pgloader Documentation

Release 3.6.7

Dimitri Fontaine

Aug 13, 2022

Table Of Contents:

pgloader loads data from various sources into PostgreSQL. It can transform the data it reads on the fly and submit raw SQL before and after the loading. It uses the *COPY* PostgreSQL protocol to stream the data into the server, and manages errors by filling a pair of *reject.dat* and *reject.log* files.

Thanks to being able to load data directly from a database source, pgloader also supports from migrations from other productions to PostgreSQL. In this mode of operations, pgloader handles both the schema and data parts of the migration, in a single unmanned command, allowing to implement **Continuous Migration**.

CHAPTER 1

Features Overview

pgloader has two modes of operation: loading from files, migrating databases. In both cases, pgloader uses the PostgreSQL COPY protocol which implements a **streaming** to send data in a very efficient way.

1.1 Loading file content in PostgreSQL

When loading from files, pgloader implements the following features:

Many source formats supported Support for a wide variety of file based formats are included in pgloader: the CSV family, fixed columns formats, dBase files (`db3`), and IBM IXF files.

The SQLite database engine is accounted for in the next section: pgloader considers SQLite as a database source and implements schema discovery from SQLite catalogs.

On the fly data transformation Often enough the data as read from a CSV file (or another format) needs some tweaking and clean-up before being sent to PostgreSQL.

For instance in the geolite example we can see that integer values are being rewritten as IP address ranges, allowing to target an `ip4r` column directly.

Full Field projections pgloader supports loading data into less fields than found on file, or more, doing some computation on the data read before sending it to PostgreSQL.

Reading files from an archive Archive formats *zip*, *tar*, and *gzip* are supported by pgloader: the archive is extracted in a temporary directly and expanded files are then loaded.

HTTP(S) support pgloader knows how to download a source file or a source archive using HTTP directly. It might be better to use `curl -O- http://... | pgloader` and read the data from *standard input*, then allowing for streaming of the data from its source down to PostgreSQL.

Target schema discovery When loading in an existing table, pgloader takes into account the existing columns and may automatically guess the CSV format for you.

On error stop / On error resume next In some cases the source data is so damaged as to be impossible to migrate in full, and when loading from a file then the default for pgloader is to use `on error resume next` option, where the rows rejected by PostgreSQL are saved away and the migration continues with the other rows.

In other cases loading only a part of the input data might not be a great idea, and in such cases it's possible to use the `on error stop` option.

Pre/Post SQL commands This feature allows pgloader commands to include SQL commands to run before and after loading a file. It might be about creating a table first, then loading the data into it, and then doing more processing on-top of the data (implementing an *ELT* pipeline then), or creating specific indexes as soon as the data has been made ready.

1.2 One-command migration to PostgreSQL

When migrating a full database in a single command, pgloader implements the following features:

One-command migration The whole migration is started with a single command line and then runs unattended. pgloader is meant to be integrated in a fully automated tooling that you can repeat as many times as needed.

Schema discovery The source database is introspected using its SQL catalogs to get the list of tables, attributes (with data types, default values, not null constraints, etc), primary key constraints, foreign key constraints, indexes, comments, etc. This feeds an internal database catalog of all the objects to migrate from the source database to the target database.

User defined casting rules Some source database have ideas about their data types that might not be compatible with PostgreSQL implementaion of equivalent data types.

For instance, SQLite since version 3 has a Dynamic Type System which of course isn't compatible with the idea of a Relation. Or MySQL accepts datetime for year zero, which doesn't exists in our calendar, and doesn't have a boolean data type.

When migrating from another source database technology to PostgreSQL, data type casting choices must be made. pgloader implements solid defaults that you can rely upon, and a facility for **user defined data type casting rules** for specific cases. The idea is to allow users to specify the how the migration should be done, in order for it to be repeatable and included in a *Continuous Migration* process.

On the fly data transformations The user defined casting rules come with on the fly rewrite of the data. For instance zero dates (it's not just the year, MySQL accepts `0000-00-00` as a valid datetime) are rewritten to NULL values by default.

Partial Migrations It is possible to include only a partial list of the source database tables in the migration, or to exclude some of the tables on the source database.

Schema only, Data only This is the **ORM compatibility** feature of pgloader, where it is possible to create the schema using your ORM and then have pgloader migrate the data targeting this already created schema.

When doing this, it is possible for pgloader to *reindex* the target schema: before loading the data from the source database into PostgreSQL using COPY, pgloader DROPs the indexes and constraints, and reinstalls the exact same definitions of them once the data has been loaded.

The reason for operating that way is of course data load performance.

Repeatable (DROP+CREATE) By default, pgloader issues DROP statements in the target PostgreSQL database before issuing any CREATE statement, so that you can repeat the migration as many times as necessary until migration specifications and rules are bug free.

The schedule the data migration to run every night (or even more often!) for the whole duration of the code migration project. See the Continuous Migration methodology for more details about the approach.

On error stop / On error resume next The default behavior of pgloader when migrating from a database is `on error stop`. The idea is to let the user fix either the migration specifications or the source data, and run the process again, until it works.

In some cases the source data is so damaged as to be impossible to migrate in full, and it might be necessary to then resort to the `on error resume next` option, where the rows rejected by PostgreSQL are saved away and the migration continues with the other rows.

Pre/Post SQL commands, Post-Schema SQL commands While pgloader takes care of rewriting the schema to PostgreSQL expectations, and even provides *user-defined data type casting rules* support to that end, sometimes it is necessary to add some specific SQL commands around the migration. It's of course supported right from pgloader itself, without having to script around it.

Online ALTER schema At times migrating to PostgreSQL is also a good opportunity to review and fix bad decisions that were made in the past, or simply that are not relevant to PostgreSQL.

The pgloader command syntax allows to ALTER pgloader's internal representation of the target catalogs so that the target schema can be created a little different from the source one. Changes supported include target a different *schema* or *table* name.

Materialized Views, or schema rewrite on-the-fly In some cases the schema rewriting goes deeper than just renaming the SQL objects to being a full normalization exercise. Because PostgreSQL is great at running a normalized schema in production under most workloads.

pgloader implements full flexibility in on-the-fly schema rewriting, by making it possible to migrate from a view definition. The view attribute list becomes a table definition in PostgreSQL, and the data is fetched by querying the view on the source system.

A SQL view allows to implement both content filtering at the column level using the SELECT projection clause, and at the row level using the WHERE restriction clause. And backfilling from reference tables thanks to JOINs.

Distribute to Citus When migrating from PostgreSQL to Citus, a important part of the process consists of adjusting the schema to the distribution key. Read Preparing Tables and Ingesting Data in the Citus documentation for a complete example showing how to do that.

When using pgloader it's possible to specify the distribution keys and reference tables and let pgloader take care of adjusting the table, indexes, primary keys and foreign key definitions all by itself.

Encoding Overrides MySQL doesn't actually enforce the encoding of the data in the database to match the encoding known in the metadata, defined at the database, table, or attribute level. Sometimes, it's necessary to override the metadata in order to make sense of the text, and pgloader makes it easy to do so.

1.3 Continuous Migration

pgloader is meant to migrate a whole database in a single command line and without any manual intervention. The goal is to be able to setup a *Continuous Integration* environment as described in the Project Methodology document of the MySQL to PostgreSQL webpage.

1. Setup your target PostgreSQL Architecture
2. Fork a Continuous Integration environment that uses PostgreSQL
3. Migrate the data over and over again every night, from production
4. As soon as the CI is all green using PostgreSQL, schedule the D-Day
5. Migrate without suprise and enjoy!

In order to be able to follow this great methodology, you need tooling to implement the third step in a fully automated way. That's pgloader.

1.3.1 Introduction

pgloader loads data from various sources into PostgreSQL. It can transform the data it reads on the fly and submit raw SQL before and after the loading. It uses the *COPY* PostgreSQL protocol to stream the data into the server, and manages errors by filling a pair of *reject.dat* and *reject.log* files.

pgloader knows how to read data from different kind of sources:

- Files
  - CSV
  - Fixed Format
  - DBF
- Databases
  - SQLite
  - MySQL
  - MS SQL Server
  - PostgreSQL
  - Redshift

pgloader knows how to target different products using the PostgreSQL Protocol:

- PostgreSQL
- Citus
- Redshift

The level of automation provided by pgloader depends on the data source type. In the case of CSV and Fixed Format files, a full description of the expected input properties must be given to pgloader. In the case of a database, pgloader connects to the live service and knows how to fetch the metadata it needs directly from it.

Features Matrix

Here's a comparison of the features supported depending on the source database engine. Some features that are not supported can be added to pgloader, it's just that nobody had the need to do so yet. Those features are marked with . Empty cells are used when the feature doesn't make sense for the selected source database.

Feature	SQLite	MySQL	MS SQL	PostgreSQL	Redshift
One-command migration	✓	✓	✓	✓	✓
Continuous Migration	✓	✓	✓	✓	✓
Schema discovery	✓	✓	✓	✓	✓
Partial Migrations	✓	✓	✓	✓	✓
Schema only	✓	✓	✓	✓	✓
Data only	✓	✓	✓	✓	✓
Repeatable (DROP+CREATE)	✓	✓	✓	✓	✓
User defined casting rules	✓	✓	✓	✓	✓
Encoding Overrides		✓			
On error stop	✓	✓	✓	✓	✓
On error resume next	✓	✓	✓	✓	✓
Pre/Post SQL commands	✓	✓	✓	✓	✓
Post-Schema SQL commands		✓	✓	✓	✓
Primary key support	✓	✓	✓	✓	✓
Foreign key support	✓	✓	✓	✓	
Online ALTER schema	✓	✓	✓	✓	✓
Materialized views		✓	✓	✓	✓
Distribute to Citus		✓	✓	✓	✓

For more details about what the features are about, see the specific reference pages for your database source.

For some of the features, missing support only means that the feature is not needed for the other sources, such as the capability to override MySQL encoding metadata about a table or a column. Only MySQL in this list is left completely unable to guarantee text encoding. Or Redshift not having foreign keys.

Commands

pgloader implements its own *Command Language*, a DSL that allows to specify every aspect of the data load and migration to implement. Some of the features provided in the language are only available for a specific source type.

Command Line

The pgloader command line accepts those two variants:

```
pgloader [<options>] [<command-file>]...
pgloader [<options>] SOURCE TARGET
```

Either you have a *command-file* containing migration specifications in the pgloader *Command Language*, or you can give a *Source* for the data and a PostgreSQL database connection *Target* where to load the data into.

1.3.2 Pgloader Quick Start

In simple cases, pgloader is very easy to use.

CSV

Load data from a CSV file into a pre-existing table in your database:

```
pgloader --type csv                                  \
         --field id --field field                    \
         --with truncate                             \
         --with "fields terminated by ','"           \
         ./test/data/matching-1.csv                  \
         postgres:///pgloader?tablename=matching
```

In that example the whole loading is driven from the command line, bypassing the need for writing a command in the pgloader command syntax entirely. As there's no command though, the extra information needed must be provided on the command line using the *–type* and *–field* and *–with* switches.

For documentation about the available syntaxes for the *–field* and *–with* switches, please refer to the CSV section later in the man page.

Note also that the PostgreSQL URI includes the target *tablename*.

Reading from STDIN

File based pgloader sources can be loaded from the standard input, as in the following example:

```
pgloader --type csv                                    \
         --field "usps,geoid,aland,awater,aland_sqmi,awater_sqmi,intptlat,intptlong" \
         --with "skip header = 1"                      \
         --with "fields terminated by '\t'"            \
         -                                             \
         postgresql:///pgloader?districts_longlat      \
         < test/data/2013_Gaz_113CDs_national.txt
```

The dash (-) character as a source is used to mean *standard input*, as usual in Unix command lines. It's possible to stream compressed content to pgloader with this technique, using the Unix pipe:

```
gunzip -c source.gz | pgloader --type csv ... - pgsql:///target?foo
```

Loading from CSV available through HTTP

The same command as just above can also be run if the CSV file happens to be found on a remote HTTP location:

```
pgloader --type csv                                              \
         --field "usps,geoid,aland,awater,aland_sqmi,awater_sqmi,intptlat,intptlong" \
         --with "skip header = 1"                                \
         --with "fields terminated by '\t'"                      \
         http://pgsql.tapoueh.org/temp/2013_Gaz_113CDs_national.txt  \
         postgresql:///pgloader?districts_longlat
```

Some more options have to be used in that case, as the file contains a one-line header (most commonly that's column names, could be a copyright notice). Also, in that case, we specify all the fields right into a single *–field* option argument.

Again, the PostgreSQL target connection string must contain the *tablename* option and you have to ensure that the target table exists and may fit the data. Here's the SQL command used in that example in case you want to try it yourself:

```
create table districts_longlat
(
         usps        text,
```

(continues on next page)

(continued from previous page)

```
         geoid       text,
         aland       bigint,
         awater      bigint,
         aland_sqmi  double precision,
         awater_sqmi double precision,
         intptlat    double precision,
         intptlong   double precision
);
```

Also notice that the same command will work against an archived version of the same data.

Streaming CSV data from an HTTP compressed file

Finally, it's important to note that pgloader first fetches the content from the HTTP URL it to a local file, then expand the archive when it's recognized to be one, and only then processes the locally expanded file.

In some cases, either because pgloader has no direct support for your archive format or maybe because expanding the archive is not feasible in your environment, you might want to *stream* the content straight from its remote location into PostgreSQL. Here's how to do that, using the old battle tested Unix Pipes trick:

```
curl http://pgsql.tapoueh.org/temp/2013_Gaz_113CDs_national.txt.gz \
| gunzip -c                                                        \
| pgloader --type csv                                             \
           --field "usps,geoid,aland,awater,aland_sqmi,awater_sqmi,intptlat,intptlong"
           --with "skip header = 1"                               \
           --with "fields terminated by '\t'"                     \
           -                                                      \
           postgresql:///pgloader?districts_longlat
```

Now the OS will take care of the streaming and buffering between the network and the commands and pgloader will take care of streaming the data down to PostgreSQL.

Migrating from SQLite

The following command will open the SQLite database, discover its tables definitions including indexes and foreign keys, migrate those definitions while *casting* the data type specifications to their PostgreSQL equivalent and then migrate the data over:

```
createdb newdb
pgloader ./test/sqlite/sqlite.db postgresql:///newdb
```

Migrating from MySQL

Just create a database where to host the MySQL data and definitions and have pgloader do the migration for you in a single command line:

```
createdb pagila
pgloader mysql://user@localhost/sakila postgresql:///pagila
```

Fetching an archived DBF file from a HTTP remote location

It's possible for pgloader to download a file from HTTP, unarchive it, and only then open it to discover the schema then load the data:

```
createdb foo
pgloader --type dbf http://www.insee.fr/fr/methodes/nomenclatures/cog/telechargement/
↪2013/dbf/historiq2013.zip postgresql:///foo
```

Here it's not possible for pgloader to guess the kind of data source it's being given, so it's necessary to use the *–type* command line switch.

1.3.3 Pgloader Tutorial

Loading CSV Data with pgloader

CSV means *comma separated values* and is often found with quite varying specifications. pgloader allows you to describe those specs in its command.

The Command

To load data with pgloader you need to define in a *command* the operations in some details. Here's our example for loading CSV data:

```
LOAD CSV
     FROM 'path/to/file.csv' (x, y, a, b, c, d)
     INTO postgresql:///pgloader?csv (a, b, d, c)

     WITH truncate,
          skip header = 1,
          fields optionally enclosed by '"',
          fields escaped by double-quote,
          fields terminated by ','

      SET client_encoding to 'latin1',
          work_mem to '12MB',
          standard_conforming_strings to 'on'

   BEFORE LOAD DO
    $$ drop table if exists csv; $$,
    $$ create table csv (
        a bigint,
        b bigint,
        c char(2),
        d text
       );
  $$;
```

The Data

This command allows loading the following CSV file content:

```
Header, with a © sign
"2.6.190.56","2.6.190.63","33996344","33996351","GB","United Kingdom"
"3.0.0.0","4.17.135.31","50331648","68257567","US","United States"
"4.17.135.32","4.17.135.63","68257568","68257599","CA","Canada"
"4.17.135.64","4.17.142.255","68257600","68259583","US","United States"
"4.17.143.0","4.17.143.15","68259584","68259599","CA","Canada"
"4.17.143.16","4.18.32.71","68259600","68296775","US","United States"
```

Loading the data

Here's how to start loading the data. Note that the ouput here has been edited so as to facilitate its browsing online:

```
$ pgloader csv.load
... LOG Starting pgloader, log system is ready.
... LOG Parsing commands from file "/Users/dim/dev/pgloader/test/csv.load"

       table name       read   imported     errors            time
-----------------  ---------  ---------  ---------  --------------
      before load          2          2          0          0.039s
-----------------  ---------  ---------  ---------  --------------
               csv          6          6          0          0.019s
-----------------  ---------  ---------  ---------  --------------
Total import time          6          6          0          0.058s
```

The result

As you can see, the command described above is filtering the input and only importing some of the columns from the example data file. Here's what gets loaded in the PostgreSQL database:

```
pgloader# table csv;
    a     |    b     | c  |       d
----------+----------+----+----------------
 33996344 | 33996351 | GB | United Kingdom
 50331648 | 68257567 | US | United States
 68257568 | 68257599 | CA | Canada
 68257600 | 68259583 | US | United States
 68259584 | 68259599 | CA | Canada
 68259600 | 68296775 | US | United States
(6 rows)
```

Loading Fixed Width Data File with pgloader

Some data providers still use a format where each column is specified with a starting index position and a given length. Usually the columns are blank-padded when the data is shorter than the full reserved range.

The Command

To load data with pgloader you need to define in a *command* the operations in some details. Here's our example for loading Fixed Width Data, using a file provided by the US census.

You can find more files from them at the [Census 2000 Gazetteer Files](http://www.census.gov/geo/maps-data/data/gazetteer2000.html).

Here's our command:

```
LOAD ARCHIVE
   FROM http://www2.census.gov/geo/docs/maps-data/data/gazetteer/places2k.zip
   INTO postgresql:///pgloader

   BEFORE LOAD DO
     $$ drop table if exists places; $$,
     $$ create table places
       (
          usps      char(2)  not null,
          fips      char(2)  not null,
          fips_code char(5),
          loc_name  varchar(64)
       );
     $$

   LOAD FIXED
        FROM FILENAME MATCHING ~/places2k.txt/
             WITH ENCODING latin1
             (
                usps           from   0 for  2,
                fips           from   2 for  2,
                fips_code      from   4 for  5,
                "LocationName" from   9 for 64 [trim right whitespace],
                p              from  73 for  9,
                h              from  82 for  9,
                land           from  91 for 14,
                water          from 105 for 14,
                ldm            from 119 for 14,
                wtm            from 131 for 14,
                lat            from 143 for 10,
                long           from 153 for 11
             )
        INTO postgresql:///pgloader?places
             (
            usps, fips, fips_code, "LocationName"
             );
```

The Data

This command allows loading the following file content, where we are only showing the first couple of lines:

```
AL0100124Abbeville city                                                    2987
↪ 1353      40301945        120383   15.560669    0.046480 31.566367 -85.251300
AL0100460Adamsville city                                                   4965
↪ 2042      50779330         14126   19.606010    0.005454 33.590411 -86.949166
AL0100484Addison town                                                       723
↪  339       9101325             0    3.514041    0.000000 34.200042 -87.177851
AL0100676Akron town                                                         521
↪  239       1436797             0    0.554750    0.000000 32.876425 -87.740978
AL0100820Alabaster city                                                   22619
↪ 8594      53023800        141711   20.472605    0.054715 33.231162 -86.823829
AL0100988Albertville city                                                 17247
↪ 7090      67212867        258738   25.951034    0.099899 34.265362 -86.211261
AL0101132Alexander City city                                              15008
↪ 6855     100534344        433413   38.816529    0.167342 32.933157 -85.936008
```

Loading the data

Let's start the *pgloader* command with our *census-places.load* command file:

```
$ pgloader census-places.load
... LOG Starting pgloader, log system is ready.
... LOG Parsing commands from file "/Users/dim/dev/pgloader/test/census-places.load"
... LOG Fetching 'http://www2.census.gov/geo/docs/maps-data/data/gazetteer/places2k.
↪zip'
... LOG Extracting files from archive '//private/var/folders/w7/
↪9n8v8pw54t1gngfff0lj16040000gn/T/pgloader//places2k.zip'

       table name       read   imported     errors            time
-----------------  ---------  ---------  ---------  --------------
         download          0          0          0          1.494s
          extract          0          0          0          1.013s
      before load          2          2          0          0.013s
-----------------  ---------  ---------  ---------  --------------
           places      25375      25375          0          0.499s
-----------------  ---------  ---------  ---------  --------------
Total import time      25375      25375          0          3.019s
```

We can see that pgloader did download the file from its HTTP URL location then *unziped* it before the loading itself.

Note that the output of the command has been edited to facilitate its browsing online.

Loading MaxMind Geolite Data with pgloader

MaxMind provides a free dataset for geolocation, which is quite popular. Using pgloader you can download the lastest version of it, extract the CSV files from the archive and load their content into your database directly.

The Command

To load data with pgloader you need to define in a *command* the operations in some details. Here's our example for loading the Geolite data:

```
/*
 * Loading from a ZIP archive containing CSV files. The full test can be
 * done with using the archive found at
 * http://geolite.maxmind.com/download/geoip/database/GeoLiteCity_CSV/GeoLiteCity-
↪latest.zip
 *
 * And a very light version of this data set is found at
 * http://pgsql.tapoueh.org/temp/foo.zip for quick testing.
 */

LOAD ARCHIVE
   FROM http://geolite.maxmind.com/download/geoip/database/GeoLiteCity_CSV/
↪GeoLiteCity-latest.zip
   INTO postgresql:///ip4r

   BEFORE LOAD DO
     $$ create extension if not exists ip4r; $$,
     $$ create schema if not exists geolite; $$,
     $$ create table if not exists geolite.location
```

(continues on next page)

(continued from previous page)

```
      (
         locid      integer primary key,
         country    text,
         region     text,
         city       text,
         postalcode text,
         location   point,
         metrocode  text,
         areacode   text
      );
    $$,
    $$ create table if not exists geolite.blocks
      (
         iprange    ip4r,
         locid      integer
      );
    $$,
    $$ drop index if exists geolite.blocks_ip4r_idx; $$,
    $$ truncate table geolite.blocks, geolite.location cascade; $$

  LOAD CSV
       FROM FILENAME MATCHING ~/GeoLiteCity-Location.csv/
            WITH ENCODING iso-8859-1
            (
               locId,
               country,
               region     null if blanks,
               city       null if blanks,
               postalCode null if blanks,
               latitude,
               longitude,
               metroCode  null if blanks,
               areaCode   null if blanks
            )
       INTO postgresql:///ip4r?geolite.location
            (
               locid,country,region,city,postalCode,
               location point using (format nil "(~a,~a)" longitude latitude),
               metroCode,areaCode
            )
       WITH skip header = 2,
            fields optionally enclosed by '"',
            fields escaped by double-quote,
            fields terminated by ','

 AND LOAD CSV
       FROM FILENAME MATCHING ~/GeoLiteCity-Blocks.csv/
            WITH ENCODING iso-8859-1
            (
               startIpNum, endIpNum, locId
            )
       INTO postgresql:///ip4r?geolite.blocks
            (
               iprange ip4r using (ip-range startIpNum endIpNum),
               locId
            )
       WITH skip header = 2,
```

(continues on next page)

(continued from previous page)

```
            fields optionally enclosed by '"',
            fields escaped by double-quote,
            fields terminated by ','

  FINALLY DO
    $$ create index blocks_ip4r_idx on geolite.blocks using gist(iprange); $$;
```

Note that while the *Geolite* data is using a pair of integers (*start*, *end*) to represent *ipv4* data, we use the very poweful ip4r PostgreSQL Extension instead.

The transformation from a pair of integers into an IP is done dynamically by the pgloader process.

Also, the location is given as a pair of *float* columns for the *longitude* and the *latitude* where PostgreSQL offers the point datatype, so the pgloader command here will actually transform the data on the fly to use the appropriate data type and its input representation.

Loading the data

Here's how to start loading the data. Note that the ouput here has been edited so as to facilitate its browsing online:

```
$ pgloader archive.load
... LOG Starting pgloader, log system is ready.
... LOG Parsing commands from file "/Users/dim/dev/pgloader/test/archive.load"
... LOG Fetching 'http://geolite.maxmind.com/download/geoip/database/GeoLiteCity_CSV/
↪GeoLiteCity-latest.zip'
... LOG Extracting files from archive '//private/var/folders/w7/
↪9n8v8pw54t1gngfff0lj16040000gn/T/pgloader//GeoLiteCity-latest.zip'

       table name       read   imported     errors            time
-----------------  ---------  ---------  ---------  --------------
         download          0          0          0         11.592s
          extract          0          0          0          1.012s
      before load          6          6          0          0.019s
-----------------  ---------  ---------  ---------  --------------
 geolite.location     470387     470387          0          7.743s
   geolite.blocks    1903155    1903155          0         16.332s
-----------------  ---------  ---------  ---------  --------------
          finally          1          1          0         31.692s
-----------------  ---------  ---------  ---------  --------------
Total import time    2373542    2373542          0        1m8.390s
```

The timing of course includes the transformation of the *1.9 million* pairs of integer into a single *ipv4 range* each. The *finally* step consists of creating the *GiST* specialized index as given in the main command:

```
CREATE INDEX blocks_ip4r_idx ON geolite.blocks USING gist(iprange);
```

That index will then be used to speed up queries wanting to find which recorded geolocation contains a specific IP address:

```
ip4r> select *
        from      geolite.location l
             join geolite.blocks b using(locid)
       where iprange >>= '8.8.8.8';

-[ RECORD 1 ]------------------
```

(continues on next page)

(continued from previous page)

```
locid      | 223
country    | US
region     |
city       |
postalcode |
location   | (-97,38)
metrocode  |
areacode   |
iprange    | 8.8.8.8-8.8.37.255

Time: 0.747 ms
```

Loading dBase files with pgloader

The dBase format is still in use in some places as modern tools such as *Filemaker* and *Excel* offer some level of support for it. Speaking of support in modern tools, pgloader is right there on the list too!

The Command

To load data with pgloader you need to define in a *command* the operations in some details. Here's our example for loading a dBase file, using a file provided by the french administration.

You can find more files from them at the Insee website.

Here's our command:

```
LOAD DBF
    FROM http://www.insee.fr/fr/methodes/nomenclatures/cog/telechargement/2013/dbf/
↪historiq2013.zip
    INTO postgresql:///pgloader
    WITH truncate, create table
     SET client_encoding TO 'latin1';
```

Note that here pgloader will benefit from the meta-data information found in the dBase file to create a PostgreSQL table capable of hosting the data as described, then load the data.

Loading the data

Let's start the *pgloader* command with our *dbf-zip.load* command file:

```
$ pgloader dbf-zip.load
... LOG Starting pgloader, log system is ready.
... LOG Parsing commands from file "/Users/dim/dev/pgloader/test/dbf-zip.load"
... LOG Fetching 'http://www.insee.fr/fr/methodes/nomenclatures/cog/telechargement/
↪2013/dbf/historiq2013.zip'
... LOG Extracting files from archive '//private/var/folders/w7/
↪9n8v8pw54t1gngfff0lj16040000gn/T/pgloader//historiq2013.zip'

       table name       read   imported     errors            time
-----------------  ---------  ---------  ---------  --------------
         download          0          0          0          0.167s
          extract          0          0          0          1.010s
 create, truncate          0          0          0          0.071s
```

(continues on next page)

(continued from previous page)

```
-----------------  ---------  ---------  ---------  --------------
     historiq2013       9181       9181          0          0.658s
-----------------  ---------  ---------  ---------  --------------
Total import time       9181       9181          0          1.906s
```

We can see that pgloader did download the file from its HTTP URL location then *unziped* it before the loading itself.

Note that the output of the command has been edited to facilitate its browsing online.

Loading SQLite files with pgloader

The SQLite database is a respected solution to manage your data with. Its embeded nature makes it a source of migrations when a projects now needs to handle more concurrency, which PostgreSQL is very good at. pgloader can help you there.

In a Single Command Line

You can

```
$ createdb chinook
$ pgloader https://github.com/lerocha/chinook-database/raw/master/ChinookDatabase/
↪DataSources/Chinook_Sqlite_AutoIncrementPKs.sqlite pgsql:///chinook
```

Done! All with the schema, data, constraints, primary keys and foreign keys, etc. We also see an error with the Chinook schema that contains several primary key definitions against the same table, which is not accepted by PostgreSQL:

```
2017-06-20T16:18:59.019000+02:00 LOG Data errors in '/private/tmp/pgloader/'
2017-06-20T16:18:59.236000+02:00 LOG Fetching 'https://github.com/lerocha/chinook-
↪database/raw/master/ChinookDatabase/DataSources/Chinook_Sqlite_AutoIncrementPKs.
↪sqlite'
2017-06-20T16:19:00.664000+02:00 ERROR Database error 42P16: multiple primary keys␣
↪for table "playlisttrack" are not allowed
QUERY: ALTER TABLE playlisttrack ADD PRIMARY KEY USING INDEX idx_66873_sqlite_
↪autoindex_playlisttrack_1;
2017-06-20T16:19:00.665000+02:00 LOG report summary reset
             table name       read   imported     errors      total time
-----------------------  ---------  ---------  ---------  --------------
                  fetch          0          0          0          0.877s
        fetch meta data         33         33          0          0.033s
         Create Schemas          0          0          0          0.003s
       Create SQL Types          0          0          0          0.006s
          Create tables         22         22          0          0.043s
         Set Table OIDs         11         11          0          0.012s
-----------------------  ---------  ---------  ---------  --------------
                  album        347        347          0          0.023s
                 artist        275        275          0          0.023s
               customer         59         59          0          0.021s
               employee          8          8          0          0.018s
                invoice        412        412          0          0.031s
                  genre         25         25          0          0.021s
            invoiceline       2240       2240          0          0.034s
              mediatype          5          5          0          0.025s
          playlisttrack       8715       8715          0          0.040s
               playlist         18         18          0          0.016s
```

(continues on next page)

(continued from previous page)

```
                  track       3503       3503          0          0.111s
------------------------  ---------  ---------  ---------  --------------
COPY Threads Completion          33         33          0          0.313s
         Create Indexes          22         22          0          0.160s
 Index Build Completion          22         22          0          0.027s
        Reset Sequences           0          0          0          0.017s
           Primary Keys          12          0          1          0.013s
    Create Foreign Keys          11         11          0          0.040s
        Create Triggers           0          0          0          0.000s
       Install Comments           0          0          0          0.000s
------------------------  ---------  ---------  ---------  --------------
      Total import time       15607      15607          0          1.669s
```

You may need to have special cases to take care of tho. In advanced case you can use the pgloader command.

The Command

To load data with pgloader you need to define in a *command* the operations in some details. Here's our command:

```
load database
     from 'sqlite/Chinook_Sqlite_AutoIncrementPKs.sqlite'
     into postgresql:///pgloader

 with include drop, create tables, create indexes, reset sequences

  set work_mem to '16MB', maintenance_work_mem to '512 MB';
```

Note that here pgloader will benefit from the meta-data information found in the SQLite file to create a PostgreSQL database capable of hosting the data as described, then load the data.

Loading the data

Let's start the *pgloader* command with our *sqlite.load* command file:

```
$ pgloader sqlite.load
... LOG Starting pgloader, log system is ready.
... LOG Parsing commands from file "/Users/dim/dev/pgloader/test/sqlite.load"
... WARNING Postgres warning: table "album" does not exist, skipping
... WARNING Postgres warning: table "artist" does not exist, skipping
... WARNING Postgres warning: table "customer" does not exist, skipping
... WARNING Postgres warning: table "employee" does not exist, skipping
... WARNING Postgres warning: table "genre" does not exist, skipping
... WARNING Postgres warning: table "invoice" does not exist, skipping
... WARNING Postgres warning: table "invoiceline" does not exist, skipping
... WARNING Postgres warning: table "mediatype" does not exist, skipping
... WARNING Postgres warning: table "playlist" does not exist, skipping
... WARNING Postgres warning: table "playlisttrack" does not exist, skipping
... WARNING Postgres warning: table "track" does not exist, skipping
             table name       read   imported     errors            time
----------------------  ---------  ---------  ---------  --------------
      create, truncate          0          0          0          0.052s
                 Album        347        347          0          0.070s
                Artist        275        275          0          0.014s
              Customer         59         59          0          0.014s
```

(continues on next page)

(continued from previous page)

```
                Employee            8            8            0            0.012s
                   Genre           25           25            0            0.018s
                 Invoice          412          412            0            0.032s
             InvoiceLine         2240         2240            0            0.077s
               MediaType            5            5            0            0.012s
                Playlist           18           18            0            0.008s
           PlaylistTrack         8715         8715            0            0.071s
                   Track         3503         3503            0            0.105s
index build completion               0            0            0            0.000s
----------------------  ---------  ---------  ---------  --------------
          Create Indexes           20           20            0            0.279s
         reset sequences            0            0            0            0.043s
----------------------  ---------  ---------  ---------  --------------
    Total streaming time        15607        15607            0            0.476s
```

We can see that pgloader did download the file from its HTTP URL location then *unziped* it before loading it.

Also, the *WARNING* messages we see here are expected as the PostgreSQL database is empty when running the command, and pgloader is using the SQL commands *DROP TABLE IF EXISTS* when the given command uses the *include drop* option.

Note that the output of the command has been edited to facilitate its browsing online.

Migrating from MySQL to PostgreSQL

If you want to migrate your data over to PostgreSQL from MySQL then pgloader is the tool of choice!

Most tools around are skipping the main problem with migrating from MySQL, which is to do with the type casting and data sanitizing that needs to be done. pgloader will not leave you alone on those topics.

In a Single Command Line

As an example, we will use the f1db database from <http://ergast.com/mrd/> which which provides a historical record of motor racing data for non-commercial purposes. You can either use their API or download the whole database at http://ergast.com/downloads/f1db.sql.gz. Once you've done that load the database in MySQL:

```
$ mysql -u root
> create database f1db;
> source f1db.sql
```

Now let's migrate this database into PostgreSQL in a single command line:

```
$ createdb f1db
$ pgloader mysql://root@localhost/f1db pgsql:///f1db
```

Done! All with schema, table definitions, constraints, indexes, primary keys, *auto_increment* columns turned into *bigserial* , foreign keys, comments, and if you had some MySQL default values such as *ON UPDATE CURRENT_TIMESTAMP* they would have been translated to a PostgreSQL before update trigger automatically.

```
$ pgloader mysql://root@localhost/f1db pgsql:///f1db
2017-06-16T08:56:14.064000+02:00 LOG Main logs in '/private/tmp/pgloader/pgloader.log'
2017-06-16T08:56:14.068000+02:00 LOG Data errors in '/private/tmp/pgloader/'
2017-06-16T08:56:19.542000+02:00 LOG report summary reset
               table name       read   imported     errors      total time
```

(continues on next page)

(continued from previous page)

```
------------------------  ---------  ---------  ---------  --------------
           fetch meta data         33         33          0          0.365s
            Create Schemas          0          0          0          0.007s
          Create SQL Types          0          0          0          0.006s
             Create tables         26         26          0          0.068s
            Set Table OIDs         13         13          0          0.012s
------------------------  ---------  ---------  ---------  --------------
  f1db.constructorresults      11011      11011          0          0.205s
            f1db.circuits         73         73          0          0.150s
        f1db.constructors        208        208          0          0.059s
f1db.constructorstandings      11766      11766          0          0.365s
             f1db.drivers        841        841          0          0.268s
            f1db.laptimes     413578     413578          0          2.892s
     f1db.driverstandings      31420      31420          0          0.583s
            f1db.pitstops       5796       5796          0          2.154s
               f1db.races        976        976          0          0.227s
          f1db.qualifying       7257       7257          0          0.228s
             f1db.seasons         68         68          0          0.527s
             f1db.results      23514      23514          0          0.658s
              f1db.status        133        133          0          0.130s
------------------------  ---------  ---------  ---------  --------------
  COPY Threads Completion         39         39          0          4.303s
           Create Indexes         20         20          0          1.497s
   Index Build Completion         20         20          0          0.214s
          Reset Sequences          0         10          0          0.058s
             Primary Keys         13         13          0          0.012s
      Create Foreign Keys          0          0          0          0.000s
          Create Triggers          0          0          0          0.001s
         Install Comments          0          0          0          0.000s
------------------------  ---------  ---------  ---------  --------------
        Total import time     506641     506641          0          5.547s
```

You may need to have special cases to take care of tho, or views that you want to materialize while doing the migration. In advanced case you can use the pgloader command.

The Command

To load data with pgloader you need to define in a *command* the operations in some details. Here's our example for loading the MySQL Sakila Sample Database.

Here's our command:

```
load database
     from      mysql://root@localhost/sakila
     into postgresql:///sakila

 WITH include drop, create tables, no truncate,
      create indexes, reset sequences, foreign keys

  SET maintenance_work_mem to '128MB', work_mem to '12MB', search_path to 'sakila'

 CAST type datetime to timestamptz
                drop default drop not null using zero-dates-to-null,
      type date drop not null drop default using zero-dates-to-null
```

(continues on next page)

(continued from previous page)

```
 MATERIALIZE VIEWS film_list, staff_list

 -- INCLUDING ONLY TABLE NAMES MATCHING ~/film/, 'actor'
 -- EXCLUDING TABLE NAMES MATCHING ~<ory>

 BEFORE LOAD DO
 $$ create schema if not exists sakila; $$;
```

Note that here pgloader will benefit from the meta-data information found in the MySQL database to create a PostgreSQL database capable of hosting the data as described, then load the data.

In particular, some specific *casting rules* are given here, to cope with date values such as *0000-00-00* that MySQL allows and PostgreSQL rejects for not existing in our calendar. It's possible to add per-column casting rules too, which is useful is some of your *tinyint* are in fact *smallint* while some others are in fact *boolean* values.

Finaly note that we are using the *MATERIALIZE VIEWS* clause of pgloader: the selected views here will be migrated over to PostgreSQL *with their contents*.

It's possible to use the *MATERIALIZE VIEWS* clause and give both the name and the SQL (in MySQL dialect) definition of view, then pgloader creates the view before loading the data, then drops it again at the end.

Loading the data

Let's start the *pgloader* command with our *sakila.load* command file:

```
$ pgloader sakila.load
... LOG Starting pgloader, log system is ready.
... LOG Parsing commands from file "/Users/dim/dev/pgloader/test/sakila.load"
   <WARNING: table "xxx" does not exists have been edited away>

             table name       read   imported     errors            time
-----------------------  ---------  ---------  ---------  --------------
            before load          1          1          0          0.007s
        fetch meta data         45         45          0          0.402s
           create, drop          0         36          0          0.208s
-----------------------  ---------  ---------  ---------  --------------
                  actor        200        200          0          0.071s
                address        603        603          0          0.035s
               category         16         16          0          0.018s
                   city        600        600          0          0.037s
                country        109        109          0          0.023s
               customer        599        599          0          0.073s
                   film       1000       1000          0          0.135s
              film_actor       5462       5462          0          0.236s
          film_category       1000       1000          0          0.070s
              film_text       1000       1000          0          0.080s
              inventory       4581       4581          0          0.136s
               language          6          6          0          0.036s
                payment      16049      16049          0          0.539s
                 rental      16044      16044          0          0.648s
                  staff          2          2          0          0.041s
                  store          2          2          0          0.036s
              film_list        997        997          0          0.247s
             staff_list          2          2          0          0.135s
Index Build Completion          0          0          0          0.000s
-----------------------  ---------  ---------  ---------  --------------
```

(continues on next page)

(continued from previous page)

```
        Create Indexes         41         41          0         0.964s
       Reset Sequences          0          1          0         0.035s
          Foreign Keys         22         22          0         0.254s
----------------------  ---------  ---------  ---------  --------------
     Total import time      48272      48272          0         3.502s
```

The *WARNING* messages we see here are expected as the PostgreSQL database is empty when running the command, and pgloader is using the SQL commands *DROP TABLE IF EXISTS* when the given command uses the *include drop* option.

Note that the output of the command has been edited to facilitate its browsing online.

1.3.4 Installing pgloader

Several distributions are available for pgcopydb.

debian packages

You can install pgloader directly from apt.postgresql.org and from official debian repositories, see packages.debian.org/pgloader.

```
$ apt-get install pgloader
```

RPM packages

The Postgres community repository for RPM packages is yum.postgresql.org and does include binary packages for pgloader.

Docker Images

Docker images are maintained for each tagged release at dockerhub, and also built from the CI/CD integration on GitHub at each commit to the *main* branch.

The DockerHub dimitri/pgloader repository is where the tagged releases are made available. The image uses the Postgres version currently in debian stable.

To use the `dimitri/pgloader` docker image:

```
$ docker run --rm -it dimitri/pgloader:latest pgloader --version
```

Or you can use the CI/CD integration that publishes packages from the main branch to the GitHub docker repository:

```
$ docker pull ghcr.io/dimitri/pgloader:latest
$ docker run --rm -it ghcr.io/dimitri/pgloader:latest pgloader --version
$ docker run --rm -it ghcr.io/dimitri/pgloader:latest pgloader --help
```

Build from sources

pgloader is a Common Lisp program, tested using the SBCL (>= 1.2.5) and Clozure CL implementations and with Quicklisp to fetch build dependencies.

When building from sources, you should always build from the current git HEAD as it's basically the only source that is managed in a way to ensure it builds aginst current set of dependencies versions.

The build system for pgloader uses a Makefile and the Quicklisp Common Lisp packages distribution system.

The modern build system for pgloader is entirely written in Common Lisp, where the historical name for our operation is *save-lisp-and-die* and can be used that way:

```
$ make save
```

The legacy build system also uses Buildapp and can be used that way:

```
$ make pgloader
```

Building from sources on debian

Install the build dependencies first, then use the Makefile:

```
$ apt-get install sbcl unzip libsqlite3-dev make curl gawk freetds-dev libzip-dev
$ cd /path/to/pgloader

$ make save
$ ./build/bin/pgloader --help
```

Building from sources on RedHat/CentOS

To build and install pgloader the Steel Bank Common Lisp package (sbcl) from EPEL, and the freetds packages are required.

It is recommended to build the RPM yourself, see below, to ensure that all installed files are properly tracked and that you can safely update to newer versions of pgloader as they're released.

To do an adhoc build and install run `boostrap-centos.sh` for CentOS 6 or `bootstrap-centos7.sh` for CentOS 7 to install the required dependencies.

Building a pgloader RPM from sources

The spec file in the root of the pgloader repository can be used to build your own RPM. For production deployments it is recommended that you build this RPM on a dedicated build box and then copy the RPM to your production environment for use; it is considered bad practice to have compilers and build tools present in production environments.

1. Install the [EPEL repo](https://fedoraproject.org/wiki/EPEL#Quickstart).
2. Install rpmbuild dependencies:

   ```
   sudo yum -y install yum-utils rpmdevtools @"Development Tools"
   ```

3. Install pgloader build dependencies:

   ```
   sudo yum-builddep pgloader.spec
   ```

4. Download pgloader source:

   ```
   spectool -g -R pgloader.spec
   ```

5. Build the source and binary RPMs (see *rpmbuild –help* for other build options):

```
rpmbuild -ba pgloader.spec
```

Building from sources on macOS

We suppose you already have `git` and `make` available, if that's not the case now is the time to install those tools. The SQLite lib that comes in MacOSX is fine, no need for extra software here.

You will need to install either SBCL or CCL separately, and when using [brew](http://brew.sh/) it's as simple as:

```
$ brew install sbcl
$ brew install clozure-cl
```

NOTE: Make sure you installed the universal binaries of Freetds, so that they can be loaded correctly.

```
$ brew install freetds --universal --build-from-source
```

Then use the normal build system for pgloader:

```
$ make save
$ ./build/bin/pgloader --version
```

Building from sources on Windows

Building pgloader on Windows is supported (in theory), thanks to Common Lisp implementations being available on that platform, and to the Common Lisp Standard for making it easy to write actually portable code.

It is recommended to have a look at the issues labelled with Windows support if you run into trouble when building pgloader, because the development team is lacking windows user and in practice we can't maintain the support for that Operating System:

If you need `pgloader.exe` on windows please condider contributing fixes for that environment and maybe longer term support then. Specifically, a CI integration with a windows build host would allow ensuring that we continue to support that target.

Building Docker image from sources

You can build a Docker image from source using SBCL by default:

```
$ docker build .
```

Or Clozure CL (CCL):

```
$ docker build -f Dockerfile.ccl .
```

More options when building from source

The `Makefile` target `save` knows how to produce a Self Contained Binary file for pgloader, found at `./build/bin/pgloader`:

```
$ make save
```

By default, the `Makefile` uses SBCL to compile your binary image, though it's possible to build using Clozure-CL.

```
$ make CL=ccl64 save
```

It is possible to to tweak the default amount of memory that the pgloader image will allow itself using when running through your data (don't ask for more than your current RAM tho). At the moment only the legacy build system includes support for this custom build:

```
$ make DYNSIZE=8192 pgloader
```

The `make pgloader` command when successful outputs a *./build/bin/pgloader* file for you to use.

1.3.5 PgLoader Reference Manual

pgloader loads data from various sources into PostgreSQL. It can transform the data it reads on the fly and submit raw SQL before and after the loading. It uses the *COPY* PostgreSQL protocol to stream the data into the server, and manages errors by filling a pair of *reject.dat* and *reject.log* files.

pgloader operates either using commands which are read from files:

```
pgloader commands.load
```

or by using arguments and options all provided on the command line:

```
pgloader SOURCE TARGET
```

Arguments

The pgloader arguments can be as many load files as needed, or a couple of connection strings to a specific input file.

Source Connection String

The source connection string format is as follows:

```
format:///absolute/path/to/file.ext
format://./relative/path/to/file.ext
```

Where format might be one of *csv*, *fixed*, *copy*, *dbf*, *db3* or *ixf*.:

```
db://user:pass@host:port/dbname
```

Where db might be of *sqlite*, *mysql* or *mssql*.

When using a file based source format, pgloader also support natively fetching the file from an http location and decompressing an archive if needed. In that case it's necessary to use the *–type* option to specify the expected format of the file. See the examples below.

Also note that some file formats require describing some implementation details such as columns to be read and delimiters and quoting when loading from csv.

For more complex loading scenarios, you will need to write a full fledge load command in the syntax described later in this document.

Target Connection String

The target connection string format is described in details later in this document, see Section Connection String.

Options

Inquiry Options

Use these options when you want to know more about how to use *pgloader*, as those options will cause *pgloader* not to load any data.

- *-h*, *–help*

  Show command usage summary and exit.

- *-V*, *–version*

  Show pgloader version string and exit.

- *-E*, *–list-encodings*

  List known encodings in this version of pgloader.

- *-U*, *–upgrade-config*

  Parse given files in the command line as *pgloader.conf* files with the *INI* syntax that was in use in pgloader versions 2.x, and output the new command syntax for pgloader on standard output.

General Options

Those options are meant to tweak *pgloader* behavior when loading data.

- *-v*, *–verbose*

  Be verbose.

- *-q*, *–quiet*

  Be quiet.

- *-d*, *–debug*

  Show debug level information messages.

- *-D*, *–root-dir*

  Set the root working directory (default to “/tmp/pgloader”).

- *-L*, *–logfile*

  Set the pgloader log file (default to “/tmp/pgloader/pgloader.log”).

- *–log-min-messages*

  Minimum level of verbosity needed for log message to make it to the logfile. One of critical, log, error, warning, notice, info or debug.

- *–client-min-messages*

  Minimum level of verbosity needed for log message to make it to the console. One of critical, log, error, warning, notice, info or debug.

- *-S*, *–summary*

  A filename where to copy the summary output. When relative, the filename is expanded into **root-dir**.

  The format of the filename defaults to being *human readable*. It is possible to have the output in machine friendly formats such as *CSV*, *COPY* (PostgreSQL's own COPY format) or *JSON* by specifying a filename with the extension resp. *.csv*, *.copy* or *.json*.

- *-l <file>*, *–load-lisp-file <file>*

  Specify a lisp <file> to compile and load into the pgloader image before reading the commands, allowing to define extra transformation function. Those functions should be defined in the *pgloader.transforms* package. This option can appear more than once in the command line.

- *–dry-run*

  Allow testing a *.load* file without actually trying to load any data. It's useful to debug it until it's ok, in particular to fix connection strings.

- *–on-error-stop*

  Alter pgloader behavior: rather than trying to be smart about error handling and continue loading good data, separating away the bad one, just stop as soon as PostgreSQL refuses anything sent to it. Useful to debug data processing, transformation function and specific type casting.

- *–self-upgrade <directory>*

  Specify a <directory> where to find pgloader sources so that one of the very first things it does is dynamically loading-in (and compiling to machine code) another version of itself, usually a newer one like a very recent git checkout.

- *–no-ssl-cert-verification*

  Uses the OpenSSL option to accept a locally issued server-side certificate, avoiding the following error message:

  ```
  SSL verify error: 20 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY
  ```

  The right way to fix the SSL issue is to use a trusted certificate, of course. Sometimes though it's useful to make progress with the pgloader setup while the certificate chain of trust is being fixed, maybe by another team. That's when this option is useful.

Command Line Only Operations

Those options are meant to be used when using *pgloader* from the command line only, rather than using a command file and the rich command clauses and parser. In simple cases, it can be much easier to use the *SOURCE* and *TARGET* directly on the command line, then tweak the loading with those options:

- *–with "option"*

  Allows setting options from the command line. You can use that option as many times as you want. The option arguments must follow the *WITH* clause for the source type of the *SOURCE* specification, as described later in this document.

- *–set "guc_name='value'"*

  Allows setting PostgreSQL configuration from the command line. Note that the option parsing is the same as when used from the *SET* command clause, in particular you must enclose the guc value with single-quotes.

- *–field "…"*

Allows setting a source field definition. Fields are accumulated in the order given on the command line. It's possible to either use a *–field* option per field in the source file, or to separate field definitions by a comma, as you would do in the *HAVING FIELDS* clause.

- *–cast "..."*

  Allows setting a specific casting rule for loading the data.

- *–type csv|fixed|db3|ixf|sqlite|mysql|mssql*

  Allows forcing the source type, in case when the *SOURCE* parsing isn't satisfying.

- *–encoding <encoding>*

  Set the encoding of the source file to load data from.

- *–before <filename>*

  Parse given filename for SQL queries and run them against the target database before loading the data from the source. The queries are parsed by pgloader itself: they need to be terminated by a semi-colon (;) and the file may include *i* or *ir* commands to *include* another file.

- *–after <filename>*

  Parse given filename for SQL queries and run them against the target database after having loaded the data from the source. The queries are parsed in the same way as with the *–before* option, see above.

More Debug Information

To get the maximum amount of debug information, you can use both the *–verbose* and the *–debug* switches at the same time, which is equivalent to saying *–client-min-messages data*. Then the log messages will show the data being processed, in the cases where the code has explicit support for it.

Batches And Retry Behaviour

To load data to PostgreSQL, pgloader uses the *COPY* streaming protocol. While this is the faster way to load data, *COPY* has an important drawback: as soon as PostgreSQL emits an error with any bit of data sent to it, whatever the problem is, the whole data set is rejected by PostgreSQL.

To work around that, pgloader cuts the data into *batches* of 25000 rows each, so that when a problem occurs it's only impacting that many rows of data. Each batch is kept in memory while the *COPY* streaming happens, in order to be able to handle errors should some happen.

When PostgreSQL rejects the whole batch, pgloader logs the error message then isolates the bad row(s) from the accepted ones by retrying the batched rows in smaller batches. To do that, pgloader parses the *CONTEXT* error message from the failed COPY, as the message contains the line number where the error was found in the batch, as in the following example:

```
CONTEXT: COPY errors, line 3, column b: "2006-13-11"
```

Using that information, pgloader will reload all rows in the batch before the erroneous one, log the erroneous one as rejected, then try loading the remaining of the batch in a single attempt, which may or may not contain other erroneous data.

At the end of a load containing rejected rows, you will find two files in the *root-dir* location, under a directory named the same as the target database of your setup. The filenames are the target table, and their extensions are *.dat* for the rejected data and *.log* for the file containing the full PostgreSQL client side logs about the rejected data.

The *.dat* file is formatted in PostgreSQL the text COPY format as documented in *http://www.postgresql.org/docs/9.2/static/sql-copy.html#AEN66609*.

It is possible to use the following WITH options to control pgloader batch behavior:

- *on error stop*, *on error resume next*

  This option controls if pgloader is using building batches of data at all. The batch implementation allows pgloader to recover errors by sending the data that PostgreSQL accepts again, and by keeping away the data that PostgreSQL rejects.

  To enable retrying the data and loading the good parts, use the option *on error resume next*, which is the default to file based data loads (such as CSV, IXF or DBF).

  When migrating from another RDMBS technology, it's best to have a reproducible loading process. In that case it's possible to use *on error stop* and fix either the casting rules, the data transformation functions or in cases the input data until your migration runs through completion. That's why *on error resume next* is the default for SQLite, MySQL and MS SQL source kinds.

A Note About Performance

pgloader has been developed with performance in mind, to be able to cope with ever growing needs in loading large amounts of data into PostgreSQL.

The basic architecture it uses is the old Unix pipe model, where a thread is responsible for loading the data (reading a CSV file, querying MySQL, etc) and fills pre-processed data into a queue. Another threads feeds from the queue, apply some more *transformations* to the input data and stream the end result to PostgreSQL using the COPY protocol.

When given a file that the PostgreSQL *COPY* command knows how to parse, and if the file contains no erroneous data, then pgloader will never be as fast as just using the PostgreSQL *COPY* command.

Note that while the *COPY* command is restricted to read either from its standard input or from a local file on the server's file system, the command line tool *psql* implements a *copy* command that knows how to stream a file local to the client over the network and into the PostgreSQL server, using the same protocol as pgloader uses.

A Note About Parallelism

pgloader uses several concurrent tasks to process the data being loaded:

- a reader task reads the data in and pushes it to a queue,
- at last one write task feeds from the queue and formats the raw into the PostgreSQL COPY format in batches (so that it's possible to then retry a failed batch without reading the data from source again), and then sends the data to PostgreSQL using the COPY protocol.

The parameter *workers* allows to control how many worker threads are allowed to be active at any time (that's the parallelism level); and the parameter *concurrency* allows to control how many tasks are started to handle the data (they may not all run at the same time, depending on the *workers* setting).

We allow *workers* simultaneous workers to be active at the same time in the context of a single table. A single unit of work consist of several kinds of workers:

- a reader getting raw data from the source,
- N writers preparing and sending the data down to PostgreSQL.

The N here is setup to the *concurrency* parameter: with a *CONCURRENCY* of 2, we start (+ 1 2) = 3 concurrent tasks, with a *concurrency* of 4 we start (+ 1 4) = 5 concurrent tasks, of which only *workers* may be active simultaneously.

The defaults are *workers = 4, concurrency = 1* when loading from a database source, and *workers = 8, concurrency = 2* when loading from something else (currently, a file). Those defaults are arbitrary and waiting for feedback from users, so please consider providing feedback if you play with the settings.

As the *CREATE INDEX* threads started by pgloader are only waiting until PostgreSQL is done with the real work, those threads are *NOT* counted into the concurrency levels as detailed here.

By default, as many *CREATE INDEX* threads as the maximum number of indexes per table are found in your source schema. It is possible to set the *max parallel create index WITH* option to another number in case there's just too many of them to create.

Source Formats

pgloader supports the following input formats:

- csv, which includes also tsv and other common variants where you can change the *separator* and the *quoting* rules and how to *escape* the *quotes* themselves;
- fixed columns file, where pgloader is flexible enough to accomodate with source files missing columns (*ragged fixed length column files* do exist);
- PostgreSLQ COPY formatted files, following the COPY TEXT documentation of PostgreSQL, such as the reject files prepared by pgloader;
- dbase files known as db3 or dbf file;
- ixf formated files, ixf being a binary storage format from IBM;
- sqlite databases with fully automated discovery of the schema and advanced cast rules;
- mysql databases with fully automated discovery of the schema and advanced cast rules;
- MS SQL databases with fully automated discovery of the schema and advanced cast rules.

Pgloader Commands Syntax

pgloader implements a Domain Specific Language allowing to setup complex data loading scripts handling computed columns and on-the-fly sanitization of the input data. For more complex data loading scenarios, you will be required to learn that DSL's syntax. It's meant to look familiar to DBA by being inspired by SQL where it makes sense, which is not that much after all.

The pgloader commands follow the same global grammar rules. Each of them might support only a subset of the general options and provide specific options.

```
LOAD <source-type>
         FROM <source-url>
       [ HAVING FIELDS <source-level-options> ]
            INTO <postgresql-url>
       [ TARGET TABLE [ "<schema>" ]."<table name>" ]
       [ TARGET COLUMNS <columns-and-options> ]

    [ WITH <load-options> ]

    [ SET <postgresql-settings> ]

[ BEFORE LOAD [ DO <sql statements> | EXECUTE <sql file> ] ... ]
[  AFTER LOAD [ DO <sql statements> | EXECUTE <sql file> ] ... ]
    ;
```

The main clauses are the *LOAD*, *FROM*, *INTO* and *WITH* clauses that each command implements. Some command then implement the *SET* command, or some specific clauses such as the *CAST* clause.

Templating with Mustache

pgloader implements the https://mustache.github.io/ templating system so that you may have dynamic parts of your commands. See the documentation for this template system online.

A specific feature of pgloader is the ability to fetch a variable from the OS environment of the pgloader process, making it possible to run pgloader as in the following example:

```
$ DBPATH=sqlite/sqlite.db pgloader ./test/sqlite-env.load
```

or in several steps:

```
$ export DBPATH=sqlite/sqlite.db
$ pgloader ./test/sqlite-env.load
```

The variable can then be used in a typical mustache fashion:

```
load database
     from '{{DBPATH}}'
     into postgresql:///pgloader;
```

It's also possible to prepare a INI file such as the following:

```
[pgloader]

DBPATH = sqlite/sqlite.db
```

And run the following command, feeding the INI values as a *context* for pgloader templating system:

```
$ pgloader --context ./test/sqlite.ini ./test/sqlite-ini.load
```

The mustache templates implementation with OS environment support replaces former *GETENV* implementation, which didn't work anyway.

Common Clauses

Some clauses are common to all commands:

FROM

The *FROM* clause specifies where to read the data from, and each command introduces its own variant of sources. For instance, the *CSV* source supports *inline*, *stdin*, a filename, a quoted filename, and a *FILENAME MATCHING* clause (see above); whereas the *MySQL* source only supports a MySQL database URI specification.

INTO

The PostgreSQL connection URI must contains the name of the target table where to load the data into. That table must have already been created in PostgreSQL, and the name might be schema qualified.

Then *INTO* option also supports an optional comma separated list of target columns, which are either the name of an input *field* or the white space separated list of the target column name, its PostgreSQL data type and a *USING* expression.

The *USING* expression can be any valid Common Lisp form and will be read with the current package set to *pgloader.transforms*, so that you can use functions defined in that package, such as functions loaded dynamically with the *–load* command line parameter.

Each *USING* expression is compiled at runtime to native code.

This feature allows pgloader to load any number of fields in a CSV file into a possibly different number of columns in the database, using custom code for that projection.

WITH

Set of options to apply to the command, using a global syntax of either:

- *key = value*
- *use option*
- *do not use option*

See each specific command for details.

All data sources specific commands support the following options:

- *on error stop*, *on error resume next*
- *batch rows = R*
- *batch size = ... MB*
- *prefetch rows = ...*

See the section BATCH BEHAVIOUR OPTIONS for more details.

In addition, the following settings are available:

- *workers = W*
- *concurrency = C*
- *max parallel create index = I*

See section A NOTE ABOUT PARALLELISM for more details.

SET

This clause allows to specify session parameters to be set for all the sessions opened by pgloader. It expects a list of parameter name, the equal sign, then the single-quoted value as a comma separated list.

The names and values of the parameters are not validated by pgloader, they are given as-is to PostgreSQL.

BEFORE LOAD DO

You can run SQL queries against the database before loading the data from the *CSV* file. Most common SQL queries are *CREATE TABLE IF NOT EXISTS* so that the data can be loaded.

Each command must be *dollar-quoted*: it must begin and end with a double dollar sign, *$$*. Dollar-quoted queries are then comma separated. No extra punctuation is expected after the last SQL query.

BEFORE LOAD EXECUTE

Same behaviour as in the *BEFORE LOAD DO* clause. Allows you to read the SQL queries from a SQL file. Implements support for PostgreSQL dollar-quoting and the *i* and *ir* include facilities as in *psql* batch mode (where they are the same thing).

AFTER LOAD DO

Same format as *BEFORE LOAD DO*, the dollar-quoted queries found in that section are executed once the load is done. That's the right time to create indexes and constraints, or re-enable triggers.

AFTER LOAD EXECUTE

Same behaviour as in the *AFTER LOAD DO* clause. Allows you to read the SQL queries from a SQL file. Implements support for PostgreSQL dollar-quoting and the *i* and *ir* include facilities as in *psql* batch mode (where they are the same thing).

AFTER CREATE SCHEMA DO

Same format as *BEFORE LOAD DO*, the dollar-quoted queries found in that section are executed once the schema has been created by pgloader, and before the data is loaded. It's the right time to ALTER TABLE or do some custom implementation on-top of what pgloader does, like maybe partitioning.

AFTER CREATE SCHEMA EXECUTE

Same behaviour as in the *AFTER CREATE SCHEMA DO* clause. Allows you to read the SQL queries from a SQL file. Implements support for PostgreSQL dollar-quoting and the *i* and *ir* include facilities as in *psql* batch mode (where they are the same thing).

Connection String

The <*postgresql-url*> parameter is expected to be given as a *Connection URI* as documented in the PostgreSQL documentation at http://www.postgresql.org/docs/9.3/static/libpq-connect.html#LIBPQ-CONNSTRING.

```
postgresql://[user[:password]@][netloc][:port][/dbname][?option=value&...]
```

Where:

- *user*

  Can contain any character, including colon (*:*) which must then be doubled (*::*) and at-sign (@) which must then be doubled (@@).

  When omitted, the *user* name defaults to the value of the *PGUSER* environment variable, and if it is unset, the value of the *USER* environment variable.

- *password*

  Can contain any character, including the at sign (@) which must then be doubled (@@). To leave the password empty, when the *user* name ends with at at sign, you then have to use the syntax user:@.

When omitted, the *password* defaults to the value of the *PGPASSWORD* environment variable if it is set, otherwise the password is left unset.

When no *password* is found either in the connection URI nor in the environment, then pgloader looks for a *.pgpass* file as documented at https://www.postgresql.org/docs/current/static/libpq-pgpass.html. The implementation is not that of *libpq* though. As with *libpq* you can set the environment variable *PGPASSFILE* to point to a *.pgpass* file, and pgloader defaults to *~/.pgpass* on unix like systems and *%APPDATA%postgresqlpgpass.conf* on windows. Matching rules and syntax are the same as with *libpq*, refer to its documentation.

- *netloc*

  Can be either a hostname in dotted notation, or an ipv4, or an Unix domain socket path. Empty is the default network location, under a system providing *unix domain socket* that method is preferred, otherwise the *netloc* default to *localhost*.

  It's possible to force the *unix domain socket* path by using the syntax *unix:/path/to/where/the/socket/file/is*, so to force a non default socket path and a non default port, you would have:

  postgresql://unix:/tmp:54321/dbname

  The *netloc* defaults to the value of the *PGHOST* environment variable, and if it is unset, to either the default *unix* socket path when running on a Unix system, and *localhost* otherwise.

  Socket path containing colons are supported by doubling the colons within the path, as in the following example:

  postgresql://unix:/tmp/project::region::instance:5432/dbname

- *dbname*

  Should be a proper identifier (letter followed by a mix of letters, digits and the punctuation signs comma (,), dash (-) and underscore (_).

  When omitted, the *dbname* defaults to the value of the environment variable *PGDATABASE*, and if that is unset, to the *user* value as determined above.

- *options*

  The optional parameters must be supplied with the form *name=value*, and you may use several parameters by separating them away using an ampersand (&) character.

  Only some options are supported here, *tablename* (which might be qualified with a schema name) *sslmode*, *host*, *port*, *dbname*, *user* and *password*.

  The *sslmode* parameter values can be one of *disable*, *allow*, *prefer* or *require*.

  For backward compatibility reasons, it's possible to specify the *tablename* option directly, without spelling out the *tablename=* parts.

  The options override the main URI components when both are given, and using the percent-encoded option parameters allow using passwords starting with a colon and bypassing other URI components parsing limitations.

Regular Expressions

Several clauses listed in the following accept *regular expressions* with the following input rules:

- A regular expression begins with a tilde sign (~),
- is then followed with an opening sign,
- then any character is allowed and considered part of the regular expression, except for the closing sign,
- then a closing sign is expected.

The opening and closing sign are allowed by pair, here's the complete list of allowed delimiters:

```
~//
~[]
~{}
~()
~<>
~""
~''
~||
~##
```

Pick the set of delimiters that don't collide with the *regular expression* you're trying to input. If your expression is such that none of the solutions allow you to enter it, the places where such expressions are allowed should allow for a list of expressions.

Comments

Any command may contain comments, following those input rules:

- the – delimiter begins a comment that ends with the end of the current line,
- the delimiters /* and */ respectively start and end a comment, which can be found in the middle of a command or span several lines.

Any place where you could enter a *whitespace* will accept a comment too.

Batch behaviour options

All pgloader commands have support for a *WITH* clause that allows for specifying options. Some options are generic and accepted by all commands, such as the *batch behaviour options*, and some options are specific to a data source kind, such as the CSV *skip header* option.

The global batch behaviour options are:

- *batch rows*

  Takes a numeric value as argument, used as the maximum number of rows allowed in a batch. The default is *25 000* and can be changed to try having better performance characteristics or to control pgloader memory usage;

- *batch size*

  Takes a memory unit as argument, such as *20 MB*, its default value. Accepted multipliers are *kB*, *MB*, *GB*, *TB* and *PB*. The case is important so as not to be confused about bits versus bytes, we're only talking bytes here.

- *prefetch rows*

  Takes a numeric value as argument, defaults to *100000*. That's the number of rows that pgloader is allowed to read in memory in each reader thread. See the *workers* setting for how many reader threads are allowed to run at the same time.

Other options are specific to each input source, please refer to specific parts of the documentation for their listing and covering.

A batch is then closed as soon as either the *batch rows* or the *batch size* threshold is crossed, whichever comes first. In cases when a batch has to be closed because of the *batch size* setting, a *debug* level log message is printed with how many rows did fit in the *oversized* batch.

1.3.6 Loading CSV data

This command instructs pgloader to load data from a *CSV* file. Because of the complexity of guessing the parameters of a CSV file, it's simpler to instruct pgloader with how to parse the data in there, using the full pgloader command syntax and CSV specifications as in the following example.

Using advanced options and a load command file

The command then would be:

```
$ pgloader csv.load
```

And the contents of the `csv.load` file could be inspired from the following:

```
LOAD CSV
   FROM 'GeoLiteCity-Blocks.csv' WITH ENCODING iso-646-us
        HAVING FIELDS
        (
           startIpNum, endIpNum, locId
        )
   INTO postgresql://user@localhost:54393/dbname
        TARGET TABLE geolite.blocks
        TARGET COLUMNS
        (
           iprange ip4r using (ip-range startIpNum endIpNum),
           locId
        )
   WITH truncate,
        skip header = 2,
        fields optionally enclosed by '"',
        fields escaped by backslash-quote,
        fields terminated by '\t'

    SET work_mem to '32 MB', maintenance_work_mem to '64 MB';
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

CSV Source Specification: FROM

Filename where to load the data from. Accepts an *ENCODING* option. Use the *–list-encodings* option to know which encoding names are supported.

The filename may be enclosed by single quotes, and could be one of the following special values:

- *inline*

  The data is found after the end of the parsed commands. Any number of empty lines between the end of the commands and the beginning of the data is accepted.

- *stdin*

  Reads the data from the standard input stream.

- *FILENAME MATCHING*

  The whole *matching* clause must follow the following rule:

```
[ ALL FILENAMES | [ FIRST ] FILENAME ]
MATCHING regexp
[ IN DIRECTORY '...' ]
```

  The *matching* clause applies given *regular expression* (see above for exact syntax, several options can be used here) to filenames. It's then possible to load data from only the first match of all of them.

  The optional *IN DIRECTORY* clause allows specifying which directory to walk for finding the data files, and can be either relative to where the command file is read from, or absolute. The given directory must exists.

Fields Specifications

The *FROM* option also supports an optional comma separated list of *field* names describing what is expected in the *CSV* data file, optionally introduced by the clause *HAVING FIELDS*.

Each field name can be either only one name or a name following with specific reader options for that field, enclosed in square brackets and comma-separated. Supported per-field reader options are:

- *terminated by*

  See the description of *field terminated by* below.

  The processing of this option is not currently implemented.

- *date format*

  When the field is expected of the date type, then this option allows to specify the date format used in the file.

  Date format string are template strings modeled against the PostgreSQL *to_char* template strings support, limited to the following patterns:

  - YYYY, YYY, YY for the year part
  - MM for the numeric month part
  - DD for the numeric day part
  - HH, HH12, HH24 for the hour part
  - am, AM, a.m., A.M.
  - pm, PM, p.m., P.M.
  - MI for the minutes part
  - SS for the seconds part
  - MS for the milliseconds part (4 digits)
  - US for the microseconds part (6 digits)
  - unparsed punctuation signs: - . * # @ T / and space

  Here's an example of a *date format* specification:

```
column-name [date format 'YYYY-MM-DD HH24-MI-SS.US']
```

- *null if*

This option takes an argument which is either the keyword *blanks* or a double-quoted string.

When *blanks* is used and the field value that is read contains only space characters, then it's automatically converted to an SQL *NULL* value.

When a double-quoted string is used and that string is read as the field value, then the field value is automatically converted to an SQL *NULL* value.

- *trim both whitespace*, *trim left whitespace*, *trim right whitespace*

  This option allows to trim whitespaces in the read data, either from both sides of the data, or only the whitespace characters found on the left of the streaing, or only those on the right of the string.

CSV Loading Options: WITH

When loading from a *CSV* file, the following options are supported:

- *truncate*

  When this option is listed, pgloader issues a *TRUNCATE* command against the PostgreSQL target table before reading the data file.

- *drop indexes*

  When this option is listed, pgloader issues *DROP INDEX* commands against all the indexes defined on the target table before copying the data, then *CREATE INDEX* commands once the *COPY* is done.

  In order to get the best performance possible, all the indexes are created in parallel and when done the primary keys are built again from the unique indexes just created. This two step process allows creating the primary key index in parallel with the other indexes, as only the *ALTER TABLE* command needs an *access exclusive lock* on the target table.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

  This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *skip header*

  Takes a numeric value as argument. Instruct pgloader to skip that many lines at the beginning of the input file.

- *csv header*

  Use the first line read after *skip header* as the list of csv field names to be found in the CSV file, using the same CSV parameters as for the CSV data.

- *trim unquoted blanks*

  When reading unquoted values in the *CSV* file, remove the blanks found in between the separator and the value. That behaviour is the default.

- *keep unquoted blanks*

  When reading unquoted values in the *CSV* file, keep blanks found in between the separator and the value.

- *fields optionally enclosed by*

  Takes a single character as argument, which must be found inside single quotes, and might be given as the printable character itself, the special value t to denote a tabulation character, the special value ‘ to denote a single-quote, or *0x* then an hexadecimal value read as the ASCII code for the character.

The following options specify the same enclosing character, a single quote:

```
fields optionally enclosed by '\''
fields optionally enclosed by '0x27'
```

This character is used as the quoting character in the *CSV* file, and defaults to double-quote.

- *fields not enclosed*

  By default, pgloader will use the double-quote character as the enclosing character. If you have a CSV file where fields are not enclosed and are using double-quote as an expected ordinary character, then use the option *fields not enclosed* for the CSV parser to accept those values.

- *fields escaped by*

  Takes either the special value *backslash-quote* or *double-quote*, or any value supported by the *fields terminated by* option (see below). This value is used to recognize escaped field separators when they are to be found within the data fields themselves. Defaults to *double-quote*.

- *csv escape mode*

  Takes either the special value *quote* (the default) or *following* and allows the CSV parser to parse either only escaped field separator or any character (including CSV data) when using the *following* value.

- *fields terminated by*

  Takes a single character as argument, which must be found inside single quotes, and might be given as the printable character itself, the special value t to denote a tabulation character, or *0x* then an hexadecimal value read as the ASCII code for the character.

  This character is used as the *field separator* when reading the *CSV* data.

- *lines terminated by*

  Takes a single character as argument, which must be found inside single quotes, and might be given as the printable character itself, the special value t to denote a tabulation character, or *0x* then an hexadecimal value read as the ASCII code for the character.

  This character is used to recognize *end-of-line* condition when reading the *CSV* data.

1.3.7 Loading Fixed Cols File Formats

This command instructs pgloader to load data from a text file containing columns arranged in a *fixed size* manner.

Using advanced options and a load command file

The command then would be:

```
$ pgloader fixed.load
```

And the contents of the `fixed.load` file could be inspired from the following:

```
LOAD FIXED
     FROM inline
          (
           a from  0 for 10,
           b from 10 for  8,
           c from 18 for  8,
           d from 26 for 17 [null if blanks, trim right whitespace]
```

(continues on next page)

(continued from previous page)

```
          )
     INTO postgresql:///pgloader
   TARGET TABLE fixed
          (
             a, b,
             c time using (time-with-no-separator c),
             d
          )

     WITH truncate

      SET work_mem to '14MB',
          standard_conforming_strings to 'on'

BEFORE LOAD DO
     $$ drop table if exists fixed; $$,
     $$ create table fixed (
         a integer,
         b date,
         c time,
         d text
        );
     $$;

 01234567892008052011431250firstline
    01234562008052115182300left blank-padded
 12345678902008052208231560another line
  2345609872014092914371500
  2345678902014092914371520
```

Note that the example comes from the test suite of pgloader, where we use the advanced feature `FROM inline` that allows embedding the source data within the command file. In most cases a more classic FROM clause loading the data from a separate file would be used.

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

Fixed File Format Source Specification: FROM

Filename where to load the data from. Accepts an *ENCODING* option. Use the *–list-encodings* option to know which encoding names are supported.

The filename may be enclosed by single quotes, and could be one of the following special values:

- *inline*

  The data is found after the end of the parsed commands. Any number of empty lines between the end of the commands and the beginning of the data is accepted.

- *stdin*

  Reads the data from the standard input stream.

- *FILENAMES MATCHING*

  The whole *matching* clause must follow the following rule:

```
[ ALL FILENAMES | [ FIRST ] FILENAME ]
MATCHING regexp
[ IN DIRECTORY '...' ]
```

The *matching* clause applies given *regular expression* (see above for exact syntax, several options can be used here) to filenames. It's then possible to load data from only the first match of all of them.

The optional *IN DIRECTORY* clause allows specifying which directory to walk for finding the data files, and can be either relative to where the command file is read from, or absolute. The given directory must exists.

Fields Specifications

The *FROM* option also supports an optional comma separated list of *field* names describing what is expected in the *FIXED* data file.

Each field name is composed of the field name followed with specific reader options for that field. Supported per-field reader options are the following, where only *start* and *length* are required.

- *start*

  Position in the line where to start reading that field's value. Can be entered with decimal digits or *0x* then hexadecimal digits.

- *length*

  How many bytes to read from the *start* position to read that field's value. Same format as *start*.

Those optional parameters must be enclosed in square brackets and comma-separated:

- *terminated by*

  See the description of *field terminated by* below.

  The processing of this option is not currently implemented.

- *date format*

  When the field is expected of the date type, then this option allows to specify the date format used in the file.

  Date format string are template strings modeled against the PostgreSQL *to_char* template strings support, limited to the following patterns:

  - YYYY, YYY, YY for the year part
  - MM for the numeric month part
  - DD for the numeric day part
  - HH, HH12, HH24 for the hour part
  - am, AM, a.m., A.M.
  - pm, PM, p.m., P.M.
  - MI for the minutes part
  - SS for the seconds part
  - MS for the milliseconds part (4 digits)
  - US for the microseconds part (6 digits)
  - unparsed punctuation signs: - . * # @ T / and space

  Here's an example of a *date format* specification:

```
column-name [date format 'YYYY-MM-DD HH24-MI-SS.US']
```

- *null if*

  This option takes an argument which is either the keyword *blanks* or a double-quoted string.

  When *blanks* is used and the field value that is read contains only space characters, then it's automatically converted to an SQL *NULL* value.

  When a double-quoted string is used and that string is read as the field value, then the field value is automatically converted to an SQL *NULL* value.

- *trim both whitespace*, *trim left whitespace*, *trim right whitespace*

  This option allows to trim whitespaces in the read data, either from both sides of the data, or only the whitespace characters found on the left of the streaing, or only those on the right of the string.

Fixed File Format Loading Options: WITH

When loading from a *FIXED* file, the following options are supported:

- *truncate*

  When this option is listed, pgloader issues a *TRUNCATE* command against the PostgreSQL target table before reading the data file.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

  This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *skip header*

  Takes a numeric value as argument. Instruct pgloader to skip that many lines at the beginning of the input file.

1.3.8 Loading COPY Formatted Files

This commands instructs pgloader to load from a file containing COPY TEXT data as described in the PostgreSQL documentation.

Using advanced options and a load command file

The command then would be:

```
$ pgloader copy.load
```

And the contents of the `copy.load` file could be inspired from the following:

```
LOAD COPY
     FROM copy://./data/track.copy
          (
            trackid, track, album, media, genre, composer,
            milliseconds, bytes, unitprice
```

(continues on next page)

(continued from previous page)

```
          )
     INTO postgresql:///pgloader
   TARGET TABLE track_full

     WITH truncate

      SET work_mem to '14MB',
          standard_conforming_strings to 'on'

BEFORE LOAD DO
     $$ drop table if exists track_full; $$,
     $$ create table track_full (
          trackid      bigserial,
          track        text,
          album        text,
          media        text,
          genre        text,
          composer     text,
          milliseconds bigint,
          bytes        bigint,
          unitprice    numeric
        );
     $$;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

COPY Formatted Files Source Specification: FROM

Filename where to load the data from. This support local files, HTTP URLs and zip files containing a single dbf file of the same name. Fetch such a zip file from an HTTP address is of course supported.

- *inline*

  The data is found after the end of the parsed commands. Any number of empty lines between the end of the commands and the beginning of the data is accepted.

- *stdin*

  Reads the data from the standard input stream.

- *FILENAMES MATCHING*

  The whole *matching* clause must follow the following rule:

  ```
  [ ALL FILENAMES | [ FIRST ] FILENAME ]
  MATCHING regexp
  [ IN DIRECTORY '...' ]
  ```

  The *matching* clause applies given *regular expression* (see above for exact syntax, several options can be used here) to filenames. It's then possible to load data from only the first match of all of them.

  The optional *IN DIRECTORY* clause allows specifying which directory to walk for finding the data files, and can be either relative to where the command file is read from, or absolute. The given directory must exists.

COPY Formatted File Options: WITH

When loading from a *COPY* file, the following options are supported:

- *delimiter*

  Takes a single character as argument, which must be found inside single quotes, and might be given as the printable character itself, the special value t to denote a tabulation character, or *0x* then an hexadecimal value read as the ASCII code for the character.

  This character is used as the *delimiter* when reading the data, in a similar way to the PostgreSQL *COPY* option.

- *null*

  Takes a quoted string as an argument (quotes can be either double quotes or single quotes) and uses that string as the *NULL* representation in the data.

  This is similar to the *null COPY* option in PostgreSQL.

- *truncate*

  When this option is listed, pgloader issues a *TRUNCATE* command against the PostgreSQL target table before reading the data file.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

  This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *skip header*

  Takes a numeric value as argument. Instruct pgloader to skip that many lines at the beginning of the input file.

1.3.9 Loading DBF data

This command instructs pgloader to load data from a *DBF* file. A default set of casting rules are provided and might be overloaded and appended to by the command.

Using advanced options and a load command file

Here's an example with a remote HTTP source and some user defined casting rules. The command then would be:

```
$ pgloader dbf.load
```

And the contents of the `dbf.load` file could be inspired from the following:

```
LOAD DBF
    FROM http://www.insee.fr/fr/methodes/nomenclatures/cog/telechargement/2013/dbf/
↪reg2013.dbf
    INTO postgresql://user@localhost/dbname
    WITH truncate, create table
    CAST column reg2013.region to integer,
         column reg2013.tncc to smallint;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

DBF Source Specification: FROM

Filename where to load the data from. This support local files, HTTP URLs and zip files containing a single dbf file of the same name. Fetch such a zip file from an HTTP address is of course supported.

DBF Loading Options: WITH

When loading from a *DBF* file, the following options are supported:

- *truncate*

  When this option is listed, pgloader issues a *TRUNCATE* command against the PostgreSQL target table before reading the data file.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

  This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *create table*

  When this option is listed, pgloader creates the table using the meta data found in the *DBF* file, which must contain a list of fields with their data type. A standard data type conversion from DBF to PostgreSQL is done.

- *table name*

  This options expects as its value the possibly qualified name of the table to create.

Default DB3 Casting Rules

When migrating from DB3 the following Casting Rules are provided:

```
type C to text using db3-trim-string
type M to text using db3-trim-string
type N to numeric using db3-numeric-to-pgsql-integer
type I to numeric using db3-numeric-to-pgsql-numeric
type L to boolean using logical-to-boolean
type D to date using db3-date-to-pgsql-date
```

1.3.10 Loading IXF Data

This command instructs pgloader to load data from an IBM *IXF* file.

Using advanced options and a load command file

The command then would be:

```
$ pgloader ixf.load
```

And the contents of the `ixf.load` file could be inspired from the following:

```
LOAD IXF
    FROM data/nsitra.test1.ixf
    INTO postgresql:///pgloader
  TARGET TABLE nsitra.test1
    WITH truncate, create table, timezone UTC

  BEFORE LOAD DO
   $$ create schema if not exists nsitra; $$,
   $$ drop table if exists nsitra.test1; $$;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

IXF Source Specification: FROM

Filename where to load the data from. This support local files, HTTP URLs and zip files containing a single ixf file of the same name. Fetch such a zip file from an HTTP address is of course supported.

IXF Loading Options: WITH

When loading from a *IXF* file, the following options are supported:

- *truncate*

  When this option is listed, pgloader issues a *TRUNCATE* command against the PostgreSQL target table before reading the data file.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

  This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *create table*

  When this option is listed, pgloader creates the table using the meta data found in the *DBF* file, which must contain a list of fields with their data type. A standard data type conversion from DBF to PostgreSQL is done.

- *table name*

  This options expects as its value the possibly qualified name of the table to create.

- *timezone*

This options allows to specify which timezone is used when parsing timestamps from an IXF file, and defaults to *UTC*. Expected values are either *UTC*, *GMT* or a single quoted location name such as *'Universal'* or *'Europe/Paris'*.

1.3.11 Loading From an Archive

This command instructs pgloader to load data from one or more files contained in an archive. Currently the only supported archive format is *ZIP*, and the archive might be downloaded from an *HTTP* URL.

Using advanced options and a load command file

The command then would be:

```
$ pgloader archive.load
```

And the contents of the `archive.load` file could be inspired from the following:

```
LOAD ARCHIVE
   FROM /Users/dim/Downloads/GeoLiteCity-latest.zip
   INTO postgresql:///ip4r

   BEFORE LOAD
     DO $$ create extension if not exists ip4r; $$,
        $$ create schema if not exists geolite; $$,

     EXECUTE 'geolite.sql'

   LOAD CSV
        FROM FILENAME MATCHING ~/GeoLiteCity-Location.csv/
             WITH ENCODING iso-8859-1
             (
                locId,
                country,
                region     null if blanks,
                city       null if blanks,
                postalCode null if blanks,
                latitude,
                longitude,
                metroCode  null if blanks,
                areaCode   null if blanks
             )
        INTO postgresql:///ip4r?geolite.location
             (
                locid,country,region,city,postalCode,
                location point using (format nil "(~a,~a)" longitude latitude),
                metroCode,areaCode
             )
        WITH skip header = 2,
             fields optionally enclosed by '"',
             fields escaped by double-quote,
             fields terminated by ','

  AND LOAD CSV
        FROM FILENAME MATCHING ~/GeoLiteCity-Blocks.csv/
             WITH ENCODING iso-8859-1
```

(continues on next page)

(continued from previous page)

```
            (
               startIpNum, endIpNum, locId
            )
       INTO postgresql:///ip4r?geolite.blocks
            (
               iprange ip4r using (ip-range startIpNum endIpNum),
               locId
            )
       WITH skip header = 2,
            fields optionally enclosed by '"',
            fields escaped by double-quote,
            fields terminated by ','

  FINALLY DO
    $$ create index blocks_ip4r_idx on geolite.blocks using gist(iprange); $$;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

Archive Source Specification: FROM

Filename or HTTP URI where to load the data from. When given an HTTP URL the linked file will get downloaded locally before processing.

If the file is a *zip* file, the command line utility *unzip* is used to expand the archive into files in *$TMPDIR*, or */tmp* if *$TMPDIR* is unset or set to a non-existing directory.

Then the following commands are used from the top level directory where the archive has been expanded.

Archive Sub Commands

- command [*AND* command ...]

  A series of commands against the contents of the archive, at the moment only *CSV*,''FIXED‘ and *DBF* commands are supported.

  Note that commands are supporting the clause *FROM FILENAME MATCHING* which allows the pgloader command not to depend on the exact names of the archive directories.

  The same clause can also be applied to several files with using the spelling *FROM ALL FILENAMES MATCHING* and a regular expression.

  The whole *matching* clause must follow the following rule:

```
FROM [ ALL FILENAMES | [ FIRST ] FILENAME ] MATCHING
```

Archive Final SQL Commands

- *FINALLY DO*

  SQL Queries to run once the data is loaded, such as *CREATE INDEX*.

1.3.12 Migrating a MySQL Database to PostgreSQL

This command instructs pgloader to load data from a database connection. pgloader supports dynamically converting the schema of the source database and the indexes building.

A default set of casting rules are provided and might be overloaded and appended to by the command.

Using default settings

Here is the simplest command line example, which might be all you need:

```
$ pgloader mysql://myuser@myhost/dbname pgsql://pguser@pghost/dbname
```

Using advanced options and a load command file

It might be that you want more flexibility than that and want to set advanced options. Then the next example is using as many options as possible, some of them even being defaults. Chances are you don't need that complex a setup, don't copy and paste it, use it only as a reference!

The command then would be:

```
$ pgloader my.load
```

And the contents of the command file `my.load` could be inspired from the following:

```
LOAD DATABASE
     FROM      mysql://root@localhost/sakila
     INTO postgresql://localhost:54393/sakila

 WITH include drop, create tables, create indexes, reset sequences,
      workers = 8, concurrency = 1,
      multiple readers per thread, rows per range = 50000

  SET PostgreSQL PARAMETERS
      maintenance_work_mem to '128MB',
      work_mem to '12MB',
      search_path to 'sakila, public, "$user"'

  SET MySQL PARAMETERS
      net_read_timeout  = '120',
      net_write_timeout = '120'

 CAST type bigint when (= precision 20) to bigserial drop typemod,
      type date drop not null drop default using zero-dates-to-null,
      -- type tinyint to boolean using tinyint-to-boolean,
      type year to integer

 MATERIALIZE VIEWS film_list, staff_list

 -- INCLUDING ONLY TABLE NAMES MATCHING ~/film/, 'actor'
 -- EXCLUDING TABLE NAMES MATCHING ~<ory>
 -- DECODING TABLE NAMES MATCHING ~/messed/, ~/encoding/ AS utf8
 -- ALTER TABLE NAMES MATCHING 'film' RENAME TO 'films'
 -- ALTER TABLE NAMES MATCHING ~/_list$/ SET SCHEMA 'mv'

 ALTER TABLE NAMES MATCHING ~/_list$/, 'sales_by_store', ~/sales_by/
```

(continues on next page)

(continued from previous page)

```
 SET SCHEMA 'mv'

ALTER TABLE NAMES MATCHING 'film' RENAME TO 'films'
ALTER TABLE NAMES MATCHING ~/./ SET (fillfactor='40')

ALTER SCHEMA 'sakila' RENAME TO 'pagila'

BEFORE LOAD DO
  $$ create schema if not exists pagila; $$,
  $$ create schema if not exists mv;     $$,
  $$ alter database sakila set search_path to pagila, mv, public; $$;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

MySQL Database Source Specification: FROM

Must be a connection URL pointing to a MySQL database.

If the connection URI contains a table name, then only this table is migrated from MySQL to PostgreSQL.

See the *SOURCE CONNECTION STRING* section above for details on how to write the connection string. The MySQL connection string accepts the same parameter *sslmode* as the PostgreSQL connection string, but the *verify* mode is not implemented (yet).

```
mysql://[user[:password]@][netloc][:port][/dbname][?option=value&...]
```

MySQL connection strings support specific options:

- `useSSL`

  The same notation rules as found in the *Connection String* parts of the documentation apply, and we have a specific MySQL option: `useSSL`. The value for `useSSL` can be either `false` or `true`.

  If both `sslmode` and `useSSL` are used in the same connection string, pgloader behavior is undefined.

The MySQL connection string also accepts the *useSSL* parameter with values being either *false* or *true*.

Environment variables described in <http://dev.mysql.com/doc/refman/5.0/en/environment-variables.html> can be used as default values too. If the user is not provided, then it defaults to *USER* environment variable value. The password can be provided with the environment variable *MYSQL_PWD*. The host can be provided with the environment variable *MYSQL_HOST* and otherwise defaults to *localhost*. The port can be provided with the environment variable *MYSQL_TCP_PORT* and otherwise defaults to *3306*.

MySQL Database Migration Options: WITH

When loading from a *MySQL* database, the following options are supported, and the default *WITH* clause is: *no truncate*, *create tables*, *include drop*, *create indexes*, *reset sequences*, *foreign keys*, *downcase identifiers*, *uniquify index names*.

- *include drop*

  When this option is listed, pgloader drops all the tables in the target PostgreSQL database whose names appear in the MySQL database. This option allows for using the same command several times in a row until you figure out all the options, starting automatically from a clean environment. Please note that *CASCADE* is used to

ensure that tables are dropped even if there are foreign keys pointing to them. This is precisely what *include drop* is intended to do: drop all target tables and recreate them.

Great care needs to be taken when using *include drop*, as it will cascade to *all* objects referencing the target tables, possibly including other tables that are not being loaded from the source DB.

- *include no drop*

  When this option is listed, pgloader will not include any *DROP* statement when loading the data.

- *truncate*

  When this option is listed, pgloader issue the *TRUNCATE* command against each PostgreSQL table just before loading data into it.

- *no truncate*

  When this option is listed, pgloader issues no *TRUNCATE* command.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

  This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *create tables*

  When this option is listed, pgloader creates the table using the meta data found in the *MySQL* file, which must contain a list of fields with their data type. A standard data type conversion from DBF to PostgreSQL is done.

- *create no tables*

  When this option is listed, pgloader skips the creation of table before loading data, target tables must then already exist.

  Also, when using *create no tables* pgloader fetches the metadata from the current target database and checks type casting, then will remove constraints and indexes prior to loading the data and install them back again once the loading is done.

- *create indexes*

  When this option is listed, pgloader gets the definitions of all the indexes found in the MySQL database and create the same set of index definitions against the PostgreSQL database.

- *create no indexes*

  When this option is listed, pgloader skips the creating indexes.

- *drop indexes*

  When this option is listed, pgloader drops the indexes in the target database before loading the data, and creates them again at the end of the data copy.

- *uniquify index names*, *preserve index names*

  MySQL index names are unique per-table whereas in PostgreSQL index names have to be unique per-schema. The default for pgloader is to change the index name by prefixing it with *idx_OID* where *OID* is the internal numeric identifier of the table the index is built against.

  In somes cases like when the DDL are entirely left to a framework it might be sensible for pgloader to refrain from handling index unique names, that is achieved by using the *preserve index names* option.

  The default is to *uniquify index names*.

Even when using the option *preserve index names*, MySQL primary key indexes named "PRIMARY" will get their names uniquified. Failing to do so would prevent the primary keys to be created again in PostgreSQL where the index names must be unique per schema.

- *drop schema*

  When this option is listed, pgloader drops the target schema in the target PostgreSQL database before creating it again and all the objects it contains. The default behavior doesn't drop the target schemas.

- *foreign keys*

  When this option is listed, pgloader gets the definitions of all the foreign keys found in the MySQL database and create the same set of foreign key definitions against the PostgreSQL database.

- *no foreign keys*

  When this option is listed, pgloader skips creating foreign keys.

- *reset sequences*

  When this option is listed, at the end of the data loading and after the indexes have all been created, pgloader resets all the PostgreSQL sequences created to the current maximum value of the column they are attached to.

  The options *schema only* and *data only* have no effects on this option.

- *reset no sequences*

  When this option is listed, pgloader skips resetting sequences after the load.

  The options *schema only* and *data only* have no effects on this option.

- *downcase identifiers*

  When this option is listed, pgloader converts all MySQL identifiers (table names, index names, column names) to *downcase*, except for PostgreSQL *reserved* keywords.

  The PostgreSQL *reserved* keywords are determined dynamically by using the system function *pg_get_keywords()*.

- *quote identifiers*

  When this option is listed, pgloader quotes all MySQL identifiers so that their case is respected. Note that you will then have to do the same thing in your application code queries.

- *schema only*

  When this option is listed pgloader refrains from migrating the data over. Note that the schema in this context includes the indexes when the option *create indexes* has been listed.

- *data only*

  When this option is listed pgloader only issues the *COPY* statements, without doing any other processing.

- *single reader per thread*, *multiple readers per thread*

  The default is *single reader per thread* and it means that each MySQL table is read by a single thread as a whole, with a single *SELECT* statement using no *WHERE* clause.

  When using *multiple readers per thread* pgloader may be able to divide the reading work into several threads, as many as the *concurrency* setting, which needs to be greater than 1 for this option to kick be activated.

  For each source table, pgloader searches for a primary key over a single numeric column, or a multiple-column primary key index for which the first column is of a numeric data type (one of *integer* or *bigint*). When such an index exists, pgloader runs a query to find the *min* and *max* values on this column, and then split that range into many ranges containing a maximum of *rows per range*.

When the range list we then obtain contains at least as many ranges than our concurrency setting, then we distribute those ranges to each reader thread.

So when all the conditions are met, pgloader then starts as many reader thread as the *concurrency* setting, and each reader thread issues several queries with a *WHERE id >= x AND id < y*, where *y - x = rows per range* or less (for the last range, depending on the max value just obtained.

- *rows per range*

  How many rows are fetched per *SELECT* query when using *multiple readers per thread*, see above for details.

- *SET MySQL PARAMETERS*

  The *SET MySQL PARAMETERS* allows setting MySQL parameters using the MySQL *SET* command each time pgloader connects to it.

MySQL Database Casting Rules

The command *CAST* introduces user-defined casting rules.

The cast clause allows to specify custom casting rules, either to overload the default casting rules or to amend them with special cases.

A casting rule is expected to follow one of the forms:

```
type <mysql-type-name> [ <guard> ... ] to <pgsql-type-name> [ <option> ... ]
column <table-name>.<column-name> [ <guards> ] to ...
```

It's possible for a *casting rule* to either match against a MySQL data type or against a given *column name* in a given *table name*. That flexibility allows to cope with cases where the type *tinyint* might have been used as a *boolean* in some cases but as a *smallint* in others.

The *casting rules* are applied in order, the first match prevents following rules to be applied, and user defined rules are evaluated first.

The supported guards are:

- *when unsigned*

  The casting rule is only applied against MySQL columns of the source type that have the keyword *unsigned* in their data type definition.

  Example of a casting rule using a *unsigned* guard:

  ```
  type smallint when unsigned to integer drop typemod
  ```

- *when default 'value'*

  The casting rule is only applied against MySQL columns of the source type that have given *value*, which must be a single-quoted or a double-quoted string.

- *when typemod expression*

  The casting rule is only applied against MySQL columns of the source type that have a *typemod* value matching the given *typemod expression*. The *typemod* is separated into its *precision* and *scale* components.

  Example of a cast rule using a *typemod* guard:

  ```
  type char when (= precision 1) to char keep typemod
  ```

  This expression casts MySQL *char(1)* column to a PostgreSQL column of type *char(1)* while allowing for the general case *char(N)* will be converted by the default cast rule into a PostgreSQL type *varchar(N)*.

- *with extra auto_increment*

  The casting rule is only applied against MySQL columns having the *extra* column *auto_increment* option set, so that it's possible to target e.g. *serial* rather than *integer*.

  The default matching behavior, when this option isn't set, is to match both columns with the extra definition and without.

  This means that if you want to implement a casting rule that target either *serial* or *integer* from a *smallint* definition depending on the *auto_increment* extra bit of information from MySQL, then you need to spell out two casting rules as following:

```
type smallint  with extra auto_increment
  to serial drop typemod keep default keep not null,

type smallint
  to integer drop typemod keep default keep not null
```

The supported casting options are:

- *drop default*, *keep default*

  When the option *drop default* is listed, pgloader drops any existing default expression in the MySQL database for columns of the source type from the *CREATE TABLE* statement it generates.

  The spelling *keep default* explicitly prevents that behaviour and can be used to overload the default casting rules.

- *drop not null*, *keep not null*, *set not null*

  When the option *drop not null* is listed, pgloader drops any existing *NOT NULL* constraint associated with the given source MySQL datatype when it creates the tables in the PostgreSQL database.

  The spelling *keep not null* explicitly prevents that behaviour and can be used to overload the default casting rules.

  When the option *set not null* is listed, pgloader sets a *NOT NULL* constraint on the target column regardless whether it has been set in the source MySQL column.

- *drop typemod*, *keep typemod*

  When the option *drop typemod* is listed, pgloader drops any existing *typemod* definition (e.g. *precision* and *scale*) from the datatype definition found in the MySQL columns of the source type when it created the tables in the PostgreSQL database.

  The spelling *keep typemod* explicitly prevents that behaviour and can be used to overload the default casting rules.

- *using*

  This option takes as its single argument the name of a function to be found in the *pgloader.transforms* Common Lisp package. See above for details.

  It's possible to augment a default cast rule (such as one that applies against *ENUM* data type for example) with a *transformation function* by omitting entirely the *type* parts of the casting rule, as in the following example:

```
column enumerate.foo using empty-string-to-null
```

MySQL Views Support

MySQL views support allows pgloader to migrate view as if they were base tables. This feature then allows for on-the-fly transformation from MySQL to PostgreSQL, as the view definition is used rather than the base data.

MATERIALIZE VIEWS

This clause allows you to implement custom data processing at the data source by providing a *view definition* against which pgloader will query the data. It's not possible to just allow for plain *SQL* because we want to know a lot about the exact data types of each column involved in the query output.

This clause expect a comma separated list of view definitions, each one being either the name of an existing view in your database or the following expression:

```
*name* `AS` `$$` *sql query* `$$`
```

The *name* and the *sql query* will be used in a *CREATE VIEW* statement at the beginning of the data loading, and the resulting view will then be dropped at the end of the data loading.

MATERIALIZE ALL VIEWS

Same behaviour as *MATERIALIZE VIEWS* using the dynamic list of views as returned by MySQL rather than asking the user to specify the list.

MySQL Partial Migration

INCLUDING ONLY TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expression* used to limit the tables to migrate to a sublist.

Example:

```
including only table names matching ~/film/, 'actor'
```

EXCLUDING TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expression* used to exclude table names from the migration. This filter only applies to the result of the *INCLUDING* filter.

```
excluding table names matching ~<ory>
```

MySQL Encoding Support

DECODING TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expressions* used to force the encoding to use when processing data from MySQL. If the data encoding known to you is different from MySQL's idea about it, this is the option to use.

```
decoding table names matching ~/messed/, ~/encoding/ AS utf8
```

You can use as many such rules as you need, all with possibly different encodings.

MySQL Schema Transformations

ALTER TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expressions* that you want to target in the pgloader *ALTER TABLE* command. Available actions are *SET SCHEMA*, *RENAME TO*, and *SET*:

```
ALTER TABLE NAMES MATCHING ~/_list$/, 'sales_by_store', ~/sales_by/
 SET SCHEMA 'mv'

ALTER TABLE NAMES MATCHING 'film' RENAME TO 'films'

ALTER TABLE NAMES MATCHING ~/./ SET (fillfactor='40')

ALTER TABLE NAMES MATCHING ~/./ SET TABLESPACE 'pg_default'
```

You can use as many such rules as you need. The list of tables to be migrated is searched in pgloader memory against the *ALTER TABLE* matching rules, and for each command pgloader stops at the first matching criteria (regexp or string).

No *ALTER TABLE* command is sent to PostgreSQL, the modification happens at the level of the pgloader in-memory representation of your source database schema. In case of a name change, the mapping is kept and reused in the *foreign key* and *index* support.

The *SET ()* action takes effect as a *WITH* clause for the *CREATE TABLE* command that pgloader will run when it has to create a table.

The *SET TABLESPACE* action takes effect as a *TABLESPACE* clause for the *CREATE TABLE* command that pgloader will run when it has to create a table.

MySQL Migration: limitations

The *database* command currently only supports MySQL source database and has the following limitations:

- Views are not migrated,

  Supporting views might require implementing a full SQL parser for the MySQL dialect with a porting engine to rewrite the SQL against PostgreSQL, including renaming functions and changing some constructs.

  While it's not theoretically impossible, don't hold your breath.

- Triggers are not migrated

  The difficulty of doing so is not yet assessed.

- Of the geometric datatypes, only the *POINT* database has been covered. The other ones should be easy enough to implement now, it's just not done yet.

Default MySQL Casting Rules

When migrating from MySQL the following Casting Rules are provided:

Numbers:

```
type int with extra auto_increment to serial when (< precision 10)
type int with extra auto_increment to bigserial when (<= 10 precision)
type int to int       when  (< precision 10)
type int to bigint    when  (<= 10 precision)
```

(continues on next page)

(continued from previous page)

```
type tinyint   with extra auto_increment to serial
type smallint  with extra auto_increment to serial
type mediumint with extra auto_increment to serial
type bigint    with extra auto_increment to bigserial

type tinyint to boolean when (= 1 precision) using tinyint-to-boolean

type bit when (= 1 precision) to boolean drop typemod using bits-to-boolean
type bit to bit drop typemod using bits-to-hex-bitstring

type bigint when signed to bigint drop typemod
type bigint when (< 19 precision) to numeric drop typemod

type tinyint when unsigned to smallint   drop typemod
type smallint when unsigned to integer  drop typemod
type mediumint when unsigned to integer  drop typemod
type integer when unsigned to bigint    drop typemod

type tinyint to smallint   drop typemod
type smallint to smallint  drop typemod
type mediumint to integer  drop typemod
type integer to integer    drop typemod
type bigint to bigint      drop typemod

type float to float        drop typemod
type double to double precision drop typemod

type numeric to numeric keep typemod
type decimal to decimal keep typemod
```

Texts:

```
type char       to char keep typemod using remove-null-characters
type varchar    to varchar keep typemod using remove-null-characters
type tinytext   to text using remove-null-characters
type text       to text using remove-null-characters
type mediumtext to text using remove-null-characters
type longtext   to text using remove-null-characters
```

Binary:

```
type binary     to bytea using byte-vector-to-bytea
type varbinary  to bytea using byte-vector-to-bytea
type tinyblob   to bytea using byte-vector-to-bytea
type blob       to bytea using byte-vector-to-bytea
type mediumblob to bytea using byte-vector-to-bytea
type longblob   to bytea using byte-vector-to-bytea
```

Date:

```
type datetime when default "0000-00-00 00:00:00" and not null
  to timestamptz drop not null drop default
     using zero-dates-to-null

type datetime when default "0000-00-00 00:00:00"
  to timestamptz drop default
     using zero-dates-to-null
```

(continues on next page)

(continued from previous page)

```
type datetime with extra on update current timestamp when not null
  to timestamptz drop not null drop default
     using zero-dates-to-null

type datetime with extra on update current timestamp
  to timestamptz drop default
     using zero-dates-to-null

type timestamp when default "0000-00-00 00:00:00" and not null
  to timestamptz drop not null drop default
      using zero-dates-to-null

type timestamp when default "0000-00-00 00:00:00"
  to timestamptz drop default
      using zero-dates-to-null

type date when default "0000-00-00" to date drop default
      using zero-dates-to-null

type date to date
type datetime to timestamptz
type timestamp to timestamptz
type year to integer drop typemod
```

Geometric:

```
type geometry   to point using convert-mysql-point
type point      to point using convert-mysql-point
type linestring to path using convert-mysql-linestring
```

Enum types are declared inline in MySQL and separately with a *CREATE TYPE* command in PostgreSQL, so each column of Enum Type is converted to a type named after the table and column names defined with the same labels in the same order.

When the source type definition is not matched in the default casting rules nor in the casting rules provided in the command, then the type name with the typemod is used.

1.3.13 Migrating a SQLite database to PostgreSQL

This command instructs pgloader to load data from a SQLite file. Automatic discovery of the schema is supported, including build of the indexes.

Using default settings

Here is the simplest command line example, which might be all you need:

```
$ pgloader sqlite:///path/to/file.db pgsql://pguser@pghost/dbname
```

Using advanced options and a load command file

The command then would be:

```
$ pgloader db.load
```

Here's an example of the `db.load` contents then:

```
load database
     from sqlite:///Users/dim/Downloads/lastfm_tags.db
     into postgresql:///tags

 with include drop, create tables, create indexes, reset sequences

  set work_mem to '16MB', maintenance_work_mem to '512 MB';
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

SQLite Database Source Specification: FROM

Path or HTTP URL to a SQLite file, might be a *.zip* file.

SQLite Database Migration Options: WITH

When loading from a *SQLite* database, the following options are supported:

When loading from a *SQLite* database, the following options are supported, and the default *WITH* clause is: *no truncate*, *create tables*, *include drop*, *create indexes*, *reset sequences*, *downcase identifiers*, *encoding 'utf-8'*.

- *include drop*

  When this option is listed, pgloader drops all the tables in the target PostgreSQL database whose names appear in the SQLite database. This option allows for using the same command several times in a row until you figure out all the options, starting automatically from a clean environment. Please note that *CASCADE* is used to ensure that tables are dropped even if there are foreign keys pointing to them. This is precisely what *include drop* is intended to do: drop all target tables and recreate them.

  Great care needs to be taken when using *include drop*, as it will cascade to *all* objects referencing the target tables, possibly including other tables that are not being loaded from the source DB.

- *include no drop*

  When this option is listed, pgloader will not include any *DROP* statement when loading the data.

- *truncate*

  When this option is listed, pgloader issue the *TRUNCATE* command against each PostgreSQL table just before loading data into it.

- *no truncate*

  When this option is listed, pgloader issues no *TRUNCATE* command.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *create tables*

  When this option is listed, pgloader creates the table using the meta data found in the *SQLite* file, which must contain a list of fields with their data type. A standard data type conversion from SQLite to PostgreSQL is done.

- *create no tables*

  When this option is listed, pgloader skips the creation of table before loading data, target tables must then already exist.

  Also, when using *create no tables* pgloader fetches the metadata from the current target database and checks type casting, then will remove constraints and indexes prior to loading the data and install them back again once the loading is done.

- *create indexes*

  When this option is listed, pgloader gets the definitions of all the indexes found in the SQLite database and create the same set of index definitions against the PostgreSQL database.

- *create no indexes*

  When this option is listed, pgloader skips the creating indexes.

- *drop indexes*

  When this option is listed, pgloader drops the indexes in the target database before loading the data, and creates them again at the end of the data copy.

- *reset sequences*

  When this option is listed, at the end of the data loading and after the indexes have all been created, pgloader resets all the PostgreSQL sequences created to the current maximum value of the column they are attached to.

- *reset no sequences*

  When this option is listed, pgloader skips resetting sequences after the load.

  The options *schema only* and *data only* have no effects on this option.

- *schema only*

  When this option is listed pgloader will refrain from migrating the data over. Note that the schema in this context includes the indexes when the option *create indexes* has been listed.

- *data only*

  When this option is listed pgloader only issues the *COPY* statements, without doing any other processing.

- *encoding*

  This option allows to control which encoding to parse the SQLite text data with. Defaults to UTF-8.

SQLite Database Casting Rules

The command *CAST* introduces user-defined casting rules.

The cast clause allows to specify custom casting rules, either to overload the default casting rules or to amend them with special cases.

SQlite Database Partial Migrations

INCLUDING ONLY TABLE NAMES LIKE

Introduce a comma separated list of table name patterns used to limit the tables to migrate to a sublist.

Example:

```
including only table names like 'Invoice%'
```

EXCLUDING TABLE NAMES LIKE

Introduce a comma separated list of table name patterns used to exclude table names from the migration. This filter only applies to the result of the *INCLUDING* filter.

```
excluding table names like 'appointments'
```

Default SQLite Casting Rules

When migrating from SQLite the following Casting Rules are provided:

Numbers:

```
type tinyint to smallint using integer-to-string
type integer to bigint   using integer-to-string

type float to float   using float-to-string
type real to real     using float-to-string
type double to double precision     using float-to-string
type numeric to numeric     using float-to-string
type decimal to numeric     using float-to-string
```

Texts:

```
type character  to text drop typemod
type varchar    to text drop typemod
type nvarchar   to text drop typemod
type char       to text drop typemod
type nchar      to text drop typemod
type nvarchar   to text drop typemod
type clob       to text drop typemod
```

Binary:

```
type blob       to bytea
```

Date:

```
type datetime    to timestamptz using sqlite-timestamp-to-timestamp
type timestamp   to timestamptz using sqlite-timestamp-to-timestamp
type timestamptz to timestamptz using sqlite-timestamp-to-timestamp
```

1.3.14 Migrating a MS SQL Database to PostgreSQL

This command instructs pgloader to load data from a MS SQL database. Automatic discovery of the schema is supported, including build of the indexes, primary and foreign keys constraints.

Using default settings

Here is the simplest command line example, which might be all you need:

```
$ pgloader mssql://user@mshost/dbname pgsql://pguser@pghost/dbname
```

Using advanced options and a load command file

The command then would be:

```
$ pgloader ms.load
```

And the contents of the command file `ms.load` could be inspired from the following:

```
load database
     from mssql://user@host/dbname
     into postgresql:///dbname

including only table names like 'GlobalAccount' in schema 'dbo'

set work_mem to '16MB', maintenance_work_mem to '512 MB'

before load do $$ drop schema if exists dbo cascade; $$;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

MS SQL Database Source Specification: FROM

Connection string to an existing MS SQL database server that listens and welcome external TCP/IP connection. As pgloader currently piggybacks on the FreeTDS driver, to change the port of the server please export the *TDSPORT* environment variable.

MS SQL Database Migration Options: WITH

When loading from a *MS SQL* database, the same options as when loading a *MYSQL* database are supported. Please refer to the MYSQL section. The following options are added:

- *create schemas*

  When this option is listed, pgloader creates the same schemas as found on the MS SQL instance. This is the default.

- *create no schemas*

  When this option is listed, pgloader refrains from creating any schemas at all, you must then ensure that the target schema do exist.

MS SQL Database Casting Rules

CAST

The cast clause allows to specify custom casting rules, either to overload the default casting rules or to amend them with special cases.

Please refer to the MS SQL CAST clause for details.

MS SQL Views Support

MS SQL views support allows pgloader to migrate view as if they were base tables. This feature then allows for on-the-fly transformation from MS SQL to PostgreSQL, as the view definition is used rather than the base data.

MATERIALIZE VIEWS

This clause allows you to implement custom data processing at the data source by providing a *view definition* against which pgloader will query the data. It's not possible to just allow for plain *SQL* because we want to know a lot about the exact data types of each column involved in the query output.

This clause expect a comma separated list of view definitions, each one being either the name of an existing view in your database or the following expression:

```
*name* `AS` `$$` *sql query* `$$`
```

The *name* and the *sql query* will be used in a *CREATE VIEW* statement at the beginning of the data loading, and the resulting view will then be dropped at the end of the data loading.

MATERIALIZE ALL VIEWS

Same behaviour as *MATERIALIZE VIEWS* using the dynamic list of views as returned by MS SQL rather than asking the user to specify the list.

MS SQL Partial Migration

INCLUDING ONLY TABLE NAMES LIKE

Introduce a comma separated list of table name patterns used to limit the tables to migrate to a sublist. More than one such clause may be used, they will be accumulated together.

Example:

```
including only table names like 'GlobalAccount' in schema 'dbo'
```

EXCLUDING TABLE NAMES LIKE

Introduce a comma separated list of table name patterns used to exclude table names from the migration. This filter only applies to the result of the *INCLUDING* filter.

```
excluding table names matching 'LocalAccount' in schema 'dbo'
```

MS SQL Schema Transformations

ALTER SCHEMA '...' RENAME TO '...'

Allows to rename a schema on the flight, so that for instance the tables found in the schema 'dbo' in your source database will get migrated into the schema 'public' in the target database with this command:

```
alter schema 'dbo' rename to 'public'
```

ALTER TABLE NAMES MATCHING ... IN SCHEMA '...'

Introduce a comma separated list of table names or *regular expressions* that you want to target in the pgloader *ALTER TABLE* command. Available actions are *SET SCHEMA*, *RENAME TO*, and *SET*:

```
ALTER TABLE NAMES MATCHING ~/_list$/, 'sales_by_store', ~/sales_by/
  IN SCHEMA 'dbo'
 SET SCHEMA 'mv'

ALTER TABLE NAMES MATCHING 'film' IN SCHEMA 'dbo' RENAME TO 'films'

ALTER TABLE NAMES MATCHING ~/./ IN SCHEMA 'dbo' SET (fillfactor='40')

ALTER TABLE NAMES MATCHING ~/./ IN SCHEMA 'dbo' SET TABLESPACE 'tlbspc'
```

You can use as many such rules as you need. The list of tables to be migrated is searched in pgloader memory against the *ALTER TABLE* matching rules, and for each command pgloader stops at the first matching criteria (regexp or string).

No *ALTER TABLE* command is sent to PostgreSQL, the modification happens at the level of the pgloader in-memory representation of your source database schema. In case of a name change, the mapping is kept and reused in the *foreign key* and *index* support.

The *SET ()* action takes effect as a *WITH* clause for the *CREATE TABLE* command that pgloader will run when it has to create a table.

The *SET TABLESPACE* action takes effect as a *TABLESPACE* clause for the *CREATE TABLE* command that pgloader will run when it has to create a table.

The matching is done in pgloader itself, with a Common Lisp regular expression lib, so doesn't depend on the *LIKE* implementation of MS SQL, nor on the lack of support for regular expressions in the engine.

MS SQL Driver setup and encoding

pgloader is using the *FreeTDS* driver, and internally expects the data to be sent in utf-8. To achieve that, you can configure the FreeTDS driver with those defaults, in the file *~/.freetds.conf*:

```
[global]
    tds version = 7.4
    client charset = UTF-8
```

Default MS SQL Casting Rules

When migrating from MS SQL the following Casting Rules are provided:

Numbers:

```
type tinyint to smallint

type float to float   using float-to-string
type real to real     using float-to-string
type double to double precision     using float-to-string
type numeric to numeric     using float-to-string
type decimal to numeric     using float-to-string
type money to numeric     using float-to-string
type smallmoney to numeric     using float-to-string
```

Texts:

```
type char      to text drop typemod
type nchar     to text drop typemod
type varchar   to text drop typemod
type nvarchar  to text drop typemod
type xml       to text drop typemod
```

Binary:

```
type binary    to bytea using byte-vector-to-bytea
type varbinary to bytea using byte-vector-to-bytea
```

Date:

```
type datetime    to timestamptz
type datetime2   to timestamptz
```

Others:

```
type bit to boolean
type hierarchyid to bytea
type geography to bytea
type uniqueidentifier to uuid using sql-server-uniqueidentifier-to-uuid
```

1.3.15 Migrating a PostgreSQL Database to PostgreSQL

This command instructs pgloader to load data from a database connection. Automatic discovery of the schema is supported, including build of the indexes, primary and foreign keys constraints. A default set of casting rules are provided and might be overloaded and appended to by the command.

Using default settings

Here is the simplest command line example, which might be all you need:

```
$ pgloader pgsql://user@source/dbname pgsql://user@target/dbname
```

Using advanced options and a load command file

Here's a short example of migrating a database from a PostgreSQL server to another. The command would then be:

```
$ pgloader pg.load
```

And the contents of the command file `pg.load` could be inspired from the following:

```
load database
  from pgsql://localhost/pgloader
  into pgsql://localhost/copy

including only table names matching 'bits', ~/utilisateur/ in schema 'mysql'
including only table names matching ~/geolocations/ in schema 'public'
;
```

Common Clauses

Please refer to *Common Clauses* for documentation about common clauses.

PostgreSQL Database Source Specification: FROM

Must be a connection URL pointing to a PostgreSQL database.

See the *SOURCE CONNECTION STRING* section above for details on how to write the connection string.

```
pgsql://[user[:password]@][netloc][:port][/dbname][?option=value&...]
```

PostgreSQL Database Migration Options: WITH

When loading from a *PostgreSQL* database, the following options are supported, and the default *WITH* clause is: *no truncate*, *create schema*, *create tables*, *include drop*, *create indexes*, *reset sequences*, *foreign keys*, *downcase identifiers*, *uniquify index names*, *reindex*.

- *include drop*

  When this option is listed, pgloader drops all the tables in the target PostgreSQL database whose names appear in the MySQL database. This option allows for using the same command several times in a row until you figure out all the options, starting automatically from a clean environment. Please note that *CASCADE* is used to ensure that tables are dropped even if there are foreign keys pointing to them. This is precisely what *include drop* is intended to do: drop all target tables and recreate them.

  Great care needs to be taken when using *include drop*, as it will cascade to *all* objects referencing the target tables, possibly including other tables that are not being loaded from the source DB.

- *include no drop*

  When this option is listed, pgloader will not include any *DROP* statement when loading the data.

- *truncate*

  When this option is listed, pgloader issue the *TRUNCATE* command against each PostgreSQL table just before loading data into it.

- *no truncate*

  When this option is listed, pgloader issues no *TRUNCATE* command.

- *disable triggers*

  When this option is listed, pgloader issues an *ALTER TABLE ... DISABLE TRIGGER ALL* command against the PostgreSQL target table before copying the data, then the command *ALTER TABLE ... ENABLE TRIGGER ALL* once the *COPY* is done.

This option allows loading data into a pre-existing table ignoring the *foreign key constraints* and user defined triggers and may result in invalid *foreign key constraints* once the data is loaded. Use with care.

- *create tables*

  When this option is listed, pgloader creates the table using the meta data found in the *MySQL* file, which must contain a list of fields with their data type. A standard data type conversion from DBF to PostgreSQL is done.

- *create no tables*

  When this option is listed, pgloader skips the creation of table before loading data, target tables must then already exist.

  Also, when using *create no tables* pgloader fetches the metadata from the current target database and checks type casting, then will remove constraints and indexes prior to loading the data and install them back again once the loading is done.

- *create indexes*

  When this option is listed, pgloader gets the definitions of all the indexes found in the MySQL database and create the same set of index definitions against the PostgreSQL database.

- *create no indexes*

  When this option is listed, pgloader skips the creating indexes.

- *drop indexes*

  When this option is listed, pgloader drops the indexes in the target database before loading the data, and creates them again at the end of the data copy.

- *reindex*

  When this option is used, pgloader does both *drop indexes* before loading the data and *create indexes* once data is loaded.

- *drop schema*

  When this option is listed, pgloader drops the target schema in the target PostgreSQL database before creating it again and all the objects it contains. The default behavior doesn't drop the target schemas.

- *foreign keys*

  When this option is listed, pgloader gets the definitions of all the foreign keys found in the MySQL database and create the same set of foreign key definitions against the PostgreSQL database.

- *no foreign keys*

  When this option is listed, pgloader skips creating foreign keys.

- *reset sequences*

  When this option is listed, at the end of the data loading and after the indexes have all been created, pgloader resets all the PostgreSQL sequences created to the current maximum value of the column they are attached to.

  The options *schema only* and *data only* have no effects on this option.

- *reset no sequences*

  When this option is listed, pgloader skips resetting sequences after the load.

  The options *schema only* and *data only* have no effects on this option.

- *downcase identifiers*

  When this option is listed, pgloader converts all MySQL identifiers (table names, index names, column names) to *downcase*, except for PostgreSQL *reserved* keywords.

The PostgreSQL *reserved* keywords are determined dynamically by using the system function *pg_get_keywords()*.

- *quote identifiers*

  When this option is listed, pgloader quotes all MySQL identifiers so that their case is respected. Note that you will then have to do the same thing in your application code queries.

- *schema only*

  When this option is listed pgloader refrains from migrating the data over. Note that the schema in this context includes the indexes when the option *create indexes* has been listed.

- *data only*

  When this option is listed pgloader only issues the *COPY* statements, without doing any other processing.

- *rows per range*

  How many rows are fetched per *SELECT* query when using *multiple readers per thread*, see above for details.

PostgreSQL Database Casting Rules

The command *CAST* introduces user-defined casting rules.

The cast clause allows to specify custom casting rules, either to overload the default casting rules or to amend them with special cases.

A casting rule is expected to follow one of the forms:

```
type <type-name> [ <guard> ... ] to <pgsql-type-name> [ <option> ... ]
column <table-name>.<column-name> [ <guards> ] to ...
```

It's possible for a *casting rule* to either match against a PostgreSQL data type or against a given *column name* in a given *table name*. So it's possible to migrate a table from a PostgreSQL database while changing and *int* column to a *bigint* one, automatically.

The *casting rules* are applied in order, the first match prevents following rules to be applied, and user defined rules are evaluated first.

The supported guards are:

- *when default 'value'*

  The casting rule is only applied against MySQL columns of the source type that have given *value*, which must be a single-quoted or a double-quoted string.

- *when typemod expression*

  The casting rule is only applied against MySQL columns of the source type that have a *typemod* value matching the given *typemod expression*. The *typemod* is separated into its *precision* and *scale* components.

  Example of a cast rule using a *typemod* guard:

  ```
  type char when (= precision 1) to char keep typemod
  ```

  This expression casts MySQL *char(1)* column to a PostgreSQL column of type *char(1)* while allowing for the general case *char(N)* will be converted by the default cast rule into a PostgreSQL type *varchar(N)*.

- *with extra auto_increment*

  The casting rule is only applied against PostgreSQL attached to a sequence. This can be the result of doing that manually, using a *serial* or a *bigserial* data type, or an *identity* column.

The supported casting options are:

- *drop default*, *keep default*

  When the option *drop default* is listed, pgloader drops any existing default expression in the MySQL database for columns of the source type from the *CREATE TABLE* statement it generates.

  The spelling *keep default* explicitly prevents that behaviour and can be used to overload the default casting rules.

- *drop not null*, *keep not null*, *set not null*

  When the option *drop not null* is listed, pgloader drops any existing *NOT NULL* constraint associated with the given source MySQL datatype when it creates the tables in the PostgreSQL database.

  The spelling *keep not null* explicitly prevents that behaviour and can be used to overload the default casting rules.

  When the option *set not null* is listed, pgloader sets a *NOT NULL* constraint on the target column regardless whether it has been set in the source MySQL column.

- *drop typemod*, *keep typemod*

  When the option *drop typemod* is listed, pgloader drops any existing *typemod* definition (e.g. *precision* and *scale*) from the datatype definition found in the MySQL columns of the source type when it created the tables in the PostgreSQL database.

  The spelling *keep typemod* explicitly prevents that behaviour and can be used to overload the default casting rules.

- *using*

  This option takes as its single argument the name of a function to be found in the *pgloader.transforms* Common Lisp package. See above for details.

  It's possible to augment a default cast rule (such as one that applies against *ENUM* data type for example) with a *transformation function* by omitting entirely the *type* parts of the casting rule, as in the following example:

```
column enumerate.foo using empty-string-to-null
```

PostgreSQL Views Support

PostgreSQL views support allows pgloader to migrate view as if they were base tables. This feature then allows for on-the-fly transformation of the source schema, as the view definition is used rather than the base data.

MATERIALIZE VIEWS

This clause allows you to implement custom data processing at the data source by providing a *view definition* against which pgloader will query the data. It's not possible to just allow for plain *SQL* because we want to know a lot about the exact data types of each column involved in the query output.

This clause expect a comma separated list of view definitions, each one being either the name of an existing view in your database or the following expression:

```
*name* `AS` `$$` *sql query* `$$`
```

The *name* and the *sql query* will be used in a *CREATE VIEW* statement at the beginning of the data loading, and the resulting view will then be dropped at the end of the data loading.

MATERIALIZE ALL VIEWS

Same behaviour as *MATERIALIZE VIEWS* using the dynamic list of views as returned by PostgreSQL rather than asking the user to specify the list.

PostgreSQL Partial Migration

INCLUDING ONLY TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expression* used to limit the tables to migrate to a sublist.

Example:

```
including only table names matching ~/film/, 'actor' in schema 'public'
```

EXCLUDING TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expression* used to exclude table names from the migration. This filter only applies to the result of the *INCLUDING* filter.

```
excluding table names matching ~<ory> in schema 'public'
```

PostgreSQL Schema Transformations

ALTER TABLE NAMES MATCHING

Introduce a comma separated list of table names or *regular expressions* that you want to target in the pgloader *ALTER TABLE* command. Available actions are *SET SCHEMA*, *RENAME TO*, and *SET*:

```
ALTER TABLE NAMES MATCHING ~/_list$/, 'sales_by_store', ~/sales_by/
  IN SCHEMA 'public'
 SET SCHEMA 'mv'

ALTER TABLE NAMES MATCHING 'film' IN SCHEMA 'public' RENAME TO 'films'

ALTER TABLE NAMES MATCHING ~/./ IN SCHEMA 'public' SET (fillfactor='40')

ALTER TABLE NAMES MATCHING ~/./ IN SCHEMA 'public' SET TABLESPACE 'pg_default'
```

You can use as many such rules as you need. The list of tables to be migrated is searched in pgloader memory against the *ALTER TABLE* matching rules, and for each command pgloader stops at the first matching criteria (regexp or string).

No *ALTER TABLE* command is sent to PostgreSQL, the modification happens at the level of the pgloader in-memory representation of your source database schema. In case of a name change, the mapping is kept and reused in the *foreign key* and *index* support.

The *SET ()* action takes effect as a *WITH* clause for the *CREATE TABLE* command that pgloader will run when it has to create a table.

The *SET TABLESPACE* action takes effect as a *TABLESPACE* clause for the *CREATE TABLE* command that pgloader will run when it has to create a table.

PostgreSQL Migration: limitations

The only PostgreSQL objects supported at this time in pgloader are extensions, schema, tables, indexes and constraints. Anything else is ignored.

- Views are not migrated,

  Supporting views might require implementing a full SQL parser for the MySQL dialect with a porting engine to rewrite the SQL against PostgreSQL, including renaming functions and changing some constructs.

  While it's not theoretically impossible, don't hold your breath.

- Triggers are not migrated

  The difficulty of doing so is not yet assessed.

- Stored Procedures and Functions are not migrated.

Default PostgreSQL Casting Rules

When migrating from PostgreSQL the following Casting Rules are provided:

```
type int with extra auto_increment to serial
type bigint with extra auto_increment to bigserial
type "character varying" to text drop typemod
```

1.3.16 Migrating a PostgreSQL Database to Citus

This command instructs pgloader to load data from a database connection. Automatic discovery of the schema is supported, including build of the indexes, primary and foreign keys constraints. A default set of casting rules are provided and might be overloaded and appended to by the command.

Automatic distribution column backfilling is supported, either from commands that specify what is the distribution column in every table, or only in the main table, then relying on foreign key constraints to discover the other distribution keys.

Here's a short example of migrating a database from a PostgreSQL server to another:

```
load database
from pgsql:///hackathon
into pgsql://localhost:9700/dim

with include drop, reset no sequences

cast column impressions.seen_at to "timestamp with time zone"

distribute companies using id
-- distribute campaigns using company_id
-- distribute ads using company_id from campaigns
-- distribute clicks using company_id from ads, campaigns
-- distribute impressions using company_id from ads, campaigns
;
```

Everything works exactly the same way as when doing a PostgreSQL to PostgreSQL migration, with the added fonctionality of this new *distribute* command.

Distribute Command

The distribute command syntax is as following:

```
distribute <table name> using <column name>
distribute <table name> using <column name> from <table> [, <table>, ...]
distribute <table name> as reference table
```

When using the distribute command, the following steps are added to pgloader operations when migrating the schema:

- if the distribution column does not exist in the table, it is added as the first column of the table
- if the distribution column does not exists in the primary key of the table, it is added as the first column of the primary of the table
- all the foreign keys that point to the table are added the distribution key automatically too, including the source tables of the foreign key constraints
- once the schema has been created on the target database, pgloader then issues Citus specific command create_reference_table() and create_distributed_table() to make the tables distributed

Those operations are done in the schema section of pgloader, before the data is loaded. When the data is loaded, the newly added columns need to be backfilled from referenced data. pgloader knows how to do that by generating a query like the following and importing the result set of such a query rather than the raw data from the source table.

Citus Migration Example

With the migration command as above, pgloader adds the column `company_id` to the tables that have a direct or indirect foreign key reference to the `companies` table.

We run pgloader using the following command, where the file ./test/citus/company.load contains the pgloader command as shown above.

```
$ pgloader --client-min-messages sql ./test/citus/company.load
```

The following SQL statements are all extracted from the log messages that the pgloader command outputs. We are going to have a look at the *impressions* table. It gets created with a new column *company_id* in the first position, as follows:

```
CREATE TABLE "public"."impressions"
(
  company_id               bigint,
  "id"                     bigserial,
  "ad_id"                  bigint default NULL,
  "seen_at"                timestamp with time zone default NULL,
  "site_url"               text default NULL,
  "cost_per_impression_usd" numeric(20,10) default NULL,
  "user_ip"                inet default NULL,
  "user_data"              jsonb default NULL
);
```

The original schema for this table does not have the *company_id* column, which means pgloader now needs to change the primary key definition, the foreign keys constraints definitions from and to this table, and also to *backfill* the *company_id* data to this table when doing the COPY phase of the migration.

Then once the tables have been created, pgloader executes the following SQL statements:

```
SELECT create_distributed_table('"public"."companies"', 'id');
SELECT create_distributed_table('"public"."campaigns"', 'company_id');
SELECT create_distributed_table('"public"."ads"', 'company_id');
SELECT create_distributed_table('"public"."clicks"', 'company_id');
SELECT create_distributed_table('"public"."impressions"', 'company_id');
```

Then when copying the data from the source PostgreSQL database to the new Citus tables, the new column (here `company_id`) needs to be backfilled from the source tables. Here's the SQL query that pgloader uses as a data source for the `ads` table in our example:

```
SELECT "campaigns".company_id::text, "ads".id::text, "ads".campaign_id::text,
       "ads".name::text, "ads".image_url::text, "ads".target_url::text,
       "ads".impressions_count::text, "ads".clicks_count::text,
       "ads".created_at::text, "ads".updated_at::text

  FROM       "public"."ads"
       JOIN "public"."campaigns"
         ON ads.campaign_id = campaigns.id
```

The `impressions` table has an indirect foreign key reference to the `company` table, which is the table where the distribution key is specified. pgloader will discover that itself from walking the PostgreSQL catalogs, and you may also use the following specification in the pgloader command to explicitely add the indirect dependency:

```
distribute impressions using company_id from ads, campaigns
```

Given this schema, the SQL query used by pgloader to fetch the data for the *impressions* table is the following, implementing online backfilling of the data:

```
SELECT "campaigns".company_id::text, "impressions".id::text,
       "impressions".ad_id::text, "impressions".seen_at::text,
       "impressions".site_url::text,
       "impressions".cost_per_impression_usd::text,
       "impressions".user_ip::text,
       "impressions".user_data::text

  FROM      "public"."impressions"

       JOIN "public"."ads"
         ON impressions.ad_id = ads.id

       JOIN "public"."campaigns"
         ON ads.campaign_id = campaigns.id
```

When the data copying is done, then pgloader also has to install the indexes supporting the primary keys, and add the foreign key definitions to the schema. Those definitions are not the same as in the source schema, because of the adding of the distribution column to the table: we need to also add the column to the primary key and the foreign key constraints.

Here's the commands issued by pgloader for the `impressions` table:

```
CREATE UNIQUE INDEX "impressions_pkey"
    ON "public"."impressions" (company_id, id);

ALTER TABLE "public"."impressions"
  ADD CONSTRAINT "impressions_ad_id_fkey"
     FOREIGN KEY(company_id,ad_id)
      REFERENCES "public"."ads"(company_id,id)
```

Given a single line of specification `distribute companies using id` then pgloader implements all the necessary schema changes on the fly when migrating to Citus, and also dynamically backfills the data.

Citus Migration: Limitations

The way pgloader implements *reset sequence* does not work with Citus at this point, so sequences need to be taken care of separately at this point.

1.3.17 Support for Redshift in pgloader

The command and behavior are the same as when migration from a PostgreSQL database source, see *Migrating a PostgreSQL Database to PostgreSQL*. pgloader automatically discovers that it's talking to a Redshift database by parsing the output of the `SELECT version()` SQL query.

Redshift as a data source

Redshift is a variant of PostgreSQL version 8.0.2, which allows pgloader to work with only a very small amount of adaptation in the catalog queries used. In other words, migrating from Redshift to PostgreSQL works just the same as when migrating from a PostgreSQL data source, including the connection string specification.

Redshift as a data destination

The Redshift variant of PostgreSQL 8.0.2 does not have support for the `COPY FROM STDIN` feature that pgloader normally relies upon. To use COPY with Redshift, the data must first be made available in an S3 bucket.

First, pgloader must authenticate to Amazon S3. pgloader uses the following setup for that:

- `~/.aws/config`

  This INI formatted file contains sections with your default region and other global values relevant to using the S3 API. pgloader parses it to get the region when it's setup in the `default` INI section.

  The environment variable `AWS_DEFAULT_REGION` can be used to override the configuration file value.

- `~/.aws/credentials`

  The INI formatted file contains your authentication setup to Amazon, with the properties `aws_access_key_id` and `aws_secret_access_key` in the section `default`. pgloader parses this file for those keys, and uses their values when communicating with Amazon S3.

  The environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` can be used to override the configuration file

- `AWS_S3_BUCKET_NAME`

  Finally, the value of the environment variable `AWS_S3_BUCKET_NAME` is used by pgloader as the name of the S3 bucket where to upload the files to COPY to the Redshift database. The bucket name defaults to `pgloader`.

Then pgloader works as usual, see the other sections of the documentation for the details, depending on the data source (files, other databases, etc). When preparing the data for PostgreSQL, pgloader now uploads each batch into a single CSV file, and then issue such as the following, for each batch:

```
COPY <target_table_name>
      FROM 's3://<s3 bucket>/<s3-filename-just-uploaded>'
      FORMAT CSV
      TIMEFORMAT 'auto'
```

(continues on next page)

(continued from previous page)

```
    REGION '<aws-region>'
    ACCESS_KEY_ID '<aws-access-key-id>'
    SECRET_ACCESS_KEY '<aws-secret-access-key>;
```

This is the only difference with a PostgreSQL core version, where pgloader can rely on the classic `COPY FROM STDIN` command, which allows to send data through the already established connection to PostgreSQL.

1.3.18 Transformation Functions

Some data types are implemented in a different enough way that a transformation function is necessary. This function must be written in *Common lisp* and is searched in the *pgloader.transforms* package.

Some default transformation function are provided with pgloader, and you can use the *–load* command line option to load and compile your own lisp file into pgloader at runtime. For your functions to be found, remember to begin your lisp file with the following form:

```
(in-package #:pgloader.transforms)
```

The provided transformation functions are:

- *zero-dates-to-null*

  When the input date is all zeroes, return *nil*, which gets loaded as a PostgreSQL *NULL* value.

- *date-with-no-separator*

  Applies *zero-dates-to-null* then transform the given date into a format that PostgreSQL will actually process:

  ```
  In:  "20041002152952"
  Out: "2004-10-02 15:29:52"
  ```

- *time-with-no-separator*

  Transform the given time into a format that PostgreSQL will actually process:

  ```
  In:  "08231560"
  Out: "08:23:15.60"
  ```

- *tinyint-to-boolean*

  As MySQL lacks a proper boolean type, *tinyint* is often used to implement that. This function transforms *0* to *'false'* and anything else to *'true'*.

- *bits-to-boolean*

  As MySQL lacks a proper boolean type, *BIT* is often used to implement that. This function transforms 1-bit bit vectors from *0* to *f* and any other value to *t*..

- *int-to-ip*

  Convert an integer into a dotted representation of an ip4.

  ```
  In:  18435761
  Out: "1.25.78.177"
  ```

- *ip-range*

  Converts a couple of integers given as strings into a range of ip4.

```
In:  "16825344" "16825599"
Out: "1.0.188.0-1.0.188.255"
```

- *convert-mysql-point*

  Converts from the *astext* representation of points in MySQL to the PostgreSQL representation.

```
In:  "POINT(48.5513589 7.6926827)"
Out: "(48.5513589,7.6926827)"
```

- *integer-to-string*

  Converts a integer string or a Common Lisp integer into a string suitable for a PostgreSQL integer. Takes care of quoted integers.

```
In:  "\"0\""
Out: "0"
```

- *float-to-string*

  Converts a Common Lisp float into a string suitable for a PostgreSQL float:

```
In:  100.0d0
Out: "100.0"
```

- *hex-to-dec*

  Converts a string containing an hexadecimal representation of a number into its decimal representation:

```
In:  "deadbeef"
Out: "3735928559"
```

- *set-to-enum-array*

  Converts a string representing a MySQL SET into a PostgreSQL Array of Enum values from the set.

```
In: "foo,bar"
Out: "{foo,bar}"
```

- *empty-string-to-null*

  Convert an empty string to a null.

- *right-trim*

  Remove whitespace at end of string.

- *remove-null-characters*

  Remove *NUL* characters (*0x0*) from given strings.

- *byte-vector-to-bytea*

  Transform a simple array of unsigned bytes to the PostgreSQL bytea Hex Format representation as documented at http://www.postgresql.org/docs/9.3/interactive/datatype-binary.html

- *sqlite-timestamp-to-timestamp*

  SQLite type system is quite interesting, so cope with it here to produce timestamp literals as expected by PostgreSQL. That covers year only on 4 digits, 0 dates to null, and proper date strings.

- *sql-server-uniqueidentifier-to-uuid*

  The SQL Server driver receives data fo type uniqueidentifier as byte vector that we then need to convert to an UUID string for PostgreSQL COPY input format to process.

- *unix-timestamp-to-timestamptz*

  Converts a unix timestamp (number of seconds elapsed since beginning of 1970) into a proper PostgreSQL timestamp format.

- *varbinary-to-string*

  Converts binary encoded string (such as a MySQL *varbinary* entry) to a decoded text, using the table's encoding that may be overloaded with the *DECODING TABLE NAMES MATCHING* clause.

1.3.19 Reporting Bugs

pgloader is a software and as such contains bugs. Most bugs are easy to solve and taken care of in a short delay. For this to be possible though, bug reports need to follow those recommandations:

- include pgloader version,
- include problematic input and output,
- include a description of the output you expected,
- explain the difference between the ouput you have and the one you expected,
- include a self-reproducing test-case

Test Cases to Reproduce Bugs

Use the *inline* source type to help reproduce a bug, as in the pgloader tests:

```
LOAD CSV
   FROM INLINE
   INTO postgresql://dim@localhost/pgloader?public."HS"

   WITH truncate,
        fields terminated by '\t',
        fields not enclosed,
        fields escaped by backslash-quote,
        quote identifiers

    SET work_mem to '128MB',
        standard_conforming_strings to 'on',
        application_name to 'my app name'

 BEFORE LOAD DO
  $$ create extension if not exists hstore; $$,
  $$ drop table if exists "HS"; $$,
  $$ CREATE TABLE "HS"
     (
        id     serial primary key,
        kv     hstore
     )
  $$;
```

(continues on next page)

(continued from previous page)

```
1     email=>foo@example.com,a=>b
2     test=>value
3     a=>b,c=>"quoted hstore value",d=>other
4     baddata
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search