Teil 2: Einen PostgreSQL-Cluster tunen

PostgreSQL-Fine-Tuning

Die PostgreSQL-Datenbank hat für die Konfiguration einige Rädchen, an denen man drehen kann. Um die beste Performance zu erreichen, müssen Sie wissen, in welche Richtung die Rädchen gedreht werden wollen. Darüber hinaus sollten Sie wissen, wie Sie eine PostgreSQL-Abfrage analysieren können.

▸ von Andreas Wenk

Im ersten Teil dieser Serie über die PostgreSQL-Datenbank [1] habe ich Ihnen gezeigt, wie Sie die PostgreSQL auf unterschiedlichen Systemen installieren und so konfigurieren, dass Sie damit arbeiten können. Sie haben auch gesehen, dass das nicht wirklich kompliziert ist. Dieser Mythos ist seitdem „officially busted". Und weil ich darin nun Übung habe, werde ich zu Anfang dieses Artikels auch gleich noch einmal ein Gerücht aus der Welt schaffen: Die PostgreSQL ist nicht lahm bzw. weist eine schlechte Performance auf. Ganz im Gegenteil. Richtig konfiguriert ist die PostgreSQL in vielen Einsatzgebieten wesentlich stabiler und performanter als z. B. eine MySQL-Datenbank. Ich habe auch bereits erwähnt, dass die Einstellungen der Konfigurationsparameter (in der *postgresql.conf*) out of the box sehr konservativ gehalten sind. Deshalb ist es unumgänglich, dass Sie die Einstellungen auf Grundlage Ihrer Applikation und der Hardware, auf der der Datenbankcluster laufen wird, anpassen. Wie Sie das tun, zeige ich Ihnen in diesem Artikel. Weiter unten werde ich Ihnen außerdem zeigen, welche Tools die PostgreSQL bietet, um eine Abfrage (Query) zu analysieren: namentlich *EXPLAIN* und *EXPLAIN ANALYZE*. Gewappnet mit diesem Wissen werden Sie in der Lage sein, die PostgreSQL auch für Anwendungen mit ein „bisschen mehr Bums dahinter" zu konfigurieren.

Welche Bereiche sind wichtig für das Thema „Performance"?

Lassen Sie uns kurz überlegen, welche Bereiche wir zum Thema „Performance" betrachten müssen. Zuerst wäre da die Hardware. Ich möchte gerne meinen Kollegen

Andreas Putzo zitieren: „Viel RAM ist gut – mehr RAM ist besser!". In der Tat ist es so, dass die PostgreSQL (und nicht nur dieses DBMS) so viel wie möglich RAM nutzt, um diverse Jobs auszuführen. Denn sobald die Festplatte (HD = Hard Disk) für Operationen genutzt werden muss, erfolgt der Zugriff rein technisch wesentlich langsamer. Deshalb ist es keine Seltenheit, dass ein Server mit 16, 32 oder 64 GB RAM ausgestattet ist. Für die Speicherung der Daten sind dann wiederum schnelle HDs wichtig, um den Zugriff so schnell wie möglich ausführen zu können. Außerdem wird ein Raid-1 oder Raid-10 empfohlen. Von Raid-5 wird abgeraten. Und zu guter Letzt sollte eine schnelle CPU auch nicht fehlen. Gerade bei Sortieroperationen wird die CPU stark beansprucht und sollte deshalb ausreichend groß dimensioniert sein.

Der nächste Punkt betrifft das Datenbankdesign. Es gibt viel Literatur zum Thema Datenbankdesign und Fehler, die gemacht werden können. Das Thema „Normalisierung" ist Ihnen sicherlich ein Begriff, wenn Sie mit der Konzeption und dem Design von Datenbanken in einem RDBMS vertraut sind. Allerdings gilt hier auch die Kosten-Nutzen-Rechnung. Eine Normalisierung nach allen Regeln der Kunst bis auf die letzte Möglichkeit (sprich alles in unterschiedlichen Tabellen auszulagern) auszureizen, führt definitiv zu Abfragen mit vielen JOIN-Statements. Das ist der Performance sicherlich nicht dienlich.

Ein weiterer wichtiger Punkt ist die zu erwartende Datenmenge und in diesem Zusammenhang der Einsatz von Indizes. Diese dienen dazu, den Zugriff auf die Daten zu erleichtern, indem die Daten z. B. durch einen B-Tree-Index (bei skalaren Datentypen) oder einen GIN- oder GiST-Index (bei der Volltextsuche) indiziert werden. Im einfachsten Fall können Sie sich einen Index, den Sie auf die Daten einer Tabellenspalte setzen, wie den Index eines Buchs vorstellen. Über die Suche im Index kann sehr schnell auf die entsprechenden Daten zugegriffen werden. Deshalb ist es sehr wichtig, Indizes einzusetzen und das dann auch noch mit Bedacht zu tun.

Auf keinen Fall zu unterschätzen ist die Query-Optimierung (Query = Abfrage). Wie schon angesprochen, werde ich weiter unten auf *EXPLAIN* und *EXPLAIN ANALYZE* eingehen. Diese beiden Befehle helfen Ihnen, einen Query-Plan zu lesen und aufgrund der Ergebnisse Ihre SQL-Query zu optimieren. Gleich im nächsten Abschnitt beschreibe ich (soweit es in der Kürze dieses Artikels möglich ist) den *VACUUM*-Mechanismus der PostgreSQL. Dieser ist sozusagen der Aufräum- und Ordnungsmechanismus. Halten wir kurz die Grundregeln für gute Performance fest:

- gute und schnelle Hardware zur Verfügung stellen
- die Parameter in der *postgresql.conf* aufgrund der benutzen Hardware und den Anforderungen an die Anwendung anpassen und ggf. erhöhen
- gute SQL-Queries schreiben
- Indizes richtig einsetzen
- die Statistiken aktuell halten und Fragmentierung von Tabellen und Indizes vermeiden

VACUUM und VACUUM FULL

Im Kasten „Die Aufgaben des *VACUUM*-Befehls" habe ich kurz zusammengefasst, was *VACUUM* tut. Um die Liste und Aufgaben besser zu verstehen, sollten Sie verstehen, was beim Schreiben oder Aktualisieren von Daten einer Tabelle passiert. Beachten Sie noch vorab, dass der *auto-vacuum Daemon* regelmäßig den Befehl *VACUUM* und *ANALYZE* ausführt.

Die PostgreSQL-Datenbank setzt auf das MVCC-Modell. Lesen Sie im Kasten „MVCC – Multiversion Concurrency Control" eine Definition. Das in der PostgreSQL eingesetzte MVCC-Modell hat nun für die Wartung der Datenbank direkte Auswirkungen. Wenn Daten in einer Tabelle verändert oder gelöscht werden (*INSERT*, *UPDATE*, *DELETE*), werden diese nicht überschrieben, sondern es wird eine neue Version erstellt und mit den ursprünglichen Daten verknüpft. Es werden also für einen gewissen Zeitraum alle Versionen der Daten aufbewahrt (im so genannten Heap der Tabelle), damit andere Transaktionen, die bei der Änderung der aktuellen Daten laufen, noch den richtigen Stand der Daten „sehen" bzw. verwenden können. Allerdings gibt es dort nun Versionen, die für keine Transaktion mehr sichtbar

FSM – die Free Space Map

In der Free Space Map wird der Speicher verwaltet, der in den Tabellen durch das entfernen veralteter Versionen von Zeilen frei ist. Standardmäßig biete die FSM Platz für 1000 Objekte. Diese Anzahl der Datenbankseiten, die in die FSM aufgenommen werden können, kann durch den Parameter *max_fsm_pages*, und die Anzahl von Tabellen und Indizes kann durch den Parameter *max_fsm_relations* in der *postgresql.conf* geändert werden. Dabei muss man wissen, dass diese globalen Einstellungen also für alle Datenbanken im Cluster gelten.

Beachtet werden muss bei der FSM, dass immer genug Objekte aufgenommen werden können. Denn wenn die FSM „voll" ist, werden Tabellen automatisch immer größer und größer und fragmentiert. (Hinweis: Die beiden Parameter wurden aus der Version 8.4 entfernt: „Free space discovered by VACUUM is now recorded in **_fsm* files, rather than in a fixed-sized shared memory area. The *max_fsm_pages* and *max_fsm_relations* settings have been removed, greatly simplifying administration of free space management.").

MVCC – Multiversion Concurrency Control

Multiversion Concurrency Control ist ein Modell, das es unterschiedlichen Benutzern ermöglicht, gleichzeitig auf die Datenbank zuzugreifen. Die Daten, auf die zugegriffen wird, werden dabei nicht gesperrt (*locking*), sondern die vom Benutzer gestartete Transaktion betrachtet einen aktuellen *snapshot* der Datenbank und seiner Daten. Änderungen an der Datenbank durch den Benutzer werden erst nach erfolgreichem Beenden seiner gestarteten Transaktion für andere Benutzer sichtbar.

Die Aufgaben des VACUUM-Befehls

Der VACUUM-Befehl führt mehrere, wichtige Aufgaben durch:

- Tabellen und Indexe defragmentieren und reorganisieren
- Tabellen von Dead Tuples (gelöschte Zeilen) bereinigen
- für reibungslosen Ablauf von Transaktionen sorgen
- Statusinformationen aktualisieren

Seit der PostgreSQL 8.3 ist der Parameter *autovacuum* in der *postgresql.conf* standardmäßig auf *on* gesetzt.

sind und somit veraltet sind. *VACUUM* untersucht nun den *Heap* und legt diese nicht mehr benötigten Versionen in der sog. Free Space Map (FSM) ab (Kasten: „FSM – Die Free Space Map"). Sollen nun Daten in die Tabelle geschrieben oder aktualisiert werden (*INSERT*, *UPDATE*), wird zuerst in der FSM nachgesehen, ob es in der Tabelle einen freien Bereich gibt, der genutzt werden kann. Die FSM verwaltet also den freien Speicherplatz aller Tabellen. Gibt es keinen Platz, werden die Daten einfach ans Ende der Tabelle geschrieben.

VACUUM FULL geht noch einen Schritt weiter. Der Heap der Tabelle wird hierbei komplett neu organisiert. Das heißt, freie Speicherbereiche werden mit den vorher sortierten, vorhandenen Daten der Reihe nach aufgefüllt. Sind alle Möglichkeiten ausgeschöpft, wird der Heap am Ende kleiner sein und nicht mehr so viel Speicherplatz benötigen. Deshalb wird der gewonnene Speicher an das Betriebssystem zurückgegeben. Man kann diesen Vorgang ansatzweise mit der Defragmentierung eines FAT-Dateisystems vergleichen. Beachten Sie unbedingt, dass bei *VACUUM FULL* im Gegensatz zum Befehl *VACUUM* die Tabellen gesperrt werden (*locking*), da am physikalischen Speicher gearbeitet wird. Deshalb sollte *VACUUM FULL* auch nur eingesetzt werden, wenn es nicht mehr anders geht. Wenn der *VACUUM*-Daemon regelmäßig und in den richtigen Zeitabständen läuft (*autovacuum*), sollte *VACUUM FULL* nicht notwendig sein. Zusammenfassend: Achten Sie immer darauf, dass *VACUUM* ordnungsgemäß und regelmäßig ausgeführt wird. Das hat erheblichen Einfluss auf die Performance der Datenbank. Fragmentierte Tabellen oder Indizes verschlechtern die Performance merklich. Sollte das einmal der Fall sein, hilft nur noch ein *VACUUM FULL* in einem Wartungsfenster (das bringt's dann allerdings). Zusammengefasst sind die Parameter *max_fsm_relations* und *max_fsm_pages* für die Einstellungen der FSM und die Parameter *vacuum_cost_limit* und *vacuum_cost_delay* für den *VACUUM*-Daemon von großer Bedeutung. Mit den zuletzt genannten Parametern können Sie *VACUUM* für eine gewisse Zeit aussetzen, wenn der Prozess die Performance nach unten zieht (die entstehenden „Kosten" werden aufgrund der I/O-Auslastung berechnet, und bei erreichen des Limits, bestimmt durch *vacuum_cost_limit*, für die Zeit, bestimmt durch *vacuum_cost_delay*, ausgesetzt).

Ausgabe 1

```
postgres=# SHOW work_mem;
 work_mem
----------
 1MB
postgres=# SET work_mem TO '24MB';
SET
postgres=# SHOW work_mem;
 work_mem
----------
 24MB
postgres=# RESET work_mem;
RESET
postgres=# SHOW work_mem;
 work_mem
----------
 1MB
```

Einstellungen in der postgresql.conf

Die Datei *potgresql.conf* kann getrost als Herzstück des PostgreSQL-Servers betrachtet werden. Sie ist sehr umfangreich und hat sehr viele Schrauben (Einstellungsparameter), an denen gedreht werden kann. Die gute Nachricht ist, dass Sie mit den richtigen Einstellungen einen sehr gut funktionierenden Datenbankcluster haben werden. Die schlechte ist, dass Sie mit den falschen Einstellungen einen unbrauchbaren erhalten werden.

Ich werde Ihnen die wichtigsten Parameter vorstellen und teilweise bzw., wenn möglich, einen allgemeinen Richtwert oder eine Empfehlung der Größenordnung für den jeweiligen Parameter angeben. Die Richtwerte sind nach allgemeinen Erfahrungen der PostgreSQL Contributor und der Community entstanden. Viele davon können Sie auch in der Onlinedokumentation [2] wiederfinden. Die Bezeichnung „Richtwert" sollte wörtlich genommen werden, denn Sie werden einige Parameter etwas genauer an Ihre Serverumgebung und Applikation anpassen wollen. Im Weiteren werden Sie dann das Programm *pgtune* kennenlernen und einsetzen, um einen an Ihre Hardware und Umgebung ausgerichteten Vorschlag für einige Einstellungen zu erhalten.

Beachten Sie, dass die Änderungen einen *reload* oder teilweise sogar einen *restart* des PostgreSQL-Clusters erfordern. Hier zur Erinnerung noch einmal die entsprechenden Kommandos:

$ /usr/bin/pg_ctlcluster 8.4 main reload bzw. *$ /usr/bin/pg_ctl -D /usr/local/pgsql/data reload* bzw. *$ /usr/bin/pg_ctlcluster 8.4 main restart* bzw. *$ /usr/bin/pg_ctl -D /usr/local/pgsql/data restart*.

Ausgabe 2

```
php_mag_03_10=# SHOW maintenance_work_
                                    mem;
 maintenance_work_mem
----------------------
 16MB
php_mag_03_10=# BEGIN;
BEGIN
php_mag_03_10=# SET LOCAL maintenance_work_
                                mem TO '128MB';
SET
php_mag_03_10=# CREATE INDEX idx_
          bestellungen_id ON bestellungen (id);
CREATE INDEX
php_mag_03_10=# SHOW maintenance_work_
                                    mem;
 maintenance_work_mem
----------------------
 128MB
php_mag_03_10=# COMMIT;
COMMIT
php_mag_03_10=# SHOW maintenance_work_
                                    mem;
 maintenance_work_mem
----------------------
 16MB
```

Während des Betriebs

Sie haben die Möglichkeit, Parameter, die zur Laufzeit änderbar sind (also kein *reload* oder *restart* des Clusters benötigen), anzusehen, zu setzen und wieder zurück zu setzen. Sie können das auch nur pro Transaktion tun. Die Befehle sind *SHOW*, *SET* und *RESET*. Ein einfaches Beispiel ist es, den Parameter *work_mem* zu erhöhen (etwas gekürzte Ausgabe aus *psql*), wie Ausgabe 1 zeigt.

Wenn Sie z. B. einen Index erstellen wollen und dafür den Parameter *maintenance_work_mem* nur für diesen Vorgang erhöhen wollen, um mehr RAM zur Verfügung zu stellen, kapseln Sie das in eine Transaktion und benutzen den Befehl

- businesskritische Anwendungen
- PHP-Webanwendungs-Entwicklung
- Softwarearchitekturplanung für skalierbare Systeme
- Beratung (Datenbank, Architektur & Qualität)
- Contributor bei Standard-Entwicklungswerkzeugen wie PHPUnit oder phpUnderControl
- Hersteller von Open Source Software

- weitgreifende Projekterfahrung
- regelmäßige Fortbildung zu fachspezifischen Themen
- agiles Projekt Management mit Scrum
- Kooperation mit Hochschulen
- Zertifizierungen (PHP5, MySQL5, CSM)

MAYFLOWER

loyale Kunden & langfristige Geschäftsbeziehungen

AGILITÄT & EXZELLENZ

Und was können wir für Sie tun?

Kontaktieren Sie uns unter http://www.mayflower.de/de/kontakt
Wir entwickeln auch für Ihr Unternehmen maßgeschneiderte IT-Lösungen.

SET LOCAL, sodass der Parameter nur für die Laufzeit der Indexerstellung geändert wird. Das ist in Ausgabe 2 zu sehen.

Sehen wir uns also die wichtigsten Parameter im Einzelnen an (der jeweilige Bereich wird als Überschrift angegeben).

RESOURCE USAGE (except WAL)

shared_buffers = 32MB: Dieser Parameter gibt an, wie viel gemeinsam genutzter Speicher (Shared Memory) dem PostgreSQL-Server zur Verfügung steht (Shared Buffer Pool der PostgresSQL). 10 – 25 % des RAM Ihres Systems werden für eine gute Performance sorgen. Bei sehr vielen Schreibzugriffen sind auch bis zu 50 % denkbar.

work_mem = 1MB: Die Einstellung für *work_mem* ist eine Obergrenze. Der Parameter gibt an, wie viel RAM für Datenbankoperationen wie Sortieroperationen oder bestimmte JOIN-Typen wie Hash-Joins und Bitmap Index Scans benötigt werden. Beachten Sie, dass die PostgreSQL auf die Festplatte ausweicht, wenn nicht genug Arbeitsspeicher zur Verfügung steht. Insofern müssen Sie den Parameter je nach Hardware und Anwendung deutlich erhöhen (siehe *pgtune* weiter unten).

maintenance_work_mem = 16MB: Mit *maintenance_work_mem* wird die Obergrenze angegeben, wie viel RAM für Wartungsarbeiten wie *VACUUM*, *CREATE INDEX*, *ALTER TABLE* und *CLUSTER* zur Verfügung steht. Hier muss also auch wieder abgewogen werden, wie viel die PostgreSQL aufräumen muss. Werden z. B. viele löschende Transaktionen ausgeführt, entstehen fragmentierte Tabellen, die „vacuumisiert" werden wollen – ergo sollte *maintenance_work_mem* höher gesetzt werden. Außerdem ist die Anpassung dieses Werts hilfreich bei der Erstellungen von Indizes (Beispiel weiter oben).

max_fsm_pages = 153600: Dieser Parameter legt fest, wie viele Datenbankseiten in der so genannten Free Space Map (FSM) abgelegt werden können. Wie schon beschrieben, werden in der FSM „Zeiger" auf von *VACUUM* freigegebenem Speicher verwaltet. Dieser Parameter muss immer groß genug angegeben werden, denn sonst müssen Dienste wie *VACUUM FULL* oder *REINDEX* zu oft ausgeführt werden. Berechnung: *max_fsm_relations * 16* (jede Datenbankseite hat eine Größe von 6 Byte). Dieser Parameter sollte auf Grund des zur Verfügung stehenden Speichers bzw. anhand des verbrauchten Speichers der größten Tabellen eingestellt werden. Informationen zum aktuellen Stand erhalten Sie durch ausführen des Befehls *VACUUM VERBOSE (INFO: …, DETAIL: …* bei der Ausgabe) (ab 8.4 entfernt).

max_fsm_relations = 1000: Dieser Wert gibt die Anzahl von Tabellen und Indizes an, die in der FSM gespeichert werden können. Der Wert kann durchaus erhöht werden, wobei aber davon auszugehen ist, dass 1000 in den meisten Fällen ausreichend ist (ab 8.4 entfernt).

WRITE AHEAD LOG

Sobald Transaktionen vorgenommen werden, werden diese im so genannten *Write Ahead Log* (WAL) protokolliert. Damit wird sichergestellt, dass die Datenbank in einem konsistenten Zustand bleibt.

wal_buffers = 64KByte: Dieser Parameter beschreibt die Größe des Puffers für das Transaktions-Log. Der Wert ist im Normalfall ausreichend, kann aber wiederum bei großen Transaktionen erhöht werden.

checkpoint_segments = 3: Dieser Parameter beschreibt die Anzahl der Segmente im Transaktions-Log, wobei jedes Segment eine Größe von 16 MB hat. Die Anzahl von *3* gibt also an, dass der PostgreSQL drei Segmente für Datenänderungen innerhalb einer Transaktion zur Verfügung stehen, bevor ein Checkpoint (Kasten: „Was ist ein Checkpoint?") ausgelöst wird. Der Wert ist ziemlich gering und sollte auf 8 – 12 erhöht werden.

checkpoint_completion_target = 0.5: Der Parameter beschreibt die maximale Dauer eines Checkpoints. Der Wert kann von 0.0 bis 1.0 eingestellt werden. Mit einem Wert von 1.0 wird das Schreiben der Datenänderungen voll ausgebremst und mit 0.0 entsprechend gar nicht.

checkpoint_warning = 30s: Wenn Checkpoints in einem geringeren Abstand als dieser Wert ausgeführt werden, wird eine Meldung im Log eingetragen. Tritt der Fall ein, sollte *checkpoint_segments* erhöht werden.

AUTOVACUUM PARAMETERS

autovacuum = on: Seit der PostgreSQL 8.3 wird *VACUUM* automatisch ausgeführt. Lassen Sie die Einstellung immer auf *on*.

vacuum_cost_limit = 200, vacuum_cost_delay = 0ms: Mit *vacuum_cost_limit* wird der Schwellwert angegeben, bei dessen Erreichen der *VACUUM*-Prozess so lange angehalten wird, wie mit *vacuum_cost_delay* bestimmt. Der Schwellwert wird auf Grund der I/O-Auslastung berechnet. Mit diesen beiden Parametern können Sie also den *VACUUM*-Daemon bei zu hoher Auslastung Ihres Servers etwas ausbremsen.

QUERY TUNING

effective_cache_size = 128MB: Dieser Parameter ist sehr wichtig und teilt dem Planer mit, wie viel Arbeitsspeicher der PostgreSQL für ein Query zur Verfügung steht. Als Faustregel kann der Wert auf 50 – 75 % des gesamten RAM des Rechners gesetzt werden (*pgtune 75 %*). Sie sollten dabei einfach das Unix-Programm *free* aufrufen und das RAM betrachten, das gecacht ist:

```
$ free -m
total used free shared buffers cached
Mem: 995 961 33 0 103 702
…
```

Wenn Sie den Wert von *total* und *cached* ins Verhältnis setzen, liegen Sie mit der

Was ist ein Checkpoint?

Ein Checkpoint ist dafür da, um alle Änderungen im Transaktions-Log in die Datenbasis zu synchronisieren. Standardmäßig geschieht das alle fünf Minuten. Mit dem Parameter *checkpoint_timeout* können Sie dieses Intervall verringern oder vergrößern.

So wird ein Query-Plan gelesen

Ein Query-Plan wird von innen nach außen gelesen. Außerdem zeigen uns die Einrückungen der jeweiligen Planknoten, dass der weiter rechts eingerückte Planknoten ein Ergebnis für den darüber liegenden Planknoten liefert.

Faustregel ziemlich gut. Wichtig ist hier zu wissen, dass Daten aus dem RAM gelesen erheblich schneller zur Verfügung stehen, als von der „langsamen" Festplatte. Je höher Sie diesen Wert setzen, desto besser wird die Performance werden, allerdings auch nicht zu hoch, da der Planer sonst falsche Entscheidungen für die Ausführung einer Anfrage vornimmt. Dieser Parameter wird übrigens ausschließlich vom Planer genutzt (Achtung bei der Aussprache des Parameters).

Zwischenergebnis

Ich habe Ihnen hier die Parameter vorgestellt, die Sie auf jeden Fall kontrollieren und ggf. justieren sollten. Achten Sie darauf, dass Sie nicht zu viele Parameter auf einmal ändern. Haben Sie einen Bereich angepasst (z. B. WAL), testen Sie zuerst Ihren Cluster (gerade unter Performancegesichtspunkten) und fahren dann mit den weiteren Einstellungen fort. Es bietet sich auch an, ein Backup der original *postgresql.conf* vor den Änderungen anzulegen. Wenn Sie die *postgresql.conf* ansehen (zu finden – je nach Installationsart – unter Debian und per aptitude installiert z. B. in */etc/postgresql/8.4/main/*), werden Sie natürlich weitaus mehr Parameter finden. Diese können hier nicht alle besprochen werden. Sie können in der Onlinedokumentation [2] mehr dazu lesen. Übrigens gibt es einige Meinungen in der PostgreSQL-Community, dass es zu viele Parameter gibt und einige entfernt werden sollten. Wir werden sehen, was in zukünftigen PostgreSQL-Versionen geschieht.

Manche in der PostgreSQL-Community meinen, es gebe zu viele Parameter in der postgresql.conf.

pgtune – die postgresql.conf erstellen lassen

pgtune [3] ist ein von Greg Smith (PostgreSQL Contributor) in Python geschriebenes CLI-Programm und liegt beim Schreiben dieser Zeilen in Version 0.9.3 vor. Prinzipiell erstellt das Programm für Sie eine alternative *postgresql.conf*-Datei auf Grundlage der im Cluster vorhandenen Datei. Dabei wird zum einen die Hardware Ihres Rechners/Servers untersucht, und Sie haben die Möglichkeit, dem Programm einige grundlegende Optionen mitzugeben.

Die Installation ist trivial. Laden Sie das mit *gzip* komprimierte *tar*-Archiv von [3] herunter oder, wenn Sie einen aktuellen Snapshot nutzen wollen, von [4] (ich habe diesen Snapshot beim Schreiben des Artikels genutzt: http://bit.ly/cBBeA6). Entpacken Sie das Archiv an irgendeine Stelle Ihres Servers und wechseln Sie in das Verzeichnis. Geben Sie jetzt wie üblich *./pgtune –h* ein, um die Hilfe aufzurufen. Das Ergebnis sehen sie in Ausgabe 3.

Am wichtigsten sind hier die Parameter *-i*, um den Pfad der vorhandenen *postgresql.conf* anzugeben, *-o*, um den Pfad für die alternative *postgresql.conf* anzugeben, und *-T*, um den Typ der Datenbank anzugeben. Die möglichen Optionen des Parameters *-T* sind *DW* (Data Warehouse [5]), *OLTP* (Online Transaction Processing [6]), *Web*, *Mixed* und *Desktop*. Je nachdem, welche Option Sie angeben, wird das Programm unterschiedliche Faktoren bei der Berechnung der Werte nutzen. Sie werden wohl immer entweder *OLTP* oder *Web* angeben. Ein gutes Beispiel ist die Berechnung des Parameters *work_mem* (folgend als Pseudocode gezeigt):

```
mem = ermittelter memory;
con = ermittelte max connections;
switch(db_type)
 case 'Web' or 'OLTP':
  work_mem = mem / con;
 case 'DW' or 'Mixed':
  work_mem = mem / con / 2;
 case 'Desktop':
 work_mem = mem / con / 6;
```

Wie Sie sehen, wird bei der Option DW *work_mem* noch um den Faktor zwei, und bei *Desktop* sogar um den Faktor 6 minimiert. Es wird also davon ausgegangen, dass für solche Typen weniger RAM benötigt wird oder zur Verfügung steht. Bei einigen anderen Parametern verhält sich das Programm genauso und

LISTING 1

```
CREATE TABLE produkte (
    id serial NOT NULL PRIMARY KEY,
    art_nr bigint NOT NULL,
    bezeichnung CHARACTER VARYING (255),
    beschreibung TEXT,
    stichwort TEXT
);
CREATE TABLE bestellungen (
 id serial NOT NULL PRIMARY KEY,
 bestell_dat TIMESTAMP WITHOUT TIME ZONE
);
CREATE TABLE bestellungen_produkte (
 bestellungen_id int NOT NULL,
  produkte_id int NOT NULL,
 menge int
);
```

LISTING 2

```
CREATE LANGUAGE plpgsql;
CREATE OR REPLACE FUNCTION create_produkt_
            dummy_data(integer) RETURNS void AS
$FUNC$
 DECLARE
 count ALIAS FOR $1;
 _cnt int := 1;
 BEGIN
 WHILE _cnt <= count LOOP
 INSERT INTO produkte (art_nr, bezeichnung,
                       beschreibung,  stichworte)
 VALUES (random() * 89764 * _cnt,'PostgreSQL
                                  Buch' || _cnt,
   'Beschreibung für das PostgreSQL Buch' || _cnt,
    'postgrsql,buch');
 _cnt := _cnt + 1;
 END LOOP;
END
$FUNC$ LANGUAGE 'plpgsql';
```

LISTING 3

```
EXPLAIN ANALYZE SELECT a.bestell_dat, b.menge,
                                  c.bezeichnung
FROM produkte c
LEFT JOIN bestellungen_produkte b ON
                          (c.id = b.produkte_id)
LEFT JOIN bestellungen a ON (a.id = b.bestellungen_
                                  id);
```

nutzt ähnliche Algorithmen. Um das Programm nun zu starten, geben Sie folgendes Kommando ein:

```
$ ./pgtune -i /etc/postgresql/8.4/main/postgresql.conf
--type="OLTP" -o postgresql-new.conf
```

Sie erhalten im Verzeichnis, in dem Sie sich gerade befinden, eine Datei namens *postgresql-new.conf*. Nutzen wir das Programm *diff*, um die Unterschiede zur originalen Datei anzusehen: *$ diff postgresql-new.conf /etc/postgresql/8.4/main/postgresql.conf*. Beachten Sie dabei, dass in diesem Fall die originale Datei „jungfräulich" war (Ausgabe 4).

Wie Sie sehen, wurde der Parameter *max_connections* und *shared_buffers* deaktiviert (mit # kommentiert). Außerdem wurden ans Ende der Datei acht Parameter mit entsprechenden berechneten Werten geschrieben. *pgtune* macht das der Übersicht wegen und um sehr einfach bzw. um auf einen Blick zu zeigen, welche Werte wie geändert wurden. In diesem Fall wurde z. B. der Wert für *max_connections* von *100* auf *200* und der Wert für *shared_buffers* von *28 MB* auf *120 MB* erhöht. Die anderen Parameter waren bislang nicht aktiviert und wurden von *pgtune* gesetzt.

Interessant ist der Unterschied der Werte, wenn Sie anstatt *-T 'Web' -T 'OLTP'* angeben. Ausgabe 5 zeigt das Ergebnis des folgenden *diffs*: *# diff postgresql-new.conf postgresql-new-oltp.conf*.

Spannend ist die Tatsache, dass *work_mem* reduziert wurde, *wal_buffers* und *checkpoint_segments* aber verdoppelt wurden. Dem anderen Datenbanktyp (hier *OLTP*) wird insofern Rechnung getragen, als mehr Wert auf Hochverfügbarkeit und Transaktionssicherheit gelegt wird.

pgtune hilft Ihnen, einen ersten Schritt in Richtung individuell angepasste Konfiguration für Ihren Datenbankcluster zu unternehmen. Die wichtigsten Parameter werden zum einen aktiviert und zum anderen auch noch so gesetzt, wie Ihre Hardware es zulässt. Nutzen Sie dieses großartige Programm, um einen guten, lauffähigen Cluster zu erhalten. Allerdings möchte ich an dieser Stelle auch erwähnen, dass das ein Anfang ist und nicht der Weisheit letzter Schluss. Das kann es auch gar nicht sein, denn viele wichtige Parameter werden nicht berücksichtigt. Es ist also weiterhin erforderlich, dass Sie sich mit der Konfiguration beschäftigen.

EXPLAIN ANALYZE – einen Query-Plan lesen

Sehen wir uns im Folgenden an, was *EXPLAIN*, *ANALYZE* und *EXPLAIN ANALYZE* tun und wie Sie einen Query-Plan lesen (lesen Sie auch den Kasten: „So wird ein Query-Plan gelesen"). Letztlich wird das dazu beitragen, den Planer besser zu verstehen und Performanceprobleme aufzudecken. Als Erstes lesen Sie sich am besten die Kurzerklärung im Kasten: „EXPLAIN, ANALYZE und EXPLAIN ANALYZE" durch, um einen Überblick zu erhalten.

Prinzipiell wird ein Query-Plan in Teilpläne (die Planknoten) untergliedert. Dabei werden zwei große Bereiche von Planknotentypen unterschieden: die Scan- und die Join-Typen. Es gibt folgende Scan-Typen:

- *Sequential Scan*: Alle Zeilen der Tabelle werden komplett durchlaufen – langsam.

Ausgabe 3

```
Usage: pgtune [options]
Options:
 --version       show program's version number and
                                                exit
 -h, --help      show this help message and exit
 -i INPUT_CONFIG, --input-config=INPUT_CONFIG
                 Input configuration file
 -o OUTPUT_CONFIG, --output-config=OUTPUT_
                                                CONFIG
                 Output configuration file, defaults to
                                        standard output
 -M TOTAL_MEMORY, --memory=TOTAL_MEMORY
          Total system memory, will attempt to
                                    detect if unspecified
 -T DB_TYPE, --type=DB_TYPE
          Database type, defaults to Mixed, valid
          options are DW, OLTP, Web, Mixed, Desktop
 -c CONNECTIONS, --connections=CONNECTIONS
          Maximum number of expected connections,
                default depends on database type
 -D, --debug     Enable debugging mode
 -S SETTINGS_DIR, --settings=SETTINGS_DIR
          Directory where settings data files are
                                              located at.
          Defaults to the directory where the script
                                    is being run from
 --doctest       run doctests
```

Ausgabe 4

```
64c64
< # max_connections = 100  # (change requires
                                              restart)
---
> max_connections = 100  # (change requires
                                              restart)
106c106
< # shared_buffers = 28MB  # min 128kB
---
> shared_buffers = 28MB  # min 128kB
499,506d498
< maintenance_work_mem = 30MB # pgtune wizard
                                          2010-01-31
< checkpoint_completion_target = 0.7 # pgtune
                                        wizard 2010-01-31
< effective_cache_size = 352MB # pgtune wizard
                                          2010-01-31
< work_mem = 2560kB # pgtune wizard 2010-01-31
< wal_buffers = 4MB # pgtune wizard 2010-01-31
< checkpoint_segments = 8 # pgtune wizard
                                          2010-01-31
< shared_buffers = 120MB # pgtune wizard
                                          2010-01-31
< max_connections = 200 # pgtune wizard
                                          2010-01-31
```

Ausgabe 5

```
500c500
< checkpoint_completion_target = 0.7 # pgtune
                                        wizard 2010-01-31
---
> checkpoint_completion_target = 0.9 # pgtune
                                        wizard 2010-01-31
502,504c502,504
< work_mem = 2560kB # pgtune wizard 2010-01-31
< wal_buffers = 4MB # pgtune wizard 2010-01-31
< checkpoint_segments = 8 # pgtune wizard
                                          2010-01-31
---
> work_mem = 1664kB # pgtune wizard 2010-01-31
> wal_buffers = 8MB # pgtune wizard 2010-01-31
> checkpoint_segments = 16 # pgtune wizard
                                          2010-01-31
506c506
< max_connections = 200 # pgtune wizard
                                          2010-01-31
---
> max_connections = 300 # pgtune wizard
                                          2010-01-31
```

- *Indexscan*: Spalte einer Tabelle hat einen Index, und diese wird im SQL-Statement abgefragt.
- *Bitmap Index Scan*: Alle Zeilen werden in einer Bitmap im RAM in der Reihenfolge, wie sie in der Tabelle vorkommen, zwischengespeichert – dann wird geprüft, ob die Zeile für die aktuelle Transaktion überhaupt sichtbar ist, und das tatsächliche Ergebnis wird zurückgegeben.

Welcher Scan-Typ genutzt wird, hängt davon ab, ob, und wenn ja, welche Indizes in den einzelnen Tabellen und den abzufragenden Spalten vorhanden sind. Dabei kann ein Sequential Scan immer genutzt werden. Bei großen Abfragen und der Nutzung eines Bitmap Index Scan ist es übrigens gut, viel RAM zu haben und die Einstellungen für *work_mem* in der *postgresql.conf* entsprechend hoch zu stellen. Der Planer wird aufgrund der gefundenen Scan-Typen einen Join-Typ für zwei Tabellen erstellen. Wenn Joins über mehrere Tabellen durchgeführt werden sollen, wird der Planer einen Baum mit Joins (Join Tree) erstellen, in dem dann immer über zwei Tabellen ein Join ausgeführt wird. Join-Typen sind die folgenden:

- *Nested Loop Join*: Zwei in einer Schleife verschachtelte Tabellen – linke Tabelle wird einmal durchsucht, wobei dabei pro Zeile jede Zeile der rechten Tabelle durchsucht wird und die gegebene Bedingung geprüft wird.
- *Hash Join*: Die Kleinere Tabelle wird zuerst durchlaufen und daraus eine Hash-Tabelle im Speicher erstellt, in der ein Schlüssel abgelegt wird. Beim Durchlaufen der größeren Tabelle wird bei zutreffender Bedingung über den Schlüssel in der Hash-Tabelle der entsprechend passende Datensatz in der kleineren Tabelle gesucht.
- *Merge Join*: Beide Tabellen werden im ersten Schritt nach den für die jeweilige Tabelle vorliegenden Bedingungen aus dem SQL-Statement sortiert. Dann werden die beiden Tabellen parallel durchlaufen und die entsprechenden zu einander gehörenden Datensätze gelesen.

Leider würde es den Rahmen dieses Artikels deutlich sprengen, um auf die einzelnen Scan- und Join-Typen näher einzugehen. Es soll an dieser Stelle ausreichen, dass es diese gibt und dass der Planer eine Kombination aus Scan- und Join-Typ für die Abfrage erstellt.

Wenn Sie nun einen Query-Plan ansehen, können Sie analysieren, ob der Planer aufgrund Ihres Queries einen guten Plan genutzt hat und ob die Performance gut ist. Wenn nicht, hat „meist" nicht der Planer etwas falsch gemacht, sondern Sie müssen an Ihrer Abfrage etwas verändern oder am generellen Setup Ihres Datenbankclusters. Das betrifft die Einstellungen der *postgresql.conf* genauso wie die eingesetzte Hardware. Erstellen wir uns nun für Anschauungszwecke ein paar Tabellen (Listing 1).

Ich habe hier der Einfachheit halber auf *Foreign Key Constraints* verzichtet, da diese hier nicht relevant sind. In Listing 7 sehen Sie noch eine kleine *User De-*

EXPLAIN, ANALYZE und EXPLAIN ANALYZE

- *EXPLAIN*: wird schlicht und ergreifend genutzt, um einen Query-Plan zu generieren und auszugeben. Er enthält geschätzte Werte des Planers für den Ausführungsaufwand und gibt an, welche Scan- und Join-Typen der Planer vorsieht.
- *ANALYZE*: sammelt Statistiken über die Inhalte der Tabellen in der Datenbank und speichert diese in der Tabelle *pg_statistics*. Der Planer erstellt auf Grundlage dieser Statistiken den bestmöglichen Query-Plan. Der *VACUUM DAEMON* nutzt *ANALYZE*, wenn sich Tabellen sehr verändert haben.
- *EXPLAIN ANALYZE*: dient dazu, um das Query tatsächlich auszuführen und einen entsprechend erweiterten *Query-Plan* zu erstellen. Beachten Sie, dass das keine Kombination aus *EXPLAIN* und *ANALYZE* ist, sondern *ANALYZE* in diesem Fall eine Option des Befehls *EXPLAIN* ist. Wenn Sie einen Query-Plan aufgrund aktueller Statistiken betrachten wollen, sollten Sie vorher *ANALYZE* (und ggf. auch *VACUUM*) ausführen. Dadurch, dass die Query tatsächlich ausgeführt wird, kann die Laufzeit unter Umständen sehr lange dauern.

Ausgabe 6

```
QUERY-PLAN
------------------------------------------------------------
Merge Left Join  (cost=218.09..4355.45
                 rows=100000 width=32) (actual time=
                 0.057..119.536 rows=100000 loops=1)
                 Merge Cond: (c.id = b.produkte_id)
 -> Index Scan using produkte_pkey on produkte
    c  (cost=0.00..3858.26 rows=100000 width=24)
(actual time=0.020..56.150 rows=100000 loops=1)
 -> Sort  (cost=218.09..222.94 rows=1940
      width=16) (actual time=0.031..0.032 rows=3
                                      loops=1)
    Sort Key: b.produkte_id
    Sort Method:  quicksort  Memory: 17kB
    -> Hash Left Join  (cost=53.65..112.15
                       rows=1940 width=16) (actual
                  time=0.019..0.023 rows=3 loops=1)
       Hash Cond: (b.bestellungen_id = a.id)
       -> Seq Scan on bestellungen_produkte b
(cost=0.00..29.40 rows=1940 width=12)
                       (actual time=0.003..0.003 rows=
                                      3 loops=1)
       -> Hash  (cost=29.40..29.40 rows=1940
                width=12) (actual time=0.008..0.008
                                      rows=3 loops=1)
          -> Seq Scan on bestellungen a  (cost=
             0.00..29.40 rows=1940 width=12) (actual
                  time=0.002..0.004 rows=3 loops=1)
Total runtime: 144.055 ms
(12 rows)
```

Ausgabe 7

```
QUERY-PLAN
------------------------------------------------------------
Hash Left Join  (cost=21.91..2867.94 rows=100000
                width=32) (actual time=0.048..109.721
                rows=100000 loops=1)
 Hash Cond: (c.id = b.produkte_id)
 -> Seq Scan on produkte c  (cost=0.00..2471.00
rows=100000 width=24) (actual time=
                0.009..37.516 rows=100000 loops=1)
 -> Hash  (cost=21.87..21.87 rows=3 width=16)
      (actual time=0.031..0.031 rows=3 loops=1)
    -> Nested Loop Left Join  (cost=0.00..21.87
       rows=3 width=16) (actual time=0.014..0.025
                                      rows=3 loops=1)
       -> Seq Scan on bestellungen_produkte b
          (cost=0.00..1.03 rows=3 width=12) (actual
                time=0.002..0.002 rows=3 loops=1)
       -> Index Scan using bestellungen_pkey on
 bestellungen a  (cost=0.00..6.93 rows=1 width=12)
          (actual time=0.004..0.005 rows=1 loops=3)
          Index Cond: (a.id = b.bestellungen_id)
 Total runtime: 131.740 ms
(9 rows)
```

Die Hochleister

Voller Einsatz für Ihren Erfolg

Mit einem starken Partner zum Erfolg:
Simon Habeck, Mitarbeiter der PlusServer AG, hat erfolgreich den höchsten Berg Afrikas, den 5.892 Meter hohen Kilimanjaro bestiegen.

Hochleister
powered by PlusServer

Managed **Server** und Managed **Services**

Bei PlusServer steht die beste Leistung für unsere Kunden an erster Stelle. Wir zeigen vollen Einsatz für Ihren Erfolg und finden so auch Lösungen für komplexe Server-Anforderungen.

Egal ob ein einzelner Root-Server oder 300 Server im Hochverfügbarkeits-Cluster, wir geben alles, um unser gemeinsames Ziel zu erreichen: Ihren Erfolg.
Das macht uns zu den Hochleistern.

Überzeugen Sie sich selbst von PlusServer,
wir beraten Sie gern:
0800 – 758 77 37 und **www.plusserver.de**

Die Server mit dem Plus:
Performante Hochleistungs-Server schon ab €129,-*/Monat.
Natürlich inklusive vollem Server-Management mit Rundum-Sorglos-Service.

plusserver

* Preis inkl. MwSt. bei 36 Monaten Vertragslaufzeit. Kürzere Laufzeiten möglich.

fined Function, um schnell viele Datensätze in die Tabelle *produkte* pumpen zu können.

Um 100 000 Datensätzte in die Tabelle zu schreiben, rufen Sie die Funktion mit einem *SELECT create_produkt_dummy_data(100000);* auf. Außerdem sollten jetzt natürlich noch Bestellungen in die Tabelle *bestellungen* aufgenommen werden und entsprechend viele Datensätze in die Tabelle *bestellungen_produkte*. Sie können dafür folgende Statements nutzen:

```
INSERT INTO bestellungen (bestell_dat)
    VALUES ('2010-01-02'), ('2010-01-02'), ('2010-01-10')
                                        -- usw. ...;
INSERT INTO bestellungen_produkte (bestellungen_id,
                                    produkte_id, menge)
VALUES (1, 234, 1), (1, 324, 1), (2, 5437, 10) – usw. ...;
```

Machen Sie sich die Mühe und schreiben Sie ein paar Datensätze in die Tabellen. Nach getaner Arbeit ist es dann auch soweit – wir sehen uns einen ersten Query-Plan an. Ich habe folgende Query vorbereitet (Listing 8). In Ausgabe 6 ist der Query-Plan dargestellt.

Die Query ist relativ trivial und ziemlich sinnfrei. Hier ist das ziemlich egal, denn wie Sie gleich sehen werden, dient es nur als Beispiel für die Beeinflussung des Planers.

Ich kann hier zwar nicht auf jede einzelne Zeile des Query-Plans eingehen, weswegen wir uns nur die wichtigsten Punkte ansehen. Zuallererst stellen wir fest, dass die Ausführung der Query 144.055 ms gedauert hat. Das versuchen wir zu optimieren bzw. zu reduzieren. Wichtig ist auch die Tatsache, dass in der ersten Zeile ein *Merge Left Join* mit der Merge Condition *c.id = b.produkte_id* ausgeführt wird. Wenn wir nun einen Index auf die Spalte *produkte_id* der Tabelle *bestellungen_produkte* setzen, können wir den Planer insofern beeinflussen, als dass er dann einen anderen Join-Typ wählen wird. Probieren wir's aus:

```
CREATE INDEX idx_bestellungen_produkte_id ON
                    bestellungen_produkte (produkte_id);
```

Dann wiederholen wir unser Query und vergleichen die beiden Query-Pläne (Ausgabe 7).

Zum einen läuft die Query nur noch 131.740 ms, und außerdem hat sich der Query-Plan ziemlich geändert. Zum einen wird anstatt des Merge Left Joins ein *Hash Left Join* genutzt (der schneller ist), und weiter unten wird die Tabelle *bestellungen a* durch einen Indexscan anstatt eines Sequential Scan analysiert, was ebenfalls einen Geschwindigkeitsvorteil bedeutet.

Durch das einfache Setzen eines Index kann bereits eine Performancesteigerung erreicht werden.

Wie Sie sehen, kann durch das einfache Setzen eines Index schon eine beträchtliche Performancesteigerung erreicht werden. Gerade wenn Sie nun mit großen Datenmengen zu tun haben (z. B. bei einer Volltextsuche) ist es unerlässlich, Ihre Statements durch das Analysieren des Query-Plans zu optimieren. Der hier gezeigte Ansatz soll verdeutlichen, dass Sie durch geeignete Maßnahmen (wie eben z. B. das Setzen eines Index) die Performance Ihrer Statements erheblich verbessern können.

Fazit und Ausblick

Performancetuning für die PostgreSQL-Datenbank ist kein Buch mit sieben Siegeln. Allerdings bedarf es etwas mehr Anstrengungen als z. B. bei einer MySQL Datenbank – zumindest was die Konfiguration betrifft. Sie haben in diesem zweiten Artikel der Serie über die PostgreSQL gelesen, dass es mehrere unterschiedliche Bereiche gibt, die es zu betrachten gilt. Hervorzuheben sind dabei die Einstellungen in der *postgresql.conf* und das Analysieren Ihrer Queries mit *EXPLAIN ANALYZE*.

An dieser Stelle sei mir ein Hinweis auf die aktuelle deutschsprachige Literatur zum Thema erlaubt. Zum einen finden Sie alle angesprochenen Bereiche in dem von mir und Thomas Pfeiffer geschriebenen Buch „PostgreSQL 8.4" beim Galileo Computing Verlag wieder [7] (Website zum Buch unter [8]). Ein sehr gutes Buch hat außerdem Peter Eisentraut (PostgreSQL-Core-Team) für den O'Reilly Verlag geschrieben: „PostgreSQL Administration" [9]. Und zu guter Letzt sei das Buch „PostgreSQL. Datenbankpraxis für Anwender, Administratoren und Entwickler" von Andreas Scherbaum [10] ans Herz gelegt. Mit diesen drei Büchern sind Sie sehr gut gewappnet, um viel Spaß mit der PostgreSQL-Datenbank zu haben.

Im dritten Teil dieser Artikelserie geht es dann schließlich um das wichtige Thema Backup und weitere, wissenswerte Dinge rund um die PostgreSQL (das soll ein bisschen heiß machen). Bis dahin alles Gute!

Links & Literatur

[1] http://www.postgresql.org

[2] http://www.postgresql.org/docs/8.4/interactive/runtime-config.html

[3] http://pgfoundry.org/projects/pgtune

[4] http://git.postgresql.org/gitweb?p=pgtune.git

[5] http://de.wikipedia.org/wiki/Data_Warehouse

[6] http://de.wikipedia.org/wiki/Online_Transaction_Processing

[7] http://bit.ly/d0pINq (Amazon – Pfeiffer, Wenk)

[8] http://www.pg-praxisbuch.de

[9] http://bit.ly/9bNxef (Amazon – Eisentraut)

[10] http://bit.ly/cwkfeC (Amazon – Scherbaum)

Andreas Wenk

Andreas Wenk ist PostgreSQL-Fan und Softwarearchitekt bei der NMMN in Hamburg. Außerdem lernt er täglich Neues über Programmierung, Internet, seine beiden Töchter und andere coole Dinge. Sie erreichen ihn unter andy@nms.de und können ihm unter @awenkhh folgen.