SokweDB Technical Documentation

Karl O. Pinc, The Meme Factory, Inc. kop@karlpinc.com

2024-03-01 16:12:05 UTC

CONTENTS

1 Technical Documentation **1**
- 1.1 Introduction 1
- 1.2 System Architecture 3
- 1.3 Entity Relationship Diagrams 7
- 1.4 Code Tables 10
- 1.5 Data Tables 13
- 1.6 Data Retrieval Views 19
- 1.7 SQL Functions 20
- 1.8 APPENDIXES 21
- 1.9 Indices and tables 23

CHAPTER

ONE

TECHNICAL DOCUMENTATION

1.1 Introduction

This document describes the design and capabilities of the SokweDB database and related systems.

1.1.1 About This Document

This document describes the SokweDB data management system, it's features and capabilities. This includes the system design principals, a description of the database's tables, and the intended usage of related programs. It does not describe the procedures used to enter data into the system or extract data from it. Nor does it describe the details of how to operate all related programs, or maintain the underlying software.

Conventions

In other words, this document describes the system's capabilities. How to use the system on a day-to-day basis, which features are used in which ways, and similar, are to be found elsewhere.[1]

This section describes the conventions used within this document, which also speaks to the conventions of the overall database design.

There are a number of conventions regarding character case. The database is case insensitive when it comes to the names used to identify content – table names and column names and so forth.[2] Within this document schema names are written with an initial capital letter, table and view names are written in all upper case, and column names are written with initial capital letters.

Unless otherwise noted all columns must contain a value. Note that an empty string, the series of characters consisting of zero characters, is a value. The empty string[3] is a string and so it is reasonable to compare it with other non-empty strings. The empty string is distinct from `NULL`[4], which means "no information", and is used when there is no value/no data at all.

Data consistency is guaranteed only when the guarantee is explicitly mentioned.[5] Care must be taken when updating data unless there is an explicit guarantee of data consistently.

This document uses a particular vocabulary, which informs the table names within the database. The selection of the vocabulary is based on the terminology in use at the JGI and may need some study for those not familiar with it.

[1] The wiki, found at https://sokwe.janegoodall.org/, is a good place to document day-to-day procedures and practices.
[2] To be precise, *unless special steps are taken* the database, and SQL, is case-insensitive when it comes to names.
[3] Written `''` in SQL.
[4] Written `NULL` in SQL.
[5] The notations in the *Entity Relationship Diagrams* are dense in constraints, constraints built into the database, which ensure data integrity.

A Guide for the Reader

This is a reference document, and as such is not expected to be read from front-to-back.

Care must be taken when querying columns which allow `NULL` values. SQL uses a three valued logic, the values being `TRUE`, `FALSE`, and `NULL`. This only comes into play when a `NULL` value is encountered but can be a particularly important factor when a single query relates multiple tables.

1.1.2 System Design

The system design emphasizes first, data integrity, and second, low long-term cost. The database engine chosen, PostgreSQL, supports concurrent multi-user data entry and retrieval, which minimizes the amount of inter-user coordination required and enhances usability.

These design goals lead to the following design elements.

Data integrity is enforced within the database. This allows any program to be used to interact with the database and update database content. Costs are kept down because generic Open Source user interfaces may be used to interact with the database. No matter the tool used, the integrity of the data is maintained.

A web-based user interface, including a wiki which allows rapid web-page development, maximizes availability. The primary method of interacting with the database is SQL, the industry standard relational query language. The amount of SQL which must be learned can be, depending on the amount of development effort spent, reduced to an absolute minimum through the use of *views* – in short, pre-packaged queries.

Data is kept secure though industry-standard practices. These include the encryption of communications, the association of accounts with individuals, the secure authentication required for account access, and the use of in-database access controls to limit the permissions of user accounts. Because individual people are granted direct, but controlled, access to the database itself there is no “middleware” which, when bypassed, has unlimited access to the data.

A minimal number of bespoke programs limits the amount of code development required, and, even more significantly, limits the long-term maintenance costs. The Open Source licensing of the SokweDB system minimizes cost by sharing long-term development of those portions of the system used by more than one institution.

Costs are kept down by minimizing the amount of user-interaction available through bespoke programs. Interacting with a person, particularly reporting errors which arise, requires a lot of programming. Therefore the system is designed around bulk input and output.

Individual accounts are given their own, private, workspaces (*schemas*). This separates private from shared data, which allows for better long-term data maintenance.

About Databases

In PostgreSQL a database is a stand-alone data store. Queries can easily interact with and combine all data kept within a single database. Access to data outside a database, from within the database, is possible but requires additional work that depends upon the data source.

About Tables

Databases store data in tables. Related singleton datum, such as a single name, a single birthdate, a mother, are kept together in a single row of a table constructed to hold this particular kind of data. Data of the same kind kept within a single table are are found in a column of the table. Columns have names, like “name”, “birthdate”, and “mother_id”.

So a table is a grid containing (classically) a single value in each cell of the grid. Each row of the table represents a physical thing, such as a chimpanzee, or an abstract thing, such as the distance to some designated chimpanzee. E.g., a row with the 2 columns: a distance in meters, and the id of the 2nd chimpanzee. Each column of a table is expected to contain the same “kind” of data; a name should not go in the “birthdate”column.

The SokweDB design endeavors to name tables in the plural, as they hold multiple rows. Column names are singular, as each column of each table holds a single value.

About Views

Views appear to be tables but they are not. Views are virtual, when queried they deliver the results of a query run against the database's actual tables. An SQL query can freely intermix the use of tables and views. When setup to do so, changing the data in a view can change data in the database's underlying tables.

Views make it easy to reuse complex or commonly used queries, or portions of queries. They allow a database designed around the capabilities of the computer to be interacted with in a fashion that makes sense to people. Although the views do not appear in the entity relationship diagrams that document the underlying database, and so are omitted from the high level overview these diagrams provide, most users will greatly benefit if they take the time to understand how the views fit into the overall database. Where views exist, most will usually find it easier to work with the views than with the underlying tables.

Views that have the structure, the corresponding columns, of the data after collection in the field and entry into electronic form, are used to upload data into the database. Inserting data into these views distributes the uploaded data into the underlying tables. These sorts of views may or may not be useful when retrieving data from the database for analysis. Investigate to see if some other view or query is better suited rather than automatically using a view created for data upload to do analysis.

About Schemas

Schemas partition databases.[1][2] They work like directories or folders do in filesystems, but can be only one level deep. A schema cannot contain another schema.

Schemas organize database content. One purpose is to allow a user to focus on the content of some schema(s) and ignore what is in other schemas.

1.2 System Architecture

Primary importance is placed on data integrity. The system is optimized for data integrity rather than maximal performance.[1]

The expectation is that the database will be read more often than written and is configured with that in mind.[2]

1.2.1 Databases

SokweDB utilizes at least 3 databases, each for a different purpose. There may also be other databases available.[1]

[1] The term "schema" is overloaded. A separate meaning defines a schema as the tables, columns and relationships between tables that exist within a database. So a schema can denote the design of a particular database.

[2] A PostgreSQL schema can be thought of as a MySQL database, or vice versa.

[1] Among other choices of configuration, SokweDB ensures that concurrent database updates by different users will not lead to data inconsistency by setting the transaction isolation to `serializable`.

[2] In particular, many indexes exist. This speeds query results but slows database writes.

[1] Particularly during periods of heavy software development, there may be a separate database dedicated to each developer.

sokwedb

The `sokwedb` database contains the final, "official", data. All research takes place in this database.

sokwedb_test

The `sokwedb_test` database is used, by everyone, for testing. Typically, this database contains a copy of the `sokwedb` data. It may be desirable to upload new data into the `sokwedb_test` database before uploading into the `sokwedb` database. This allows the data to be cleaned and examinations made before upload into production.

sokwedb_dev

The `sokwedb_dev` database is for software and database development. It is primarily used by the system's developers to try new features. After coordinating with the developers, it could used by anyone to test something that seems particularly dangerous and might interfere with normal operations if tested in the `sokwedb_test` database.

1.2.2 Special Data Values

As much as possible SokweDB utilizes a controlled vocabulary within the system's data store. To provide the system's users[1] with control over the codes used, this vocabulary may be tailored by adding or deleting codes to or from the tables which define the system's vocabulary.

At times, SokweDB recognizes that particular codes have special meanings, for example, the *BIOGRAPHY_DATA* table's `F` (female) Sex code. The meaning of these codes is fixed into the logic of the system. As examples, an individual must be female to be allowed to have a menstruation, or, the individual must be in the community to be sighted in the community. Some of these codes, like sex, are not defined in tables, they are hardcoded into the system. Others are defined in support or other tables. Because these codes have intrinsic meaning, they cannot be removed from the SokweDB system nor should their presence in the data be used to code a different meaning from that which the code presently has. For example, the meaning of *DEPARTTYPES* code value `O` (alive[2]) should not be changed to mean "death due to meteorite impact" because the system's programs would then allow individuals to have sexual cycles after death. Each of the "special" values that the system requires retain particular meaning is listed in the Special Values section of the code table's documentation. For further information on the meaning of the "special" values, see the description of the table(s) that contain the code values. Should the meaning of one of these "special" values need to be changed, the logic in the SokweDB programs should be adjusted to reflect the change.

SokweDB prevents ordinary users from altering rows that give meaning to special values in an attempt to prevent mis-configuration of the system. Only users with permissions to modify a table's triggers may alter the table's special values.[3] This is not a panacea. To return to the example above, not only does the system expect a *DEPARTTYPES* code of `O` to mean alive, it also expects `O` to be the only code in *DEPARTTYPES* that means alive. If another *DEPARTTYPES* code is created to indicate a more specific sort of "alive-ness", unless re-programmed the system will consider all individuals given that code to be dead, not alive. A careful review of the documentation should be undertaken before modifying the content of tables that instantiate special values.

[1] As opposed to the system's programmers.

[2] Specifically, "still alive and present as of the last census date".

[3] Rather than create another role just to control the alteration of special values the choice was made to use PostgreSQL's `TRIGGER` privilege. This allows superusers (or the somewhat less privileged) the necessary access. This conveniently separates regular users from those who can do more.

1.2.3 Dates and Times

In SQL dates and times are written as strings, so are enclosed in single quotes (or dollar quotes). But the system must be told that the data is a date, or a time, etc.[1] The easy way to type this is to follow the string with two colons and then the name of the appropriate data type. This is best illustrated with an example:

```
SELECT '1707-05-23'::DATE;
```

Input and Output Representations

Date values are always output in YYYY-MM-DD format.[2] This is unambiguous and more universal than most date representations.

Date values may be input in a wide variety of formats. Ideally, they would be input as YYYY-MM-DD but when this is not the case the system first attempts to recognize dates as if they were written in MM-DD-YYYY format.

For further information on date and time representation see the PostgreSQL documentation, either that on dates and times or the details of date/time interpretation.

Time Zones

SokweDB contains few, if any, time-zone aware columns. For this reason, and reasons given below, most users will not need to concern themselves with time zones.

Date + time combinations, called timestamps, may or may not be time zone aware. This is also true of plain, 24-hour, time values. Time zone aware values display differently depending on the time zone in which they are viewed – or at least they can display differently. A time zone aware time value which displays as 10:00AM in the US/Eastern time zone would display as 9:00AM in the US/Central time zone.

Dates and times *without* a time zone, most time-related data recorded in SokweDB, are as-of the time recorded in the field. So in Gombe time, and the time values won't change no matter where viewed.

Some other dates and times, perhaps those involving administrative actions like, perhaps, the automatically recorded time of database updates, may be time zone aware.

By default, time-zone aware data is input and output in the `UCT` time zone. If you wish to have time-related data be input and output in a different time zone you must tell the server which time zone you are in.[3] This does not happen automatically. Further, the change to your time zone only lasts for the duration of your connection to the database. Practically speaking, this usually, depending on the tool you use to access the database, means that you must change your time zone every time you submit SQL statements to the server.

To sum up, most of the time-related values you work with will be in Gombe time. The rest are in `UCT` unless you put some work into changing your time zone.

1.2.4 Users and Database Permissions

Each person should have their own login/username, which should not be shared.

The database associates each login with specific permissions to objects (tables, etc.) within the database.

To access the data in the database permission must be granted. This is done per user login.

There are 2 ordinary levels of permission. Their names are:

reader
: Permission to query database content.

[1] The string is said to be *cast* to the desired data type.
[2] This is the ISO 8601 format.
[3] E.g. `SET TIME ZONE 'US/Mountain';`

writer

All the permissions of `reader` plus permission to alter the content of the database.

Ordinary permissions are database *roles*. They can be granted with SQL, e.g.:

```
GRANT reader TO someuser;
```

Or grants can be made through some other mechanism.

The Administrator Permission Level

The `admin` permission level has maximal permissions. It is used to create *superusers*.

Permissions are implemented as PostgreSQL roles. It is the `admin` role that owns all the SokweDB database objects, the tables, views, etc.

Superusers (aka Administrators)

Superusers have permission to do anything with a database, create and destroy tables, create and destroy user logins, etc.[1] Only a few people are expected to have superuser privileges.

Those people with superuser privileges will typically have 2 logins, one ordinary login and a second login with superuser privileges. The superuser login should be used only when necessary, as when a new person is given access to SokweDB and a new database login must be created. Ordinary interactions with the database, data entry, data retrieval, etc., should be done with a non-superuser login.

Developers

Developers, the users who maintain the database structure, etc., must be *superusers*.

1.2.5 Schemas

Each SokweDB database contains a number of *schemas*. Some of these schemas will not be documented herein; it is likely that some schemas will be created to hold shared data, data not part of SokweDB itself but related to it.

The sokwedb schema

The `sokwedb` schema contains the data collected in the field. It is the primary schema of interest to the researcher.

The codes schema

The `codes` schema contains those tables which control the data vocabulary defining the codes able to be recorded in the database. Because the codes defined in this schema are often used and well-known the schema's tables are not often of interest. This schema exists so that the `sokwedb` schema is not cluttered with un-interesting tables.

Most of the tables in this schema contain one row per defined code. The codes are usually kept in a column that has the name of table, but a name which is singular instead of plural. There is also a `Description` column, which describes the coded value.

A few of the tables in this schema are more complex, and contain more than 2 columns. These are often tables which contain 'meta information' involving the mechanics of the data collection process. Things like lists of researchers, observers, or equipment used in the data collection process.

[1] The Azure cloud platform does not allow logins (aka roles) to have the `SUPERUSER` *role attribute*. Instead, the `CREATEDB` and `CREATEROLE` attributes are most that can be given. The SokweDB `admin` group (aka role) has `CREATEDB` and `CREATEROLE` privileges. This is enough that there is no need for the actual `SUPERUSER` attribute.

Membership in the `admin` role and having the role attributes `CREATEROLE` and `CREATEDB` is what grants a role (login) superuser privileges.

The upload schema

The `upload` schema contains the views used during the data upload process. These are, usually, of interest only to those who upload data into the database.

The lib schema

The `lib` schema (short for "library") contains things used by SokweDB's internal mechanisms. The end-user should not need to be concerned with its content.

The per-user private schemas

Individuals are given their own schemas in which they can do whatever they wish. So each regular account/login/username has an associated schema with a name the same as that of the account.

> **Caution:** It is usually bad practice to grant another person access to something in a private schema. It is often better to create another, shared, schema. In this way individuals, and their accounts and private schemas, can come and go without affecting the work of the institution.

Because of the schema search order the schema name must be used to qualify anything created in the user's schema. E.g., to create the table `foo` in the user `mylogin`'s schema:

```
CREATE TABLE mylogin.foo (somecolumn INTEGER);
```

1.3 Entity Relationship Diagrams

Entity Relationship Diagrams, or ER Diagrams, are graphic pictures of how rows in the database's tables relate to other rows. They are dense in information about what data exists and in what tables it can be found.

Most tables have have an id, or key, column that contains a number unique to that row within its table. The id can be used, in perpetuity, to refer to its related row and distinguish it from all the other rows of the table. Ids are arbitrary, although for convenience they are often sequentially generated integers. The name of the column holding the id value is not always Id, although it sometimes is.

A relationship is established between the rows of two tables when an id value from one table appears as data in the other. The relationship notion is made most clear by way of diagrams and examples. The relationship concept is at the heart of relational databases and, while the underlying idea is rather simple, it took many years to develop relational database concepts so don't expect a full understanding immediately.

When an id value of a row in one table appears as data in a second table, the data in the second table can be used to retrieve the identified row from the first table.[1] When an id value of a row in the first table appears as data only once in the second table, the two tables are said to have a one-to-one relationship. One row in the first table relates to one (or possibly zero) row(s) in the second table. When a row's id value can appear in more than one row of a second table, the two tables are said to have a one-to-many relationship. One row of the first table can be related to many rows in the second table. One-to-many relationships are more common than one-to-one relationships. In the ER diagrams each table (entity) is a box, and each box contains a list of the table's columns. The lines between the boxes represent the relationships between the tables.

For example, individuals can transfer between communities zero or more times. A single individual is one kind of entity recorded in the database, one transfer of a single individual between communities is a second kind of entity. Each individual is represented in the database as a single row in a table. Each transfer is likewise represented by a single row in a different table.

[1] And the reverse is true. The id of a row in the first table can be used to find the row in the second table that holds it.

The ER diagram of this shows that a row representing a one individual can be "connected to" zero or more rows recording community transfers. The diagram also shows that a transfer **must** be "connected to" exactly one individual, the individual who is changing communities. A transfer should not be able to exist unless the individual transferring also exists.

1.3.1 Key

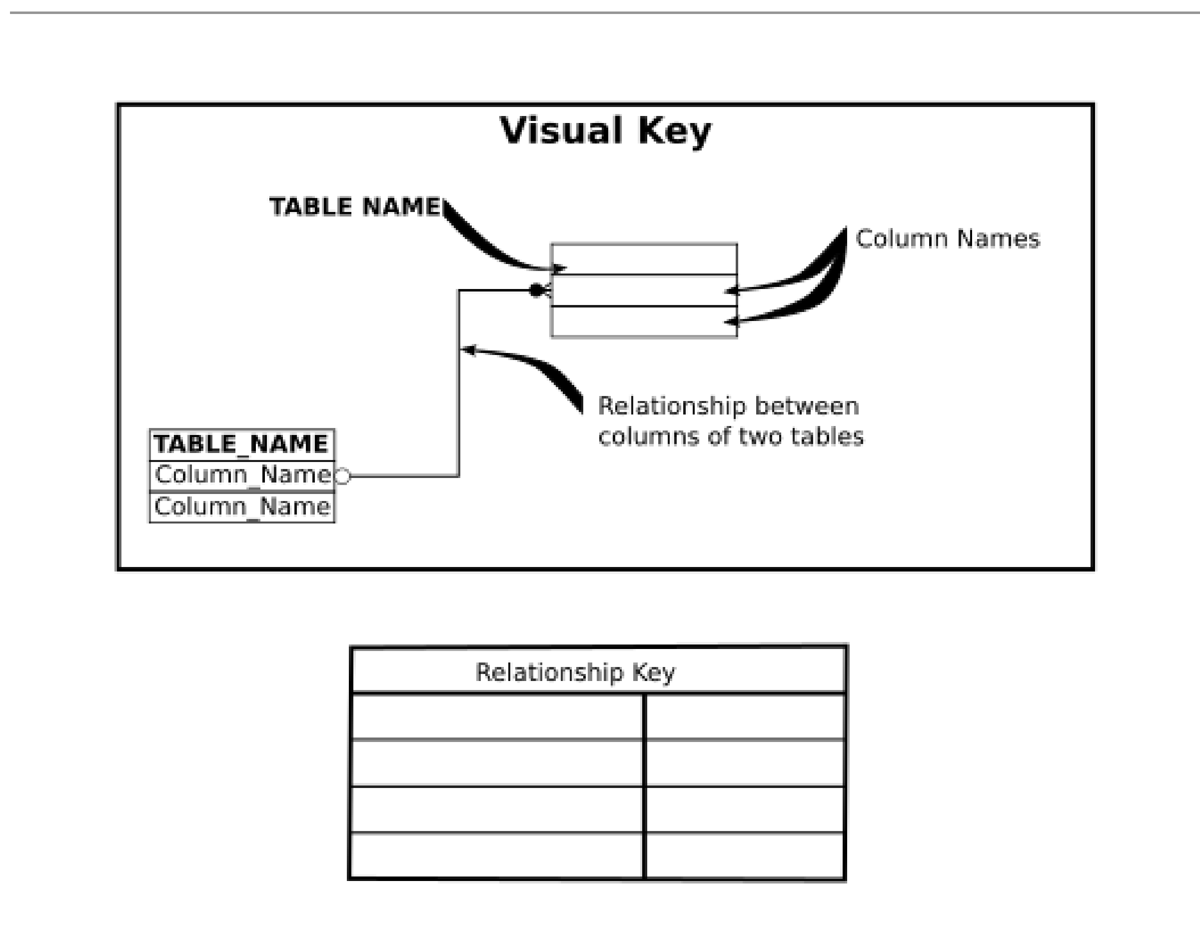

Fig. 1: Key to the Entity Relationship Diagrams

1.3.2 Demography

Contents

- *Code Tables*
  - *COMM_IDS (Community IDentifiers)*
  - *COMM_MEMBS_SOURCES (COMMunity MEMBershipS SOURCES)*
  - *DEPARTTYPES (community Departure reasons)*
  - *ENTRYTYPES (community Entry reasons)*
  - *PEOPLE*

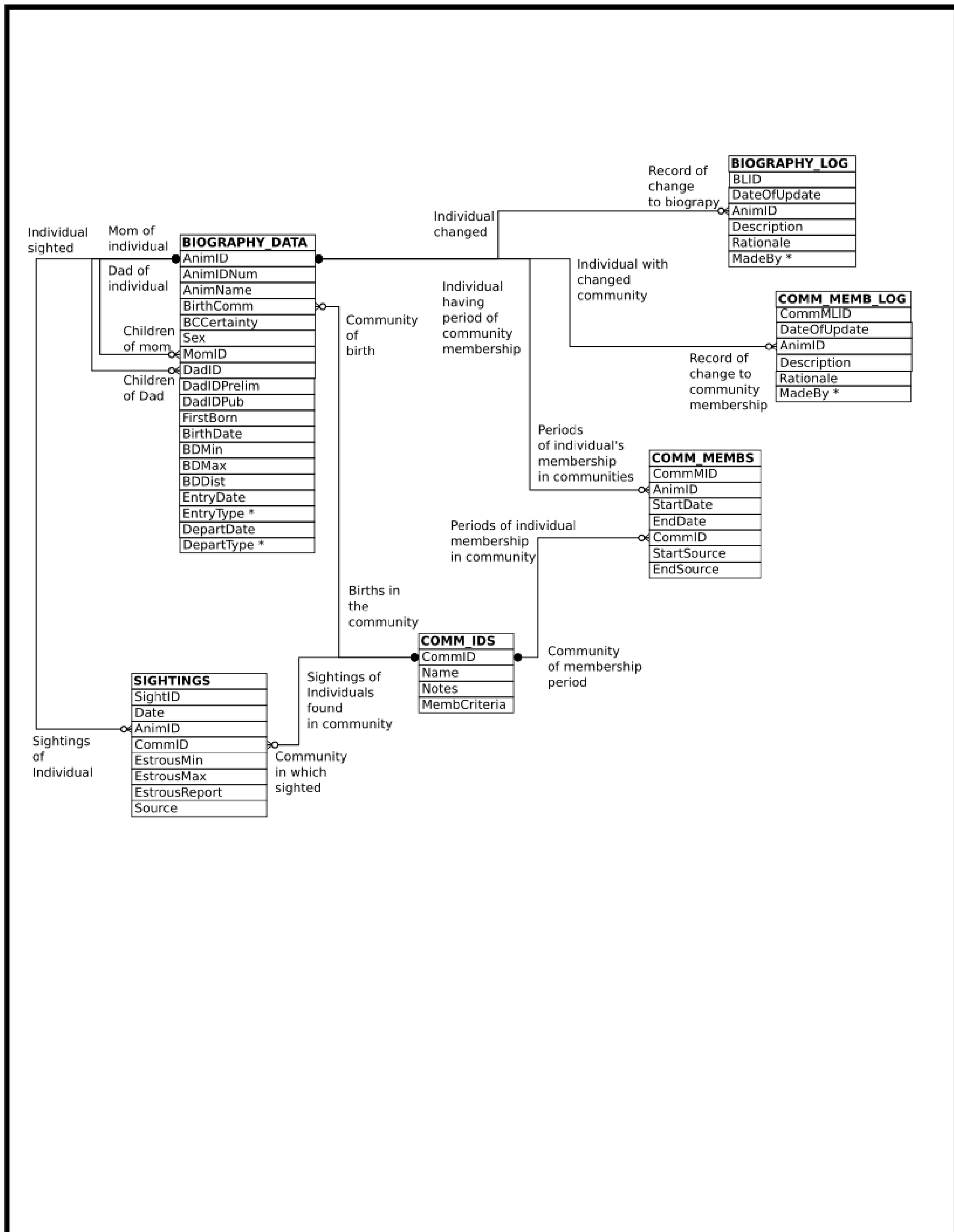

Fig. 2: Demography

1.4 Code Tables

1.4.1 COMM_IDS (Community IDentifiers)

Each row represents a community. There is one row for every chimpanzee community on which any information has been recorded, whether presently under-study or not. This includes the special community named `Unknown`, which is used when an individual cannot be assigned to a community.

Special Values

The *CommID* value of `Unknown` is special.

Column Descriptions

CommID (Community IDentifier)

A short sequence of characters which identify a community. Each value stored in this column must be unique. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters. This column may not contain whitespace characters.

Name

The name of the community. Each value stored in this column must be unique. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character. This column may *not* be `NULL`.

Notes

Notes on the community, how membership is assigned and other relevant information. This column may be empty text. It need not contain characters, but it may not contain only whitespace characters. This column may *not* be `NULL`.

MembCriteria (Membership Criteria)

A description of the requirements to be a member of the community. This column may be empty text. It need not contain characters, but it may not contain only whitespace characters. This column may *not* be `NULL`.

1.4.2 COMM_MEMBS_SOURCES (COMMunity MEMBershipS SOURCES)

Contains one row for each data source from which community membership information is derived.

Special Values

None.

Column Descriptions

CommMembsSource

A somewhat abbreviated description which identifies a data source from which community membership can be derived. Each value stored in this column must be unique. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters. This column may not contain whitespace characters.

Description

A longer description of the CommMembsSource identifier. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character.

1.4.3 DEPARTTYPES (community Departure reasons)

Contains one row for each way a chimpanzee can leave a community, plus a special row with a code of `Unknown` to indicates the individual was in the community when last observed.

Special Values

The *DepartType* value of `Unknown` is special. It indicates the individual is still alive and in the community. Only individuals that are in a community may be observed in the community.

Column Descriptions

DepartType

A one character code identifying a way an individual can leave a community. Each value stored in this column must be unique. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters. This column may not contain whitespace characters. This column may not contain lower case letters.

Description

A description of the code. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character.

1.4.4 ENTRYTYPES (community Entry reasons)

Contains one row for each way a chimpanzee can enter a community.

Special Values

None.

Column Descriptions

EntryType

A one character code identifying a way an individual can enter a community. Each value stored in this column must be unique. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters. This column may not contain whitespace characters. This column may not contain lower case letters.

Description

A description of the code. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character. This column may *not* be `NULL`.

1.4.5 PEOPLE

Contains one row for each person involved in data collection. Anyone who's identity is recorded in SokweDB must have a row representing them in this table.

Note that, for reasons of simplicity and performance, SokweDB accepts only the ASCII character set.[1] (Those characters found on a standard U.S. keyboard.)

Special Values

None.

Column Descriptions

Person

A short character string used to identify the person. Each value stored in this column must be unique. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters. This column may not contain whitespace characters.

Name

The name of the person. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character. This column may *not* be `NULL`.

[1] The database could accept the full-set of Unicode characters, providing glyphs for all languages and cultures as well as a full set of emoji. And if collation (sorting) was configured so as *not* to support language and cultural conventions then there would be no performance impact.

But this would allow, say, chimpanzees to be named with Chinese ideographic glyphs, allowing arbitrary glyphs to be used anywhere text might appear. Because this is not desired we would need to institute additional controls to keep the data clean. It is simpler to live with the ASCII character set when it comes to recording people's names.

Description

A description of the person, should such be necessary to distinguish them from other people with the same or similar name. This column may be empty text. It need not contain characters, but it may not contain only whitespace characters. This column may *not* be NULL.

Active

A boolean value. Whether or not the row can be used in new data. When this column is FALSE the *Name* value cannot be used in rows that are newly inserted into the database. Further, when rows are updated an existing value cannot be set to the inactive *Name*. This column may *not* be NULL.

Unlike most other data validation checks, *Active* can be changed from TRUE to FALSE even though the *Person* value is used elsewhere in the database. This allows time for existing data to be cleaned while preventing undesirable values from appearing in new data.

1.5 Data Tables

1.5.1 BIOGRAPHY_DATA

Each row represents a chimpanzee. This table contains one row for each chimpanzee on which data has ever been recorded (in SokweDB), and an additional row for UNK a generic value used when a chimpanzee is unrecognized. BIOGRAPHY_DATA contains the basic demographic data of individual chimpanzees.

Note: The *BIOGRAPHY* view may be preferred to using the *BIOGRAPHY_DATA* table.

A mother must be female; the *Sex* must be F (female) of the *BIOGRAPHY_DATA* row identified by an offspring's *MomID*.

A father must be male; the *Sex* must be M (male) of the *BIOGRAPHY_DATA* row identified by an offspring's *DadID*.

A female cannot be too young when giving birth. The difference between the mother's maximum birthdate, the *BDMin* of the *BIOGRAPHY_DATA* row identified by an offspring's *MomID*, and the offspring's minimum birthdate, the *BDMin* of the offspring, cannot be less than 8 years.

A male cannot be too young when becoming a parent. The difference between the father's maximum birthdate, the *BDMin* of the *BIOGRAPHY_DATA* row identified by an offspring's *DadID*, and the offspring's minimum birthdate, the *BDMin* of the offspring, cannot be less than 10 years.

When the individual is not the first recorded offspring of their mother, based on the *BirthDate* of all recorded maternal siblings, the *FirstBorn* value must be N (not first born).

The date the individual entered the study (*EntryDate*) may not be before the individual's birth date (*BirthDate*).

The date the individual left the study (*DepartDate*) may not be before the date the individual entered the study (*EntryDate*).

The maximum age of an individual, the time span between the individual's earliest possible birth date (*BDMin*) and their *DepartDate*, may not be more than 70 years.

DadPrelim must be NULL when *DadID* is NULL. Otherwise *DadPrelim* must not be NULL. *DadIDPub* must be NULL when *DadID* is NULL. Otherwise *DadIDPub* must not be NULL.

The row defining the unknown individual, the BIOGRAPHY_DATA row having an *AnimID* value of UNK, is *special* and cannot be altered or deleted by ordinary user accounts.

AnimID (Animal IDentifier)

A short sequence of characters which uniquely identify the chimpanzee. Each value stored in this column must be unique. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters. This column may not contain whitespace characters. The value of this column cannot be changed.

AnimIDNum (Animal IDentifier Number)

A unique positive integer used to identify the chimpanzee in SIV papers. Each value stored in this column must be unique. This column may be `NULL`.

These are the former Ch numbers from Beatrice Hahn's lab.

AnimName (Animal Name)

The name of the chimpanzee. Each value stored in this column must be unique. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character. This column may *not* be `NULL`.

BirthComm (Birth Community)

The *COMM_IDS*.*CommID* of the community in which the individual was born. This column may be `NULL` when the birth community is unknown.

BCCertainty (Birth Community Certainty)

A code indicating the certainty of the *BirthComm*, the certainty of the birth community. Only 2 values are allowed, `C` when the birth community is certain and `U` when the birth community is uncertain. This column may *not* be `NULL`.

Sex

A code indicating the sex of the individual. Only 3 values are allowed: `M` for males, `F` for females, and `U` when the sex is unknown. This column may *not* be `NULL`.

MomID (Mother's AnimID)

The *AnimID* of the individual's mother, when known. This column may be `NULL`.

DadID (Father's AnimID)

The *AnimID* of the individual's father, or `NULL` when not known.

This column may be `NULL`.

DadPrelim (Is Paternity Preliminary?)

A boolean value. When `TRUE`, the paternity assignment is preliminary. This column may be `NULL`.

DadIDPub (Publication of Paternity)

Citation of the publication where paternity was declared, or 'Unknown' when paternity has not yet been published. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character. This column may be `NULL`.

FirstBorn

A code indicating whether the individual is the mother's first born. One of 3 values are allowed: `Y` means known to be the first born, `N` means known not to be the first born, and `U` means that the firstborn status is unknown. This column may *not* be `NULL`.

BirthDate

The individual's (often estimated) birth date. This column may *not* be `NULL`. This value may not be before `1900-01-01` or after the current date.

BDMin (Minimum Birth Date)

The earliest possible birthdate. If born into the study, this is the last date prior to birth that the mother was seen without the infant. This column may *not* be `NULL`. This value may not be before `1900-01-01` or after the current date.

BDMin (Maximum Birth Date)

The latest possible birthdate. If born into the study, this is date of the first sighting of the infant. This column may *not* be `NULL`. This value may not be before `1900-01-01` or after the current date.

BDDist (Birth Date Distribution)

The probability distribution of the likelihood of birth between *BDMin* and *BDMax*. Only one of 2 values are allowed, `N` when it is most likely that the actual birthdate is closer to *BirthDate* than to *BDMin* or *BDMax*, or `U` when any birthdate between *BDMin* and *BDMax* is equally likely. This column may *not* be `NULL`.

EntryDate

The date the individual entered the study; the date first seen. This column may *not* be `NULL`.

EntryType

An *ENTRYTYPES.EntryType* code indicating how the individual entered the study. This column may *not* be `NULL`.

DepartDate

The date the individual was last in the study, or the date the individual was last seen. For living individuals this is the date of last census. This column may *not* be NULL.

DepartType

A *DEPARTTYPES.DepartType* code indicating that the individual is still alive and under study, or how the individual left the study. The *DEPARTTYPES.DepartType* 0 value is special. It indicates the individual is still alive and in the community. This column may *not* be NULL.

1.5.2 BIOGRAPHY_LOG

Each row documents a change made to a *BIOGRAPHY_DATA* row. Changes have been logged since 2013-07-01.

BLID (Biography Log IDentifier)

A unique, automatically generated, positive integer which serves to identify the row. The value of this column cannot be changed. This column may *not* be NULL.

DateOfUpdate

The date on which the update was made. This date cannot be before 2013-07-01 and cannot be after the current date.

AnimID (Animal IDentifier)

The *BIOGRAPHY_DATA.AnimID* which identifies the chimpanzee who's information was updated. This column may *not* be NULL.

Description

A description of the changes made. This column may not be empty text, its textual values must contain characters. This column may *not* be NULL.

Rationale

The rationale for the change to the *BIOGRAPHY_DATA* data. This column may not be empty text, its textual values must contain characters. This column may *not* be NULL.

MadeBy

The *PEOPLE.Person* designating the researcher who made the update. This column may *not* be NULL.

1.5.3 COMM_MEMBS

Each row represents an un-interrupted series of days during which the given chimpanzee is a member of the given community. The unit of time is the day; it is not possible to place any given chimpanzee in more than one community within a single day. Leaving a community, whether to join another or not, ends this period of community membership. Another row in `COMM_MEMBS` is required to record a newer period of community membership, whether in the same or a different community.

An individual may not be recorded in more than one community on any given day, although there may be days during which the individual is not placed in any community. Further, an individual may not be placed in the same community, by use of two `COMM_MEMBS` rows, on the same day. There can be no "overlap" of `COMM_MEMBS` rows. The *StartDate* to *EndDate* intervals, of all the `COMM_MEMBS` rows with a given *AnimID*, may not overlap.

Two `COMM_MEMBS` rows may not be used to place a single individual in the same community on successive days. Instead, combine the two `COMM_MEMBS` rows into one. The *StartDate* of an individual with a given *CommID* may not be the day after the *EndDate* of a `COMM_MEMBS` row having the same *AnimID* value.

An individual may not be placed in a community unless that individual is under study; the *StartDate* may not be before the individual's *BIOGRAPHY_DATA*.*EntryDate* and the *EndDate* may not be after the individual's *BIOGRAPHY_DATA*.*DepartDate*.

The *StartDate* must not be after the *EndDate*.

CommMID (Community Memberships IDentifier)

A unique, automatically generated, positive integer which serves to identify the row. The value of this column cannot be changed. This column may *not* be `NULL`.

AnimID (Animal IDentifier)

The *BIOGRAPHY_DATA*.*AnimID* which identifies the chimpanzee who's community membership is recorded. This column may *not* be `NULL`.

StartDate

The date on which the individual joined the community; the start date, inclusive, of the interval of continuous membership. This column may *not* be `NULL`.

EndDate

The last date on which the individual was a community member; the end date, inclusive, of the interval of continuous membership This column may *not* be `NULL`.

CommID (Community IDentifier)

The *COMM_IDS*.*CommID* identifying the community in which the row records membership. This column may *not* be `NULL`.

StartSource

The *COMM_MEMBS_SOURCES.CommMembsSource* value which identifies the data source used to determine the *StartDate*. This column may *not* be `NULL`.

EndSource

The *COMM_MEMBS_SOURCES.CommMembsSource* value which identifies the data source used to determine the *EndDate*. This column may *not* be `NULL`.

1.5.4 COMM_MEMB_LOG

Each row is a log entry describing a change in an individual chimpanzee's community membership. All community membership changes starting from `2013-12-01` are recorded here.

This is meta-information which records history concerning the state of the database.

CommMLID (Community Membership Log IDentifier)

A unique, automatically generated, positive integer which serves to identify the row. The value of this column cannot be changed. This column may *not* be `NULL`.

DateOfUpdate

The date the database was updated to reflect the change in community membership. This value cannot be before `2013-12-01` and cannot be after the current date. This column may *not* be `NULL`.

AnimID (Animal IDentifier)

The *BIOGRAPHY_DATA.AnimID* which identifies the chimpanzee who's community membership was updated. This column may *not* be `NULL`.

Description

A description of the change made to community membership. This column may *not* be `NULL`. This column may not be empty text, its textual values must contain characters and must contain at least one non-whitespace character.

Rationale

The rationale for the change in community membership. This column may be empty text. It need not contain characters, but it may not contain only whitespace characters. This column may *not* be `NULL`.

MadeBy

The *PEOPLE.Person* designating the researcher who determined that the community membership should be changed. This column may *not* be `NULL`.

1.6 Data Retrieval Views

The views appearing in this section exist for convenience in querying. Some exist to make the data look more like the "old" data, as it appeared in the old MS Access database. Others reproduce common query patterns, eliminiating the need to connect (join) multiple tables.

View	One row for each	Purpose	Tables/Views used
BIOGRAPHY	*BIOGRAPHY_DATA* row	Reproduce "old" data	*BIOGRAPHY_DATA*

1.6.1 BIOGRAPHY

Each row represents a chimpanzee and is a transformation of the corresponding *BIOGRAPHY_DATA* row, making the data more like the traditional format and therefore, in one sense, easier to work with. This view contains one row for each chimpanzee on which data has ever been recorded (in SokweDB), and an additional row for `UNK` a generic value used when a chimpanzee is unrecognized. BIOGRAPHY contains the basic demographic data of individual chimpanzees.

Definition

```
CREATE OR REPLACE VIEW biography (
   animid
  ,animidnum
  ,animname
  ,birthcomm
  ,bccertainty
  ,sex
  ,momid
  ,dadid
  ,dadidpub
  ,firstborn
  ,birthdate
  ,bdmin
  ,bdmax
  ,bddist
  ,entrydate
  ,entrytype
  ,departdate
  ,departtype)
  AS
    SELECT
        biography_data.animid
       ,biography_data.animidnum
       ,biography_data.animname
       ,biography_data.birthcomm
       ,biography_data.bccertainty
       ,biography_data.sex
       ,biography_data.momid
       ,CASE
          WHEN biography_data.dadprelim
            THEN biography_data.dadid || '_prelim'
          ELSE biography_data.dadid
       END CASE
       ,biography_data.dadidpub
```

(continues on next page)

(continued from previous page)

```
       ,biography_data.firstborn
       ,biography_data.birthdate
       ,biography_data.bdmin
       ,biography_data.bdmax
       ,biography_data.bddist
       ,biography_data.entrydate
       ,biography_data.entrytype
       ,biography_data.departdate
       ,biography_data.departtype
     FROM biography_data;
```

Columns in the BIOGRAPHY View

Table 1: The Columns in BIOGRAPHY

Column	From	Description
AnimID	*BIOGRAPHY_DATA.AnimID*	Animal IDentifier
AnimIDNum	*BIOGRAPHY_DATA.AnimIDNum*	Animal IDentifier Number
AnimName	*BIOGRAPHY_DATA.AnimName*	Animal Name
BirthComm	*BIOGRAPHY_DATA.BirthComm*	Birth Community
BCCertainty	*BIOGRAPHY_DATA.BCCertainty*	Certainty of *BirthComm*
Sex	*BIOGRAPHY_DATA.Sex*	Individual's Sex
MomID	*BIOGRAPHY_DATA.MomID*	*AnimID* of the individual's mother
DadID	*BIOGRAPHY_DATA.DadID* *BIOGRAPHY_DATA.DadPrelim*	*AnimID* of the individual's father, suffixed with `_prelim` if *DadPrelim* is `TRUE`
DadIDPub	*BIOGRAPHY_DATA.DadIDPub*	Publication of Paternity citation
FirstBorn	*BIOGRAPHY_DATA.FirstBorn*	First born status code
BirthDate	*BIOGRAPHY_DATA.BirthDate*	Birth Date
BDMin	*BIOGRAPHY_DATA.BDMin*	Minimum Birth Date
BDMax	*BIOGRAPHY_DATA.BDMax*	Maximum Birth Date
BDDist	*BIOGRAPHY_DATA.BDDist*	Birth Date Distribution
EntryDate	*BIOGRAPHY_DATA.EntryDate*	Date of study Entry
EntryType	*BIOGRAPHY_DATA.EntryType*	Entry status code
DepartDate	*BIOGRAPHY_DATA.DepartDate*	Date last seen
DepartType	*BIOGRAPHY_DATA.DepartType*	Depart date status code

Operations Allowed

None.

1.7 SQL Functions

The functions documented here may be useful when writing SQL.

The database contains a large number of functions but only those documented below are expected to be used directly by the database users.

1.7.1 julian() – Convert a date to an integer which counts up by day

Synopsis

```
julian(date DATE) INT
julian(date TIMESTAMP) INT
```

Input

date

A `DATE` or a `TIMESTAMP`.

Description

Convert a date or a timestamp to a julian date. Supply this function with a `DATE` (or a `TIMESTAMP`) and it returns the integer that represents the given date as the number of days since a particular reference date. This number is known as the Julian date representation of the given date. (Day number 2,361,222 is September 14, 1752.) Legal values for the date are between September 14, 1752 and December 31, 9999, inclusive.

1.7.2 julian_to() – Convert an integer which counts days to a date

Synopsis

```
julian_to(julian INT) DATE
```

Input

julian

An integer representing a julian date.

Description

Converts a julian date value to a regular date value. This function reverses the *julian()* function.

1.8 APPENDIXES

Contents

- *APPENDIXES*
  - *Technologies Used*

1.8.1 Technologies Used

These technologies are used by SokweDB. The desire is to keep the number of technologies to a minimum to keep development simple.

Operating System Components and Services

- The Microsoft Azure cloud
- The Linux Operating System Kernel
- The Ubuntu Linux Distribution
- The Internet/Web/Web Browsers and related technology
- The PostgreSQL database engine
- The Postfix Mail Transfer Agent
- The Nginx webserver
- The gitweb source code repository web interface
- The Letsencrypt.org security certificate toolset and services
- The MediaWiki wiki engine
- The php-fpm PHP interpreter

Although the operating system level components have been chosen with care, they are more-or-less interchangeable with similar, stock, components. Each may be swapped out when this is found convenient. The exception is the PostgreSQL database engine. SokweDB depends upon specific PostgreSQL characteristics and features.

Development Tools

- The SQL database query and construction language
- The PL/pgSQL PostgreSQL database extension language
- The Python3 programming language
- The Pyramid web development framework
- The M4 macro programming language
- The PHP programming language (deprecated)
- The git revision control system
- The bash shell scripting language
- The make build system tool

Documentation Tools

- The ReStructured Text (RST) markup lanugage
- The Sphinx RST processor
- The Inkscape SVG vector graphics editor

1.9 Indices and tables

- genindex
- search