PostgreSQL optimieren

Open-Source-Tag Magdeburg - 11. Oktober 2008

Andreas 'ads' Scherbaum

Web: http://andreas.scherbaum.la/
E-Mail: andreas@scherbaum.biz
PGP: 9F67 73D3 43AA B30E CA8F 56E5 3002 8D24 4813 B5FE

11. Oktober 2008

Inhaltsverzeichnis

Was ist PostgreSQL?

- Relationale Datenbank
- BSD Lizenz, weltweit aktive Community
- Zahlreiche Features und Funktionen (Foreign Keys, Transaktionen, Trigger)
- Läuft auf zahlreichen Betriebssystemen und diverser Hardware
- Weitgehendes Einhalten der SQL-Standards - Dokumentation der Abweichungen
- Im Schnitt pro Jahr ein Minor-Release mit neuen Features

Ursachen schlechter Performance

- Es gibt vielfältige Ursachen für eine zu geringe Performance:
- Falsche (oder fehlende) Konfiguration
- Anwendung stellt Anfragen falsch
- Datenbank Layout ist falsch oder ungenügend (fehlende Indizies, fehlende Normalisierung)
- Anwendungen fragen zu viele (unnötige) Daten ab (z.B.: *SELECT *, fehlendes WHERE*)
- Latenzzeiten (z.B. Netzwerk, Zugriffszeiten der Festplatten)
- Ungenügende Hardware
- ...

Der Dozent

- Name: Andreas Scherbaum
- Selbstständig im Bereich Datenbanken, Linux auf Kleingeräten, Entwicklung von Webanwendungen
- Arbeit mit Datenbanken seit 1997, mit PostgreSQL seit 1999
- Gründungsmitglied der Deutschen und der Europäischen PostgreSQL User Group
- Board of Directors - European PostgreSQL User Group

Ziele dieses Workshops

- Grundlegende Details zur Steigerung der Performance
- Performancesteigerung durch optimale Nutzung der Hardware
- Zwischenschritt: Sicherstellen gleicher Konfiguration bei allen Beteiligten
- Konfiguration, Anpassen an die vorhandene Hardware
- Analyse von Anfragen und Einsatz von Indizes
- information_schema und pg_* sinnvoll nutzen
- Aufräumen: der Einsatz von VACUUM
- MVCC & Transaktionen - Wozu braucht man so etwas kompliziertes?
- Volltextsuche sinnvoll nutzen
- Partitionierung, Replikation, horizontales Skalieren

Performance - Ungenügende Hardware

- Mehr RAM, mehr RAM, noch mehr RAM
- Daten die im Speicher vorgehalten werden, müssen nicht von der Platte gelesen werden
- Langsame Festplatten gegen schnellere ersetzen, ggf. ein RAID oder Tablespaces nutzen
- Es gilt:

  die Performance verbessert sich mit der Anzahl der Spindeln

- Schwachstellen im I/O-System herausfinden
- SSD-Platten in Betracht ziehen: sehr schnell beim Lesen, jedoch derzeit noch Probleme beim Schreiben

Performancesteigerung durch Hardware

- Soviel RAM wie notwendig: die wichtigen Tabellen/Daten sollen komplett im RAM liegen
- RAID kann Plattenzugriffe verteilen - aber nur unabhängig von der tatsächlichen Nutzung
- Mittels Tablespaces kann man Zugriffe gezielt auf verschiedene Platten verteilen - z. B. Tabelle und Index auf zwei Spindeln legen
- Wenig genutzte Daten können auf langsame (und günstige) Datenträger ausgelagert werden

Voraussetzung für diesen Workshop

- Voller Zugriff auf das System
- PostgreSQL ist bereits installiert
- Position der PostgreSQL Konfigurationsdateien ist bekannt
- `psql` ist bekannt
- Eine Datenbank `workshop` wird angelegt

PostgreSQL installieren

- Debian/Ubuntu: apt-get install postgresql-8.3 postgresql-client-8.3
- Red Hat: yum install postgresql && service postgresql initdb && service postgresql start
- SuSE: über YaST
- Mac OS X: (über Macports) port install postgresql
- Windows: MSI Paket von der Webseite herunterladen

Datenbank erstellen

- CREATE DATABASE workshop;
- Folgende Zeile in *pg_hba.conf* aufnehmen und Datenbank neu laden:

Example

```
local workshop all trust
```

Performance - falsche Konfiguration

- *shared_buffers* zu gering
- FSM - Free-Space-Map nicht angepasst
- Bei umfangreichen Operationen *work_mem* erhöhen
- *maintenance_work_mem* kann mehr Speicher für VACUUM bereitstellen
- *temp_buffers*: bei Überschreiten werden Daten auf die Festplatte ausgelagert

Hinweis: die standardmäßige PostgreSQL Konfiguration ist auf geringen Ressourcenverbrauch ausgelegt.

Performance - Anwendung stellt Anfragen falsch

- Mittels *log_min_duration_statement* langlaufende Anfragen ermitteln
- Anfragen mittels *EXPLAIN* und *EXPLAIN ANALYZE* überprüfen
- Gleiche Anfragen mittels *Prepared Statement* aufrufen: erspart den Parser Schritt
- Zahlreiche INSERT-Operationen mittels *Prepared Statement* oder besser *COPY* durchführen

Performance - Datenbank Layout ist falsch oder ungenügend

- Testweise prüfen, ob ein Index auf einer Spalte eine Verbesserung der Laufzeit bewirkt
- Wichtig: ANALYZE für die Tabelle nicht vergessen
- Tabellen normalisieren - in der Regel ergeben sich dadurch für die Datenbank besser handhabbare Anfragen

Konfiguration - Übersicht

- Alle Konfigurationen für PostgreSQL werden in der Datei *postgresql.conf* vorgenommen
- Alle Einstellungen für Zugangsberechtigungen werden in der Datei *pg_hba.conf* vorgenommen
- Nach Änderungen mittels *pg_ctl reload* die Konfiguration neu laden
- Ein Neustart ist nur selten notwendig und ist in der Konfiguration vermerkt

Konfiguration - Speichernutzung

- *shared_buffers*: Verfügbarer Speicher für alle PostgreSQL Prozesse
- Ev. */etc/sysctl.conf* anpassen (*kernel.shmmax* und *kernel.shmall*)
- *http://www.postgresql.org/docs/current/static/kernel-resources.html*
- *work_mem*: Speicher für Sortieroperationen und Hashing Tabellen
- *maintenance_work_mem*: Speicher für VACUUM und ANALYZE
- *temp_buffers*: Speicher für temporäre Tabellen

Konfiguration - Speichernutzung

- *max_fsm_pages*: bestimmt die Verwaltung von Disk Pages
- *max_fsm_relations*: bestimmt die Anzahl von Tabellen und Indizes
- VACUUM VERBOSE Ausgabe beachten
- *max_stack_depth*: Größe des nutzbaren Stacks (ulimit beachten)

Konfiguration - Logdateien

- *log_filename*: Format für Logdateien in pg_log (siehe strftime())
- *log_rotation_age*: Logfiles nach Zeit rotieren
- *log_rotation_size*: Logfile nach Größe rotieren
- *log_error_verbosity*: PostgreSQL wird gesprächiger (verbose)
- *log_min_error_statement*: minimaler Loglevel (info)
- *log_min_duration_statement*: Ausführungszeit von Anfragen loggen
- *redirect_stderr*: leitet Ausgaben in eine Logdatei um

Konfiguration - Netzwerk

- *listen_addresses*: genutzte Netzwerkinterfaces ('*', 'localhost', IP-Adressen)
- *port*: Netzwerkport (Default: 5432)
- *max_connections*: maximale Anzahl Netzwerkverbindungen
- *superuser_reserved_connections*: Reserve für den Admin + VACUUM

Konfiguration - Netzwerk

- *unix_socket_directory*: Verzeichnis für den Unix-Domain-Socket
- *unix_socket_group*: Gruppenname für Domainsocket File
- *unix_socket_permissions*: Integer Wert für Zugriffsrechte auf Domainsocket File

Konfiguration - Übung

- PostgreSQL an alle Netzwerkinterfaces binden
- 42 maximale Verbindungen zulassen
- 200 MB Shared Memory einstellen
- Speicher für Sortieroperationen auf 3 MB einstellen
- Logfiles täglich neu erstellen, der Wochentag erscheint im Dateinamen
- Queries mit einer Laufzeit über 5 Sekunden protokollieren

Shared Memory - SHMMAX

- Maximale Größe eines einzelnen Shared Memory Segments
- Auf 32-Bit Systemen maximal 4 GB - jedoch effektiv weniger
- Aktuellen SHMMAX Wert abfragen: `cat /proc/sys/kernel/shmmax`
- Neuen Wert setzen: `sysctl -w kernel.shmmax=220000000`

Shared Memory - SHMALL

- Maximale Anzahl der Shared Memory Pages (systemweit)
- Sollte mindestens folgenden Wert haben: `ceil(shmmax/PAGE_SIZE)`
- Page Size abfragen: `getconf PAGE_SIZE`
- Aktuellen SHMALL Wert abfragen: `cat /proc/sys/kernel/shmall`
- Neuen Wert setzen: `sysctl -w kernel.shmall=55000`

Konfiguration - Lösung der Übung

- listen_addresses = '*'
- max_connections = 42
- shared_buffers = 200MB
- *kernel.shmmax* und *kernel.shmall* anpassen
- work_mem = 3MB
- log_filename = 'postgresql-%a.log'
- log_min_duration_statement = 5000

Konfiguration - Planer

- *enable_bitmapscan*, *enable_hashagg*, *enable_hashjoin*, *enable_hashjoin*, *enable_mergejoin*, *enable_nestloop*, *enable_seqscan*, *enable_sort*, *enable_tidscan*
- Alle Werte sind per Default *on* und beeinflussen den Planer
- Zur Laufzeit umschaltbar: *SET enable_seqscan = off*
- *effective_cache_size*: ungefähre Größe des Caches im Betriebssystem
- *geqo*: Generic Query Optimizer ein- oder ausschalten

Konfiguration - Planer

- *seq_page_cost*: Kosten für das Lesen eines Blocks während einer laufenden Leseoperation (1.0)
- *random_page_cost*: Kosten für das Lesen eines zufälligen Blocks (4.0)
- *cpu_tuple_cost*: Kosten für das Bearbeiten einer Zeile (row) (0.01)
- *cpu_index_tuple_cost*: Kosten für das Bearbeiten eines Eintrags im Index (0.005)
- *cpu_operator_cost*: Kosten für den Aufruf eines Operators oder einer Funktion (0.0025)

Konfiguration - Statistiken

- *track_activities*: in 8.3, aktiviere Statistiken für ausgeführte Befehle
- *stats_start_collector*: in 8.2, aktiviere Statistiken
- *stats_block_level*: in 8.2, aktiviere Statistiken für blockbasierte Operationen
- *stats_row_level*: in 8.2, aktiviere Staristiken für zeilenbasierte Operationen
- *track_counts*: in 8.3, aktiviere Statistiken für Datenbankaktivitäten (Autovacuum)

Konfiguration - Statistiken loggen

- *log_statement_stats*: Statistiken über den ausgeführten Befehl
- *log_parser_stats*: Statistiken über die Parser Operationen
- *log_planner_stats*: Statistiken über die Planer Operationen
- *log_executor_stats*: Statistiken über die effektiv ausgeführten Operationen
- Hinweis: *log_statement_stats* kann nicht zusammen mit einer anderen Option aktiviert werden

Konfiguration - Write Ahead Log (WAL)

- *fsync*: aktiviert oder deaktiviert synchrones Schreiben. *off* kann zu Datenverlusten und einer beschädigten Datenbank führen!
- *synchronous_commit*: bei *off* wird nicht auf das Beenden des Schreibens gewartet. Datenverluste sind möglich, aber keine beschädigte Datenbank.
- *wal_buffers*: Größe des Speicherbereichs zum Schreiben eines Eintrags in das WAL.
- *checkpoint_segments*: Anzahl WAL-Logfiles, die rotiert werden

Analyse von Anfragen - Warum?

- Langsame Anfragen schneller gestalten
- Performance-Schwachstellen entdecken
- Ungenutzte Indizies herausfinden

Analyse von Anfragen - Wie

- *EXPLAIN*: liefert den Queryplan für eine Anfrage
- *EXPLAIN ANALYZE*: führt die Anfrage zusätzlich aus und misst die Laufzeiten

Analyse von Anfragen - Übung

- Struktur des System-Views *pg_tables* herausfinden
- Struktur der beteiligten Tabellen herausfinden
- Die *pg_tables* Anfrage mit *EXPLAIN* und *EXPLAIN ANALYZE* analysieren
- Bonus: Verwendung von Indizies erzwingen und die Analyse wiederholen

Analyse von Anfragen - Lösung

```
\dS

\d pg_catalog.pg_tables

EXPLAIN <query>
EXPLAIN ANALYZE <query>

SET enable_seqscan = off;

EXPLAIN <query>
EXPLAIN ANALYZE <query>
```

Der Planer

- Jede Anfrage wird vom Planer bearbeitet und in einen Queryplan übersetzt
- Zu einzelnen Teilen der Anfrage werden statistische Informationen aus den Tabellen geholt
- Aufgrund dieser Informationen wird entschieden, ob z.B. ein Index genutzt wird
- Der Planer kann ggf. Anfragen intern umschreiben

Beispiel Tabelle

```
CREATE TABLE planer_test (
  id          SERIAL         NOT NULL UNIQUE,
  inhalt      VARCHAR        NOT NULL
);

INSERT INTO planer_test (inhalt) VALUES ('Inhalt 001');
INSERT INTO planer_test (inhalt) VALUES ('Inhalt 002');
INSERT INTO planer_test (inhalt) VALUES ('Inhalt 003');
INSERT INTO planer_test (inhalt) VALUES ('Inhalt 004');
INSERT INTO planer_test (inhalt) VALUES ('Inhalt 005');

ANALYZE VERBOSE planer_test;
```

Analyse von Anfragen

- PostgreSQL kennt den *EXPLAIN* Befehl, den man vor einen Query setzen kann
- *EXPLAIN* liefert detaillierte Aussagen, wie PostgreSQL die Abfrage bearbeiten würde

```
postgres=# EXPLAIN SELECT id, inhalt FROM planer_test WHERE id=1;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on planer_test  (cost=0.00..1.06 rows=1 width=18)
   Filter: (id = 1)
(2 rows)
```

Analyse von Anfragen

- *Seq Scan on planer_test (cost=0.00..1.06 rows=1 width=18)*
- Sequentieller Scan auf die Tabelle
- Gesamtkosten: 1.06
- Kosten bis zum Lesen des ersten brauchbaren Wertes: 0.00
- Erwartete Anzahl Spalten: 1

Analyse von Anfragen

- Frage: Warum ein sequentieller Scan, wenn wir doch einen Index auf *id* haben?
- Erzwingen wir die Nutzung des Index:
- *SET enable_seqscan=0;*
- Einschätzung: Was wird passieren?

Analyse von Anfragen

```
postgres=# EXPLAIN SELECT id FROM planer_test WHERE id=1;
                                    QUERY PLAN
-------------------------------------------------------------------
 Index Scan using planer_test_id_key on planer_test
                          (cost=0.00..3.01 rows=1 width=4)
   Index Cond: (id = 1)
(2 rows)
```

- Was ist passiert?
- Index wird gelesen: Kosten 1-2
- Da sich alle Daten in einem Block befinden, muss dieser Block danach sowieso gelesen werden
- Resultat: Kosten 3

Analyse von Anfragen - Erkentnisse

- PostgreSQL erstellt jeden Queryplan dynamisch und abhängig vom Zustand der Daten
- Zwei Anfragen auf die gleiche Tabelle, bei unterschiedlichen Datenmengen, können anders ausgeführt werden
- Daraus resultiert: PostgreSQL sollte gute Informationen über die Daten in den Tabellen haben
- Das wichtigste Hilfsmittel ist *ANALYZE* (bzw. *VACUUM ANALYZE*)
- Der Autovacuum Daemon übernimmt die meiste Arbeit

Analyse von Anfragen - Ausführungszeit messen

```
postgres=# EXPLAIN ANALYZE SELECT id FROM planer_test WHERE id=1;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 Seq Scan on planer_test  (cost=0.00..1.06 rows=1 width=4)
                          (actual time=0.017..0.025 rows=1 loops=1)
   Filter: (id = 1)
 Total runtime: 0.066 ms
(3 rows)
```

- *EXPLAIN ANALYZE* führt die Anfrage aus und misst die Zeit
- Die gemessenen Zeiten bewegen sich innerhalb der vorab geschätzten Werte
- Sollte es hier grobe Abweichungen geben, hat der Planer falsche Daten!
- Vorsicht: es können Daten verändert werden.

Analyse von Anfragen - Negativbeispiel

```
postgres=# EXPLAIN ANALYZE SELECT id FROM planer_test WHERE id=10;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 Seq Scan on planer_test  (cost=0.00..1.06 rows=1 width=4)
                          (actual time=0.022..0.022 rows=0 loops=1)
   Filter: (id = 10)
 Total runtime: 0.060 ms
(3 rows)
```

- Eine sequentielle Suche nach einem nicht vorhandenen Wert
- Die Datenbank muss die gesamte Tabelle lesen (minimal 0.022 gleich 0.022)
- Ein Index hätte hier womöglich schneller Auskunft gegeben

information_schema

- Das Information Schema stellt einen standardisierten Weg bereit, Informationen über die Struktur der Datenbank auszulesen
- Dazu gibt es Views im Schema *information_schema*
- Z.B. Tabellen, Spalten, Views, User, Sequencen, Trigger, Privilegien u.v.m.
- *http://www.postgresql.org/docs/current/static/information-schema.html*

information_schema - Übung

- Mit *information_schema* vertraut machen
- Ggf. Fragen stellen
- Herausfinden wie man alle Spalten einer Tabelle anzeigen kann

information_schema - Lösung

```
SELECT table_schema,table_name,column_name,data_type
  FROM information_schema.columns
 WHERE table_schema='<schema name>'
   AND table_name='<tabellen name>';
```

pg_* Systemtabellen

- Nicht SQL-konform, eine PostgreSQL-Erweiterung
- Seit vielen Versionen vorhanden
- Struktur kann sich verändern
- Wesentlich weitreichendere Informationen als im *information_schema*
- Änderungen können die Datenbank unbrauchbar werden lassen

pg_* Systemtabellen - Übung

- Mit Systemtabellen vertraut machen
- Ggf. Fragen stellen
- Herausfinden wie man alle Spalten einer Tabelle anzeigen kann

pg_* Systemtabellen - Lösung

```
psql mit den Optionen -eE starten
\dt <tabelle>

Ausgabe des Queries nutzen
```

VACUUM - ein Staubsauger?

- Durch MVCC bleiben alte Daten nach Abschluß einer Transaktion in einer Tabelle zurück
- *VACUUM* gibt diesen Platz wieder frei
- *ANALYZE* aktualisiert die Statistiken für den Planer
- Shortcut: *VACUUM ANALYZE*
- Der Autovacuum Daemon kann die Arbeit automatisiert erledigen
- Neu in PG 8.3: HOT - *Heap-Only-Tuples*
- quasi ein kleines Vacuum innerhalb des Blocks
- Voraussetzung: kein Wert im Index wird geändert

VACUUM automatisiert

- PostgreSQL besitzt einen eingebauten Daemon namens *autovacuum*
- Dieser überprüft regelmäßig alle Datenbanken und enthaltene Tabellen
- Die Zeit zwischen zwei Läufen wird mittels *autovacuum_naptime* eingestellt
- Der Daemon selbst wird mittels *autovacuum* ein- und ausgeschaltet
- Einzelne Tabellen in einer Datenbank können über die *pg_autovacuum* Tabelle konfiguriert werden

Konfiguration - kostenbasiertes VACUUM

- *vacuum_cost_delay*: Anzahl Millisekunden für die Ruhephase des VACUUM-Befehls, 0 zum Deaktivieren (Default)
- *vacuum_cost_page_hit*: erwartete Kosten für das VACUUM eines Blocks, der sich im Shared RAM befindet (1)
- *vacuum_cost_page_miss*: erwartete Kosten für das VACUUM eines Blocks auf der Festplatte (10)
- *vacuum_cost_page_dirty*: erwartete Kosten, wenn ein vorher sauberer Block geschrieben werden muss (20)
- *vacuum_cost_limit*: Kostenlimit für eine Runde (200)

Konfiguration - kostenbasiertes Auto-VACUUM

- *autovacuum* und *track_counts* müssen aktiviert sein
- *autovacuum_max_workers*: Anzahl gleichzeitiger VACUUM Prozesse (3)
- *autovacuum_naptime*: Zeit zwischen Beenden einer und dem Prüfen der nächsten Datenbank (1min)
- *autovacuum_vacuum_cost_delay*: Anzahl Millisekunden für die Ruhepause. Der Wert -1 führt zur Nutzung von *vacuum_cost_delay* (20)
- *autovacuum_vacuum_cost_limit*: Kostenlimit für eine Runde, wird über alle aktiven Worker verteilt. -1 führt zur Nutzung von *vacuum_cost_limit* (-1)

Konfiguration - kostenbasiertes Auto-VACUUM

- *autovacuum_vacuum_threshold*: Anzahl der notwendigen geänderten Tuples für VACUUM (50)
- *autovacuum_analyze_threshold*: Anzahl der notwendigen geänderten Tuples für ANALYZE (50)
- *autovacuum_vacuum_scale_factor*: Prozentwert über die Tabellengröße, der zu *autovacuum_vacuum_threshold* addiert wird (0.2 - 20%)
- *autovacuum_analyze_scale_factor*: Prozentwert über die Tabellengröße, der zu *autovacuum_analyze_threshold* addiert wird (0.1 - 10%)

VACUUM - Übung

- Die Datenbank mittels *VACUUM ANALYZE VERBOSE* aufräumen
- Auf die Ausgaben (am Ende) für die Anzahl benötigter Einträge für die Free-Space-Map achten

Transaktionen

- Transaktionen kapseln beliebig viele Aktionen in der Datenbank zu einer einzelnen Aktion nach aussen
- Beispiel: Banküberweisung
- Schritt 1: Geld wird von Ihrem Konto abgebucht
- Schritt 2: Geld wird auf das andere Konto gebucht
- Was passiert, wenn zwischen Schritt 1 und 2 ein Fehler auftritt?

Transaktionen - ACID

- A C I D
- A: Atomarität (Atomicity) - ganz oder gar nicht
- C: Konsistenz (Consistency) - Datenbestand ist vor und nach der Transaktion konsistent
- I: Isolation (Isolation) - alle Aktionen (Transaktionen) sind voneinander abgekapselt
- D: Dauerhaftigkeit (Durability) - bestätigte Änderungen bleiben erhalten

Transaktionen - Nutzung

- Transaktion starten:
- *BEGIN* oder *START TRANSACTION*
- Transaktion abschliessen:
- *COMMIT*
- Transaktion zurückrollen:
- *ROLLBACK*
- Hinweis: PHP bietet keine eingebauten Funktionen für Transaktionen, Perl DBI kennt *begin_work*, *commit* und *rollback*

Isolation von Transaktionen

- Beispiel:
- Transaktion 1 startet mit eigener aktueller Ansicht der Datenbank (I - Isolation)
- Transaktion 2 startet
- Transaktion 2 ändert einen Wert in einer Tabelle
- Transaktion 2 committed ihre Änderungen, diese werden sichtbar
- Aber: Transaktion 1 muss weiterhin den alten Wert sehen können, da dieser beim Start der Transaktion gültig war

Isolation von Transaktionen - Übung

- Zwei Verbindungen zur Datenbank öffnen
- Eine Tabelle erstellen
- In beiden Verbindungen eine Transaktion starten
- Den Inhalt der Tabelle in beiden Verbindungen anzeigen lassen
- In einer Transaktion einen Datensatz einfügen
- Den Inhalt der Tabelle in beiden Verbindungen anzeigen lassen

Isolation von Transaktionen - Übung

```
CREATE TABLE transaktion_test (
  id              SERIAL        NOT NULL
                                PRIMARY KEY,
  wert            VARCHAR       NOT NULL
);

BEGIN;
SELECT * FROM transaktion_test;

INSERT INTO transaktion_test (wert) VALUES ('etwas Inhalt');

SELECT * FROM transaktion_test;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Isolation von Transaktionen - die Nachteile

- Das notwendige Vorhalten älterer Versionen eines Datensatzes wird *Multiversion Concurrency Control* (kurz: MVCC) genannt
- Diese alten Versionen belegen Platz auf der Festplatte und im Index
- Diese Überreste können mit dem Befehl *VACUUM* aufgeräumt werden, Syntax:
- *VACUUM ANALYZE tabelle*
- Hinweis: *VACUUM* kann nicht innerhalb einer Transaktion aufgerufen werden

Isolation von Transaktionen - Transaktionslevel

- SQL definiert 4 verschiedene Transaktionslevel:
  - *READ UNCOMMITTED* (in PG: *READ COMMITTED*)
  - *READ COMMITTED*
  - *REPEATABLE READ* (in PG: *SERIALIZABLE*)
  - *SERIALIZABLE*
- PostgreSQL nutzt nur *READ COMMITTED* und *SERIALIZABLE*

Isolation von Transaktionen - Transaktionslevel

- Unterschiede zwischen *READ COMMITTED* und *SERIALIZABLE*:
- *READ COMMITTED*: Daten, die vor dem Beginn des Query, aber nach Beginn der Transaktion sichtbar waren, werden angezeigt
- *SERIALIZABLE*: Nur Daten, die vor Beginn der Transaktion sichtbar waren, werden angezeigt
- Hinweis: bei *SERIALIZABLE* kann es zu Transaktionsabbrüchen kommen, die Applikation muss dies abfangen!

Isolation von Transaktionen - Transaktionslevel

- PostgreSQL Default: *READ COMMITTED*
- Ändern mit: *SET TRANSACTION ISOLATION LEVEL SERIALIZABLE*
- Befehl muss nach *START TRANSACTION*/*BEGIN* aufgerufen werden
- Die Einstellung kann nach der ersten Daten-ändernden Anweisung (inkl. *SELECT*) nicht mehr geändert werden

Volltextsuche - Vorteile

- Einen einzelnen Begriff in einem geschriebenen Text zu finden ist kompliziert:
- Das Wort könnte dekliniert oder konjugiert sein
- Das Wort könnte in der Mehrzahl vorliegen
- Diese Fälle kann man mit einer simplen Suche mittels *LIKE* nicht abbilden

Volltextsuche

- Die Lösung: Volltextsuche
- Der Text wird von einem Parser zerlegt und von einem Stemmer normalisiert
- Dem Stemmer muss in der Regel die zu nutzende Sprache mitgeteilt werden

Volltextsuche - Beispiel

```
CREATE TABLE volltextsuche (
  id              SERIAL            NOT NULL
                                    PRIMARY KEY,
  text_deutsch    TEXT              NOT NULL,
  text_geparst    TSVECTOR
);
CREATE INDEX vt_suche ON volltextsuche USING gist(text_geparst);

INSERT INTO volltextsuche
            (text_deutsch, text_geparst)
     VALUES ('Ein Text ueber Katzen und Elefanten',
             to_tsvector('german',
                         'Ein Text ueber Katzen und Elefanten'));
```

Volltextsuche - Beispiel

```
SELECT id, text_deutsch
  FROM volltextsuche
 WHERE text_geparst @@ to_tsquery('german', 'Elefanten');

SELECT id, text_deutsch
  FROM volltextsuche
 WHERE text_geparst @@ to_tsquery('german', 'Elefant & Katze');

SELECT id, text_deutsch
  FROM volltextsuche
 WHERE text_geparst @@ to_tsquery('german', 'Elefant & Maus');
```

Volltextsuche - Dokumentation

- Zwei URLs mit weiteren Informationen zur Gestaltung komplexer Anfragen

`http://www.postgresql.org/docs/current/static/textsearch.html`

`http://rhodesmill.org/brandon/projects/tsearch2-guide.html`

Volltextsuche - Übung

- Eine Funktion als Trigger erstellen, die die Spalte 'text_geparst' automatisch füllt

Hinweis: Vorher mittels `createlang plpgsql` pl/pgSQL aktivieren

Volltextsuche - Beispiel

```
CREATE OR REPLACE FUNCTION update_tsearch_vectors()
          RETURNS TRIGGER
AS $$
DECLARE
BEGIN

      NEW.text_geparst := to_tsvector('german', NEW.text_deutsch);

  RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
```

Volltextsuche - Beispiel

```
CREATE TRIGGER vt_trigger
        BEFORE INSERT OR UPDATE
            ON volltextsuche FOR EACH ROW
       EXECUTE PROCEDURE update_tsearch_vectors();
```

Partitionierung - Vorteile

- Daten werden ausgehend von der tatsächlichen Situation gespeichert
- Sequentielle Scans durchlaufen nicht den gesamten Datenbestand
- Bestimmte Verwaltungsarbeiten lassen sich effizient durchführen
- Daten können auf unterschiedliche Medien aufgeteilt werden

Partitionierung bei PostgreSQL

- Tabellen werden durch Inheritance (Vererbung) von einer Mastertabelle erstellt:

```
CREATE TABLE log_data (
  id              SERIAL          NOT NULL
                                  PRIMARY KEY,
  insert_at       TIMESTAMPTZ     NOT NULL,
  data            VARCHAR(100)    NOT NULL
);
```

Partitionierung bei PostgreSQL

```
CREATE TABLE log_data_200809 ( ) INHERITS (log_data);
CREATE TABLE log_data_200810 ( ) INHERITS (log_data);

INSERT INTO log_data_200810
            (insert_at, data)
     VALUES (NOW(), 'Datensatz 1');

SELECT * FROM log_data_200810;
SELECT * FROM log_data;
```

Korrekte Timestamps erzwingen

```
CREATE TABLE log_data_200809 ( ) INHERITS (log_data);
CREATE TABLE log_data_200810 (
  CHECK (insert_at >= DATE '2008-10-01' AND
         insert_at < DATE '2008-11-01')
) INHERITS (log_data);
```

Nutzung des Timestamp bei einer Abfrage

- Wenn PostgreSQL einen Contraint auf der vererbten Tabelle vorfindet, wird dieser dazu genutzt, die komplette Tabelle gf. aus der Abfrage auszuschließen

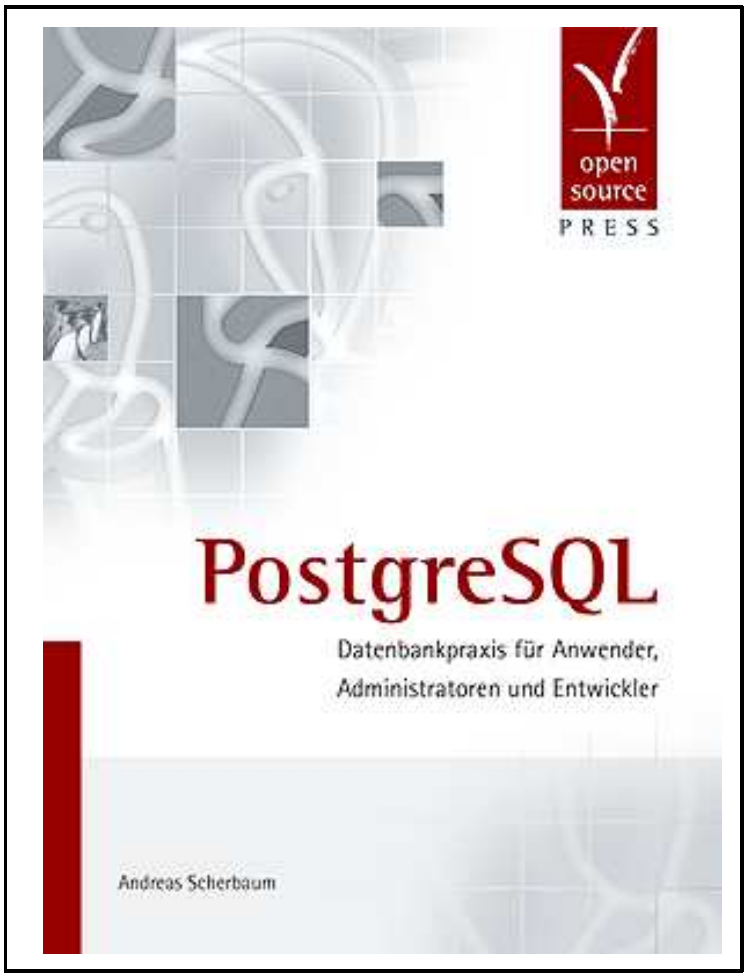
```
SET constraint_exclusion = on;

EXPLAIN SELECT * FROM log_data_200809 WHERE insert_at='2008-10-11';
```

Schulungen

- Open Source School GmbH aus München bietet PostgreSQL Schulungen an:
- `Administration von PostgreSQL`
- nächste Termine:
- 30. März - 01. April 2009
- 08. Juni - 10. Juni 2009

- Nähere Infos unter:
- `http://www.opensourceschool.de`

PostgreSQL Buch



PostgreSQL - Datenbankpraxis für Anwender, Administratoren und Entwickler

Erscheint Anfang 2009 im Verlag Open Source Press

Ende

`http://andreas.scherbaum.la/`

Fragen?

Andreas 'ads' Scherbaum <andreas@scherbaum.biz>
PostgreSQL User Group Germany
European PostgreSQL User Group