Mit CD! | Ausgabe 4.08 Juni | Juli | www.phpmagazin.de | S&S

PHP magazin

Deutschland 9,80 € Österreich 10,80 € | Schweiz 19,20 SFr
Niederlande 11,25 € | Luxemburg 11,25 €

Firefox 3
Die neuen Features für Developer

WordPress
Mit eigenen Extensions WordPress ergänzen

PHP & JavaScript
Den Datenaustausch per JSON serverübergreifend meistern

Zend Framework
Benutzerauthentifizierung mit Zend_Auth und Zend_ACL

Ready for Change

So gelingt der Umstieg von MySQL auf PostgreSQL

Progressive Enhancement

Schlüssel zur Webentwicklung

CD-INHALT

webEdition 5 (Trial) • Komodo Edit 4.3 • WordPress 2.5 • TYPO3 4.1.6
PostgreSQL 8.3.1 • Firefox 3.0 • Zend Framework 1.5.1 • Mozilla prism 0.9

Datenträger enthält nur Lehr- oder Infoprogramme

4 195649 609809 04

Umstieg von MySQL auf PostgreSQL – die ersten Schritte

Ich bin dann mal ...!

Die MySQL-Datenbank ist wohl die verbreitetste Datenbank unter den PHP- und Webapplikationsentwicklern. Gerade durch ihre einfache Integration in PHP ist dies sehr gut nachvollziehbar. Viele Entwickler wissen allerdings noch nicht, dass die PostgreSQL-Datenbank nicht nur eine sehr gute Alternative darstellt, sondern einen weitaus größeren Funktionsumfang aufweist und näher als MySQL am SQL-Standard arbeitet. Wir führen in diesem Artikel den Beweis zu dieser These und zeigen Ihnen, dass der Umstieg nicht schwer ist.

▶ von Thomas Pfeiffer und Andreas Wenk

Bevor wir mit der Installation und den ersten Gehversuchen mit der PostgreSQL starten, sollen beide Datenbanken im Vergleich betrachtet werden. Die Gegenüberstellung soll Ihnen helfen, für sich zu entscheiden, ob Sie den Wechsel zur PostgreSQL vollziehen möchten, oder ob MySQL die richtige Wahl für Sie ist.

Eines der wichtigsten Themen beim Vergleich von Datenbanken ist sicherlich die Performance. Im direkten Vergleich zwischen MySQL und PostgreSQL muss aus unserer Sicht berücksichtigt werden, in welcher Art von Applikation die Datenbank zum Einsatz kommen soll. Man muss ganz klar sagen, dass bei einer Hochlastanwendung mit sehr vielen *SELECT*-Abfragen und unter dem Einsatz der Storage Engine MYISAM MySQL klar die Nase vorn hat. Das hat z.B. der Autor des Artikels zur „SwooDoo"-Suchmaschine in der letzten Ausgabe des PHP Magazins bereits sehr anschaulich dargestellt.

Mit der Einführung der Storage-Engine INNODB kann man in MySQL Transaktionen ausführen. Allerdings ist INNODB um einiges langsamer als MYISAM und deshalb gibt es hier keinen Vorteil für die MySQL. Ganz im Gegenteil – unter Verwendung von „Stored Procedures" in PostgreSQL hat diese gegenüber MySQL sogar die Nase vorn.

Kommen wir noch zu einem immer wichtiger werdenden Thema: Volltextindizierung. Das können beide Datenbanken –MySQL allerdings nur innerhalb der Storage Engine MYISAM. Und da kommen wir auch zu einem Nachteil. Volltextindizierung und Transaktionssicherheit lassen sich nicht kombinieren. Ein klares Plus für PostgreSQL. Natürlich stellt sich die Frage, ob diese Kombination notwendig ist. In Businessapplikationen mit weitreichenden Suchfunktionalitäten ist es allerdings fast nicht wegzudenken. Lesen Sie zu diesem Thema in der nächsten Ausgabe des PHP Magazins auch den Anschlussartikel zu diesem, mit dem Thema „*PostgreSQL und die Erweiterung tsearch2 – Suchmaschine im Eigenbau*".

PostgreSQL stellt eine große Anzahl von Datentypen zur Verfügung. Weiterhin ist es dem Benutzer zudem möglich, eigene

Typen per *CREATE TYPE* hinzuzufügen. Letzteres geht in MySQL nicht.

In einem weiteren Punkt unserer kurzen Gegenüberstellung sei noch auf die Erweiterbarkeit von SQL in PostgreSQL hingewiesen. Darunter versteht sich die Nutzung von Stored Procedures, welche als Funktionen in der Datenbank gespeichert und dann in einem *SELECT*-Statement durch den Aufruf der Funktion inkl. der Übergabe etwaiger Parameter ausgeführt werden. Die Umsetzung kann in SQL, in prozeduralen Sprachen wie PL/pgSQL, mit internen Funktionen oder mit in C geschriebenen Funktionen erfolgen. In diesem Artikel werden wir die Funktionsweisen in SQL und PL/pgSQL erläutern. In MySQL stehen Stored Procedures und Stored Functions ab der Version 5 zur Verfügung – allerdings nur unter Verwendung von SQL.

Abschließend betrachten wir Trigger. Trigger sind Funktionen, die vor oder nach einem *INSERT*-, *UPDATE*- oder *DELETE*-Statement ausgeführt werden können und durch diese ausgelöst werden. PostgreSQL unterstützt Trigger in diversen prozeduralen Sprachen, nicht jedoch in SQL. In MySQL sind Trigger ab der Version 5 verfügbar.

Wie Sie sehen, ist der Umfang der beiden Datenbanken ähnlich, und eines muss man klar sagen: MySQL hat mit dem Release der Version 5.0 in Sachen Funktionsumfang gut aufgeholt, und auf der anderen Seite ist der Vorwurf der schlechten Performance bei PostgreSQL mit der Version 8.1 nur noch schwer haltbar. Wir sehen viele Vorteile bei PostgreSQL und arbeiten deshalb in unseren Projekten hauptsächlich mit dieser Datenbank. Im Übrigen ist die Nutzung der PostgreSQL-Schnittstelle unter PHP genauso einfach wie MySQL, mit dem Unterschied, dass es keinen objektorientierten Stil wie bei der *mysqli*-Erweiterung gibt. Spätestens nach dem Erstellen einer DB-Wrapper-Klasse hat man damit ohnehin nicht mehr viel zu tun.

Datenbank: Schema und Template

Ein für die Businessapplikationsentwicklung nicht zu unterschätzender Aspekt ist die Zugriffsstruktur einer Datenbank – das so genannte „Schema". Eine Datenbank in PostgreSQL kann ein oder mehrere Schemata enthalten, wobei das Schema *public* automatisch beim Anlegen einer Datenbank erstellt wird. Wenn Sie Objekte einem bestimmten Schema nicht explizit zuweisen, werden diese Objekte automatisch dem Schema *public* zugewiesen. Daraus lässt sich schlussfolgern, dass das Schema *public* im Prinzip einer Umgebung gleichkommt, in welcher keine Schemata eingesetzt werden.

Aber wofür braucht man Schemata? Ganz klar – wenn Sie eine reine „Ein Benutzer"-Umgebung betreiben, brauchen Sie keine Schemata einzusetzen. Gehen wir aber davon aus, dass wir eine Applikation entwickelt haben, die eine Datenbankgrundstruktur enthält und zudem von vielen unterschiedlichen Kunden genutzt werden soll, ist der Einsatz von Schemata ein sinnvoller Weg, um eine gute Struktur, einfache Wartbarkeit und Sicherheit unseres Datenbankservers umzusetzen. Alle Tabellen und sonstigen Datenbankobjekte, auf die jeder Datenbankbenutzer Zugriff haben soll, legen wir z.B. im Schema *public* ab (oder einem eigenen, auf welches alle Datenbankbenutzer Zugriff haben) und die kundenspezifischen Tabellen und Objekte legen wir in jeweils einem Schema ab, auf welches nur der jeweilige Datenbankbenutzer des Kunden Zugriff hat. MySQL unterstützt seit der Version 5.0 Schemata.

Ein weiterer hilfreicher Mechanismus in der PostgreSQL ist die Art und Weise, wie neue Datenbanken erstellt werden. Dies geschieht mit *Templates* (engl. Vorlage). Es gibt zwei Systemtemplates: *template0* und *template1*. Das sind die Vorlagedatenbanken die genutzt werden, um neue Datenbanken per *CREATE DATABASE* zu erstellen. Wird nichts weiter angegeben, wird die Datenbank *template1* kopiert und mit dem angegebenen Datenbanknamen erstellt. *template1* ist eine Datenbank, die Sie mit eigenen Objekten erweitern können, die dann beim Erzeugen neuer Datenbanken automatisch verfügbar sind. Weiter unten sehen Sie ein Beispiel, in welchem wir dem *template1* die Sprache PL/pgSQL zugewiesen haben. Haben Sie dies einmal getan, werden zukünftige Datenbanken die Sprache PL/pgSLQ ebenfalls enthalten. *template0* ist im Gegensatz dazu eine „jungfräuliche" Datenbank und sollte nicht

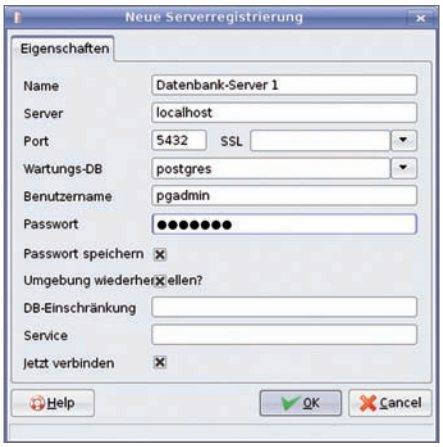


Abb. 1: Ein neuer Server ist schnell angelegt

verändert werden. Somit haben Sie immer die Möglichkeit, auf eine „ursprüngliche" Datenbank zurück zu greifen, müssen dies beim Erstellen einer Datenbank dann aber auch explizit angeben:

```
CREATE DATABESE ihr_datenbank_name TEMPLATE template0;
```

Sie können auch eigene Templates erstellen und verfahren beim Erstellen einer neuen Datenbank dann genauso wie mit *template0*.

Installation der PostgreSQL

Auf einem Debian-basierten System führen Sie einfach

```
apt-get install postgresql-8.1
```

aus, um PostgreSQL zu installieren. Die Datenbank wird dann in der Regel gleich gestartet.

Als Erstes wollen wir einen zusätzlichen administrativen Benutzer anlegen, den wir später dazu verwenden, um uns über *PgAdmin* mit dem Datenbankserver zu verbinden. Starten Sie dazu ein Terminal mit Root-Rechten und wechseln Sie den Benutzer mit

```
su – postgres
```

Danach legen Sie einen neuen Benutzer *pgadmin* mit dem Befehl

```
createuser -a -d -P -N -U postgres pgadmin
```

an. Damit sagen wir PostgreSQL: Lege eine Rolle an, der es erlaubt ist, weitere

Abb. 2: Das Anlegen einer neuen Login-Rolle

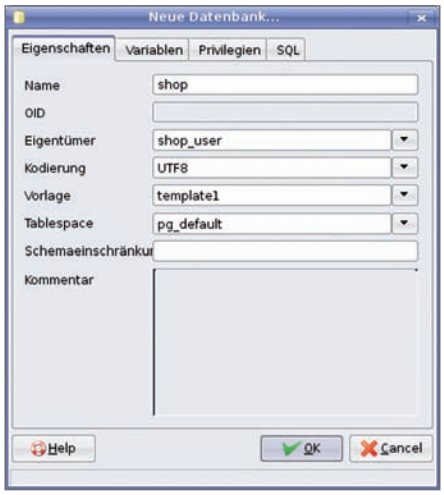


Abb. 3: Auch eine neue Datenbank ist schnell angelegt

Benutzer anzulegen (*-a*) und neue Datenbanken zu erzeugen (*-d*). Diese Rolle bekommt ein Passwort (*-P*), das wir zunächst einmal nicht verschlüsselt speichern (*-N*). Das Ganze erledigen wir als Datenbank-Superuser postgres (*-U postgres*), der bei der Installation mit angelegt wurde und mit dessen Rechten die Datenbank ausgeführt wird. Der Name der neu erzeugten Rolle soll *pgadmin* sein. Hat alles wie gewünscht geklappt, gibt der Befehl dann CREATE ROLE zurück.

Da wir aber von Natur aus misstrauisch sind, wollen wir noch einmal mithilfe des Command-Line-Clients der PostgreSQL überprüfen, ob das Anlegen des Benutzers erfolgreich war. Rufen Sie dazu das Programm *psql* mit der Datenbank *postgres* auf:

```
psql postgres
```

Wir überprüfen die Einträge in der Benutzertabelle mit:

```
SELECT * FROM pg_user;
```

Hier sollten Sie Ihren soeben erstellten Benutzer wiederfinden. Das Programm verlassen Sie durch die Eingabe *\q*.

Konfigurationsdateien anpassen

Es bleiben noch zwei Dateien, die wir bearbeiten müssen, bevor wir das Terminal wieder verlassen dürfen. Die erste ist die Datei *postgresql.conf*, bei Debian zu finden unter */etc/postgresql/8.1/main/postgresql.conf*. Hier werden die Grundeinstellungen des Datenbankservers vorgenommen. Die Zeile

```
listen_addresses = 'localhost'
```

legt fest, von welchen Adressen aus Sie sich mit dem Datenbankserver verbinden dürfen. Für Ihre PHP-Anwendung dürfte diese Einstellung in der Regel ok sein, wenn diese auf der gleichen Maschine wie Ihre Datenbank läuft. Wollen Sie sich auch von anderen Rechnern über z.B. *PgAdmin* mit dem Datenbankserver verbinden, ändern Sie hier den Eintrag entsprechend ab. Wollen oder können Sie dies aus Sicherheitserwägungen nicht tun, zeigen wir später, wie Sie die Verbindung zum Datenbankserver durch einen SSH-Tunnel herstellen können.

Die nächsten Einträge, die wir abändern wollen, betreffen das Logging der Datenbank. Entfernen Sie die Kommentare vor den folgenden Zeilen:

```
log_destination = 'stderr'
redirect_stderr = on
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_statement = 'all'
```

Damit werden dann tägliche Logfiles im Verzeichnis */var/lib/postgresql/8.1/main/pg_log/* erstellt – während der Entwicklung einer Anwendung eine nicht zu unterschätzende Hilfe.

Die letzten Einträge, die wir in dieser Datei zunächst modifizieren wollen, betreffen das *auto-vacuum*. Regelmäßiges Vacuum muss in PostgreSQL ausgeführt werden um

- Speicherplatz von gelöschten oder aktualisierten Zeilen wiederzugewinnen
- die vom PostgreSQL-Anfrageplaner verwendeten Datenstatistiken zu aktualisieren
- dem Verlust sehr alter Daten durch den Überlauf von Transaktionsnummern vorzubeugen

Mit der Version 8.1 von PostgreSQL existiert ein *autovacuum daemon*, der sich automatisch um die Bereinigung der Datenbank kümmert. Um ihn zu aktivieren, müssen in der Konfiguration die folgenden Schalter gesetzt sein:

```
autovacuum = on
stats_start_collector = on
stats_row_level = on
```

Es sollte genügen, die entsprechenden Kommentare zu entfernen.

Der letzte Schritt ist dann noch das Bearbeiten der Datei *pg_hba.conf*. Hier legen wir fest, welche Benutzer und Hosts sich mit welchen Datenbanken verbinden dürfen. *hba* steht hier für „host-based authentication". Die Datei ist wie folgt aufgebaut:

```
# TYPE DATABASE    USER      CIDR-ADDRESS    METHOD
host               all       all     127.0.0.1/32
password
host               all       pgadmin
192.168.1.20/32    password
```

Der erste Eintrag erlaubt dem Verbindungstyp *host* (TCP/IP-Verbindungen) den Zugriff auf alle Datenbanken mit allen Benutzernamen von der lokalen Maschine (127.0.0.1), sofern das richtige Passwort verwendet wird. Das Passwort ist hier wieder unverschlüsselt. In nichtvertrauenswürdigen Netzen kann man die Methode auch auf *md5* oder andere setzen [1]. Die zweite Zeile würde damit dem Benutzer *pgadmin* auf alle Datenbanken den Zugriff erlauben, sofern die Anmeldung von der IP Adresse 192.168.1.20 aus erfolgt. Welche Einträge Sie hier benötigen, hängt von der gewünschten Zu-

griffsart ab. In der Regel genügt aber ein Eintrag wie in der ersten Zeile.

Sind diese Arbeiten erledigt, können Sie den Datenbankserver mit

```
/etc/init.d/postgresql-8.1 restart
```

neu starten und die Shell fürs erste verlassen.

PgAdmin

Nachdem Ihr Datenbankserver jetzt läuft, können wir uns der Erstellung der ersten Datenbank zuwenden.

Als sehr komfortables Tool für den Umgang mit dem PostgreSQL-Server hat sich PgAdmin [2] herausgestellt. Nachdem Sie das Programm installiert haben, können Sie einen neuen Server hinzufügen (Abb. 1).

Sofern Ihr Datenbankserver auf der gleichen Maschine wie PgAdmin läuft, können Sie hier als Server einfach *localhost* eintragen, ansonsten die Adresse der entfernten Maschine. Im zweiten Fall müsste die Maschine dann jedoch auf Port 5432 lauschen. Tut Sie dies nicht, weil z.B. Ihre Firewall dies nicht zulässt, können Sie die Verbindung auch durch einen SSH-Tunnel herstellen. Diesen Tunnel erzeugt man bei einem Linux-System auf der Clientseite mit:

```
ssh user@remotehost -L 5432:localhost:5432
```

Damit werden alle Anfragen auf den Port 5432 auf der lokalen Maschine durch eine verschlüsselte Verbindung an den entfernten Server weitergeleitet.

Als erstes wollen wir jetzt mithilfe von PgAdmin einen neuen Benutzer anlegen, den wir dann später als Eigentümer für unsere noch zu erstellende Datenbank verwenden. Wählen Sie hierzu im linken Baum der Applikation Login-Rollen und legen einen neuen Benutzer an (Abb. 2).

Danach erstellen wir unsere erste eigene Datenbank – ebenfalls mithilfe von PgAdmin. Wählen Sie dazu im linken Baum *Datenbanken* aus und legen Sie eine neue Datenbank an (Abb. 3).

Das Anlegen einer neuen Datenbank funktioniert, indem eine bestehende kopiert wird. Wird hier nichts anderes angegeben, verwendet PostgreSQL die Systemdatenbank *template1* als Vorlage. Und hier jetzt bitte kurz STOP! Da wir später die Sprache PL/pgSQL verwenden möchten, um benutzerdefinierte Funktionen zu erstellen, öffnen wir doch noch einmal schnell ein Terminal und setzen den Befehl

```
createlang plpgsql template1
```

als Benutzer postgres ab. Damit wird die Sprache PL/pgSQL in der Datenbank *template1* installiert. Alle Datenbanken, die wir dann später aus dieser Vorlage heraus erzeugen, verfügen somit automatisch über diese Sprache. Damit wäre dann die erste Datenbank erzeugt, und wir können uns an das Erstellen der Tabellen machen.

Tabellen

Das erste, was der MySQL-Umsteiger vielleicht vermissen mag, sind so lieb gewordene Dinge wie *auto_increment*-Felder und den Datentyp *set*. Hat man also in MySQL noch folgendes Statement zum Erzeugen einer Tabelle absetzen können

```
CREATE TABLE products (
 id int(11) NOT NULL auto_increment,
 title varchar(255) NOT NULL,
 price numeric (10,2) DEFAULT 0.00,
 is_active set('0', '1') default '1',
 PRIMARY KEY (id)
);
```

stellt sich jetzt natürlich die Frage, wie muss das Ganze in PostgreSQL aussehen?

Das, was in der MySQL der Datentyp *auto_increment* war, ist in der PostgreSQL *serial*. Dies ist kein reeller Datentyp, sondern nur die abgekürzte Schreibweise für die Erzeugung einer Spalte mit eindeutiger, automatischer Nummerierung. Somit ist

```
CREATE TABLE products (
 id serial
);
```

also nur die Kurzschreibweise von:

```
CREATE SEQUENCE products_id_seq;
CREATE TABLE products (
 id integer DEFAULT nextval('products_id_seq') NOT NULL
);
```

Soweit, so gut. Was aber machen wir mit dem Datentyp *set*? Natürlich kann man hierfür einfach ein Feld vom Typ *char(1)* verwenden, allerdings prüft dieses dann noch nicht, ob tatsächlich nur die erlaubten Werte *0* und *1* verwendet werden. Um das zu ermöglichen, führen wir für dieses Feld eine so genannte „Beschränkung", im Datenbankjargon als „Constraint" bekannt, ein. In unserem Fall verwenden wir einen Check Constraint, der es erlaubt, einen Ausdruck anzugeben, den jeder Wert der Spalte erfüllen muss. Die vollständige Übersetzung des obigen MySQL-*CREATE*-Statements ins postgres'sche sieht dann wie folgt aus:

```
CREATE TABLE products (
 id serial,
 title varchar(255) NOT NULL,
 price numeric (10,2) DEFAULT 0.00,
 is_active char(1) DEFAULT '1', CHECK (is_active IN ('0','1')),
 PRIMARY KEY (id)
);
```

Fremdschlüssel – Foreign Key Constraints

Foreign Key Constraints geben an, dass Werte in einer Spalte mit Werten einer Spalte in einer anderen Tabelle übereinstimmen müssen. Bleiben wir der Einfachheit halber bei der soeben erstellten Tabelle *products* als Grundlage. Wir möchten jetzt noch drei weitere Tabellen erstellen.

LISTING 1

Tabelle für das Löschen von Produkten vorbereiten

```
DROP TABLE orders;
CREATE TABLE orders (
 id serial,
 customers_id integer REFERENCES customers (id) ON
DELETE CASCADE,
 order_dat timestamp default current_timestamp,
 PRIMARY KEY (id)
);

DROP TABLE orders_articles;
CREATE TABLE orders_articles (
 orders_id integer REFERENCES orders (id) ON DELETE
CASCADE,
 products_id integer REFERENCES products (id) ON
DELETE RESTRICT,
 quantity integer DEFAULT 1,
 PRIMARY KEY (orders_id, products_id)
);
```

In der ersten Tabelle werden wir unsere Kunden speichern

```
CREATE TABLE customers (
 id serial,
 first_name varchar(255) DEFAULT NULL,
 last_name varchar(255) NOT NULL,
 primary key(id)
);
```

In der zweiten Tabelle wollen wir die Bestellungen der Kunden ablegen

```
CREATE TABLE orders (
 id serial,
 customers_id integer REFERENCES customers (id),
 order_dat timestamp default current_timestamp,
 PRIMARY KEY (id)
);
```

und in der dritten Tabelle die zu der Bestellung gehörenden Produkte

```
CREATE TABLE orders_articles (
 orders_id integer REFERENCES orders (id),
 products_id integer REFERENCES products (id),
 quantity integer DEFAULT 1,
 PRIMARY KEY (orders_id, products_id)
);
```

Damit ist es jetzt unmöglich geworden, in der Tabelle *orders* einen Datensatz zu erzeugen, zu dem es keine Kundennummer aus der Tabelle *customers* gibt, und in *orders_articles* Einträge anzulegen, die zu keinem gültigen Produkt oder zu keiner gültigen Bestellung passen. Durch diese Fremdschlüsselbeziehung können wir jetzt ein Fehlverhalten unserer Applikation abfangen und sind sicher, dass die Datenbank immer in einem konsistenten Zustand ist.

Aber eigentlich war das nur der halbe Spaß. Was soll denn passieren, wenn eine Bestellung gelöscht wird? Dann möchten wir ja eigentlich, dass auch alle zu dieser Bestellung gehörenden Einträge aus der Tabelle *orders_articles* entfernt werden. Wenn ein Kunde gelöscht werden soll, wollen wir sogar alle zu diesem Kunden gehörenden Bestellungen in *orders* und alle von diesem Kunden bestellten Artikel in *orders_articles* löschen.

Und was ist, wenn ein Produkt gelöscht wird? In dem Fall müssen wir sicherstellen, dass ein Produkt nur dann gelöscht werden kann, wenn es in keiner Bestellung mehr verwendet wird. Wir ändern unsere Tabellen *orders* und *orders_articles* also noch einmal ab (Listing 1).

CASCADE bedeutet in diesem Fall, dass automatisch alle dazugehörigen Einträge aus der Tabelle *orders_articles* entfernt werden, wenn eine Bestellung in der Tabelle *orders* gelöscht wird. Somit genügt ein einfaches

```
DELETE FROM orders where id = 23;
```

um alle notwendigen Änderungen in beiden Tabellen durch das Datenbanksystem ausführen zu lassen.

RESTRICT hingegen bedeutet, dass das Datenbanksystem sich weigern wird, einen Eintrag in der Tabelle *products* zu löschen, solange es in der Tabelle *orders_articles* noch Einträge gibt, die darauf verweisen. Wird hingegen ein Kunde gelöscht, wird das Datenbanksystem automatisch alle zu diesem Kunden passenden Einträge in *orders* löschen und alle mit diesen Bestellungen verknüpften Einträge in *orders_articles*. Auch hier genügt dann ein *DELETE*-Statement wie

```
DELETE FROM customers where id = 42;
```

um alle Änderungen durchführen zu lassen.

Grundsätzlich ist es natürlich möglich, all diese Schritte auch über Ihren PHP-Code ausführen zu lassen, die Fehleranfälligkeit einer umfangreichen Anwendung wird aber deutlich minimiert, wenn Sie dem Datenbanksystem solche „Regeln" mitgeben. Mal ganz davon abgesehen, dass sich auch Ihr Programmcode dadurch stark verkürzen kann, und das ist ja auch nicht das Schlechteste.

Benutzerdefinierte Funktionen – Stored Procedures

Richtig spannend wird die Geschichte dann, wenn wir PostgreSQL mit eigenen Funktionen erweitern. Damit ist es möglich, Berechnungen und Anfragen *innerhalb* des Datenbankservers zusammenzufassen. Muss eine Anwendung also viele aufwändige Datenbankoperationen durchführen, können Sie diese sozusagen „in das Innere" Ihres Servers verlagern und so das Ausmaß an Client-/Serverkommunikation verringern. Sie erhalten damit unter Umständen einen erheblichen Performancegewinn. Aber fangen wir langsam an.

Am einfachsten benutzen Sie den SQL-Editor von PgAdmin und geben folgenden Code ein. Achten Sie darauf, dass Sie vorher die richtige Datenbank ausgewählt haben.

```
CREATE OR REPLACE FUNCTION eins()
RETURNS INTEGER AS
$BODY$
        SELECT 1 as ergebnis;
$BODY$
LANGUAGE SQL;
```

LISTING 2

Artikel einer Bestellung hinzufügen

```
CREATE OR REPLACE FUNCTION add_item(integer,
                                    integer, integer)
RETURNS numeric AS
$BODY$
DECLARE
 v_total numeric;
 prod_seen integer;
 v_order_id ALIAS FOR $1;
 v_prod_id ALIAS FOR $2;
 v_quantity ALIAS FOR $3;
BEGIN
 -- Erst mal prüfen, ob dieses Produkt in derselben
 --Bestellung schon mal geordert wurde
 -- Wenn ja wird ein update durchgeführt, ansonsten
 -- ein insert.
 prod_seen := 0;
 SELECT INTO prod_seen products_id FROM
 orders_articles WHERE orders_id = v_order_id AND
                                products_id = v_prod_id;

 IF prod_seen <> 0 THEN
  UPDATE orders_articles SET quantity = quantity
                                + v_quantity
   WHERE orders_id = v_order_id AND products_id
                                = v_prod_id;
 ELSE
  INSERT INTO orders_articles (orders_id, products_
                                id, quantity)
   VALUES (v_order_id, v_prod_id, v_quantity);
 END IF;
 -- Summe der gegenwärtigen Bestellung bilden und
 -- als Funtionsergebnis zurückliefern
 SELECT INTO v_total SUM(a.quantity * b.price) AS total
 FROM orders_articles a, products b
 WHERE
 a.products_id = b.id AND
 a.orders_id = v_order_id;

 RETURN v_total;
END
$BODY$
LANGUAGE 'plpgsql';
```

Jetzt können Sie die Funktion aufrufen, indem Sie folgendes *SELECT*-Statement absetzen:

```
select eins();
```

Extrem unspektakulär, aber Ihre erste eigene Funktion, die dann auch in PgAdmin im Baum Ihrer Datenbank unter *Funktionen* auftaucht. Probieren wir eine weitere, eigentlich unnütze Funktion zu erstellen. Wir möchten jetzt eine Produkt-ID übergeben und wollen von der Funktion den Preis des Produkts geliefert bekommen:

```
CREATE OR REPLACE FUNCTION get_price(integer)
RETURNS numeric AS
$BODY$
 SELECT price FROM products WHERE id = $1;
$BODY$
LANGUAGE SQL;
```

Wichtig ist hierbei, dass Sie in der Signatur der Funktion bereits den richtigen Typ angeben. Da die Spalte *id* in der zu prüfenden Tabelle *products* vom Typ *Integer* ist, müssen wir dies in den Funktionsparametern entsprechend benennen. Um dann diesen Wert verwenden zu können, verwenden wir im Funktionskörper die Schreibweise *$1* für das erste Argument, *$2* für das zweite, usw. Sie können jetzt

```
select get_price(1);
```

verwenden, um sich den Preis für das Produkt mit der ID *1* ausgeben zu lassen.

Und wieso die Rede von PL/pgSQL? Scheinbar wurde hier durch die Angabe von *LANGUAGE SQL;* in der jeweils letzten Zeile der Funktionen nur reines SQL verwendet. Tatsächlich ist es so, dass die Verwendung von SQL als Sprache für benutzerdefinierte Funktionen es uns erlaubt, eine beliebige Liste von SQL-Befehlen ausführen zu lassen. So etwas wie Kontrollstrukturen haben wir damit aber nicht. Möchten wir also z.B. der Funktion sagen: „Führe diesen Anweisungsblock nur unter dieser und jener Bedingung aus“ oder „Führe den Block *n* aus“, dann kommen wir mit reinem SQL nicht sehr weit. Genau für solche Dinge eignet sich aber die Sprache PL/pgSQL.

Bleiben wir weiterhin beim Shopbeispiel und probieren es mit folgender Aufgabe: Wir möchten eine Funktion schreiben, die einen Artikel zu einer Bestellung eines Kunden hinzufügt. Ist der Artikel in der Bestellung bereits vorhanden, soll die neue Bestellmenge zu der bereits vorhandenen addiert werden, andernfalls wird einfach ein neuer Datensatz eingefügt. Als Ergebnis soll die Funktion die Summe der aktuellen Bestellung liefern (Listing 2).

Im *DECLARE*-Abschnitt der Funktion können wir ein paar Variablen benennen, mit denen wir im Folgenden arbeiten wollen. Außerdem können wir hier Aliase für die übergebenen Parameter vergeben. Das erleichtert die Arbeit innerhalb der Funktion.

Im eigentlichen Funktionskörper weisen wir der vorher deklarierten Variablen *prod_seen* erst den initialen Wert *0* zu. Danach benutzen wir die *SELECT INTO*-Anweisung, um der Variablen *prod_seen* das Ergebnis der Abfrage zuzuweisen. Die *SELECT INTO*-Anweisung erlaubt uns, genau eine Zeile des Ergebnisses eines *SELECT*-Statements einer *Record*-Variablen oder einer Liste von skalaren Variablen zuzuweisen. Möchten Sie also z.B. zwei Variablen einen Wert zuweisen, schreiben Sie

```
SELECT INTO var1, var2 spalte1, spalte2 FROM ...
```

um der Variablen *var1* den Wert von *spalte1* und *var2* den Wert von *spalte2* zuzuweisen. Achten Sie auf das fehlende Komma zwischen *var2* und *spalte1*!

Um das Ergebnis einer Abfrage einer *Record*-Variablen zuzuweisen, schreiben Sie hingegen so etwas wie:

```
SELECT INTO rec * FROM ...
```

Die Variable *rec* müssen Sie dabei vorher im *DECLARE*-Abschnitt mit *rec RECORD* benennen. Auf die einzelnen Werte der Record-Variablen können Sie dann über die Punktnotation zugreifen, also z.B.

```
IF rec.spalte1 > 0 THEN ...
```

Aber zurück zu unserem Beispiel. Mit der Bedingung *IF prod_seen <> 0* finden wir dann heraus, ob in dieser Bestellung bereits ein Produkt mit dieser Produkt-ID in der Tabelle *orders_articles* existiert. Ist dies der Fall, führen wir ein Update durch, indem wir einfach zu der bereits vorhandenen Menge die neue Menge addieren. Ist das Produkt in der Bestellung noch nicht vorhanden, legen wir einen neuen Datensatz an.

LISTING 3

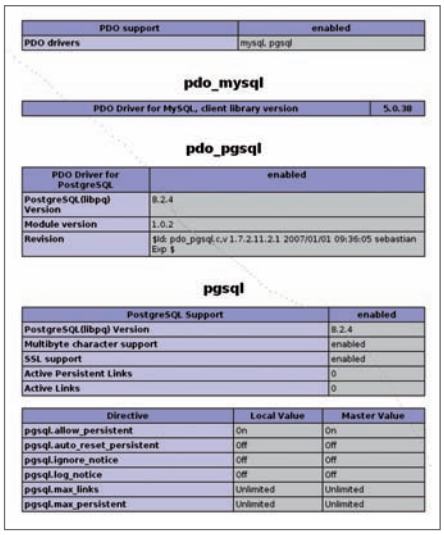
Definition der Trigger-Funktion

```
CREATE OR REPLACE FUNCTION log_product()
RETURNS trigger AS
$BODY$
BEGIN
 IF NEW.title != OLD.title OR NEW.price != OLD.
                                        price THEN
  INSERT INTO log_products (title_before, price_
                                        before,
   title_after, price_after) VALUES (
   OLD.title, OLD.price,
   NEW.title, NEW.price);
 END IF;
 RETURN NULL;
END
$BODY$
LANGUAGE 'plpgsql';
```

LISTING 4

Methode, um unser Array von Statements innerhalb einer Transaktion abzuarbeiten

```
public function transaction() {

 $ok = 1;
 pg_query($this->dbh, 'BEGIN');

 while ($sql = array_shift($this->sql)) {
  if(!pg_query($this->dbh, $sql)){
   $ok = 0;
   $this->set_error();
  }
 }

 if ($ok) {
  pg_query($this->dbh, 'COMMIT');
  $this->sql = array();

  return true;
 } else {
  $error = 'Transaktion fehlgeschlagen! Fehler: '.
pg_last_error($this->dbh);
  $this->set_error($error);
  $this->sql = array();

  return false;
 }
}
```



PDO support	enabled
PDO drivers	mysql, pgsql

pdo_mysql

PDO Driver for MySQL, client library version	5.0.38

pdo_pgsql

PDO Driver for PostgreSQL	enabled
PostgreSQL(libpq) Version	8.2.4
Module version	1.0.2
Revision	$Id: pdo_pgsql.c,v 1.7.2.11.2.1 2007/01/01 09:36:05 sebastian Exp $

pgsql

PostgreSQL Support	enabled
PostgreSQL(libpq) Version	8.2.4
Multibyte character support	enabled
SSL support	enabled
Active Persistent Links	0
Active Links	0

Directive	Local Value	Master Value
pgsql.allow_persistent	On	On
pgsql.auto_reset_persistent	Off	Off
pgsql.ignore_notice	Off	Off
pgsql.log_notice	Off	Off
pgsql.max_links	Unlimited	Unlimited
pgsql.max_persistent	Unlimited	Unlimited

Abb. 4: Ausgabe von phpinfo() nach dem Neustart des Servers

Als Letztes verwenden wir wieder die *SELECT-INTO*-Anweisung, um der Variablen *v_total* die Summe der aktuellen Bestellung zuzuweisen. Danach lassen wir der Funktion diesen Wert mit der Anweisung *RETURN v_total* zurückliefern. Drei neue Produkte mit der ID *2* können Sie jetzt in die Bestellung mit der ID *1* aufnehmen, indem Sie folgendes Statement absetzen:

```
SELECT add_item(1, 2, 3);
```

Über Sinn und Unsinn der Funktion kann man natürlich streiten, aber sie zeigt, wie Sie Teile Ihrer Programmlogik ins Innere Ihres Datenbankservers verlagern können. Sobald Sie hier etwas umfangreichere Aktionen ausführen, wird Ihre Anwendung mit einem deutlichen Performancegewinn belohnt.

Trigger

Nun können wir zwar mit benutzerdefinierten Funktionen Aktualisierungen in mehreren Tabellen in einem Rutsch vornehmen, was aber, wenn wir bestimmte Aktionen innerhalb der Datenbank *vor* oder *nach* einer schreibenden Aktion vornehmen möchten?

Hierfür eignen sich Trigger. Die Befehle *INSERT*, *UPDATE* und *DELETE* können Trigger-Funktionen auslösen, die dann vor oder nach diesem Befehl ausgeführt werden. Beim Erstellen eines Triggers ist es wichtig, dass die Trigger-Funktion vor dem eigentlichen Erzeugen des Triggers erzeugt wird. Ein Beispiel: Wir möchten, dass jede Veränderung an den Feldern *title* und *price* in der Tabelle *products* in eine spezielle Log-Tabelle geschrieben wird. Dabei möchten wir den alten Wert, den neuen Wert und einen aktuellen Zeitstempel wegschreiben. Wir benötigen also zunächst eine Tabelle, die diese Log-Einträge aufnimmt:

```
CREATE TABLE log_products (
 id serial,
 title_before varchar(255) DEFAULT NULL,
 price_before numeric(10,2) DEFAULT NULL,
 title_after varchar(255) DEFAULT NULL,
 price_after numeric(10,2) DEFAULT NULL,
 modified timestamp DEFAULT current_timestamp,
 primary key(id)
);
```

Danach können wir dann die entsprechende Trigger-Funktion definieren (Listing 3).

Diese Funktion wird also nur ein *INSERT* in die Tabelle *log_products* ausführen, wenn die Bezeichnung oder der Preis verändert wurden. Auf die Inhalte vor und nach der Aktualisierung können wir über die Syntax *OLD.[Feldname]* bzw. *NEW.[Feldname]* zugreifen. Jetzt müssen wir noch den eigentlichen Trigger erzeugen. Dies tun wir mit

```
CREATE TRIGGER log_procducts_trigger AFTER UPDATE ON
                                                products
FOR EACH ROW EXECUTE PROCEDURE log_product();
```

Dieser Trigger wird nun nach einem erfolgreichen Update der Tabelle *products* ausgeführt, und zwar für jede betroffene Zeile der Abfrage.

Talk to me

Bislang haben wir mit PgAdmin sozusagen direkt auf dem Datenbankserver gearbeitet, und das auch noch als privilegierter Benutzer pgadmin. Jetzt soll unsere PHP-Anwendung aber mit dem PostgreSQL-Server sprechen. Zunächst muss natürlich sichergestellt sein, dass das entsprechende PHP-Modul installiert und geladen ist. Unter Debian installieren Sie dieses am einfachsten per:

```
apt-get install php5-pgsql
```

Nach dem Neustart des Webservers sollte die Ausgabe von phpinfo(); derjenigen in Abbildung 4 ähneln. Um PHP mit der PostgreSQL zu verbinden, verwenden wir

```
$dbh = pg_connect("host=localhost dbname=shop user=
                                shop_user password=geheim");
```

Und um eine Abfrage an den Server zu schicken und das Ergebnis in einem Array zu hinterlegen, verwenden wir

```
$qs = "select * from products";
$result = array();
$query = pg_query($dbh, $qs);

$fields = pg_num_fields($query);
$rows = pg_num_rows($query);

$i = 0;
while($row = pg_fetch_assoc($query)) {
 $result[$i] = $row;
 $i++;
}
```

Wenn Ihr PHP-Skript jetzt ausgeführt wird, ist es durchaus möglich, dass Sie eine Fehlermeldung wie *Query failed: FEHLER: keine Berechtigung für Relation products bekommen*. Das liegt daran, dass alle Objekte in der Datenbank, die soeben erstellt wurden, durch den User *pgadmin* angelegt wurden. Nun ist es aber so, dass zunächst nur der Eigentümer eines Objekts, in diesem Fall *pgadmin*, etwas damit anstellen darf. Damit auch andere User diese Objekte nutzen können, muss der Besitzer Privilegien erteilen. Sie können nun explizit die Rechte für ein Objekt vergeben, indem Sie z.B. mit

```
GRANT select, insert, update, delete ON products TO
                                                shop_user;
```

dem Datenbanksystem mitteilen, welcher Benutzer mit welchen Rechten auf bestimmte Objekte zugreifen darf.

Transaktionen

Ein zentrales Konzept von Datenbanken sind Transaktionen. Mit ihnen ist es möglich, mehrere Schritte zu einer einzigen Operation zusammenzufassen. Diese Operation wird als „atomar" betrachtet, was nichts anderes bedeutet, als dass sie

entweder ganz oder gar nicht ausgeführt wird. Was bedeutet das in der Praxis?

Nehmen wir an, wir hätten in der Tabelle *customers* eine zusätzliche Spalte, in der wir Bonuspunkte speichern. Jetzt möchten wir aus irgendeinem unerfindlichen Grund einem Kunden 100 Punkte abziehen und einem anderen Kunden gutschreiben. Was wir tun müssten, ist also Folgendes:

```
UPDATE customers set bonus = bonus - 100 where id = 1;
UPDATE customers set bonus = bonus + 100 where id = 2;
```

Ein zugegebenermaßen recht einfaches Beispiel. Trotzdem – was passiert, wenn zwischen den beiden Statements irgendetwas schief geht, z.B. weil es den Kunden mit der ID *2* nicht gibt oder unser Server abstürzt? Dann hätte der erste Kunde 100 Bonuspunkte weniger, der zweite aber keine zusätzlichen 100, da diese sich zwischenzeitlich ins Nirvana verabschiedet haben. Deshalb ist es wichtig, solche Operationen in einer Transaktion zu kapseln:

```
BEGIN;
UPDATE customers SET bonus = bonus - 100 WHERE id = 1;
UPDATE customers SET bonus = bonus + 100 WHERE id = 2;
COMMIT;
```

Das Wörtchen *BEGIN* kennzeichnet den Anfang der Transaktion. Danach kommen die auszuführenden Statements. Ist alles glatt gegangen, wird ein *COMMIT* ausgelöst und alle Änderungen werden dauerhaft gespeichert. Dabei garantiert uns das Datenbanksystem, dass alle notwendigen Änderungen auf ein dauerhaftes Speichermedium, in der Regel also die Festplatte, geschrieben wurden, bevor die Transaktion als abgeschlossen bestätigt wird. Geht hingegen etwas schief, löst die Datenbank ein *ROLLBACK* aus und der Ursprungszustand der Daten wird wieder hergestellt.

Wenn wir jetzt jedoch in unserem PHP-Code per *pg_query($dbh, $qs);* ein Statement ausführen lassen, ist es leider so, dass PostgreSQL zunächst jeden Befehl innerhalb einer Transaktion ausführt. Zwei Befehle, zwei Transaktionen – nützt uns also leider gar nichts. Eine Möglichkeit, dies elegant zu lösen, ist es, zwei PHP-Methoden zu schreiben, die wir mit einigen anderen in einer speziellen Datenbankklasse hinterlegt haben. Die erste Methode ist recht einfach und soll nur einem Array unsere Statements hinzufügen:

```
...
private $sql = array();
...
public function sqlAdd($qs) {
        array_push($this->sql, $qs);
}
```

Die zweite Methode soll dieses Array von Statements dann innerhalb einer Transaktion abarbeiten und somit sicherstellen, dass entweder alle Statements oder keines ausgeführt wird (Listing 4). Das *COMMIT* wird hierbei nur dann ausgelöst, wenn *pg_query* keinen Fehler gemeldet hat. Um ein explizites *ROLLBACK* kümmert sich der Datenbankserver selbst. Für das obige Beispiel würde das folgenden Code bedeuten:

```
$db->sqlAdd("UPDATE customers SET bonus = bonus - 100
                                        WHERE id = 1");
$db->sqlAdd("UPDATE customers SET bonus = bonus + 100
                                        WHERE id = 2");
$result = $db->transaction();
```

Fazit

Wir haben Ihnen gezeigt, dass der Umstieg von MySQL nicht schwer ist. Klar ist, dass man, wie bei allen Architekturänderungen, ein gewisses Maß an Zeit investieren muss, um mit der neuen Materie vertraut zu werden. Es bleibt festzuhalten, dass beide Datenbanken ihre Vorteile haben. MySQL in der Performance, was viele *SELECT*-Abfragen angeht und PostgreSQL, was den Funktionsumfang anbetrifft. Unsere eigene Erfahrung hat gezeigt, dass wir PostgreSQL für unsere Projekte nicht mehr missen möchten. In einem Forum für PostgreSQL war nach einer langen Diskussion, ob einer der Beteiligten den Umstieg wagen soll oder nicht, zu lesen: „Also je länger ich mich mit PostgreSQL beschäftige, desto besser finde ich sie." Antwort eines Moderators: „Und wieder einer !". In diesem Sinne – haben Sie Mut und „checken" Sie PostgreSQL aus – vielleicht wollen auch Sie sie nicht mehr missen.

Fensterbank

Wenn Sie den PostgreSQL-Server unter Windows einrichten möchten, verwenden Sie den PostgreSQL-Installer für Windows. Die aktuelle Version erhalten Sie auf den PostgreSQL-Downloadseiten. Geben Sie dem standardmäßigen User *postgres* für den User, unter dem der Datenbankserver laufen wird, ein Passwort und bestätigen Sie gegebenenfalls, dass der PostgreSQL-Server als Dienst eingerichtet wird, wenn Sie möchten, dass die Datenbank automatisch mit dem System gestartet wird.

Im nächsten Schritt „Datenbankcluster initialisieren" können Sie dann bereits bestimmen, ob Ihr Server auch Verbindungen für andere Server als *localhost* annehmen soll. Als Name für den Datenbank-Superuser können Sie hier dann entweder den vorgeschlagenen user postgres mit einem anderen Passwort verwenden oder, anlehnend an unsere Linux-Installation, den User *pgadmin* eintragen.

Um die vorgestellten Beispiele nachvollziehen zu können, sollten Sie im Folgenden der Installation der Sprache PL/pgSQL zustimmen. Die *Contrib*-Module bieten zusätzliche, teils sehr spezielle Funktionalität für den Datenbankserver. Falls Sie später *tsearch2* verwenden möchten, sollten Sie hier neben dem Administratorpaket auch noch B-Tree GiST zur Installation auswählen.

Der Datenbankserver sollte dann nach erfolgreicher Installation bereits gestartet werden. Angenehm ist hierbei, dass PgAdmin bereits mit installiert wird. Die Konfigurationsdateien werden dann in der Regel unter *C:\Programme\PostgreSQL\[Version]\data* zu finden sein, die Log-Datei unter *C:\Programme\PostgreSQL\[Versionsnummer]\data\pg_log.*

Thomas Pfeiffer & Andreas Wenk

Thomas Pfeiffer und Andreas Wenk sind Applikationsentwickler bei der NMMN – New Media Markets & Networks GmbH in Hamburg. Dort entwickeln sie das Ressourcen-Management-System eUNIQUE unter heftigem Einsatz von PHP und PostgreSQL. Sie erreichen die beiden unter tp@nmmn.com und aw@nmmn.com, oder über die Webpräsenz www.e-unique.com.

Links & Literatur

[1] http://www.postgresql.org/files/documentation/books/pghandbuch/html/client-authentication.html

[2] http://www.pgadmin.org/