Einführung in PostgreSQL und Postgis

Was ist PostgreSQL?

PostgreSQL (http://www.postgresql.org/) ist ein Open Source relationales Datenbanksystem. PostgreSQL ist Multi-Plattform und läuft auf fast allen Unix Systemen, inklusive Linux und MacOSX, aber auch unter Windows (nur NT Serie). PostgreSQL ist seit über 15 Jahren in Entwicklung und ist neben MySQL die populärste Open Source Datenbank. PostgreSQL implementiert die meisten der ANSI SQL92, SQL99 und SQL2003 Datentypen und Abfragen. Programmierschnittstellen existieren zu praktisch allen Programmier- und Scriptingsprachen wie C/C++, Java, .NET, Perl, PHP, Python, Ruby. Ausserdem kann optional die ODBC und/oder JDBC Schnittstelle aktiviert werden, um Applikationen einen datenbankunabhängigen Zugriff auf PostgreSQL Datenbanken zu ermöglichen.

Im Gegensatz zur noch populäreren MySQL Datenbank unterstützt PostgreSQL bereits seit längerem Enterprise Features wie Triggers, Views, Sub-Selects, Stored Procedures (in verschiedenen Programmiersprachen), Transaktionen, Load Balancing (Lastaufteilung zwischen verschiedenen Datenbankrechnern), Replikation, Hot Backups im laufenden Betrieb, und einen ausgeklügelten Query Planner und Optimierer. PostgreSQL skaliert sehr gut, sowohl bei der Anzahl Benutzer als auch bei der verwalteten Datenmenge. So ist die Datenbankgrösse unlimitiert, eine Tabelle kann maximal 32 Terrabyte gross werden und ein einzelner Record kann maximal 1.6 Terrabyte, und ein einzelner Feldinhalt kann maximal 1 Gigabyte gross werden.

PostgreSQL ist mit Sicherheit die ausgereifteste OpenSource Datenbank und reicht in vielen Bereichen an die Performanz und Flexibilität von kommerziellen Datenbanken, wie Oracle, Sybase oder IBM DB2 heran. Die liberale BSD Lizenz erlaubt das Modifizieren und Verteilen des Quellcodes, sowohl innerhalb von Open Source als auch innerhalb kommerzieller Systeme. Zu den Sponsoren und Unterstützern von PostgreSQL gehören etwa Firmen wie Fujitsu-Siemens, Skype, NTT DoCoMo, Redhat, Sun Microsystems oder Google. Zu den Benutzern gehören grosse Firmen wie Apple, Affilias DNS services, Verisign, Sun, Cisco, Sony, TiVo, USPS (United States Postal Service), Skype, Telstra, NTT, BASF, die UNO, NASA, US Army, viele Regierungsorganisationen (z.B. US-amerikanisches Arbeitsamt) und Universitäten.

PostgreSQL Geschichte

Postgres, der Vorgänger von PostgreSQL wurde zuerst an der University of California at Berkeley (UCB) entwickelt. Der erste Entwickler (1986) war Michael Stonebraker, später CTO von Informix. Michael war damals Computer Science Professor und entwickelte Postgres als Nachfolgeprojekt von Ingres (später von Computer Associates gekauft). Postgres wurde im Rahmen eines Forschungsprojektes über objektrelationale Datenbanksysteme entwickelt. Daher unterstützt PostgreSQL heute die Vererbung von Tabellendefinitionen, sowohl einfache, als auch multiple Vererbung. Tabellendefinitionen können als Basisklassen für abgeleitete Tabellendefinitionen dienen. Die Daten selber können ebenfalls zwischen Eltern- und Kindklassen geteilt werden.

Ein erster Postgres Prototyp war 1988 erhältlich, 1993 existierte eine grosse Userbasis, welche mit ihren Supportanfragen und Feature-Requests die damalige Postgres Organisation überforderte. Das Projekt wurde 1993 eingestellt. Stonebraker wurde CTO von Informix, welche 2001 für 1 Milliarde US Dollar von IBM übernommen wurde. Da die BSD Lizenz sowohl Open Source als auch kommerzielle Ableger erlaubte, begannen 1994/95 zwei PhD Studenten (Andrew Yu und Jolly Chen) Postgres weiterzuentwickeln indem sie Postgres' proprietäre Abfragesprache mit einem erweiterten Subsets des SQL93 Standards ersetzten. Das System hiess nun Postgres95. Ab 1996 wurde Postgres95 zu einem richtigen OpenSource Projekt ausserhalb der Universität, zuerst unterstützt von Hub.org (einer Provider Firma). Im gleichen

Jahr wurde Postgres95 zum heutigen Namen PostgreSQL umbenannt um die neuen SQL Abfragefeatures zu betonen. Heute gibt es circa jedes Jahr einen neuen Major Release, dazwischen auch Bugfix-Releases. Version 8 wurde 2005 veröffentlicht, seit 2005 wird PostgreSQL auch mit SUN Solaris ausgeliefert. Version 8 unterstützte erstmals Microsoft Windows und brachte zahlreiche neue Features und Performanzverbesserungen. Der Release 8.1 brachte wichtige Neuerungen im Bereich Skalierungsfähigkeit. Heute entwickeln einige grosse IT Firmen und kleinere Dienstleister die Datenbank weiter. Teilweise werden spezialisierte Ableger und kommerzieller Distributionen vertrieben. Erste professionelle Benchmarks von Sun im August 2007 zeigen dass PostgreSQL nur ca. 12% langsamer ist als Oracle auf vergleichbarer Hardware, dies aber bei deutlich tieferen Kosten und geringeren Wartungsaufwänden.

PostgreSQL Support und Dokumentation
Wie bei Open Source Software meist üblich, werden die Supportkanäle Web, IRC, sowie Mailinglisten und Foren genutzt. Eine ausführliche Dokumentation ist unter http://www.postgresql.org/docs/manuals/ zu finden. Benutzer haben die Möglichkeit direkt zu den Dokumentationen Kommentare abzugeben. Mittlerweile gibt es mehr als 15 Bücher zu PostgreSQL. Eine Liste ist unter http://www.postgresql.org/docs/books/ zu finden. Die Mailinglisten und Archive sind unter http://www.postgresql.org/community/lists/ . Schliesslich gibt es mittlerweile eine stattliche Anzahl von Firmen die kommerziellen Support zu PostgreSQL und Postgis leisten (http://www.postgresql.org/support/professional_support)

Was ist Postgis?
Postgis (http://postgis.refractions.net/) ist eine räumliche Erweiterung von PostgreSQL. Es erweitert die Datenbank um räumliche Datentypen, Operatoren, Indizes und Funktionen. Dazu gibt es eine Anzahl von Utilities, etwa für die Wartung und Import und Export. Zusammen mit der OpenSource Bibliothek Geos (http://geos.refractions.net/) können räumliche Manipulationen und Analysen durchgeführt werden, etwa intersections, buffer, unions, etc. Zusammen mit der proj4 Bibliothek (http://proj.maptools.org/) können die Geodaten auch in andere Projektionssysteme überführt werden. Postgis wird von praktisch allen grösseren OpenSource GIS Projekten unterstützt. Ab ArcGIS Version 9.3 wird auch ESRI Postgis über ArcSDE unterstützen. Autodesk unterstützt Postgis über die FDO Schnittstelle (http://fdo.osgeo.org/). Über GDAL (http://www.gdal.org/) und können andere GIS-Formate nach Postgis geschrieben werden und umgekehrt. Ebenso unterstützt FME (http://www.safe.com/), das Schweizer Taschenmesser bei der Geodatenkonvertierung und Manipulation, Postgis. Schliesslich kann Postgis auch als Backend für den UMN Mapserver (http://mapserver.gis.umn.edu/) eingesetzt werden.

Postgis Geschichte
Postgis wird von der Firma Refractions entwickelt und weiterentwickelt. Einen ersten Release gab es 2001. Die stabile Version 1.0 wurde im April 2005 veröffentlicht. Im September 2006 wurde Postgis 1.1 als OpenGIS compliant zertifiziert. Die Firma Refractions, wie auch viele andere Open Source Firmen leisten heute auch kommerziellen Support für die Datenbank. Postgis wird in vielen grossen Projekten eingesetzt, etwa beim IGN (französische Landestopografie), GlobeExplorer, EU-Joint Research Centre, UC Davis, InfoTerra, KOGIS/Swisstopo, Kantone Solothurn/Thurgau/Graubünden und vielen mehr.

PostgreSQL Support und Dokumentation
Sämtliche Ressourcen zu Postgis können unter http://postgis.refractions.net/ heruntergeladen werden: Dokumentationen, Mailinglisten Archive, Download Quellcode, etc. Support funktioniert v.a. Über Mailinglisten, kommerzieller Support kann von der Fa. Refractions aber auch vielen anderen GIS Firmen geleistet werden. Dazu braucht es Supportverträge oder Projektaufträge.

Einige Datenbankbegriffe

Client/Server System

Ein Client/Server System trennt die zentrale Datenhaltung und zentrale Funktionen von der Anzeige und Interaktion (Frontend). PostgreSQL ist ein Client/Server System. Der Server Prozess (der PostgreSQL Server) läuft permanent am Server, koordiniert die Datenhaltung und nimmt von den Clients SQL Anfragen entgegen. Der Serverprozess koordiniert auch den konkurrierenden Zugriff mehrerer Benutzer und stellt sicher, dass die Daten ständig konsistent vorgehalten werden und mehrere Nutzer nicht gleichzeitig die selben Daten ändern. Im Gegensatz zu primitiveren Datenbanken, bei denen ganze Datenbanken oder Tabellen beim Zugriff eines einzelnen Benutzers gesperrt werden, wird bei PostgreSQL das sogenannte multi-version concurrency control system eingesetzt um Konkurrenzsituationen zu reduzieren und zu vermeiden.

Als Clients können beliebige Programme agieren. Einerseits dedizierte PostgreSQL Clients, wie die Kommandozeilen-Schnittstelle “psql”, oder der graphische PostgreSQL Administrator “pgAdmin”. Zusätzlich können Webapplikationen oder GIS Systeme als Clients fungieren oder beliebige Programme die die ODBC oder JDBC Schnittstelle unterstützen, wie z.B. Office Software. Schliesslich können selbst geschriebene Programme als Clients agieren. Client und Server können sich entweder am gleichen Rechner befinden oder können über TCP-IP Netzwerkverbindungen kommunizieren. Der PostgreSQL Serverprozess (mit dem Namen “postmaster”) horcht per default am TCP-IP Port 5432.

Daemon/Service

Ein Daemon ist unter Unix/Linux ein Hintergrunddienst der unmittelbar nach dem Betriebssystem und Netzwerkstart hochgefahren wird, ohne dass sich dazu Benutzer manuell einloggen müssen. Daemons lauschen an diversen Schnittstellen (Geräteschnittstellen oder Netzwerkports) auf Anfragen und liefern Antworten an die Clients zurück. Unter Windows gibt es ähnliche Konzepte, dort heissen die Daemons “service”. Bei PostgreSQL läuft permanent ein sogenannter Master Serverprozess, wenn sich ein Client verbindet, wird ein “fork”, ein Ableger des Masterprozesses gestartet. Für jede aufrechte Client-Serververbindung wird solch ein fork gestartet. Wenn sich der Client verabschiedet, wird auch der Server-Fork Prozess gestoppt. Die Anzahl der maximal zulässigen Client-Verbindungen kann in der Konfigurationsdatei “postgresql.conf” mit der Variable “max_connections” kontrolliert werden.

Datenbank (database)

Ein Containerelement (typischerweise Projektspezifisch) das verschiedene PostgreSQL Objekte, wie z.B. Schemen, Tabellen, Operatoren, Funktionen, etc. aufnehmen kann. Datenbanken werden mit dem Befehl “createdb dbname” angelegt und mit dem Befehl “dropdb dbname” gelöscht. Achtung, beim Löschen werden alle Kindelemente mitgelöscht!

Schema

Jede Datenbank kann ein oder mehrere Schemen beinhalten. Per default wird bei einer neuen Datenbank in PostgreSQL das Schema “public” erstellt. Wenn nur eine Schema angelegt wird, ist das Schema auch weiterhin belanglos, ansonsten muss jeweils zusätzlich zum Objektnamen noch der Schemenname angegeben werden, etwa “public.gemeinden” um die Tabelle “gemeinden” innerhalb des Schemas “public” zu bezeichnen. Schemen sind wie Datenbanken Containerobjekte welche weitere Objekte wie Tabellen, Datentypen, Funktionen, etc. beinhalten können. Innerhalb einer Datenbank können Objektnamen mehrmals vorkommen wenn sie in unterschiedlichen Schemen definiert sind. Schemen werden verwendet um Tabellen logisch zu

gruppieren, Namenskonflikte zu vermeiden, oder Benutzern bestimmte private Bereiche innerhalb einer Datenbank zuzuweisen. Der Befehl "`CREATE SCHEMA schemaname`" erstellt ein neues Schema innerhalb einer Datenbank, der Befehl "`DROP SCHEMA schemaname CASCADING`" löscht eine Schema inklusive aller Kindobjekte.

Tabelle (Table, Relation)

Die Tabelle (auch als Relation bezeichnet) dient zum Speichern von Daten innerhalb eines Schemas. Die Tabelle besteht aus **Zeilen (Records)** und **Spalten (Columns)**. Die Anzahl der Records in einer Tabelle ist in PostgreSQL prinzipiell unbeschränkt, eine Tabelle kann aber derzeit maximal 32 Terrabytes gross werden. Für jede Spalte muss ein Datentyp definiert sein. Der folgende Befehl kreiert eine neue Tabelle mit 3 Spalten:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

Der Befehl "`DROP TABLE products`" löscht die Tabelle.

Datentypen (Data Types)

Der Datentyp schränkt die gültigen Wertebereiche innerhalb einer Spalte ein. Diese Einschränkung dient einerseits einer gewissen Qualitätssicherung, andererseits können durch die Einschränkungen erst gewisse Funktionen ermöglicht werden, die Speicherung optimiert werden, Indizes beschleunigt werden, etc. PostgreSQL kennt eine Vielzahl an Datentypen (siehe http://www.postgresql.org/docs/8.3/interactive/datatype.html). Bei Bedarf können eigene Datentypen mit dem Befehl "`CREATE TYPE name AS ( attribute_name data_type [, ... ])`" selber kreiert werden. Alle derzeit verfügbaren nativen Datentypen fallen in die folgenden Untergruppen:

- Numerische Typen (Numeric Types)
- Monetäre Typen (Monetary Types)
- Zeichen Typen (Character Types)
- Binäre Datentypen (Binary Data Types)
- Datums- und Zeitformate (Date/Time Types)
- Boolean Types
- Geometrie Typen (Geometric Types)
- Netzwerkadress Typen (Network Address Types)
- Bit String Types (für Bit Masken)
- Feld Typen (Array Types, Multidimensionale Felder)
- Komposit Typen (Composite Types, Zusammenfassung von Daten und Funktionen zu Objekten)
- Objekt Identifikator Typen (Object Identifier Types, Systemintern verwendet)
- Pseudo Typen (Pseudo Types, intern verwendet oder für Entwickler)

Operatoren (Operators) und Funktionen (Functions)

Operatoren können mit existierenden Daten Vergleiche und Verknüpfungen herstellen. Operatoren kommen meist im “WHERE” Teil von Abfragen oder Datenmanipulationen vor. Im Gegensatz zu Operatoren, die für bedingte Anweisungen verwendet werden, können Funktionen Daten manipulieren und verändern, etwa Berechnungen vornehmen, aggregieren oder Zeichen extrahieren. Wie bei den Datentypen können zusätzlich zu den nativen Operatoren und Funktionen auch eigene Operatoren und Funktionen mit Hilfe von PL/PgSQL oder Scripting- und Programmiersprachen definiert werden. PostgreSQL stellt die folgenden Gruppen von Operatoren und Funktionen nativ zur Verfügung.

- Logische Operatoren (Logical Operators, AND, OR, NOT)
- Vergleichsoperatoren (Comparison Operators, z.B. <, >, >=, <=, =, != oder <>)
- Mathematische Operatoren und Funktionen (Mathematical Operators and Functions, +, -, *, /, %, ^, !, Trigonometrische Funktionen, etc.)
- Zeichenoperatoren und Funktionen (String Operators and Functions, z.B. Substring(), trim(), upper(), lower(), etc.)
- Binäre Zeichenfunktionen und Operatoren (Binary String Functions and Operators, Zeichenmanipulationen und Operatoren auf Byte Ebene)
- Bit Zeichenfunktionen und Operatoren (Bit String Functions and Operators, Zeichenmanipulationen und Operatoren auf Bit Ebene)
- Zeichenmuster-Vergleichsoperatoren (Pattern Matching, LIKE Operator (wildcards) und regular expressions)
- Datentyp Formatierungs Funktionen (Data Type Formatting Functions, z.B. to_char(), to_date(), to_number, etc.)
- Datums- und Zeitfunktionen und Operatoren (Date/Time Functions and Operators, Datumskalkulationen, age(), date_part(), now(), etc.)
- Geometriefunktionen und Operatoren (Geometric Functions and Operators, nicht zu verwechseln mit den Postgis-Funktionen und Operatoren!, z.B. Is left of, is right of, distance between, contains, etc.)
- Netzwerkadress Funktionen und Operatoren (Network Address Functions and Operators, z.B. <, >, netmask(), broadcast(), masklen(), etc.)
- Sequenzfunktionen (Sequence Manipulation Functions, aktuellen und nächsten Wert einer Sequenz abfragen, oder neu initialisieren)
- Bedingte Anweisungen (Conditional Expressions, z.B. CASE, COALESCE, NULLIF, GREATEST, LEAST)
- Feldfunktionen und -operatoren (Array Functions and Operators, z.B. array_append(), array_cat(), array_to_string(), string_to_array(), etc.)
- Aggregierungsfunktionen (Aggregate Functions, z.B. Avg(), max(), min(), count(), etc.)
- Verschachtelte Ausdrücke (Subquery Expressions, z.B. EXISTS, IN, NOT IN, ANY, SOME, etc.)
- Zeilen- und Feldvergleichsoperatoren (Row and Array Comparisons, Vergleiche für Gruppen von Werten in Zeilen und Feldern)
- Seriengeneratoren (Set Returning Functions, können Reihen mit bestimmten Intervallen generieren)
- System Informationen (System Information Functions, z.B. Version(), current_database(), etc.)

- Systemadministrationsfunktionen (System Administration Functions, zum Setzen und Abfragen von run-time Konfigurationsparametern)

Index

Ein Datenbankindex ist eine Datenbankstruktur die die Performanz von Abfragen, Sortierungen und Manipulationen steigern kann. Dabei werden Teile der Daten (Schlüsselwerte) in eine Reihenfolge gebracht und Verweise auf die detaillierteren Daten vorgenommen. Indizes kann man über einzelne Spalten oder Kombinationen von Spalten definieren. In PostgreSQL kann man sogar Indizes über die Ergebnisse von Funktionen definieren. Je nach Datentyp kommen unterschiedliche Indextypen zum Einsatz. Manche Indextypen sind etwa auf Zahlenspalten spezialisiert, andere auf Textvergleiche, wieder andere auf Geometriedatentypen. GIST (Abkürzung für Generalized Search Tree) ist eine generelle Index Infrastruktur die einem Datenbankbenutzer oder Entwickler erlaubt seine eigenen Indexkritierien umzusetzen. Aus Benutzersicht ist es dabei wichtig, die richtigen Parameter zu verwenden um sicherzugehen dass der richtige Index verwendet wird. Bei Postgis wird GIST etwa für die räumlichen Indizes verwendet. Ob ein Index tatsächlich verwendet wird zeigt die Ausgabe des EXPLAIN Befehls, welcher den Abfrageplan (query plan) und die Laufzeiten zeigt.

Trigger

Ein Trigger ist ein Auslöser der Aktionen vor oder nach bestimmten Ereignissen (events) auslöst. So kann zum Beispiel ein Trigger veranlassen, dass eine abhängige Spalte neu berechnet wird, wenn ein Wert einer gewissen Spalte verändert wird. Die Ereignisse sind entweder "Insert", "Update" oder "Delete". Die allgemeine Syntax lautet:

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE funcname ( arguments )
```

Es wird unterschieden „before" and „after" Triggern. Before Trigger dienen etwa zur Datenmanipulation vor dem Einfügen/Update der Daten, after Trigger kann man etwa zum Archivieren von Daten in separaten Log-Tabellen verwenden. Weiters wird unterschieden zwischen per-row und per-statement Triggern unterschieden. Wie der Name sagt wird der per-row Trigger für jeden einzelnen betroffenen Record ausgeführt, der per-Statement Trigger einmal pro INSERT/UPDATE/DELETE statement, egal wie viele Records vom Statement betroffen sind. Per-row Trigger können performance-technisch problematisch sein, wenn sehr viele Records vom Statement betroffen sind. Trigger können erst einer Tabelle zugewiesen werden, wenn die referenzierte Triggerfunktion bereits existiert. Hier ein Beispiel:

```
CREATE TRIGGER insert_landwirtschaftsflaechen
  BEFORE INSERT
  ON test.landwirtschaftsflaechen
  FOR EACH ROW
  EXECUTE PROCEDURE test.insert_gis();
```

In diesem Beispiel wird vor jedem dem Ausführen des INSERT statements die Triggerfunktion insert_gis() ausgeführt.

Constraints

Constraints sind Einschränkungen die darüber wachen ob gewisse Kriterien eingehalten werden beim Einfüllen der Daten. Mit den Datentypen kann man zwar den Datentyp, nicht aber den Wertebereich sicherstellen. PostgreSQL kennt verschiedene Constraint Typen. Die so genannten **Check Constraints** erlauben der Datenspaltendefinition einen Ausdruck

mitzugeben der mit “true” evaluieren muss damit die Daten eingefüllt werden können. Im folgenden Beispiel wird sichergestellt dass in die Tabelle mega_cities nur Städte mit einer Einwohnerzahl von über 1 Million eingetragen werden können:

```
CREATE TABLE mega_cities (
    gid integer,
    name text,
    population numeric CHECK (population > 1000000)
);
```

Weiters gibt es die **NOT NULL** constraints die sicherstellen, dass in eine Spalte Werte eingetragen werden müssen. So könnte man etwa in der obigen Tabellendefinition zusätzlich sicherstellen, dass in die Spalten gid und name zwingend Werte eingefüllt werden:

```
CREATE TABLE mega_cities (
    gid integer NOT NULL,
    name text NOT NULL,
    population numeric CHECK (population > 1000000)
);
```

Ausserdem gibt es den UNIQUE constraint, der sicherstellt dass jeder eingegebene Wert eindeutig ist, also nicht mehr als ein mal vorkommt. So könnte man obige Tabellendefinition wie folgt verbessern, um sicherzustellen, dass die gid eindeutig ist.

```
CREATE TABLE mega_cities (
    gid integer UNIQUE NOT NULL,
    name text NOT NULL,
    population numeric CHECK (population > 1000000)
);
```

Primärschlüssel (Primary Key)

Der Primärschlüssel ist im Prinzip ein spezieller constraint als Kombination von UNIQUE und NOT NULL. Jede Tabelle kann nur einen Primary Key haben, aber mehr als einen UNIQUE NOT NULL constraint. In der obigen Tabellendefinition ist es z.B. sinnvoll die Spalte gid als Primary Key definieren:

```
CREATE TABLE mega_cities (
    gid integer PRIMARY KEY,
    name text NOT NULL,
    population numeric CHECK (population > 1000000)
);
```

Fremdschlüssel (Foreign Key)

Fremdschlüssel können erzwingen, dass in eine Spalte nur Werte eingetragen werden die in der Spalte einer anderen Tabelle vorkommen. So könnte ich etwa in der Tabelle mega_cities eine Spalte “country” haben in der nur Werte aus der Tabelle “countries”, Spalte “name” vorkommen dürfen. So kann ich etwa auch unterschiedliche Schreibweisen verhindern. Dies könnte über die folgenden zwei Tabellendefinition realisiert werden.

```
CREATE TABLE countries (
     gid integer PRIMARY KEY,
     name text NOT NULL UNIQUE
);

CREATE TABLE mega_cities (
     gid integer PRIMARY KEY,
     name text NOT NULL,
     country text REFERENCES countries (name),
     population numeric CHECK (population > 1000000)
);
```

Nun ergibt sich jedoch die Problematik, dass wir definieren müssen was passiert wenn in der Tabelle countries ein Wert gelöscht wird, der in der Tabelle mega_cities in der Spalte country referenziert wird. Wir haben verschiedene Möglichkeiten: “ON DELETE RESTRICT” verhindert das Löschen in der Tabelle “countries” wenn der Wert referenziert wird. “ON DELETE CASCADE” bewirkt, dass Records die nun durch die Löschung inkorrekte Referenzen haben automatisch gelöscht werden. Die Option “SET NULL” setzt den Wert auf Null und “SET DEFAULT” setzt zurück auf den default Wert, falls dieser gesetzt wurde. Hier die Variante mit dem kaskadierendem Löschen:

```
CREATE TABLE mega_cities (
      gid integer PRIMARY KEY,
      name text NOT NULL,
      country text REFERENCES countries (name) ON DELETE CASCADE,
      population numeric CHECK (population > 1000000)
);
```

View

Mit der Hilfe von Views kann man ein virtuelles Extrakt von anderen physisch existenten Tabellen oder Views herstellen, das sich dem Benutzer aus Abfragesicht wie eine Tabelle präsentiert. So kann man etwa Ausschnitte oder weniger heikle, z.b. aggregierte Daten, dem Benutzer zugänglich machen, falls man die heiklen detaillierten Daten nicht direkt zur Verfügung stellen kann, etwa aus Datenschutzgründen. Ausserdem kann man komplexe Tabellen für den Benutzer vereinfachen indem man nur die notwendigen Records und/oder Spalten zur Verfügung stellt. Schliesslich lassen sich mit Views Datenbankstrukturen konsistent halten, auch wenn sich die Strukturen der zugrundeliegenden Tabellen ändern. In PostgreSQL sind Views im Moment immer nur temporär und read only. Materialized Views können aber über Drittmodule nachgerüstet werden. Schreibbare Views können über Regeln (RULEs) realisiert werden.

Mit dem folgenden Befehl erstelle ich etwa einen View für europäische Länder als Extrakt eines weltweiten Datensatzes.

```
CREATE VIEW european_countries AS
      SELECT gid, the_geom, name, population FROM world_countries
      WHERE continent = 'Europe';
```

Mit “`DROP VIEW european_countries;`” kann man den View wieder löschen. Die Originaldaten in der(n) Ursprungstabelle(n) bleiben natürlich erhalten.

Regeln (RULEs)

Mit Regeln kann man das Default-Verhalten von INSERT, UPDATE und DELETE Statements überschreiben. Die Möglichkeiten überlappen sich etwas mit den Triggern in Kombination mit den Trigger-Funktionen aber es gibt einige Fälle bei denen nur entweder die Rules oder die Trigger funktionieren oder besser geeignet sind. Regeln manipulieren die ursprüngliche SQL Abfrage bevor sie ausgeführt wird und sind daher im Vergleich zu row-level Triggern häufig schneller/optimierter.

Hier ein Beispiel für eine DELETE Rule, bei der statt dem Löschen ein Update durchgeführt wird und das Archiv-Datum gesetzt wird:

```
CREATE OR REPLACE RULE landwirtschaftsflaechen_del AS
 ON DELETE TO lw.landwirtschaftsflaechen
 DO INSTEAD UPDATE lw.landwirtschaftsflaechen SET archive_date = now()
  WHERE landwirtschaftsflaechen.gid = old.gid AND
   landwirtschaftsflaechen.archive_date IS NULL;
```

Transaktionen (Transactions)

Transaktionen erlauben die Bündelung mehrerer Einzelaktionen in einen einzelnen Schritt (Alles oder Nichts wird ausgeführt, atomic). Nur bei erfolgreichem Abschluss aller Einzelaktionen innerhalb einer Transaktion kann diese abgeschlossen werden. Wenn dies nicht möglich ist, werden alle Einzelschritte rückgängig gemacht (rollback). Während die Transaktion läuft wird noch der alte Zustand ausgeliefert. PostgreSQL kennt bezüglich Transaktionen die Befehle “BEGIN” und “COMMIT” für den Beginn und Abschluss einer Transaktion. Dazwischen kann der Benutzer “SAVEPOINT”s setzen und “ROLLBACK”s bis zu einem SAVEPOINT machen. Beispiel:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Vererbung (Inheritance)

Vererbung ist ein Konzept aus dem objektorientierten Datenbankdesign. Kindtabellen können von ihren Elterntabellen Spaltendefinitionen und Daten erben. Nehmen wir an wir haben eine Tabelle Städte und wollen eine spezialisierte Tabelle Hauptstädte mit zusätzlichen Attributen erstellen.

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int
);

CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

Die Tabelle capitals erbt alle Spalten der Tabelle cities und fügt noch die Spalte “state” hinzu. Eine Tabelle kann auch von mehreren anderen Tabellen erben. Bei der Abfrage kann man festlegen ob man die vererbten Tabellen mit abfragen will oder nicht. So würde der folgende Befehl nur Städte und keine Hauptstädte retournieren. Wenn man das Schlüsselwort “ONLY” weglässt würde man sowohl Städte als auch Hauptstädte, also Resultate von beiden Tabellen, erhalten:

```
SELECT name, altitude
     FROM ONLY cities
     WHERE altitude > 500;
```

Man kann bei der Abfrage herausfinden von welcher Quelltabelle eine Antwort stammt, indem man die Tabellen Object-ID (tableoid) abfrägt und einen Join auf die Systemtabelle “pg_class” macht:

```
SELECT p.relname, c.name, c.altitude
     FROM cities c, pg_class p
     WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

Zu beachten ist, dass bei den Befehlen INSERT und UPDATE die Zieltabelle explizit angegeben werden muss. Wenn man z.B. eine Hauptstadt aktualisieren will, muss man es in der "capitals" Tabelle machen. Check constraints und NOT NULL constraints werden von der Eltern zur Kindtabelle vererbt, andere Constraint-Typen nicht. Es gibt noch ein paar weitere Feinheiten zu beachten. Bitte konsultieren Sie dazu die Dokumentation auf der PostgreSQL Dokumentation.

Rolle (role)

Die Zugriffsberechtigungen werden in PostgreSQL mit "Rollen" (role) und Zugriffsberechtigungen ("privileges") umgesetzt. Eine Rolle kann entweder ein Benutzer (user) sein oder eine Gruppe (group). Berechtigungen können entweder auf Benutzer- oder Gruppenebene gesetzt werden. Rollen werden mit "`CREATE ROLE name`" und "`DROP ROLE name`" erstellt und gelöscht. Beim Installieren von PostgreSQL wird automatisch ein Superuser (per default postgres) erstellt. Beim Erstellen einer Rolle kann man auch Superuser-Rechte mit abgeben. Existierenden Rollen kann man generelle Superuser-Rechte hinzufügen oder entfernen mit dem Befehl "`ALTER ROLE`".

Zugriffsberechtigungen (privileges)

Wenn ein neues Objekt kreiert wird bekommt es automatisch einen Besitzer (owner). Der Besitzer kann später geändert werden. Der Besitzer der ein Objekt erstellt hat automatisch die vollen Rechte darauf. Zusätzlich kann man anderen Rollen (Benutzer und Gruppen) Rechte geben. Rechte können auf jedem Objekt vergeben werden, also z.B. Tabellen, Spalten, Records, etc. Zu Beachten ist, dass auf Abhängigkeiten ebenfalls die Rechte vergeben werden müssen. Verwendet also eine Tabelle eine Sequenz, so müssen die Rechte auf der Sequenz ebenfalls vergeben werden. Rechte werden mit dem Befehl "`GRANT`" vergeben werden und mit dem Befehl "`REVOKE`" entfernt. Z.B:

```
GRANT UPDATE ON accounts TO Joe;
REVOKE ALL ON accounts FROM public;
```

PostgreSQL kennt die folgenden Rechte:

- SELECT
- INSERT
- UPDATE
- DELETE
- REFERENCES
- TRIGGER
- CREATE
- CONNECT
- TEMPORARY
- EXECUTE
- USAGE

PL/pgSQL und andere DB-serverseitige Programmiersprachen

PL/pgSQL (Procedural Language/PostgreSQL Structured Query Language) ist die PostgreSQL eigene Programmiersprache zum Erzeugen von eigenen Datentypen, Funktionen, Trigger Prozeduren, Berechnungen und Operatoren. Sie ist an Oracle PL/SQL angelehnt. Als alternative Programmiersprachen stehen neben PL/pgSQL auch TCL, Perl, Python, C und Java zur Verfügung. Implementationen in C, PL/pgSQL und Java sind typischerweise performanter als interpretierte Sprachen wie Perl und Python. Letztere eignen sich aber besser für schnelles Entwickeln. Während PL/pgSQL standardmässig immer zur Verfügung steht, müssen andere Sprachen speziell kompiliert oder als Zusatzmodule nachgerüstet werden.

Beispiel für ein PL/pgSQL Programm:

```
CREATE OR REPLACE FUNCTION test.insert_gis()
  RETURNS "trigger" AS
$BODY$
BEGIN
      IF NEW.create_date IS NULL THEN
            NEW.create_date := now();
            NEW.archive_date := NULL;
            NEW.id := NEW.gid;
            -- Remember who changed the record
            NEW.last_user := current_user;
      END IF;
      RETURN NEW;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Dieses Pl/PgSQL Programm (der fett-gedruckte Code) nimmt vom INSERT statement Daten entgegen, ergänzt und modifziert bestimmte Werte und schreibt danach die modifizierten Werte in die Datenbank. Leider können bei der Verarbeitung von alten und neuen Records (OLD, NEW) in Triggerfunktionen keine Variablen (im Sinne von assoziativen Feldern) verwendet werden. Dies erschwert das Schreiben von generischen Triggerfunktionen. Derartige generische Funktionen müssen in einer anderen Programmiersprache, wie im Beispiel unten, geschrieben werden.

Beispiel für ein pl/Perl Programm:

```
CREATE OR REPLACE FUNCTION test.update_gis()
  RETURNS "trigger" AS
$BODY$
my ($sql_attrib,$sql_insert,$sql_values,$rv);

if ($_TD->{old}{archive_date} != undefined) {
  return "SKIP"; #quietly disallow
}
else {
  if ($_TD->{new}{archive_date} == undef) {
    $sql_insert = "INSERT INTO ".$_TD->{table_schema}."."$_TD-
>{table_name}." (";
    $sql_values = '';
    #prepare SQL statement to get all column_names and
    #data types (udt_name)
    $sql_attrib = "SELECT column_name, udt_name FROM
information_schema.columns WHERE table_schema = '".$_TD->{table_schema}."'
AND table_name = '".$_TD->{table_name}."';";
    my $rv = spi_exec_query($sql_attrib);
    my $nrows = $rv->{processed};
    #loop over all column names and concatenate SQL INSERT statement
```

```
    foreach my $rn (0 .. $nrows - 1) {
      my $row = $rv->{rows}[$rn];
      if ($row->{column_name} ne "gid") {
        if ($row->{column_name} eq "id") {
          $sql_insert .= "id,";
          $sql_values .= $_TD->{old}{gid}.",";
        }
        elsif ($row->{column_name} eq "last_user") {
          $sql_insert .= "last_user,";
          $sql_values .= "'".$_TD->{old}{last_user}."',";
        }
        elsif ($row->{column_name} eq "create_date") {
          $sql_insert .= "create_date,";
          $sql_values .= "'".$_TD->{old}{create_date}."',";
        }
        elsif ($row->{column_name} eq "archive_date") {
          $sql_insert .= "archive_date,";
          $sql_values .= "now(),";
        }
        else {
          $sql_insert .= "\"".$row->{column_name}."\",";
          if ($_TD->{old}{$row->{column_name}} eq undefined || $_TD->{old}
{$row->{column_name}} eq "") {
            $sql_values .= "NULL,";
           }
           else {
             if ($row->{udt_name} eq "text" || $row->{udt_name} eq
"geometry") {
               $sql_values .= "'".$_TD->{old}{$row->{column_name}}."',";
             }
             else {
               $sql_values .= $_TD->{old}{$row->{column_name}}.",";
             }
          }
        }
      }
    }
    chop($sql_insert);
    chop($sql_values);
    $sql_insert .= ") VALUES (".$sql_values.");";
    $rv = spi_exec_query($sql_insert);
    $_TD->{new}{create_date} = "now()";
    # Remember who changed the record
    $rv = spi_exec_query("SELECT current_user;");
    $_TD->{new}{last_user} = $rv->{rows}[0]->{current_user};
    return "MODIFY";
  }
  else {
    return "SKIP";
  }
}
$BODY$
LANGUAGE 'plperl' VOLATILE;
```

Dieses plperl Programm wird von einem UPDATE Trigger aufgerufen. Anstelle eines Updates wird der Record dupliziert und der alte Stand, versehen mit einem Ablaufdatum in die gleiche Tabelle mittels INSERT neu eingefügt. Dieser Baustein dient der Historisierung von Tabellen.

Derzeitige Limitierungen von PostgreSQL

- Keine Table-Joins über verschiedene Datenbanken hinweg ohne Zusatzwerkzeuge (geht auch bei Oracle nicht) – aber es werden verschiedene Schemen innerhalb einer Datenbank unterstützt. Abfragen und Joins über mehrere Datenbanken hinweg sollen ab der kommenden Version 8.4 unterstützt werden.
- Views sind “read only” - diese Limitierung kann aber mit Hilfe von RULEs umgangen werden. Siehe Dokument „Historisierte Tabellen“.

Derzeitige Limitierungen von Postgis

- Keine direkte Topologie in der Datenhaltung (nur on the fly)
- Limitierter Support für Kurven