Aktiv-Aktiv PostgreSQL Cluster: Eine Einführung

Kathegorie
docker, PostGreSQL, software

Eine der größten Herausforderungen für Datenbank-Entwickler besteht heutzutage darin, sicherzustellen, dass ihre Daten immer verfügbar sind, damit sie die Hochverfügbarkeitsanforderungen ihrer Anwendungen erfüllen können.

Die PostgreSQL-Replikation hat in den letzten Hauptversionen erhebliche Fortschritte gemacht. Neben verschiedener Verbesserungen und Erweiterungen werden jedoch Anwendungsfälle, in denen die Anwendung Zugriff auf eine aktualisierte Datenbank in mehr als einer geografischen Region benötigt, häufig als "Aktiv-Aktiv"-Cluster bezeichnet, nicht berücksichtigt.

Genauer gesagt handelt es sich bei einem Aktiv-Aktiv-Cluster um einen Cluster, bei dem eine Anwendung auf jeder Instanz Änderungen vornehmen kann, die dann in allen Instanzen repliziert werden, sodass jede Instanz im Cluster verwendet werden kann, um:

- nahezu keine Downtime entstehen zu lassen, da die neue Instanz sich bereits im Lese-/ Schreibzustand befindet (es ist keine Neukonfiguration notwendig),
- die Latenz für Benutzer in geographisch verteilten Clustern, durch Bereitstellung physisch näherer Instanzen, zu verbessern,
- nahezu keine Downtime entstehen zu lassen, während Upgrades eingespielt werden.

Wir werden uns mit potenziellen Referenzarchitekturen und Konfigurationen beschäftigen, die aktiv-aktiv PostgreSQL-Konfigurationen mit Open-Source-Software ermöglichen.

Denise
Robinson

Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

SymmetricDS

Eine Open Source-Lösung, die Aktiv-Aktiv-Datenbankkonfigurationen ermöglicht, ist SymmetricDS.

SymmetricsDS ist eine Open Source-Lösung, die Datenreplikation und -synchronisation für eine Vielzahl von Datenbanken, einschließlich PostgreSQL, bietet.

Einige der interessantesten Funktionen von SymmetricsDS sind:

- Ein **webbasierter Transport Layer**.
- Jeder Host, bzw. jede Datenbank die Java unterstützt kann synchronisiert werden. Somit wird **plattformübergreifendes** Arbeiten ermöglicht.
- Änderungen von verschiedenen Datenbankanbietern können synchronisiert werden, wodurch **datenbankübergreifendes** Arbeiten ermöglicht wird.

Wir zeigen Dir hier Deine ersten Schritte mit SymmetricDS mit zwei PostgreSQL-Datenbanken in einer Aktiv-Aktiv-Konfiguration.

Anforderungen

SymmetricDS ist in Java implementiert und erfordert eine Java-Laufzeitumgebung. Für die demonstrierte Umgebung habe ich openjdk-11.0.5

Des Weiteren benötigst Du SymmetricsDS:
https://www.symmetricds.org/download

Und Zu guter Letzt solltest Du über einen laufenden Postgres-Container verfügen. Solltest Du hierzu noch weitere Informationen benötigen, kannst Du diese gerne in diesem Artikel nachlesen.

Erste Schritte

Erstelle zunächst eine Datenbank mit dem Namen **sales** auf Deiner PostgreSQL-Instanz:

```
create database sales;
```

Jetzt brauchen wir einige Tabellen zum Synchronisieren. Erstelle in der **sales**-Datenbank zwei Tabellen, mit denen Du Verkäufe verfolgen kannst:

```
CREATE TABLE item ( id serial PRIMARY KEY, description text, price numeric (8,2)
);
```

und

```
CREATE TABLE sale ( id serial PRIMARY KEY, item_id int REFERENCES item(id), price
numeric(8,2) );
```

Als Nächstes ist es erforderlich eine Engine zu konfigurieren. Jede Engine steuert die Synchronisation für eine Datenbank und dient sozusagen als "Kennung" für die Datenbank, die Du mit SymmetricDS synchronisieren möchtest.
Um eine Engine zu konfigurieren, musst Du eine Properties-Datei im Engine-Unterverzeichnis erstellen. Hierzu erfordert es die folgende Konfiguration:

```
sync.url=http\://192.168.1.24\:31415/sync/sales
group.id=primary
db.init.sql=
registration.url=
db.driver=org.postgresql.Driver
db.user=rep
db.password=foo
db.url=jdbc\:postgresql\://192.168.1.24/sales?protocolVersion\=3&stringtype\=un-
specified&socketTimeout\=300&tcpKeepAlive\=true
engine.name=sales
external.id=1
db.validation.query=select 1
cluster.lock.enabled=false
```


Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Hier gibt es verschiedene Einstellungen für die Verbindung zur Datenbank: den **user**, das **Passwort** und die **Datenbank**, die Du replizieren möchtest.

Zusätzlich haben wir den Namen der Engine **sales**, die **group ID** und die **external ID**. Die Verwendung der beiden letztgenannten Optionen erläutern wir später.

An diesem Punkt können wir das System mit dem Befehl **sym_service** starten. Du kannst diesen Befehl in dem Verzeichnis ausführen, indem Du die SymmetricDS-Binärdateien installiert hast:

```
bin/sym_service start
```

Wenn Du SymmetricDS korrekt verwendest, sollte die Ausgabe in Deiner Konsole ähnlich aussehen:

```
Waiting for server to start

......

Started

Make sure you check the logs to see if there are any issues


_____ __ _ ____ _____
/ ___/ __ _____ __ ___ __ ___ _/ /_ ____(_)___ / __ | / ___/
\__ \ / / / / _ `_ \/ _ \ `_ \/ _ \/_ __// __/ / __/ / / / / \__ \
___/ // /_/ / // // / // // / __// / / / / /_ / /_/ / ___/ /
/____/ \__ /_//_//_/_//_//_/\___/ \_/ /_/ /_/\__/ /_____/ /____/
/____/
```

```
+--------------------------------------------------------------------+

| Copyright (C) 2007-2018 JumpMind, Inc. |
|
| Licensed under the GNU General Public License version 3. |
| This software comes with ABSOLUTELY NO WARRANTY. |
| See http://www.gnu.org/licenses/gpl.html |
+--------------------------------------------------------------------+

2020-08-27 10:34:49,937 INFO [startup] [SymmetricWebServer] [main] About to start
SymmetricDS web server on host:port 0.0.0.0:31415
2020-08-27 10:34:50,373 INFO [sales] [AbstractSymmetricEngine] [symmetric-engine-
startup-1] Initializing connection to database
2020-08-27 10:34:50,658 INFO [sales] [JdbcDatabasePlatformFactory] [symmetric-
engine-startup-1] Detected database 'PostgreSQL', version '11', protocol
'postgresql'
2020-08-27 10:34:50,672 INFO [sales] [JdbcDatabasePlatformFactory] [symmetric-
engine-startup-1] The IDatabasePlatform being used is org.jumpmind.db.platform.
postgresql.PostgreSqlDatabasePlatform
2020-08-27 10:34:50,798 INFO [sales] [PostgreSqlSymmetricDialect] [symmetric-
engine-startup-1] The DbDialect being used is org.jumpmind.symmetric.
db.postgresql.PostgreSqlSymmetricDialect
2020-08-27 10:34:50,856 INFO [sales] [ExtensionService] [symmetric-engine-
startup-1] Found 0 extension points from the database that will be registered
2020-08-27 10:34:50,861 INFO [sales] [StagingManager] [symmetric-engine-
startup-1] The staging directory was initialized at the following location: tmp/
sales
2020-08-27 10:34:51,730 INFO [sales] [ExtensionService] [symmetric-engine-
startup-1] Found 0 extension points from the database that will be registered
2020-08-27 10:34:51,731 INFO [sales] [ClientExtensionService] [symmetric-engine-
startup-1] Found 7 extension points from spring that will be registered
2020-08-27 10:34:51,740 INFO [sales] [AbstractSymmetricEngine] [symmetric-engine-
startup-1] Initializing SymmetricDS database
2020-08-27 10:34:51,740 INFO [sales] [PostgreSqlSymmetricDialect] [symmetric-
engine-startup-1] Checking if SymmetricDS tables need created or altered
```


Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Beim Start von SymmetricDS wird eine Verbindung zu der in der Properties-Datei angegebenen Datenbank hergestellt. Beim ersten Mal werden die für die Synchronisierung erforderlichen Tabellen erstellt.

Zum Vergleich siehst Du hier die relevanten Tabellen:

- **sym_node**: Bestimmt den Datenknoten und konfiguriert Dinge wie node Id, node group, external Id und sync URL
- **sym_node_identity**: Eindeutige Identität für diesen Knoten
- **sym_trigger**: Hier gibst Du an, welche Tabellen repliziert werden und welcher Router verwendet werden soll
- **sym_router**: Erstelle einen "Router", um die zu synchronisierenden Tabellen weiterzuleiten

Wir müssen diesen Tabellen einige Daten hinzufügen, damit unsere Instanzen korrekt miteinander synchronisiert werden. Hierzu erstellen wir erst eine **node group link**:

```
INSERT INTO sym_node_group_link (source_node_group_id,target_node_group_id,data_
event_action)
VALUES ('primary','primary','P');
```

Als Nächstes erstellen wir eine Route:

```
INSERT INTO sym_router (router_id,source_node_group_id,target_node_group_
id,router_type,router_expression,sync_on_update,sync_on_insert,sync_on_
delete,use_source_catalog_schema,create_time,last_update_by,last_update_time)
VALUES ('primary_2_primary', 'primary', 'primary', 'default', NULL, 1, 1, 1, 0,
CURRENT_TIMESTAMP, 'console', CURRENT_TIMESTAMP);
```

Schließlich fügen wir einige der Parameter hinzu, die zum Verwalten der Synchronisation erforderlich sind:

```
INSERT INTO sym_parameter (external_id, node_group_id, param_key, param_value,
create_time, last_update_by, last_update_time)
VALUES ('ALL', 'ALL', 'push.thread.per.server.count', '10', CURRENT_TIMESTAMP,
'console', CURRENT_TIMESTAMP);

INSERT INTO sym_parameter (external_id, node_group_id, param_key, param_value)
VALUES ('ALL', 'ALL', 'job.pull.period.time.ms', 2000);

INSERT INTO sym_parameter (external_id, node_group_id, param_key, param_value)
VALUES ('ALL', 'ALL', 'job.push.period.time.ms', 2000);
```

An diesem Punkt ist der Router nun eingerichtet. Wir müssen als nächstes einige Trigger hinzufügen, damit SymmetricDS versteht, was zu tun ist, wenn die Zieltabellen Änderungen erhalten:

```
INSERT INTO sym_trigger (trigger_id, source_schema_name, source_table_name,
channel_id, sync_on_update, sync_on_insert, sync_on_delete, sync_on_update_
condition, sync_on_insert_condition, sync_on_delete_condition, last_update_time,
create_time)
VALUES ('public.item', 'public', 'item', 'default', 1, 1, 1, '1=1', '1=1', '1=1',
CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);

INSERT INTO sym_trigger (trigger_id, source_schema_name, source_table_name,
channel_id, sync_on_update, sync_on_insert, sync_on_delete, sync_on_update_
condition, sync_on_insert_condition, sync_on_delete_condition, last_update_time,
create_time)
VALUES ('public.sale', 'public', 'sale', 'default', 1, 1, 1, '1=1', '1=1', '1=1',
CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);

INSERT INTO sym_trigger_router (trigger_id, router_id, enabled, initial_load_
order, create_time, last_update_time)
VALUES ('public.item', 'primary_2_primary', 1, 10, CURRENT_TIMESTAMP, CURRENT_
TIMESTAMP);

INSERT INTO sym_trigger_router (trigger_id, router_id, enabled, initial_load_
order, create_time, last_update_time)
VALUES ('public.sale', 'primary_2_primary', 1, 10, CURRENT_TIMESTAMP, CURRENT_
TIMESTAMP);
```


Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Wie wir sehen können, gibt es jetzt sowohl in der **item**- als auch in der **sale**-Tabelle Trigger.

```
\d item

Table "public.item"
Column      | Type         | Collation | Nullable | Default
------------+--------------+-----------+----------+---------------------------------
id          | integer      |           | not null | nextval('item_id_seq'::regclass)
description | text         |           |          |
price       | numeric(8,2) |           |          |

Indexes:
    "item_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "sale" CONSTRAINT "sale_item_id_fkey" FOREIGN KEY (item_id) REFERENCES item(id)
Triggers:
    sym_on_d_for_pblctm_prmry AFTER DELETE ON item FOR EACH ROW EXECUTE PROCEDURE fsym_on_d_for_pblctm_prmry()
    sym_on_i_for_pblctm_prmry AFTER INSERT ON item FOR EACH ROW EXECUTE PROCEDURE fsym_on_i_for_pblctm_prmry()
    sym_on_u_for_pblctm_prmry AFTER UPDATE ON item FOR EACH ROW EXECUTE PROCEDURE fsym_on_u_for_pblctm_prmry()

\d sale

Table "public.sale"
Column   | Type         | Collation | Nullable | Default
---------+--------------+-----------+----------+---------------------------------
id       | integer      |           | not null | nextval('sale_id_seq'::regclass)
item_id  | integer      |           |          |
price    | numeric(8,2) |           |          |

Indexes:
    "sale_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "sale_item_id_fkey" FOREIGN KEY (item_id) REFERENCES item(id)
Triggers:
    sym_on_d_for_pblcsl_prmry AFTER DELETE ON sale FOR EACH ROW EXECUTE PROCEDURE fsym_on_d_for_pblcsl_prmry()
    sym_on_i_for_pblcsl_prmry AFTER INSERT ON sale FOR EACH ROW EXECUTE PROCEDURE fsym_on_i_for_pblcsl_prmry()
    sym_on_u_for_pblcsl_prmry AFTER UPDATE ON sale FOR EACH ROW EXECUTE PROCEDURE fsym_on_u_for_pblcsl_prmry()
```

Du hast jetzt Deine erste aktive Datenbank eingerichtet!

So richtest Du die zweite aktive Datenbank ein

Um unsere Aktiv-Aktiv-Umgebung erstellen können, benötigen wir eine zweite aktive Datenbank. Dazu erstellen wir wie folgt eine weitere Datenbank mit dem Namen **sales2**:

```
create database sales2;
```

Anschließend erstellen wir eine sales2 Properties-Datei mit der folgenden Konfiguration:

```
db.connection.properties=
db.password=foo
sync.url=http\://192.168.1.27\:31415/sync/sales2
group.id=primary
db.init.sql=
db.driver=org.postgresql.Driver
db.user=rep
engine.name=sales2
external.id=sales2
db.validation.query=select 1
cluster.lock.enabled=false
registration.url=http\://192.168.1.24\:31415/sync/sales2
db.url=jdbc\:postgresql\://192.168.1.27/sales2?protocolVersion\=3&stringtype\=un-
specified&socketTimeout\=300&tcpKeepAlive\=true
```

Wie zuvor kannst Du mit folgendem Befehl SymmetricDS für die zweite Instanz starten:

```
bin/sym_service start
```

Beim Überprüfen der Protokolle, solltest Du eine Zeile finden, die ungefähr so aussieht:

```
Using registration URL of http://192.168.1.24:31415/sync/sales2/registrati-
on?nodeGroupId=primary&externalId=sales2&syncURL=http%3A%2F%2F192.168.1.27%3A
31415%2Fsync%2Fsales2&schemaVersion=%3F&databaseType=PostgreSQL&databaseVer-
sion=10.6&symmetricVersion=3.9.15&deploymentType=server&hostName=ubuntu2&ipAd-
dress=192.168.1.27
```


Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Diese URL enthält eine Kennung, mit der sich die beiden PostgreSQL-Instanzen über SymmetricDS gegenseitig erkennen können. Auf dem ersten Host (**sales**) führst Du den folgenden Befehl aus:

```
bin/symadmin -e sales2 open-registration primary sales2
```

Dieser Befehl weist den Datenknoten an, einen weiteren Knoten mit der external id von sales2 aufzunehmen, um sich in der primären Gruppe zu registrieren. Wenn wir die Protokolle überprüfen, können wir feststellen, dass die Trigger vom ersten Server zum zweiten synchronisiert und unser Knoten registriert wurden.

```
2018-12-06 09: 36: 29,856 INFO [sales2] [TriggerRouterService] [sales2-job-4]
Synchronisierung der Trigger abgeschlossen
2018-12-06 09: 36: 29,889 INFO [sales2] [RegistrationService] [sales2-job-4 ]
Erfolgreich registrierter Knoten [id = sales2]
```

Weitere Bestätigungen hierfür findest Du in der Tabelle **sym_node** auf dem ersten Server.

Wenn Du die folgende Abfrage für die **sales2**-Instanz ausführst:

```
SELECT node_id, node_group_id, external_id FROM sym_node;
```

solltest Du folgendes sehen:

```
node_id  | node_group_id | external_id

---------+---------------+-------------
1        | primary       | 1
sales    | primary       | sales
```

Zu diesem Zeitpunkt werden alle **sym_ ***-Tabellen synchronisiert, aber die **item**- und **sale**-Tabellen müssen verschoben werden.

Auf dem **ersten** Knoten können wir Folgendes ausführen:

```
bin/symadmin -e sales2 send-schema -n sales2
```

Wechselst Du nach der Ausführung zum Knoten **sales2**, dann kannst Du sehen, dass das Schema importiert wurde. Hier ist zum Beispiel die **item**-Tabelle :

```
Table "public.item"
Column       | Type         | Collation | Nullable | Default
-------------+--------------+-----------+----------+-----------------------------
id           | integer      |           | not null | nextval('item_id_seq'::regclas
description  | text         |           |          |
price        | numeric(8,2) |           |          |

Indexes:
    "item_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "sale" CONSTRAINT "sale_item_id_fkey" FOREIGN KEY (item_id) REFERENCES it
Triggers:
    sym_on_d_for_pblctm_prmry AFTER DELETE ON item FOR EACH ROW EXECUTE PROCEDURE f
    sym_on_i_for_pblctm_prmry AFTER INSERT ON item FOR EACH ROW EXECUTE PROCEDURE f
    sym_on_u_for_pblctm_prmry AFTER UPDATE ON item FOR EACH ROW EXECUTE PROCEDURE f
```

Jetzt sind wir bereit, um zu sehen, ob wir Daten von dem sales-Knoten auf dem sales2-Knoten replizieren können.

Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

So testest Du die Aktiv-Aktiv-Replikation

Wir wollen als Nächstes überprüfen, ob wir Daten auf dem **sales**-Knoten schreiben und sie auf der **sales2**-Instanz anzeigen lassen können. Führe dazu auf dem sales-Knoten die folgende Abfrage aus:

```
INSERT INTO item (description, price) VALUES ('screw', .05);
```

Wenn alles eingerichtet ist und funktioniert, sollte Dir auf dem **sales2**-Knoten Folgendes angezeigt werden, wenn Du die folgende Abfrage ausführst:

```
TABLE item;

id  | description | price
---+--------------+-------
 1   | screw           | 0.05
```

Hinweis: **TABLE** ist eine andere Schreibweise für **SELECT * FROM**.

Die nächste große Frage ist, ob wir Daten in **sales2** einfügen und in **sales** abrufen können.

Führe auf **sales2** den folgenden Befehl aus:

```
INSERT INTO item (description, price) VALUES ('hammer', 10);
```

Mit **TABLE item**; schauen wir uns zunächst an, welche Daten in **sales2** enthalten sind:

```
id  | description | price
---+--------------+-------
1   | screw           | 0.05
2   | hammer       | 10.00
```

Und sehen anschließend mit dem selben Befehl **TABLE item**; bei **sales**, dass die Daten übernommen wurden:

```
id  | description | price
---+--------------+-------
1   | screw           | 0.05
2   | hammer       | 10.00
```

Damit haben wir jetzt ein funktionierende Aktiv-Aktiv-Replikation.

Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Next Steps

Nachdem eine Aktiv-Aktiv-Replikation eingerichtet ist, bietet es sich an sich als Nächstes damit zu beschäftigen, welche Konflikte bei der Synchronisation auftreten können. SymmetricsDS bietet hier verschiedene Lösungen zur **Konflikterkennung**, z.B.:

- **USE_PK_DATA** – Gibt an, dass nur der Primärschlüssel zum Erkennen eines Konflikts verwendet wird. Wenn eine Zeile mit demselben Primärschlüssel vorhanden ist, wird während einer Aktualisierung oder eines Löschvorgangs kein Konflikt festgestellt. Aktualisierungs- und Löschzeilen werden nur mithilfe der Primärschlüsselspalten aufgelöst. Wenn während des Einfügens bereits eine Zeile vorhanden ist, wird ein Konflikt erkannt.
- **USE_CHANGED_DATA** – Gibt an, dass der Primärschlüssel sowie alle Daten, die sich im Quellsystem geändert haben, zur Erkennung eines Konflikts verwendet werden. Wenn während des Einfügens bereits eine Zeile vorhanden ist, wird ein Konflikt erkannt.

Zur **Konfliktlösung** bieten sich auch mehrere Möglichkeiten an, z.B.:

- **MANUAL** – Zeigt an, dass bei Erkennung eines Konflikts der Batch fehlerhaft bleibt, bis ein manueller Eingriff erfolgt. Eine fehlerhafte Zeile wird in die Tabelle sym_incoming_error eingefügt. Die Konflikterkennungs-ID, die den Konflikt erkannt hat, wird zusammen mit den alten, den neuen und den "aktuellen Daten" in den Spalten old_data, new_data und cur_data, aufgezeichnet. Zum Auflösen kann die Spalte "resolve_data" manuell ausgefüllt werden, die beim nächsten Ladeversuch anstelle der ursprünglichen Quelldaten verwendet wird. Die Flag resolve_ignore kann auch verwendet werden, um anzugeben, dass die Zeile beim nächsten Ladeversuch ignoriert werden soll.
- **IGNORE** – Gibt an, dass das System die eingehende Änderung automatisch ignorieren sollte, wenn ein Konflikt erkannt wird. Die Spalte resolve_row_only steuert, ob der gesamte Batch ignoriert werden soll oder nur die in Konflikt stehende Zeile.

Du solltest jetzt dazu in der Lage sein, ein Aktiv-Aktiv-Cluster einzurichten und von verschieden Instanzen Änderungen vornehmen zu können, die dann auf anderen Instanzen repliziert werden.

Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612