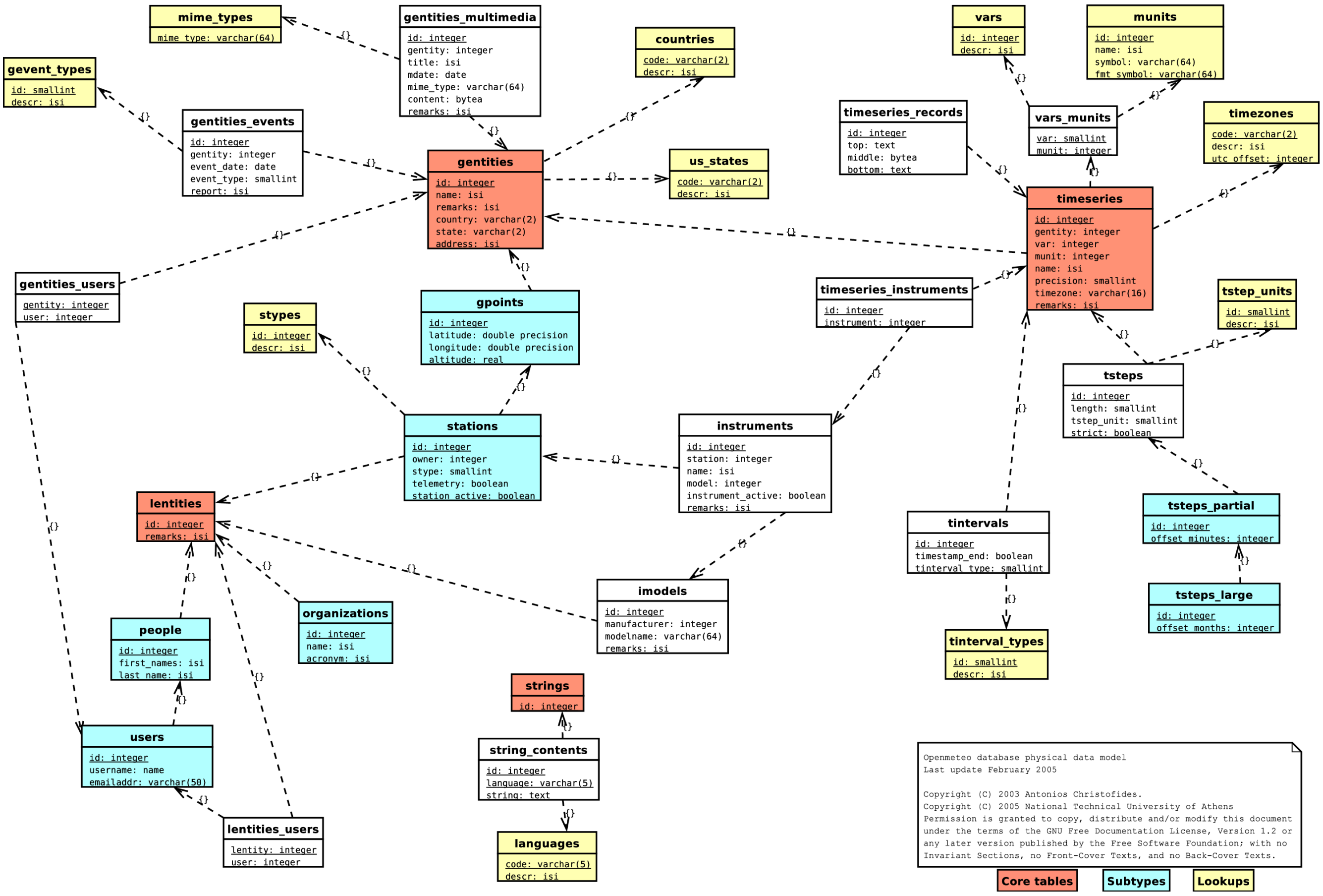
Openmeteo database description



Naming conventions

The database contains tables, triggers, functions, views, rules, constraints and sequences. The diagram shows only the tables and some constraints.

Tables are named normally, for example, the table holding information about meteorological instruments is called `instruments`. Triggers occurring on insert, update, or delete have names that begin with `ti_`, `tu_` or `td_` respectively, and continue with the table name. These triggers call functions the names of which usually begin with `fi_`, `fu_` or `fd_` and continue with the table name. Occasionally triggers associated with a table call functions associated with another table if it so happens that the tables are similar enough that one set of functions can do the job for both.

The names of views start with a v and continue with the name of a table. For example, the `vstations` view corresponds to the `stations` table. In fact, `vstations`, like most views of the database, is multi-table, but `stations` can be considered the basic table. PostgreSQL does not support updatable views directly, so for each view there is a set of three rules that serve to give the illusion of an updatable view. The names of these rules begin with `ri_`, `ru_`, and `rd_` and continue with the view name.

The names of the constraints begin with `pk_` (for primary keys), `uk_` (for unique keys), `fk_` (for foreign keys), and `ch_` (for check constraints), and continue with the table name. The names for unique and foreign key constraints continue with an underscore and a column name (it so happens that all such keys are currently simple, that is, single-column; if, in the future, we have a complex foreign or unique key, we'll use both columns in the name, or only the first one if it's too long).

Strings

There are very few *varchar*varchar and *text*text columns in the database. Instead, integer string identifiers are used, and the actual strings are stored in the `string_contents` table. The reason for this design is to enable translations in an unlimited number of languages.

Let's see an example. The `people` table has a `last_name` column. This is an integer foreign key to the `strings` table (which contains only one column, the `id`). There can be as many translations of the full name in the `string_contents` table, whose key is the composite (`id`, `language`).

The diagram does not show that `last_name` is a foreign key to `strings`, because there are so many such constraints that if they were shown, the diagram would get cluttered with arrows. However, the diagram shows all such keys with "isi" (for integer string identifier) as the data type, rather than "integer", so that you understand that it is actually a foreign key to `strings`.

Application code does not need (and in fact is not allowed) to insert integer string identifiers. For example, when you insert to the `people` table, you don't specify `last_name`; a trigger will create it automatically and add it to the `strings` table. Triggers also ensure that the identifier does not change when updating rows.

Views with English strings

Due to the fact that most strings are integer identifiers, tables are hard to read. For this reason, most tables have a corresponding view that contains all rows of the table plus the English versions of its strings. For example, for the `stypes` (station types) table, which is a lookup consisting of an `id` and a `descr`, there is the `vstypes` view with columns `id`, `descr` and `descr_en`. These views should not normally be used by the application code; they are mainly intended for the database administrator, so that he can easily check what's in the table or insert rows in lookup tables with the English versions of the strings.

In constrast, views corresponding to subtype tables are intended to be used by application code (see below).

Supertypes and subtypes

The database contains some groups of entity hierarchies, the most notable being gentities and lentities.

Examples of gentities (short for geographical entities) are measuring stations, cities, boreholes and watersheds. A gentity can be a point (e.g. stations and boreholes), a surface (e.g. lakes and watersheds), a line (e.g. aqueducts), or a network (e.g. a river). Currently the only implemented gentity subtype is the measuring station. Other gentities will be implemented only when and if it becomes necessary or useful. A measuring station is thus a subtype of a point, which is a subtype of a gentity, and is stored in three tables: `stations`, `gpoints` and `gentities`.

Lentities can be either individuals or groups of individuals or legal entities. Currently the implemented subtypes are people and organizations. An individual is thus stored in two tables: `people` and `lentities`; likewise, an organisation is stored in `organizations` and `lentities`.

For each low-level subtype, namely station, individual and organization, there is a corresponding view that contains the columns from all tables, subtype and supertype. Thus, the `vstations` view contains all columns from `stations`, `gpoints` and `gentities`. Application code should use either the view, or the tables, depending on what is more convenient. For example, if it is essentially the same application code that accepts input from the user in order to fill in `people` and `organizations`, it may be better to use the tables; if the code for the two cases is completely separate, it may be better to use the views. If the application code uses the tables, the way integrity rules work requires that a row is first inserted to the lowest level table first, then to the higher level table, and so on up to the supertype table (it is the opposite of what you'd expect, and it works because the relevant foreign key constraints are deferred; the opposite wouldn't work well because I couldn't find a way to ensure that a row in a supertype table always has one and only one corresponding row in a subtype table).

In addition to the columns from all supertype and subtype tables, the subtype views also contain columns with the English versions of the strings. Again, these columns are intended for the database administrator and should not be used by application code.

History tables

Most tables have a corresponding history table that logs operations. History tables are not shown on the diagram. For example, for the `imodels` table there is the `imodels_history` table. Each time a row is inserted, updated or deleted on `imodels`, a row is added to `imodels_history` showing what happened. The history tables contain as many columns as the corresponding main tables, plus the columns `_seq`, `_operation`, `_when`, and `_who`. `_seq` is a sequence generated integer used as the history table's primary key, `_operation` is one of `I`, `U` and `D`, `_when` is the timestamp when the operation occurred, and `_who` is the user name of the user who caused the change. The columns that correspond to the columns of the main table are all nullable and have no constraints, and they are filled in with the new values (for deletes, the columns that comprise the main table's primary key are filled in with the old values and the rest are null). For the few tables for which primary key value changes are allowed, the history table contains columns for both the old and the new value of the primary key.

History tables are automatically taken care of by insert, update and delete triggers. Currently they are not used at all by application code. They exist in case they come in handy for undoing possible major user errors, or in case application code is written in the future to show possibly useful logging information. There are no history tables for timeseries_records, and for the content field of gentities_multimedia, as this would require large amounts of disk space.

Time series records

The time series records are naturally stored in the timeseries_records table, in three chunks: top, middle and bottom. 'top' and 'bottom' are plain text, whereas 'middle' is plain text compressed in gzip file format. The concatenation of top, uncompressed middle, and bottom, is the entire time series in a plain text comma-separated values file. Each line of that file has the format

> YYYY-MM-DD HH:mm,value,flags

The value is a number, which uses the dot as the decimal separator. The third field contains optional flags or short comments; there are no restrictions on the format of the flags field; however, non-ascii characters should be avoided, and if present they must be encoded in UTF-8. The line separator is the two byte sequence used by DOS and Windows systems.

'top' stores the first few lines of the file, up to around 100. 'bottom' stores the last few lines of the file, at least one. 'middle' stores all the rest. 'Bottom' is not null; if a timeseries is empty, there must be no row in timeseries_records. If it contains only a few records (up to, say, 1000), they must all be stored in 'bottom', the other two fields being empty. If it contains more records, around 100 must be stored in 'top', 100 'bottom', and the rest in 'middle'. Appending a record to the timeseries must be done by appending to 'bottom', regardless of how many records it contains already.

The database does not enforce any of all these requirements; they must be taken care of by the application software.

The advantages of this paradoxical system of storage are: (1) Large time series are uncompressed on the client, thus easing network and

Unexpected indentation.

> server load.

Block quote ends without a blank line; unexpected unindent.

1. Very little disk space is used (20 times less than storing time series in an (id, date, value, flags) table).
2. If 'top' and 'bottom' are kept small, it is very fast to perform the frequently needed operations of retrieving the first and last records and appending a record. All other operations must practically retrieve/update the entire time series, which experience has shown that it is what is done anyway.

Time steps and time stamps

Modelling the time step of a time series is an unexpectedly complex undertaking, and I believe that what I did sucks big time. Still, I couldn't come up with a better solution, so until you do so, here it is. Let's look at the problem first.

Examples of time steps that I have encountered include five minutes, ten minutes, an hour, six hours, a day, a month, and a year. However, I have the nasty feeling that many other steps, such as weekly and quarterly, are common. The National Meteorological Service of Greece also makes temperature measurements three times a day, at 08:00, 14:00, and 20:00, which is a special case of a six-hour step in which one every three measurements is missing. In addition, annual time steps can be very different; they can refer to the calendar year (for example, 2003, meaning the period of time starting at 2003-01-01 00:00 and ending at 2004-01-01 00:00), or they can refer to the hydrological year. The hydrological year in Greece begins at 1 October, but in Australia it could be something near 1 April, and I'll be surprised if it's only confined to these two cases. Another problem is that, for daily time steps, some services in Greece consider the day 2003-11-19 to be the 24 hour interval ending at 2003-11-19 08:00, whereas others consider it to be the 24 hour interval *starting* at 2003-11-19 00:00. As if all this was not enough, timeseries are occasionally measured at different times from those they should; for example, once in a while it will happen that a temperature measurement that occurs daily at 08:00 will be recorded at 10:00 because the observer overslept; and an automatic station measuring at :10, :20, :30 etc. will occasionally measure at :11, :21, :31 etc., because of a setup error, a clock lapse, or a software error.

The time step of a time series is measured either in minutes or in months. The minute and the month thus comprise the two rows stored in `tstep_units`. Each row of the `timeseries` table has a corresponding row in `tsteps` that tells the time step of that timeseries, consisting of a length and the unit, such as 1440 minutes or 12 months. The boolean column `strict` shows whether disturbances are allowed. If it is *guaranteed* that the records of a time series are 60 minutes apart from each other (or, more precisely, multiples of 60 minutes apart, since there may be missing values), then that time series has a time step of 60 minutes strict. Note that timeseries consisting of raw measurements are almost never strict; one day you *will* oversleep, and your automatic station's clock *will* slip, no matter how precise it is, how careful you are, what kind of NTP server you use for synchronising it, and how many decades it has worked without a single problem. The only way to guarantee time step strictness is to process a timeseries of raw measurements and produce a new one, with any disturbances eliminated.

The time stamp of a time series record may refer to a moment in time or to an interval. For measurements of instantaneous variables, it is a moment in time; in most other cases it is the start or end of an interval. The information stored in `tintervals` shows which condition holds. If for a row of `timeseries` there is no corresponding row in `tintervals`, that series' timestamps refer to moments in time; otherwise they refer to intervals, and the boolean `timestamp_end` tells whether the timestamp is the end or the start of the interval. `tinterval_type` tells if the value is the sum, average, maximum, minimum, or vector average of the variable over the interval.

For time steps smaller than a day, what we've done so far is adequate. Such small time steps, however, have the convenient feature that the time stamp is accurate to the minute, and thus specifies the moment in time (whether it means itself or the start or end of an interval) precisely (in a meteorological time scale). In contrast, time steps of a day or longer are frequently communicated as "19 November 2003" or "February 1964", and additional information is needed to know what they mean. We store such additional information in `tsteps_partial`, meaning the time steps for which time stamps are partially stored and displayed. Not all time steps of a day or larger are such; it depends on how the time stamps are stored. If the file containing the data for a daily time series only stores the year, month, and day, then the step is tstep_partial; if it also stores hour and minute, it is not. Attribute `offset_minutes` is the number of minutes we must add to a partial timestamp to get to the moment. For example, an offset of 480 minutes for a daily time series means that 2003-11-19 is to be interpreted as 2003-11-19 08:00, and an offset of -475 minutes for a monthly time series means that 2003-11 actually means 2003-10-31 18:05.

Using `offset_minutes` for monthly time series may seem far fetched, but actually allows flexibility in cases when it is important. We've seen cases where a large rainfall at 30 March caused a large flow at 1 April, resulting in a high total rainfall for March and high total flow for April; thus, if the daily values were summed the conventional way, monthly rainfalls did not correlate well to monthly flows. The solution is, for example, to sum rainfalls using an offset of -2880 for the resulting monthly timeseries: "March 2003" will then mean "2003-02-27 00:00", that is, the interval that starts at that moment and ends at 2003-03-30 00:00.

Time steps larger than monthly have an additional offset measured in months, which is stored in `tsteps_large`. In an annual time series, an offset of zero months means the year starts at January; an offset of 9 months means the year starts at October. Application software should render 2003 as 2003-2004 if `offset_months` is larger than zero. Negative month offsets are allowed by the database, although I'm not sure they are useful in something, and may be disallowed in the future if we don't find a use for them.

Time stamps are of little use if we don't know the time zone, and the system has the restriction that each time series is in a single time zone. There must be no changing to daylight saving time; in the unhappy event there is a timeseries that changes time zone twice a year, it must be converted before being entered in the database.

Other comments

`munits` holds the units of measurement. `symbol` is the symbol used for that unit in plain Unicode, whereas `fmt_symbol` intends to hold the symbol in some formatting language such as MathML. I haven't decided yet, it's for the future.

`gentities_events` is a kind of log where notable events that happen to a station throughout its life are stored.

`timezones.utc_offset` is the number of minutes east (for positive) or west (for negative) of UTC.

Permissions

The database's security system is used. Users are created using the PostgreSQL's CREATE USER command (and are also added to the `users` table). The usual GRANTs do very little, and there are triggers that implement a kind of row-level security system.

Table `gentities_users` contains the users that have permission to update or delete a given gentity. The database automatically adds a row with the current user whenever a gentity is created (i.e. whenever a row is inserted to `gentities`). Then, whenever that row is to be updated or deleted, or a related row in another table is to be inserted, updated or deleted, a trigger checks gentities_users to see whether the current user is allowed to do so. Thus, if, for example, you try to insert or update or delete a row in `tsteps` with id=42, a trigger will verify that you are among the users that are allowed to alter the gentity to which timeseries 42 refers.

`lentities_users` is the same, but for lentities. In addition, a user is always allowed to alter himself.

The security system is far from perfect, and some aspects have to be taken care of by application code. See the Known problems for more information.

Integrity

Integrity, except for primary key, foreign key, unique, and check constraints, is taken care of by explicitly written triggers that are fired on insert, update and delete to tables only, not views. Views do not make any assumptions about underlying integrity rules. For example, the update rule for `vpeople` pretends that it doesn't know that the integer string identifiers for `last_name` and `first_name` cannot be changed, and it does change them; but if the new value is different from the old value, the underlying update trigger for `people` will deny the change.

Known problems

You can find the list of known problems at http://bugzilla.openmeteo.org/buglist.cgi?product=openmeteo&component=Database&bug_status=NEW&bug_status=ASSIGNED&bug_status=REOPENED

If you are going to do any application development, it is very important that you go through the list carefully.

Design FAQ

Why is the database designed this way?

> I'm not a database expert, and I have little knowledge of relational theory. I have some experience, but the only serious book I've read is "Practical issues in database management" by Fabian Pascal. If you read that book you'll understand some of my choices, namely my avoidance of nullable fields (especially to denote inapplicability rather than missing information), the way I've implemented supertypes and subtypes, and my tendency to

write lots of code so that the database, not the application code, ensures integrity. I believe there is some value in my design, but I'm doubtful about several things and I do want comments. Still, if your only experience is databases with five or so tables in MS Access, I'd prefer that you did some reading before you started a discussion.

Why does the database creation script suck?

The database creation script is huge, boring, buggy, and full of repetitions and redundancies. It's terrible! I hope it is so because of my inexperience. It's the first time I make a database with so many procedures, triggers, and rules, and I didn't know how to do a better job. In addition, because I've been working on that database for several months, I decided that at last I have to stop eternal experimentation and produce some results. So here it is. If you can propose improvements, or, better, implement them, or, better, if you want to rewrite the silly thing from scratch, please go ahead.

Credits

The history of this design is long. It begins in 1992 with the Hydroscope project, which was the code name for the creation of a National Databank for Hydrological and Meteorological Information, undertaken by the National Technical University of Athens (http://www.itia.ntua.gr/e/projinfo/1/). After that there followed several other research projects, most notably the Modernisation of the Supervision and Management of The Water Resource System of Athens (http://www.itia.ntua.gr/e/projinfo/14/). The openmeteo database contains lots of ideas from the database designed for the latter project, which in turn contains lots of ideas from previous projects, notably from the Hydroscope project. The databases for these two projects were designed by Nassos Papakostas, and their designs are available (in Greek) at http://www.itia.ntua.gr/g/docinfo/337/ and http://www.itia.ntua.gr/g/docinfo/414/.

Legal