazu z repozytorium Docker o nazwie postgres:13.4. Każdy kontener ma zamontować trwały wolumin, którego nazwa rozpoczyna się od frazy *pgsql-rbd1-disk* w katalogu */data*. Definicje szablonów woluminów opisane są za pomocą klucza volumeClaimTemplates, który umożliwia utworzenie woluminu dla każdej repliki Pod. Klucz volumeClaimTemplates.spec określa między innymi tryb dostępu do woluminu oraz jego rozmiar. Klucz *env* umożliwia przekazanie do kontenera wartości zmiennych środowiskowych, które są zazwyczaj wykorzystywane przez kontener do swojej inicjalizacji. W naszym przypadku przekazano wartości zmiennych, które określają hasło użytkownika *postgres* oraz lokalizację katalogu zawierającego konfigurację i dane bazy danych. Zauważ, że katalog ten znajduje się na zamontowanym trwałym woluminie. Klucz *template.spec.node.selector* umożliwia określenie kryterium wyboru węzłów klastra, na których mają

zostać uruchomione repliki *Pod*. W naszym przypadku posłużymy się nazwą hosta, aby uruchomić Pod na węźle k3d-rbdcluster-agent-0. Przypisanie Pod do węzłów klastra jest opcjonalne, rozpraszanie replik Pod w celu równoważenia obciążenia jest automatyczne.
Klucz spec.replicas wskazuje na liczbę żądanych replik Pod. W naszym przypadku wykorzystamy jedną replikę. Klucz spec.selector jest listą etykiet, którą muszą posiadać Pod aby kontroler mógł nimi zarządzać. Zauważ, że wartość klucza *spec.selector.matchLabels* (app: pgsql-rbd1) pasuje do wartości klucza *spec.template.metadata.labels*.
Repliki Pod, które zastaną utworzone przez kontroler mają dynamiczne adresy IP, które mogą się zmieniać przy skalowaniu lub odtwarzaniu Pod, dodatkowo adresy te działają w wewnętrznej sieci klastra. Z tego powodu trzeba utworzyć dodatkową usługę, która będzie miała stałe zewnętrzne IP za pośrednictwem której będzie można komunikować się z replikami Pod. W tym celu można wykorzystać *LoadBalancer*. Zadaniem tej usługi jest przekierowywanie połączeń do replik Pod w taki sposób aby równoważyć ich obciążenie.
Drugi manifest w pliku manifestów opisuje usługę *LoadBalancer*. Wartość klucza *kind* wskazuje, że manifest dotyczy usługi. Klucz *metadata.name* umożliwia nazwanie usługi. Węzeł *spec* określa specyfikację usługi. Klucz *spec.type* umożliwia wskazanie, że nasza usługa ma być typu *LoadBalancer*. Klucz *spec.ports* specyfikuje odwzorowanie portów, w naszym przypadku port 5432, na którym nasłuchuje baza danych Postgres uruchomiona w replikach Pod ma być odwzorowana w ten sam port adresów usługi *LoadBalancer*. Klucz *spec.selector* wskazuje etykiety Pod, dla których usługa *LoadBalancer* będzie wykonywać swoje zadanie.
Opuść program *less* wybierając przycisk *q*.

15. Rozpocznij wdrożenie komponentów z pliku manifestów za pomocą polecenia:
    `kubectl apply -f rbd1.yaml`
16. Obserwuj postęp wdrożenia *StatefulSet* wykorzystując poniższe polecenie:
    `kubectl get sts --watch`
    Wdrożenie wymaga pobranie obrazu kontenera z repozytorium Docker w związku z tym zajmuje chwilę. Wdrożenie zakończy w momencie pojawienia się na terminalu wiersza, w którym liczba działających replik będzie równa liczbie żądanych replik, np.:
    ```
    pgsql-rbd1    1/1     1m
    ```
    W tym momencie przerwij wykonanie polecenia wykorzystując kombinację Ctrl-c.
17. Teraz przygotujemy drugą bazę danych. Skopiuj plik manifestów do pliku rbd2.yaml:
    `cp rbd1.yaml rbd2.yaml`
18. Użyj swojego ulubionego edytora tekstu wykonać następujące zmiany w pliku rbd2.yaml [Raport]:
    - zamień wszystkie wystąpienia tekstu rbd1 na rbd2,
    - zamień nazwę hosta k3d-rbdcluster-agent-0 na k3d-rbdcluster-agent-1,
    - ustal w drugim manifeście wartość klucza spec.ports.port na 5433.

19. Wykonaj wdrożenie zawartości pliku rbd2.yaml [Raport].
20. Zainstaluj klienta Postgres wykonując następujące polecenia:

```
sudo yum install -y \
https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm
sudo yum -y install postgresql14-14.0
```

21. Otwórz dwa kolejne okna lub zakładki terminala, w pierwszym ustaw znak zachęty za pomocą polecenia `export PS1='[\u@rbd1 \W]\$ '`. W drugim terminalu ustaw znak zachęty za pomocą polecenia `export PS1='[\u@rbd2 \W]\$ '`. Pierwszy terminal o nazwie *rbd1* będzie służył do operowania na bazie danych *RBD1*, natomiast drugi terminal o nazwie *rbd2* będzie wykorzystywany do wykonywania operacji na bazie danych *RBD2*.

2. Transparentny dostęp do zdalnych danych

Celem tego punktu jest zaprezentowanie mechanizmów zdalnego serwera, odwzorowania użytkownika oraz importowania schematu, które umożliwiają transparenty dostęp do rozproszonych danych. Mechanizmy te są implementacją standardu ISO/IEC 9075-9 (SQL/MED).

1. W terminalu *rbd1* uruchom narzędzie *psql* logując się jako użytkownik *postgres* do bazy danych *RBD1*. Wykorzystaj polecenie
   `psql -U postgres -h localhost -p 5432 postgres`
   Użyj hasła rbd1 w celu uwierzytelnienia użytkownika postgres.
   Uwaga: jeżeli narzędzie psql nieoczekiwanie traci połączenie ze serwerem zamiast adresu localhost użyj jednego z adresów węzła loadbalancer. W celu pozyskania tych adresów użyj polecenia: `kubectl get svc`. Wykorzystaj jeden z adresów z
   kolumny EXTERNAL-IP.
2. W narzędziu *psql* ustaw znak zachęty dla pierwszego wiersza komendy za pomocą polecenia `\set PROMPT1 '%n@RBD1:%>%# '`.
3. W terminalu *rbd2* uruchom narzędzie *psql* logując się do bazy danych *RBD2*. Zwróć uwagę na podanie odpowiedniego numeru portu. W narzędziu *psql* ustaw odpowiedni znak zachęty.
4. W terminalu *rbd1* za pomocą narzędzia *psql* zainstaluj w bazie danych *RBD1* rozszerzenie *foreign-data wrapper* wykonując polecenie `CREATE EXTENSION postgres_fdw;`.
5. W terminalu *rbd2* w bazie danych *RBD2* również zainstaluj rozszerzenie *foreign-data wrapper*.
6. Pobierz skrypty SQL wykonując w terminalu pomocniczym polecenia:
   `wget www.cs.put.poznan.pl/jjezierski/RBDv2/pracownicy.sql`
   `wget www.cs.put.poznan.pl/jjezierski/RBDv2/zespoly.sql`

7. W bazie danych *RBD1* utwórz tabelę *pracownicy* uruchamiając skrypt w narzędziu *psql* poleceniem `\i ~/pracownicy.sql`.
8. W bazie danych *RBD2* utwórz tabelę *zespoly* uruchamiając skrypt `~/zespoly.sql`.

9. Utwórz w bazie danych *RBD1* definicję zdalnego serwera *rbd2* za pomocą poniższego polecenia:
   ```
   CREATE SERVER rbd2
   FOREIGN DATA WRAPPER postgres_fdw
   OPTIONS (host 'pgsql-rbd2-lb', port '5433', dbname 'postgres');
   ```
10. Utwórz w bazie danych *RBD1* odwzorowanie zdalnego użytkownika *postgres* serwera *rbd2*, wykorzystaj poniższe polecenie:
    ```
    CREATE USER MAPPING FOR postgres SERVER rbd2
    OPTIONS (user 'postgres', password 'rbd2');
    ```
11. Zaimportuj do bazy danych *RBD1* fragment schematu *public* zawierający tabelę *zespoly* udostępniany przez serwer *rbd2*, wykorzystaj poniższe polecenie:
    ```
    IMPORT FOREIGN SCHEMA public LIMIT TO (zespoly)
    FROM SERVER rbd2 INTO public;
    ```
12. W terminalu *rbd1* wykonaj poniższe polecenia testujące zdalny dostęp do tabeli *zespoly*:
    ```
    SELECT * FROM zespoly;
    SELECT nazwisko, nazwa FROM pracownicy natural join zespoly;
    ```
13. Utwórz w bazie danych RBD2 odpowiednie obiekty umożliwiające zdalny dostęp do tabeli pracownicy znajdującej się w bazie danych RBD1. [Raport]

3. Rozproszone transakcje

Celem zadania jest przedstawienie rozproszonych transakcji, czyli takich transakcji, które modyfikują dane w więcej niż w jednej bazie danych. Istotnym problem do rozwiązania przez producentów rozproszonych systemów baz danych jest zapewnienie atomowości zatwierdzenia zmian wprowadzonych do różnych węzłów rozproszonej bazy danych. Problem ten rozwiązuje się wykorzystując wielofazowe algorytmy zatwierdzenia transakcji. Najpopularniejszym algorytmem jest algorytm dwufazowy o nazwie Two Phase Commit (2PC). Zapoznaj się z tym artykułem opisującym protokół 2PC. System Postgresql w pełni wspiera ten protokół.

3.1. Rozproszona transakcja zakończona sukcesem

1. W terminalu *rbd1* za pomocą narzędzia *psql* wykonaj w bazie danych RBD1 poniższą transakcję rozproszoną:
   ```
   begin;
   update zespoly set adres='PIOTROWO 1' where id_zesp=10;
   update pracownicy set placa_pod=999 where id_prac=100;
   commit;
   ```
2. Sprawdź w terminalu rbd2 stan tabel *pracownicy* i *zespoly*. Czy posiadają one zmiany wprowadzone przez transakcję z poprzedniego punkty? [Raport]

3.2. Rozproszona transakcja zakończona awarią

Niestety system Postgresql nie posiada narzędzi diagnostycznych w celu wywołania awarii podczas wykonywania operacji COMMIT. Z tego powodu będziesz się musiał wcielić w rolę koordynatora transakcji, a poszczególne części składowe rozproszonej transakcji będziesz realizował jako lokalne transakcje. Poniższy scenariusz symuluje awarię operacji COMMIT podczas fazy PREPARE.

1. W terminalu *rbd1* w narzędziu *psql* zmień wartość parametru `max_prepared_transactions` na wartość 10 w pliku konfiguracyjnym bazy danych rbd1. Wykorzystaj w tym celu poniższe polecenie:
   `ALTER SYSTEM set max_prepared_transactions =10;`
   Zaaplikowanie nowej wartość tego parametru wymaga restartu bazy danych, który wykonasz z poziomu Kubernetes w następnym kroku.
2. Otwórz nowe okno terminala, które dalej będzie nazywane pomocniczym. Zrestartuj StatefulSet pgsql-rbd1, w tym celu wykonaj w terminalu pomocniczym poniższe polecenie:
   `kubectl rollout restart sts pgsql-rbd1`
3. Powtórz kroki od 1 do 2 dla bazy rdb2.
4. W terminalu *rbd1* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD1* za pomocą polecenia: `\c postgres postgres localhost 5432`.
5. W terminalu *rbd2* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD2* za pomocą polecenia: `\c postgres postgres localhost 5433`.
6. Lokalnie w bazie danych *RBD1* rozpocznij lokalną transakcję *distributed-transaction01-rbd1* stanowiącą część transakcji rozproszonej:
   ```
   begin;
   update pracownicy set placa_pod=888 where id_prac=100;
   ```
7. Lokalnie w bazie danych *RBD2* rozpocznij lokalną transakcję *distributed-transaction01-rbd2* stanowiącą część transakcji rozproszonej:
   ```
   begin;
   update zespoly set adres='PIOTROWO 99' where id_zesp=10;
   ```
8. Jako koordynator transakcji wysyłasz do bazy danych *RBD1* polecenie wejścia w stan PREPARE, w tym celu w bazie danych *RBD1* wykonujesz polecenie:
   `PREPARE TRANSACTION 'distributed-transaction01-rbd1';`
9. Teraz zasymuluj awarię bazy danych *RBD2*, w tym celu w pomocniczym terminalu wykonaj polecenie, które zmniejsza liczbę replik StatefullSet pgsql-rbd2 to zera:
   `kubectl scale sts pgsql-rbd2 --replicas=0`
10. Jako koordynator transakcji wysyłasz do bazy danych *RBD2* polecenie wejścia w stan PREPARE, w tym celu w bazie danych RBD2 wykonujesz polecenie:
    `PREPARE TRANSACTION 'distributed-transaction01-rbd2';`
    W celu zwiększenia złożoności awarii załóżmy, że nie otrzymałeś informacji czy polecenie się powiodło - jako koordynator masz wątpliwość czy transakcja lokalna w bazie danych *RBD2* weszła w stan PREPARE czy też nie.

11. W celu jeszcze większego skomplikowania scenariusza awarii w pomocniczym terminalu zasymuluj teraz awarię bazy danych *RBD1* wykonując polecenie:
    `kubectl scale sts pgsql-rbd1 --replicas=0`
12. Teraz jako koordynator czekasz aż obie bazy danych zostaną odtworzone.
13. Wykonaj odtworzenie obu bazy danych wykonując w pomocniczym terminalu polecenia, które zwiększają liczbę replik obu StatefulSet do jeden:
    `kubectl scale sts pgsql-rbd1 --replicas=1`
    `kubectl scale sts pgsql-rbd2 --replicas=1`
14. W terminalu *rbd1* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD1* za pomocą polecenia: `\c postgres postgres localhost 5432`.
15. W bazie danych *RBD1* sprawdź status transakcji lokalnych, które weszły w stan PREPARED wydając zapytanie:
    `select * from pg_prepared_xacts;`
    Okazuje się, że transakcja lokalna `distributed-transaction01-rbd1` weszła w stan PREPARED.
16. W terminalu *rbd2* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD2* za pomocą polecenia: `\c postgres postgres localhost 5433`.
17. W bazie danych *RBD2* sprawdź status transakcji lokalnych, które weszły w stan PREPARED wydając zapytanie:
    `select * from pg_prepared_xacts;`
    Okazuje się, że transakcja lokalna `distributed-transaction01-rbd2` NIE weszła w stan PREPARED.
18. Na postawie uzyskanych informacji, jako globalnych koordynator, decydujesz o wycofaniu rozproszonej transakcji. W tym celu należy wycofać wszystkie lokalne transakcje, które weszły w stan PREPARED. Transakcja lokalna `distributed-transaction01-rbd2` została automatycznie wycofana przy odtwarzaniu bazy danych *RBD2* ponieważ nie weszła w stan PREPARE. Należy wycofać jedynie lokalną transakcję `distributed-transaction01-rbd1`. W tym celu w bazie danych RBD1 wykonaj polecenie:
    `ROLLBACK PREPARED 'distributed-transaction01-rbd1';`
    Sprawdź zawartość tabel *pracownicy* i *zespoly* aby upewnić się, że zmiany wprowadzone przez przerwaną rozproszona transakcję zostały wycofane.
19. Zrealizuj scenariusz awarii zatwierdzenia rozproszonej transakcji, w którym wszystkie lokalne transakcje weszły w stan PREPARED. [Raport]

4. Rozproszone zakleszczenie

System Postgresql, jak praktycznie wszystkie systemy zarządzania bazami danych, do synchronizacji transakcji wykorzystuje blokady transakcyjne. Celem ćwiczenia jest sprawdzenie w jaki sposób system ten radzi sobie z rozproszonym zakleszczeniem.

1. W bazie danych *RBD1* rozpocznij rozproszoną transakcję T1 za pomocą poniższych poleceń SQL:
   `begin;`
   `update pracownicy set placa_pod=placa_pod+10 where id_prac=100;`

2. W bazie danych *RBD2* rozpocznij rozproszoną transakcję T2 za pomocą poniższych poleceń SQL:
   `begin;`
   `update zespoly set adres='PIOTROWO 88' where id_zesp=10;`

3. W bazie danych *RBD1* kontynuuj rozproszoną transakcję T1 za pomocą poniższego polecenia SQL:
   `update zespoly set adres='PIOTROWO 77' where id_zesp=10;`

4. W bazie danych *RBD2* kontynuuj rozproszoną transakcję T2 za pomocą poniższego polecenia SQL:
   `update pracownicy set placa_pod=placa_pod+10 where id_prac=100;`
5. Co się stało? Poczekaj jeszcze 1 minutę. Coś się zmieniło?
6. Niestety system Postgresql nie wykrywa rozproszonych zakleszczeń.
7. W terminalu *rbd2* za pomocą klawiszy Ctr-C przerwij oczekiwanie na blokadę.
8. W terminalu *rbd1* dokończ transakcję T1 za pomocą polecenia SQL `COMMIT;`.
9. W terminalu *rbd2* spróbuj zatwierdzić transakcję T2. Co się stało?
10. W terminalu *rbd2* zrestartuj całą transakcję T2 przewidzianą dla bazy danych RBD2 i zatwierdź transakcję.
11. W celu uniknięcia zakleszczenia przed pierwszym poleceniem modyfikującym zdalny obiekt należałoby wykonać polecenie `SET LOCAL statement_timeout = 10000`, które na czas do końca transakcji ogranicza czas wykonania dalszych poleceń w transakcji do wskazanego w milisekundach czasu (w tym przypadku 10s).
12. Zrealizuj powyższe transakcje wykorzystując parametr statement_timeout w celu uniknięcia zakleszczenia. Jakie wady posiada to rozwiązanie? [Raport]

5. Prosta replikacja danych

Replikacja jest procesem powielania tych samych danych w różnych bazach danych w rozproszonym systemie informatycznym. Proces ten zwiększa niezawodność dostępu do danych oraz wydajność. Celem zadania jest zapoznanie się z prostym mechanizmem asynchronicznej replikacji typu master-slaves implementowanej za pomocą materializowanych perspektyw. Asynchroniczność replikacji oznacza, że uaktualnianie zawartości materializowanej perspektywy następuje niezależnie od transakcji modyfikującej dane źródłowe. W replikacji master-slaves dane mogą być modyfikowane jedynie w bazie danych zawierającej tabele źródłowe, repliki służą jedynie do odczytu.

1. W bazie danych RBD1 utwórz materializowaną perspektywę za pomocą poniższego polecenia:
   `CREATE MATERIALIZED VIEW zespoly_replika AS SELECT * FROM zespoly;`
2. Sprawdź zawartość materializowanej perspektywy *zespoly_replika*.
3. W bazie danych RBD2 wykonaj poniższe polecenie w celu zmiany danych źródłowych w tabeli *zespoly*.
   `update zespoly set adres='PIOTROWO 43' where id_zesp=10;`

4. Sprawdź zawartość materializowanej perspektywy *zespoly_replika*. Czy się zmieniała? Dlaczego?
5. W bazie danych RBD1 odśwież zawartość materializowanej perspektywy za pomocą poniższego polecenia:
   `REFRESH MATERIALIZED VIEW zespoly_replika;`
6. Sprawdź zawartość materializowanej perspektywy *zespoly_replika*. Jaka jest wada takiego sposobu odświeżania materializowanych perspektyw? [Raport]
7. W następnym etapie będziemy automatycznie, cyklicznie odświeżać materializowaną perspektywę. W tym celu zainstalujemy rozszerzenie pg_cron, które nie znajduje się w standardowej dystrybucji systemu Postgres. Zmiany wykonane w plikach kontenera poza katalogami, w których są zamontowane trwałe woluminy są ulotne. Restart, skalowanie lub odtwarzanie Pod powoduje utratę zmian, które zostały wprowadzone do systemu plikowego kontenera przez instalator. Z tego powodu w dalszej części tutorialu utworzymy własny obraz kontenera w terminalu pomocniczym.
8. Pobierz plik *dockerfile* za pomocą poniższego polecenia:
   `wget http://www.cs.put.poznan.pl/jjezierski/RBDv2/pg_cron.dockerfile`
9. Przeglądnij ten plik wykorzystując poniższe polecenie:
   `less pg_cron.dockerfile`
   Dyrektywa FROM wskazuje na źródłowy obraz, który wykorzystamy jako bazę dla naszego obrazu. Dyrektywa RUN uruchamia polecenie wewnątrz źródłowego obrazu. Wykorzystamy sekwencję poleceń, które zainstaluje pliki rozszerzenia pg_cron w systemie plikowym naszej wersji obrazu. Zauważ, że do instalacji jest wykorzystywane polecenie *apt-get* a nie *yum*, wynika to faktu, że obraz kontenera wykorzystuje dystrybucję Debian Linuxa. Zamiast wielu dyrektyw RUN do uruchomienia każdego polecenia instalacji oddzielnie zastosowano sekwencję poleceń w jednej dyrektywie. Takie rozwiązanie minimalizuje liczbę warstw obrazu kontenera, które są generowane przez każdą dyrektywę, co optymalizuje rozmiar docelowego obrazu.
   Opuść program *less* wybierając przycisk *q*.
10. Utwórz własną wersję obrazu wykonując polecenie:
    `docker build -t rbd/postgres13:1.0 - < pg_cron.dockerfile`
    Przełącznik *t* polecenia *docker build* służy do wskazania nazwy i wersji naszego obrazu.
11. Utworzony obraz jest dostępny jedynie lokalnie. W celu zaimportowania go do klastra wykonaj poniższe polecenie:
    `k3d image import rbd/postgres13:1.0 --cluster RBDcluster`
12. Zmodyfikuj plik manifestów rbd1.yaml, ustaw wartość klucza *image* na wartość *rbd/postgres13:1.0* w celu zastosowania własnego obrazu kontenera.
13. Wykonaj wdrożenie z wykorzystaniem zmodyfikowanego pliku manifestów.
14. Zrestartuj połączenie z psql do bazy danych RBD1.
15. W celu załadowania do Postgresa zainstalowanej w obrazie biblioteki pg_cron wykonaj w bazie RBD1 następujące polecenie:
    `alter system set shared_preload_libraries = 'pg_cron';`
16. W terminalu pomocniczym zrestartuj Pod pgsql-rbd1, w celu zaaplikowanie zmian wprowadzonych w poprzednim kroku.
17. Zrestartuj połączenie z psql do bazy danych RBD1.

18. Utwórz w bazie danych RBD1 rozszerzenie pg_cron wykonując w psql poniższe polecenie:
    `CREATE EXTENSION pg_cron;`
19. Powtórz kroki od 12 do 18 dla bazy danych RBD2.
20. Wykorzystaj dokumentację rozszerzenia pg_cron do przygotowanie polecenia, które automatycznie, cyklicznie (np. co 5 minut) będzie odświeżać materializowaną perspektywę zespoly_replika [Raport].
21. Usuń klaster RBDcluster wykonując w terminalu pomocniczym polecenie:
    `k3d cluster delete RBDcluster`