Historisierung von PostGIS-Daten als Grundlage zur Langzeitarchivierung von Geodaten

Andreas Neumann
Stadt Uster
GIS-Kompetenzzentrum
Oberlandstrasse 78
CH-8610 Uster
andreas.neumann@stadt-uster.ch

Dr. Horst Düster
Amt für Geoinformation Kantons Solothurn
SO!GIS Koordination
Rötistrasse 4
CH-4501 Solothurn
horst.duester@bd.so.ch

Die Dimension Zeit wird in vielen GEO-Datenmodellen vernachlässigt, obwohl sie neben den Geometrie- und Attributdaten eine ebenso wichtige Dimension im GIS darstellt. Häufig ist es gewünscht, alte Datenstände einer Datenbank zu rekonstruieren. Zudem ist eine Nachvollziehbarkeit nützlich: wer hat wann welche Werte in welchem Record geändert? Zeitliche Analysen greifen ebenfalls auf historische Datenstände zurück. Schliesslich könnte der Datenbankadministrator das Historisierungslog dazu verwenden Änderungen gezielt rückgängig zu machen. Schliesslich wird mit dem Historisierungsansatz ein räumlich-zeitliches Archiv über die digitalen Geoinformationen des betrachteten Raumes unterhalten, das im laufenden Betrieb unterhalten wird. Das Kopien auf externe Speichermedien ist nicht notwendig.

Mit geeigneten Datenbankmitteln ist es möglich, sämtliche Änderungen in geographischen Datenbanken weitgehend automatisch zu historisieren und auszuwerten. Die Präsentation zeigt wie mit einem System von Triggern, Regeln, Views, sowie Stored Procedures eine historisierte Datenbank entsteht. Das System wurde auf Basis Postgis und QuantumGIS umgesetzt, lässt sich aber auch auf andere Systeme übertragen die die nötigen Datenbank-Features zur Verfügung stellen (z.B. Oracle Spatial). Umgesetzt wurden 2 verschiedene Ansätze: Beim **Repository-Ansatz** des Kantons Solothurn werden Daten ausgecheckt, extern bearbeitet und danach inkrementell wieder eingecheckt. Oder es werden Daten von externen Produktionssystemen eingecheckt und historisiert. Der **Live-Historisierungsansatz** wie er bei der Stadt Uster zum Einsatz kommt geht auf einen Ansatz der Firma Varlena ((http://www.varlena.com/GeneralBits/Tidbits/tt.pdf) sowie das alte „TimeTravel“ Feature von PostgreSQL zurück. Interessant ist die Tatsache, dass in früheren PostgreSQL Versionen (bis Version 6) das sogenannte TimeTravel fixer Bestandteil der Datenbank war. Dieses Feature wurde jedoch in späteren Versionen entfernt, da der überwiegende Anteil der Nutzer mehr Interesse an besserer Performanz als an Historisierung hatte.

Bei der gewählten Strategie werden Records nie gelöscht oder bei Änderungen überschrieben, sondern archiviert. Es werden wie beim ersten Ansatz 2 Datumsstempel verwendet: „create_date“ (wird beim Anlegen eines Records auf den aktuellen Zeitstempel gesetzt) und „archive_date“. Das „archive_date“ wird dann gesetzt, wenn der Record durch Löschung oder Aktualisierung obsolet wird. Die jeweils aktuelle Version der Tabelle wird über einen View dargestellt, der diejenigen Records selektiert, die noch kein „archive_date“ gesetzt haben(archive_date IS NULL). Ältere Stände können ebenfalls über Views oder SELECT Statements abgeleitet werden in dem über create_date und archive_date die zum damaligen Zeitpunkt gültigen Records selektiert werden. Zusätzlich zur Datumsspalte wird auch der verantwortliche Nutzer/Bearbeiter mitabgespeichert werden (create_user und last_user), sowie Informationen zur Veränderung (change_type, change_affected_columns). Letzteres dient zur späteren Visualisierung oder besseren

Nachverfolgung der Änderungen. Falls das Datenvolumen oder die Änderungsrate einer Datentabelle beim gewählten Historisierungsansatz zu einem Performance-Problem wird, kann man die Historisierungsdaten auch in separate Tabellen auslagern.

Die für den Historisierungsansatz nötigen Regeln und Trigger richtet der Datenbankmanager einmal ein. Danach kann der Desktop-GIS Nutzer die historisierten Views wie normale Datenbanktabellen benutzen. Die Nachführung der Datumsstempel, Benutzer- und Änderungsinformationen verläuft in der Datenbank vom Benutzer unbemerkt im Hintergrund ab. Die dafür benötigten Spalten können im View sichtbar gemacht oder versteckt werden. Die Desktop-GIS Software muss nicht angepasst oder verändert werden. Lediglich für das benutzerfreundliche Laden alter Datenstände muss in das GUI der Desktop-GIS Software eingegriffen werden, etwa durch ein separates Plugin.

Die Präsentation zeigt die technische Umsetzung, die Vor- und Nachteile der beiden Ansätze, sowie einige Werkzeuge (PL/pgSQL oder PL/Perl-Funktionen) die das Einrichten von und den Umgang mit historisierten Tabellen erleichtern. Zudem werden die Abgleichungsroutinen des Solothurner Geodatenrepositories vorgestellt, die den Abgleich mit externen Daten ermöglichen. Schliesslich werden einige Gedanken formuliert, wie man Änderungen in historisierten Tabellen visualisieren und auswerten könnte.

Umsetzung des Live-Historisierungsansatzes:

Folgend sollen die notwendigen Schritte zur Implementierung des Live-Historisierungsansatzes gezeigt werden. In der Praxis werden diese Arbeitsschritte von einer PL/Perl oder PL/pgSQL Funktion automatisiert, trotzdem ist es vom Verständnis her sinnvoll, die Mechanismen dahinter zu kennen. Daher werden diese hier Schritt für Schritt aufgezeigt:

Anlegen von zentralen generischen Triggerfunktionen für Insert und Update:

Vorbereitung: Die Triggerfunktionen verwenden einen neuen Datentyp „change_type“ der zuerst wie folgt definiert werden muss:

```
CREATE TYPE change_type AS ENUM ('insert','update','delete');
```

Für die Historisierung müssen zwei Triggerfunktionen erstellt werden. Diese werden generisch geschrieben, können daher einmal zentral abgelegt (etwa im 'public' Schema) und für mehrere historisierte Tabellen verwendet werden. Es ist relativ schwierig mit PL/pgSQL generische Trigger-funktionen zu schreiben die mit beliebigen Spaltennamen umgehen können. Daher wurden diese beiden Triggerfunktionen mit PL/Perl umgesetzt.

1.Triggerfunktion „insert_timegis()“

Diese Triggerfunktion wird vor einem neuen INSERT Statement ausgeführt. Sie setzt das „create_date“ auf „now()“ (jetzt), das „archive_date“ auf NULL, die „id“ auf den Wert der Spalte „gid“, den „create_user“ auf den derzeit eingeloggten Datenbankuser, den „change_type“ auf 'insert' und die „change_affected_columns“ auf die Spalten für die tatsächlich Daten, und keine „NULL-Werte“ geliefert wurden. Code der Funktion siehe Anhang.

2.Triggerfunktion „update_timegis()“

Diese Triggerfunktion führt das UPDATE auf dem aktuellen Record aus, führt aber parallel dazu ein INSERT Statement aus mit einem Record der die alten Daten vor dem UPDATE enthält. Zusätzlich wird in dem neu einzufügenden Record (der die alten Daten enthält), das „archive_date“ auf „now()“ gesetzt, also der Record damit auf obsolet gesetzt. Der Wert des Attributs „id“ wird auf den Wert des ursprünglichen Original-Records gesetzt damit die Verbindung mit den historischen Daten gegeben ist. Beim aktualisierten Record wird das „create_date“ auf „now()“ gesetzt, das „archive_date“ auf 'NULL', der „last_user“ auf den 'current_user', der „change_type“ auf 'update' und die „change_affected_columns“ auf die Spalten in denen tatsächlich Änderungen vorgenommen wurden. Code der Funktion siehe Anhang.

Zusätzlich zu den Historisierungsspalten können beide Triggerfunktionen auch noch die Fläche und Länge von Polygon/Multipolygon respektive Linestring/Multilinestring-Features aktualisieren.

Manipulation der zu historisierenden Tabellen

1. Anlegen von zusätzlichen Spalten auf Ursprungstabelle

Jeder zu historisierenden Tabelle werden 7 zusätzliche Spalten hinzugefügt werden: „create_date“ (timestamp), „archive_date“ (timestamp), „create_user“ (text), „last_user“ (text), id (integer, muss gleich wie gid sein), change_type (enumeration), change_affected_column (text). Die id dient zum Verbinden von abgeleiteten Records mit der ursprünglichen gid. Alle abgeleiteten (historisierten) Records haben die gleiche id wie die ursprüngliche gid bei der Erstellung eines Records. Die gid selber muss eindeutig sein, da sie den Primärschlüssel darstellt. Die Spalte change_type zeigt an, ob der Record gerade eingefügt wurde (insert), upgedatet wurde (update) oder gelöscht (delete). Die Spalte change_affected_column zeigt an welche Spalten von den Veränderungen betroffen wurden.

Beispiel:

```
CREATE TABLE test.landwirtschaftsflaechen
(
  gid serial NOT NULL,
  id integer,
  fruchtart text NOT NULL,
  landwirt text NOT NULL,
  area numeric,
  the_geom geometry,
  create_date timestamp without time zone,
  archive_date timestamp without time zone,
  last_user text,
  create_user text,
  change_type change_type,
  change_affected_columns text,
  CONSTRAINT pkey_lwflaechen_gid PRIMARY KEY (gid),
  CONSTRAINT enforce_dims_the_geom CHECK (ndims(the_geom) = 2),
  CONSTRAINT enforce_geotype_the_geom CHECK (geometrytype(the_geom) = 'MULTIPOLYGON'::text OR
the_geom IS NULL),
  CONSTRAINT enforce_srid_the_geom CHECK (srid(the_geom) = 21781)
)
WITH (OIDS=FALSE);
```

2. Erstellen einer „Delete-Rule“ auf Ursprungstabelle

Die Delete-Rule überschreibt das DELETE statement und wandelt es in ein UPDATE Statement um, welches das „archive_date“ auf den jetzigen Zeitpunkt setzt und den

„last_user“ auf den aktuellen Datenbank-User setzt. Zudem wird der change_type für den „gelöschten“ Record auf „delete“ gesetzt.

Beispiel:

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

3. Erstellen von Triggern auf Ursprungstabelle

Für die Ursprungstabelle müssen noch 2 Trigger angelegt werden. Diese rufen die beiden generischen Triggerfunktionen (insert_timegis() und update_timegis()) auf. Es handelt sich um einen INSERT und einen UPDATE Trigger, in beiden Fällen um einen 'row-level before' Trigger.

Beispiel:

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

CREATE TRIGGER update_landwirtschaftsflaechen
  BEFORE UPDATE
  ON test.landwirtschaftsflaechen
    FOR EACH ROW
    EXECUTE PROCEDURE update_timegis();
```

4. Erstellen von Indizes auf der Ursprungstabelle

Damit man später effiziente Selektionen auf dem historisierten Datenbestand durchführen kann, sollte man Indizes mindestens auf den Spalten „create_date“ und „archive_date“ anlegen. Optional könnte man Indizes auch auf den Spalten „create_user“, „last_user“, „change_type“ und „change_affected_columns“ anlegen, wenn man diese effizient auswerten möchte.

Beispiel:

```
CREATE UNIQUE INDEX in_test_landwirtschaftsflaechen_gid_btree ON test.landwirtschaftsflaechen
 USING btree (gid); CREATE INDEX in_test_landwirtschaftsflaechen_id_btree ON test.landwirtschafts-
flaechen
 USING btree (id); CREATE INDEX in_test_landwirtschaftsflaechen_create_date_btree ON test.landwirt-
schaftsflaechen
 USING btree (create_date); CREATE INDEX in_test_landwirtschaftsflaechen_archive_date_btree ON
test.landwirtschaftsflaechen
 USING btree (archive_date);
```

5. Erstellen eines Views für den aktuellen Datenbestand

Schliesslich braucht es einen View (einen Tabellenausschnitt), der aus der Mastertabelle die Records filtert die derzeit aktuell sind. Dies geschieht über einen View, der alle Records selektiert deren archive_date „NULL“ ist. Dies sind die aktuell gültigen Records. Für den gerade erstellten View sollte in der Tabelle „public.geometry_columns“ ein Eintrag pro Geometriespalte erstellt werden.

Beispiel:

```
CREATE OR REPLACE VIEW test.lwflaechen AS
  SELECT * FROM test.landwirtschaftsflaechen
    WHERE landwirtschaftsflaechen.archive_date IS NULL;
```

6. Erstellen von Regeln für den eben erstellten View

Views können in PostgreSQL nicht direkt editiert werden. Da die Datenbank nicht wissen kann in welchen Tabellen die Daten eines Views abgelegt werden sollen, müssen dazu manuell Regeln erstellt werden. Dafür stehen in diesen Regeln fast unbegrenzte Möglichkeiten für die Datenmanipulation offen, auch über mehrere Tabellen hinweg. In unserem Fall müssen wir eine 'insert', eine 'update' und eine 'delete' Rule erstellen die definieren welche Spalten in welchen Tabellen (in unserem Fall primär in der Ursprungstabelle) manipuliert werden. Dabei müssen lediglich die Spalten berücksichtigt werden die nicht automatisch über die Sequenzen und Triggerfunktionen manipuliert werden – also alle Spalten ausser gid/id, Länge/Fläche und den anderen von der Historisierungsfunktion verwendeten Spalten.

Beispiel:

```
CREATE OR REPLACE RULE "_DELETE" AS
  ON DELETE TO test.lwflaechen DO INSTEAD
   DELETE FROM test.landwirtschaftsflaechen WHERE landwirtschaftsflaechen.gid = old.gid;


CREATE OR REPLACE RULE "_INSERT" AS
  ON INSERT TO test.lwflaechen DO INSTEAD
   INSERT INTO test.landwirtschaftsflaechen (fruchtart, landwirt, the_geom)
     VALUES (new.fruchtart, new.landwirt, new.the_geom);


CREATE OR REPLACE RULE "_UPDATE" AS
  ON UPDATE TO test.lwflaechen DO INSTEAD
   UPDATE test.landwirtschaftsflaechen SET landwirt = new.landwirt, fruchtart = new.fruchtart,
    the_geom = new.the_geom
    WHERE landwirtschaftsflaechen.gid = new.gid;
```

7. Erstellen von historischen Views (optional)

Optional können nun auch historische Views angelegt werden: Es werden alle Records selektiert, deren „create_date“ älter als der Zeitstempel des gewünschten Datenstands ist und deren „archive_date“ entweder „NULL“ oder jünger als der Zeitstempel des anvisierten Datenstands ist.

Beispiel:

```
CREATE OR REPLACE VIEW test."lwflaechen_2008-05-15" AS
 SELECT * FROM test.landwirtschaftsflaechen
   WHERE landwirtschaftsflaechen.create_date < '2008-05-16 00:00:00'::timestamp
    AND (landwirtschaftsflaechen.archive_date IS NULL OR
     landwirtschaftsflaechen.archive_date > '2008-05-16 00:00:00'::timestamp )
   ORDER BY landwirtschaftsflaechen.gid;
```

Pl/Perl Funktionen für das Management von historisierten Tabellen/Views

add_history()

Die Funktion „add_history('name_schema','name_table','name_schema_view','name_view')“ führt die oben erwähnten Schritte 1-6 automatisch durch, sofern der Primärschlüssel 'gid' heisst und die benötigten Spaltennamen noch frei sind. Die Funktion kann auch auf

bereits historisierten Tabellen ausgeführt werden. Es werden dann nur die fehlenden Komponenten (Regeln, Trigger, Indizes oder Views) erstellt die noch nicht vorhanden sind. Bereits existierende Komponenten werden nicht überschrieben.

create_historic_view()

Die Funktion „create_historic_view('name_table_schema', 'name_table', 'name_schema_view', 'name_historic_view', 'timestamp')“ erstellt einen View mit dem historischen Zustand entsprechend dem spezifizierten Zeitstempel.

show_history_for_record()

Die Funktion „show_history_for_record('name_historicized_table',gid)“ zeigt für einen einzelnen Record in der historisierten Tabelle die Änderungshistorie an. Pro Änderung wird ein Record ausgegeben, mit Zeitstempel, Änderungsart, betroffene Spalten, alter Wert und neuer Wert.

Umsetzung des Repository-Ansatzes:

Sollen Massendaten in die Datenbank eingepflegt werden, kann der Live-Historisierungs Ansatz nicht verwendet werden. In diesem Fall wird ein im Jahre 2003 vom Kanton Solothurn entwickeltes Verfahren eingesetzt, das mit Quellcode Versionierungssystenen wie SVN oder CVS vergleichbar ist. In der Regel wird in Solothurn nicht auf den produktiven Daten gearbeitet. Vor der Bearbeitung eines Datums wird eine Kopie aus dem Repository gezogen, die dann unabhängig von den produktiven Daten bearbeitet wird. So können nach der Bearbeitung diverse Qualitätssicherungsschritte durchgeführt werden, und erst wenn diese Erfogreich durchlaufen sind, werden die Änderungen in das Repository überführt. Eine parallele Bearbeitung verschiederer Bearbeiter an einem Datum ist möglich. Eine Konfliktbehandlung ist momentan nicht realisiert, aber auf der Basis des Repositories ohne weiteres möglich. Auf die gleiche Weise lassen sich auf externen Produktionssystemen erzeugte Daten (z.B. Amtliche Vermessung) in das zentrale Repository einpflegen.

Eine Abgleichsroutine steuert den Prozess des Datenabgleiches. Dazu werden die neuen Daten zunächst in ein paralleles Postgres-Schema eingelesen. Das PL/pgSQL Script „updatelayer('newSchema.newTable', 'oldSchema.oldTable')“ vergleicht die beiden Tabellenstrukturen und gleicht diese ab, indem nur die Veränderungen zwischen den beiden Zeitständen berücksichtigt werden. Dieser Prozess dauert z.B. für die Daten der amtlichen Vermessung mit mehreren Millionen Objekten weniger als 10 Minuten.

Vor dem Abgleich werden diverse Prüfungen durchgeführt:

- Existieren beide Tabellen?
- Wie heissen die Geometriespalten und sind diese in der Tabelle „public.geometry_columns“ registriert?
- Entsprechen die Tabellenstrukturen von alter und neuer Tabelle einander?
- Gibt es Primärschlüssel?
- Existieren die für die Historisierung notwendigen Spalten in der alten (zentralen) Tabelle?

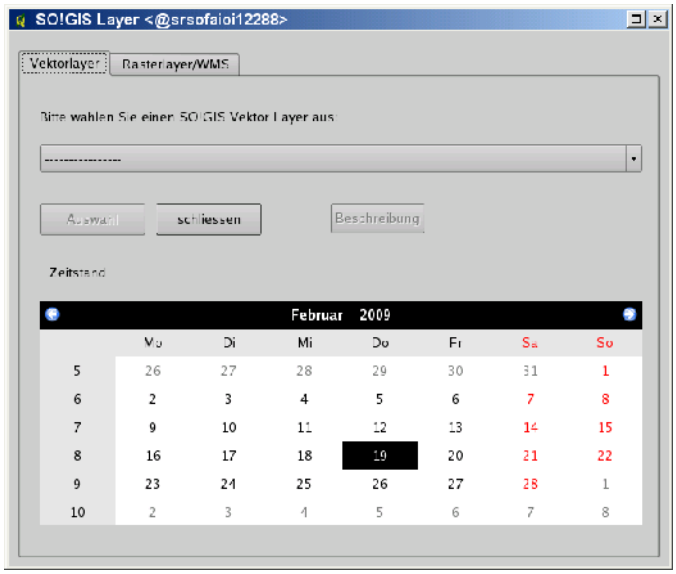
Der eigentliche Abgleich erfolgt dann in zwei Schritten:

1. Finden aller Records, die in der produktiven Tabelle enthalten sind und nicht in der neuen Tabelle. Diese werden als gelöscht behandelt und in der produktiven Tabelle als archiviert markiert.

2. Finden aller Records, die in der neuen Tabelle enthalten sind und in der produktiven Tabelle nicht. Diese werden als neu behandelt und in die produktive Tabelle eingefügt.

Alle restlichen Records werden nicht berücksichtigt. Nach diesen zwei Schritten sind die nicht archivierten Records der produktiven Tabelle und die neue Tabelle identisch. Damit die archivierten Records nicht den produktiven Bereich der Produktionstabelle belasten, werden über einen partiellen Index nur die produktiven Records indiziert. Damit existieren für die angehängten GIS-Systeme in der Regel nur die produktiven Objekte der Tabelle.

Die GIS-Anwender können auf sehr einfache Weise auf die verschiedenen Zeitstände zugreifen. Dazu ist im Kanton Solothurn mit PyQt ein benutzerfreundlicher Ladedialog für das Desktop-GIS QGIS entwickelt worden, der das Laden von aktuellen wie auch historischen Sichten auf die Daten über einen Kalender erlaubt:



Code siehe: http://www.sogis1.so.ch/postgis_historisation/

Visualisierung der Änderungshistorie

Beim Erstellen historischer Zeitstände ist es relevant die Änderungshistorie zu kennen. Die zusätzlichen Spalten der Historisierung können für eine Visualisierung der Änderung entlang einer Zeitachse herangezogen werden. Über „change_type“ und „change_affected_columns“ kann die Art der Änderung visualisiert werden. „create_user“ und „last_user“ geben Auskunft wer die Records manipuliert hat.

Die Änderungen könnten in zeitlich unterschiedlichen Granularitäten visualisiert werden, z.B. Gruppiert nach Stunden, Tagen, Wochen, Monaten. Balken- oder Liniendiagramme könnten die Änderungs-aktivitäten, auch getrennt nach User, darstellen.

Derzeit wird mit einem QGIS-Plugin experimentiert das über ein Kalenderwidget das Laden historischer Stände erlaubt und über interaktive SVG-Diagramme die Historie des Datensatzes visualiert. Die Visualisierung ist noch stark in Entwicklung – deshalb werden hier noch keine Screenshots gezeigt.

Zusammenfassung

Die vorgestellten beiden Varianten zur Live-Historisierung und zum Repository-Abgleich haben sich bereits in der Praxis bewährt (Kanton Solothurn, Stadt Uster). Beide Ansätze sind noch nicht vollständig kompatibel zu einander, da sie bei der Historisierung von teilweise unterschiedlichen Philosophien und Infrastrukturen ausgehen, sind aber vom Ansatz her ähnlich umgesetzt. Performancemässig könnte zumindest der Live-Historisierungsansatz aus Uster noch etwas verbessert werden. Feedback diesbezüglich wird gerne entgegengenommen.

Über die Hilfsfunktionen „add_history()“, „create_historic_view()“, und „updatelayer()“ kann die Historisierung mit verhältnismässig wenig Aufwand eingeführt und gewartet werden. Die Funktion „show_history_for_record()“ hilft für die Verfolgung von Änderungen eines einzelnen Records. Die Arbeiten zur Darstellung der Änderungshistorie haben erst begonnen. Ebenso gibt es noch kaum Ansätze zum gezielten Rollback von Änderungen unter zu Hilfenahme der Änderungshistorie. Das Themenfeld „Historisierung“ bietet somit noch weiteres Potential für Forschungen, Entwicklungen und Verbesserungen.

Anhang Funktionen, Trigger und Regeln

Im Anhang sind lediglich die Funktionen „insert_timegis()“ und „update_timegis()“ enthalten. Sämtliche weiteren Funktionen können (aufgrund des grösseren Codeumfangs und der Aktualität) unter http://www.sogis1.so.ch/postgis_historisation/ heruntergeladen werden.

Triggerfunktion insert_timegis()

```
CREATE OR REPLACE FUNCTION insert_timegis()
  RETURNS trigger AS

$BODY$
my ($sql,$rv, $affected_columns);
if ($_TD->{new}{create_date} == undef) {
$_TD->{new}{create_date} = "now()";
$_TD->{new}{archive_date} = undef;
$_TD->{new}{id} = $_TD->{new}{gid};
$_TD->{new}{create_user} = "current_user()";

# Remember who inserted the record and set change_type
$rv = spi_exec_query("SELECT current_user;");
$_TD->{new}{create_user} = $rv->{rows}[0]->{current_user};
$_TD->{new}{change_type} = "insert";

# prepare SQL statement to get all column_names and data types (udt_name)
$sql_attrib = "SELECT column_name FROM information_schema.columns WHERE table_schema = '".$_TD-
>{table_schema}."' AND table_name = '".$_TD->{table_name}."' AND column_name NOT IN
('gid','id','create_date','archive_date','create_user','last_user','change_type','change_affected_co
lumns');";

my $rv = spi_exec_query($sql_attrib);
my $nrows = $rv->{processed};
```

```
# loop over all column names, concatenate text string for change_affected_columns,
# also deal with area and length
$affected_columns = "";
foreach my $rn (0 .. $nrows - 1) {
	my $row = $rv->{rows}[$rn];
	if ($row->{column_name} eq "area" || $row->{column_name} eq "flaeche") {
		$rv = spi_exec_query("SELECT ST_AREA('$_TD->{new}{the_geom}') AS polyarea");
		$_TD->{new}{area} = $rv->{rows}[0]->{polyarea};
	}
	elsif ($row->{column_name} eq "length" || $row->{column_name} eq "laenge") {
		$rv = spi_exec_query("SELECT ST_LENGTH('$_TD->{new}{the_geom}') AS linelength");
		$_TD->{new}{length} = $rv->{rows}[0]->{linelength};
	}
	else {
		if ($_TD->{new}{$row->{column_name}}) {
			$affected_columns .= $row->{column_name}.",";
		}
	}
}

chop($affected_columns);
$_TD->{new}{change_affected_columns} = $affected_columns;

}
return "MODIFY";
$BODY$
  LANGUAGE 'plperl' VOLATILE
  COST 100;
```

Triggerfunktion update_timegis()

```
CREATE OR REPLACE FUNCTION update_timegis()
  RETURNS trigger AS
$BODY$
my ($sql_attrib,$sql_insert,$sql_values,$rv);

if ($_TD->{old}{archive_date} != undef) {
  return "SKIP"; #quietly disallow
}
else {
  if ($_TD->{new}{archive_date} == undef) {
    # prepare insert statement for new record representing the old (archived) data values
    $sql_insert = "INSERT INTO ".$_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};
    my $change_affected_columns = "";
    my $areaColumnName = undef;
    my $lengthColumnName = undef;

    #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 "archive_date") {
          $sql_insert .= "archive_date,";
          $sql_values .= "now(),";
        }
        else {
          $sql_insert .= "\"".$row->{column_name}."\",";
          # deal with NULL values
          if ($_TD->{old}{$row->{column_name}} eq undefined || $_TD->{old}{$row->{column_name}} eq
"") {
            $sql_values .= "NULL,";
          }
          else {
            # deal with data types that need to be quoted
            if ($row->{udt_name} eq "text" || $row->{udt_name} eq "varchar" || $row->{udt_name} eq
"geometry" || $row->{udt_name} eq "timestamp" || $row->{udt_name} eq "change_type" || $row-
>{udt_name} eq "bool" || $row->{udt_name} eq "ltree") {
              $sql_values .= "'".$_TD->{old}{$row->{column_name}}."',";
              # now test if new value is different from old value (quoted data types)
              # exclude columns that are used only for historization
              if ($row->{column_name} ne "change_type" && $row->{column_name} ne
"change_affected_columns" && $row->{column_name} ne "create_date" && $row->{column_name} ne
"create_user" && $row->{column_name} ne "last_user") {


                if ($_TD->{new}{$row->{column_name}} ne $_TD->{old}{$row->{column_name}}) {
                  $change_affected_columns .= $row->{column_name}.",";
                }
              }
            }
            # now deal with data types that don't need to be quoted
            else {
              $sql_values .= $_TD->{old}{$row->{column_name}}.",";
              # eal with area and length
              if ($row->{column_name} eq "area" || $row->{column_name} eq "flaeche") {
                $areaColumnName = $row->{column_name};
              }
              if ($row->{column_name} eq "length" || $row->{column_name} eq "laenge") {
                $lengthColumnName = $row->{column_name};
              }
              #now test if new value is different from old value (non-quoted data types)
              if ($row->{column_name} ne "area" && $row->{column_name} ne "length") {
                if ($_TD->{new}{$row->{column_name}} != $_TD->{old}{$row->{column_name}}) {
                  $change_affected_columns .= $row->{column_name}.",";
```

```
              }
            }
          }
        }
      }
    }

    chop($sql_insert); #get rid of the last comma
    chop($sql_values); #get rid of the last comma
    chop($change_affected_columns); #get rid of the last comma
    $sql_insert .= ") VALUES (".$sql_values.");";

    # execute the insert statement
    $rv = spi_exec_query($sql_insert);

    # now deal with the values that need update (current, up-to-date record)
    $_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};
    # set change_type
    $_TD->{new}{change_type} = 'update';
    # set change_affected_columns
    $_TD->{new}{change_affected_columns} = $change_affected_columns;

    # see if we need to update the area or length
    if ($areaColumnName) {
      if ($_TD->{new}{the_geom} ne $_TD->{old}{the_geom}) {
        $rv = spi_exec_query("SELECT ST_AREA('$_TD->{new}{the_geom}') AS polyarea");
        $_TD->{new}{$areaColumnName} = $rv->{rows}[0]->{polyarea};
      }
    }
    if ($lengthColumnName) {
      if ($_TD->{new}{the_geom} ne $_TD->{old}{the_geom}) {
        $rv = spi_exec_query("SELECT ST_LENGTH('$_TD->{new}{the_geom}') AS linelength");
        $_TD->{new}{$lengthColumnName} = $rv->{rows}[0]->{linelength};
      }
    }
  }
  return "MODIFY";
}
$BODY$
  LANGUAGE 'plperl' VOLATILE
  COST 100;
```