Oliphant Documentation

Release 0.1

Dave Jones

Mar 05, 2018

Contents

This package provides a set of extensions for PostgreSQL 9.x (or above). The following extensions are currently included:

- *assert* - A set of routines for asserting the truth of various things (equality or inequality of values, existence of tables and columns, etc.); useful for building test suites.
- *auth* - A set of routines for bulk manipulation of authorizations including copying all authorizations from one user to another, and transferring all authorizations from one table to another.
- *history* - A set of routines for simplifying the creation of temporal tables; tables that track the state of another table via a series of triggers on the base table.
- *merge* - Routines for bulk transfer of data between similarly structured tables or views.

CHAPTER 1

Links

- The code is licensed under the MIT license
- The source code can be obtained from GitHub, which also hosts the bug tracker
- The documentation (which includes installation and usage examples) can be read on ReadTheDocs

CHAPTER 2

Table of Contents

2.1 Server installation

The following sections detail adding the Oliphant extensions to your PostgreSQL server installation. Please select an installation method which meets your needs.

2.1.1 Ubuntu installation

The following assumes you already have a PostgreSQL server installed on your Ubuntu machine. To install the pre-requisites, clone the Oliphant repository and install the extensions within your PostgreSQL server:

```
$ sudo apt-get install git make postgresql-server-dev-all
$ git clone https://github.com/waveform-computing/oliphant.git
$ cd oliphant
$ sudo make install
```

2.1.2 Microsoft Windows installation

Pull requests for instructions gratefully received.

2.1.3 Mac OS X installation

Pull requests for instructions gratefully received.

2.1.4 Development installation

If you wish to develop oliphant itself, it is easiest to use the `develop` target of the makefile. This does something similar to `install`, but creates symlinks within your PostgreSQL extension directory which makes it a bit easier to hack on the code. The following assumes Ubuntu:

```
$ sudo apt-get install git make postgresql-server-dev-all
$ git clone https://github.com/waveform-computing/oliphant.git
$ cd oliphant
$ sudo make develop
```

2.2 Database installation

Once the Oliphant extensions have been added to your PostgreSQL installation you can install them in the database(s) of your choice. To do this manually simply use the CREATE EXTENSION with the extensions, e.g.:

```
db=# CREATE EXTENSION auth;
CREATE EXTENSION
db=# CREATE EXTENSION history;
CREATE EXTENSION
```

Alternatively, you can use the `installdb` target of the makefile. This will install all extensions available in Oliphant into the target database. This defaults to the same database as your username but you can edit the makefile to change this:

```
$ make installdb
for m in assert auth history merge; do \
                psql -d db -c "CREATE EXTENSION $m"; \
        done
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
```

2.3 Test suite execution

With the extensions installed in a database (see *Database installation* above), simply make the `test` target. All tests are echoed to stdout and any test that fails causes the script to immediately stop allowing the developer to diagnose the failure.

2.4 Quick start

This chapter provides a quick walk through of the major functionality available in the various extensions contained in Oliphant. Each section deals with the functions of an individual extension.

2.4.1 `assert` extension

The functions of the *`assert`* extension are primarily intended for the construction of test suites. Each performs a relatively simple, obvious function, raising an error in the case of failure. For example, to ensure that a particular table exists, use *`assert_table_exists()`*:

```
CREATE TABLE foo (i integer NOT NULL, PRIMARY KEY);

SELECT assert_table_exists('foo');
```

Or to ensure that some value equals another value, use *`assert_equals()`* (this has overridden variants for all common types):

```
INSERT INTO foo VALUES (1), (2), (3), (4);

SELECT assert_equals(10, (SELECT SUM(i) FROM foo));
```

Likewise, various similar functions are provided:

- *assert_not_equals()*
- *assert_is_null()*
- *assert_is_not_null()*
- *assert_column_exists()*
- *assert_function_exists()*
- *assert_trigger_exists()*

One of the more interesting functions is *assert_raises()* which can be used to check that something produces a specific SQLSTATE:

```
SELECT assert_raises('23505', 'INSERT INTO foo VALUES (1)');
```

2.4.2 auth extension

The functions of the *auth* extension are intended for bulk manipulation of role based authorizations. For example, use *copy_role_auths()* to copy all roles from user1 to user2:

```
SELECT copy_role_auths('user1', 'user2');
```

This will execute the minimum required GRANTs to provide user2 with all roles that user1 has, which user2 currently does not. Naturally, the caller must have the necessary authority to execute such GRANTs (thus, usage of this routine is generally only useful for superusers).

Please note that ownership of objects is *not* transferred by this routine. That can be easily accomplished with the REASSIGN OWNED statement instead.

If you wish to move all authorizations from one user to another this can be accomplished with the similar procedure:

```
SELECT move_role_auths('user1', 'user2');
```

A couple of other procedures can be used to manipulate table authorizations. To store and restore the authorizations associated with a table:

```
SELECT store_table_auths('foo');
SELECT restore_table_auths('foo');
```

This may seem pointless in and of itself until you understand that the authorizations are stored in the `stored_table_auths` table which allows you to manipulate them between storage and restoration. For example, to copy all authorizations from one table to another:

```
SELECT store_table_auths('foo');
UPDATE stored_table_auths SET table_name = 'bar'
WHERE table_name = 'foo';
SELECT restore_table_auths('bar');
```

Alternatively, to copy only the SELECT privileges:

```
SELECT store_table_auths('foo');
DELETE FROM stored_table_auths
WHERE table_name = 'foo'
AND privilege_type <> 'SELECT';
```

```sql
UPDATE stored_table_auths SET table_name = 'bar'
WHERE table_name = 'foo';
SELECT restore_table_auths('bar');
```

Of course, even without manipulation it can be useful when one wishes to drop and recreate the table for any reason (e.g. to change the structure in a way not supported by ALTER TABLE):

```sql
SELECT store_table_auths('foo');
DROP TABLE foo;
CREATE TABLE foo (i integer NOT NULL);
SELECT restore_table_auths('foo');
```

2.4.3 `merge` extension

Warning: This extension does not, and is not intended to, solve the UPSERT problem. It is intended solely for bulk transfers between similarly structured relations.

The *auto_insert()* function constructs an INSERT..SELECT statement for every column with the same name in both table1 and table2. Consider the following example definitions:

```sql
CREATE TABLE table1 (
    i integer NOT NULL PRIMARY KEY,
    j integer NOT NULL,
    k text
);

CREATE TABLE table2 (
    i integer NOT NULL PRIMARY KEY,
    j integer NOT NULL,
    k text,
    d timestamp DEFAULT current_timestamp NOT NULL
);
```

With these definitions, the following statements are equivalent:

```sql
SELECT auto_insert('table1', 'table2');

INSERT INTO table2 (i, j, k) SELECT i, j, k FROM table1;
```

The *auto_merge()* function constructs the PostgreSQL equivalent of an UPSERT or MERGE statement using writeable CTEs. Given the table definitions above, the following statements are equivalent:

```sql
SELECT auto_merge('table1', 'table2');

WITH upsert AS (
    UPDATE table2 AS dest SET
        i = src.i,
        j = src.j,
        k = src.k
    FROM table1 AS src
    WHERE src.i = dest.i
    RETURN src.i
)
INSERT INTO table2 (i, j, k)
SELECT i, j, k FROM table1
WHERE ROW (i) NOT IN (
    SELECT i
    FROM upsert
);
```

Finally, the *auto_delete()* function is used to remove all rows from table2 that do not exist in table1. Again, with the table definitions used above, the following statements are equivalent:

```
SELECT auto_delete('table1', 'table2');

DELETE FROM table2 WHERE ROW (i) IN (
    SELECT i FROM table2
    EXCEPT
    SELECT i FROM table1
);
```

2.4.4 `history` extension

Warning: It is strongly recommended that you read the *full usage chapter* on the temporal data functions to understand their precise effect and how to query and maintain the resulting structures. This section is intended as a brief introduction and/or refresher and does not discuss the complexities of temporal data at all.

In this section, the following example tables will be used:

```
CREATE TABLE employees (
    user_id     integer NOT NULL PRIMARY KEY,
    name        varchar(100) NOT NULL,
    dob         date NOT NULL,
    dept        char(4) NOT NULL,
    is_manager  boolean DEFAULT false NOT NULL,
    salary      numeric(8) NOT NULL
);
```

In order to track the history of changes to a particular table, construct a history table and set of triggers to maintain the content of the history table. The second parameter in the calls below specifies the resolution of changes that will be kept (this can be any interval supported by PostgreSQL):

```
SELECT create_history_table('employees', 'day');
SELECT create_history_triggers('employees', 'day');
```

The history table will have the same structure as the "base" table (in this case "employees"), but with the addition of two extra fields: effective and expiry as the first and second columns respectively. With the "day" resolution, these columns will have the "date" type. These two columns represent the inclusive range of dates on which a row was present within the base table.

The history table will initially be populated with the rows from the base table, with the effective date set to the current date, and expiry set to 9999-12-31 (to indicate each row is "current").

As changes are made to the base table, the history table will be automatically updated by triggers. To query the state of the base table at a particular point in time, X, simply use the following query:

```
SELECT * FROM employees_history WHERE X BETWEEN effective AND expiry;
```

To view the changes as a set of insertions, updates, and deletions, along with the ability to easily see "before" and "after" values for updates, construct a "changes" view with the following procedure:

```
SELECT create_history_changes('employees_history');
```

The resulting view will be called “employees_changes” by default. It will have a “changed” column (the date or timestamp) on which the change took place, a “change” column (containing the string “INSERT”, “UPDATE”, or “DELETE” depending on what operation took place), and two columns for each column in the base table, prefixed with “old_” and “new_” giving the “before” and “after” values for each column.

For example, to find all rows where an employee received a salary increase:

```
SELECT * FROM employees_changes
WHERE change = 'UPDATE'
AND new_salary > old_salary;
```

It is also possible to construct a view which provides snapshots of the base table over time. This is particularly useful for aggregation queries. For example:

```
SELECT create_history_snapshots('employees_history', 'month');

SELECT snapshot, dept, count(*) AS monthly_dept_headcount
FROM employees_by_month
GROUP BY snapshot, dept;
```

2.5 The `assert` Extension

The assert extension grew out of a desire to construct a test suite using SQL statements alone. It can be installed and removed in the standard manner:

```
CREATE EXTENSION assert;
DROP EXTENSION assert;
```

It is a relocatable, pure SQL extension which therefore requires no external libraries or compilation, and consists entirely of user-callable functions.

2.5.1 Usage

The most basic routines in the extension are *assert_is_null()* and *assert_is_not_null()* which test whether the given value is or is not NULL respectively, raising SQLSTATEs UTA06 and UTA07 respectively. These functions have two overloaded variants, one using the polymorphic `anyelement` type and the other `text` which should cover the vast majority of use cases:

```
db=# SELECT assert_is_null(null::date);
 assert_is_null
----------------

(1 row)

db=# SELECT assert_is_null('');
ERROR:   is not NULL
db=# SELECT assert_is_null(1);
ERROR:  1 is not NULL
```

Similarly, *assert_equals()* and *assert_not_equals()* test whether the two provided values are equal or not. If the assertion fails, SQLSTATE UTA08 is raised by *assert_equals()* and UTA09 by *assert_not_equals()*. Again, two overloaded variants exist to cover all necessary types:

```
db=# SELECT assert_equals(1, 1);
 assert_equals
---------------

(1 row)
```

```
db=# SELECT assert_equals('foo', 'bar');
ERROR:  foo does not equal bar
```

A set of functions for asserting the existing of various structures are also provided: `assert_table_exists()` (which works for any relation-like structure such as tables and views), `assert_column_exists()` (for testing individual columns within a relation), `assert_function_exists()`, and `assert_trigger_exists()`:

```
db=# CREATE TABLE foo (i integer NOT NULL);
CREATE TABLE
db=# SELECT assert_table_exists('foo');
 assert_table_exists
---------------------

(1 row)

db=# SELECT assert_table_exists('bar');
ERROR:  Table public.bar does not exist
CONTEXT:  SQL function "assert_table_exists" statement 1
db=# SELECT assert_column_exists('foo', 'i');
 assert_column_exists
----------------------

(1 row)
```

Note that with a bit of querying knowledge, it is actually more efficient to test a whole table structure using `assert_equals()`. For example:

```
CREATE TABLE bar (
    i integer NOT NULL PRIMARY KEY,
    j integer NOT NULL
);

SELECT assert_equals(4::bigint, (
    SELECT count(*)
    FROM (
        SELECT attnum, attname
        FROM pg_catalog.pg_attribute
        WHERE attrelid = 'bar'::regclass
        AND attnum > 0

        INTERSECT

        VALUES
            (1, 'i'),
            (2, 'j'),
    ) AS t));
```

Naturally, one could extend this technique to include tests for the column types, nullability, etc.

Finally, the `assert_raises()` function can be used to test whether arbitrary SQL raises an expected SQL-STATE. This is especially useful when building test suites for extensions (naturally, this function is used extensively within the test suite for the `assert` extension!):

```
db=# SELECT assert_raises('UTA08', 'SELECT assert_equals(1, 2)');
 assert_raises
---------------

(1 row)

db=# SELECT assert_raises('UTA08', 'SELECT assert_equals(1, 1)');
```

```
ERROR:  SELECT assert_equals(1, 1) did not signal SQLSTATE UTA08
```

2.5.2 API

`assert.`**`assert_equals`**(*a*, *b*)

Parameters

- **a** – The first value to compare
- **b** – The second value to compare

Raises SQLSTATE 'UTA08' if *a* and *b* are not equal. If either *a* or *b* are NULL, the assertion will succeed (no exception will be raised). See `assert_is_null()` for this instead.

`assert.`**`assert_not_equals`**(*a*, *b*)

Parameters

- **a** – The first value to compare
- **b** – The second value to compare

Raises SQLSTATE 'UTA09' if *a* and *b* are equal. If either *a* or *b* are NULL, the assertion will succeed (no exception will be raised). See `assert_is_null()` for this instead.

`assert.`**`assert_is_null`**(*a*)

Parameters **a** – The value to test

Raises SQLSTATE 'UTA06' if *a* is not NULL.

`assert.`**`assert_is_not_null`**(*a*)

Parameters **a** – The value to test

Raises SQLSTATE 'UTA07' if *a* is NULL.

`assert.`**`assert_table_exists`**(*aschema*, *atable*)
`assert.`**`assert_table_exists`**(*atable*)

Parameters

- **aschema** – The schema containing the table to test
- **atable** – The table to test for existence

Tests whether the table named *atable* within the schema *aschema* exists. If *aschema* is omitted it defaults to the current schema. Raises SQLSTATE 'UTA02' if the table does not exist.

`assert.`**`assert_column_exists`**(*aschema*, *atable*, *acolumn*)
`assert.`**`assert_column_exists`**(*atable*, *acolumn*)

Parameters

- **aschema** – The schema containing the table to test
- **atable** – The table containing the column to test
- **acolumn** – The column to test for existence

Tests whether the column named *acolumn* exists in the table identified by *aschema* and *atable*. If *aschema* is omitted it defaults to the current schema. Raises SQLSTATE 'UTA03' if the column does not exist.

`assert.`**`assert_trigger_exists`**(*aschema*, *atable*, *atrigger*)
`assert.`**`assert_trigger_exists`**(*atable*, *atrigger*)

Parameters

- **aschema** – The schema containing the table to test

- **atable** – The table containing the column to test
- **atrigger** – The trigger to test for existence

Tests whether the trigger named *atrigger* exists for the table identified by *aschema* and *atable*. If *aschema* is omitted it defaults to the current schema. Raises SQLSTATE 'UTA04' if the column does not exist.

`assert.`**`assert_function_exists`**(*aschema*, *atable*, *argtypes*)
`assert.`**`assert_function_exists`**(*atable*, *argtypes*)

Parameters

- **aschema** – The schema containing the function to test
- **atable** – The table to test for existence
- **argtypes** – An array of type names to match against the parameters of the function

Tests whether the function named *afunction* with the parameter types given by the array *argtypes* exists within the schema *aschema*. If *aschema* is omitted it defaults to the current schema. Raises SQLSTATE 'UTA05' if the table does not exist.

`assert.`**`assert_raises`**(*state*, *sql*)

Parameters

- **state** – The SQLSTATE to test for
- **sql** – The SQL to execute to test if it fails correctly

Tests whether the execution of the statement in *sql* results in the SQLSTATE *state* being raised. Raises SQLSTATE UTA01 in the event that *state* is not raised, or that a different SQLSTATE is raised.

2.6 The `auth` Extension

The auth extension was originally created to manage bulk authorization transfers in large scale data warehouses. It can be installed and removed in the standard manner:

```
CREATE EXTENSION auth;
DROP EXTENSION auth;
```

It is a relocatable, pure SQL extension which therefore requires no external libraries or compilation, and consists mostly of user-callable functions, with one table for storage.

2.6.1 Usage

Ideally large data warehouses would have a relatively small set of well defined roles which would be assigned to user IDs, granting them access to exactly what they need. Unfortunately, reality is rarely so well ordered. Data warehouses have a nasty habit of growing over time, and sometimes developers or admins will cut corners and grant authorities on objects directly to users instead of going via roles.

This leads to all sorts of issues when people inevitably move around, join and leave organizations, or simply go on holiday and need to grant another person access to the same objects while they're away. While one solution to such issues is simply to share the password for the user, this is hardly ideal from a security perspective.

The `auth` extension provides a partial solution to such issues with the *`copy_role_auths()`* function which, given a source and a target username will grant authorities to the target which the source has, but the target currently does not.

Note: Note that the current implementation only covers tables (and table-like objects such as views), functions, and roles. Authorizations for other objects are *not* copied.

For example:

```
SELECT copy_role_auths('fred', 'barney');
```

Another function, *remove_role_auths()* is provided to remove all authorities from a user, and this is also used in the implementation of *move_role_auths()* which simply performs a copy of authorities, then removes them from the source user:

```
SELECT move_role_auths('fred', 'barney');
SELECT remove_role_auths('wilma');
```

Note: Removing authorities from a user does *not* remove the authorities they derive from being the owner of an object. To remove these as well, see REASSIGN OWNED.

The other set of functions provided by the `auth` extension have to do with the manipulation of table authorities. The *store_table_auths()* function stores all authorizations for a table in the `stored_table_auths` table (constructed by the extension). The *restore_table_auths()* function can then be used to restore the authorizations to the table (removing them from `stored_table_auths` in the process).

These routines can be used in a number of scenarios. The simplest is when a table needs to be reconstructed (to deal with some structural change that cannot be accomplished with ALTER TABLE) and you wish to maintain the authorizations for the table:

```
SELECT store_table_auths('foo');
DROP TABLE foo;
CREATE TABLE foo (i integer NOT NULL, j integer NOT NULL);
-- Reload data into foo (e.g. from an export)
SELECT restore_table_auths('foo');
```

However, given that the `stored_table_auths` table can itself be manipulated, it can also be used for other effects. For example, to copy the authorizations from one table to another:

```
SELECT store_table_auths('foo');
CREATE TABLE bar (i integer NOT NULL);
UPDATE stored_table_auths SET table_name = 'bar' WHERE table_name = 'foo';
SELECT restore_table_auths('bar');
```

Or, to ensure that anyone who can `SELECT` from table `foo`, can also `SELECT` from the view `bar` (ignoring other privileges like `INSERT`, `UPDATE`, and such like):

```
SELECT store_table_auths('foo');
CREATE VIEW bar AS SELECT * FROM foo;
DELETE FROM stored_table_auths
    WHERE table_name = 'foo'
    AND privilege_type <> 'SELECT';
UPDATE stored_table_auths SET table_name = 'bar'
    WHERE table_name = 'foo';
SELECT restore_table_auths('bar');
```

See *create_history_table()* for an example of this usage.

2.6.2 API

`auth.`**`role_auths`**(*auth_name*)

Parameters **`auth_name`** – The role to retrieve authorizations for

This is a table function which returns one row for each privilege held by the specified authorization name. The rows have the following structure:

Column	Type	Description
object_type	varchar(20)	'TABLE', 'FUNCTION', or 'ROLE'
object_id	oid	The oid of the table or function, NULL if object_type is 'ROLE'
auth	varchar(140)	The name of the authorization, e.g. 'SELECT', 'EXECUTE', 'REFERENCES', or the name of the role if object_type is 'ROLE'
suffix	varchar(20)	The string 'WITH GRANT OPTION' or 'WITH ADMIN OPTION' if the authority was granted with these options. A blank string otherwise.

At present, the function is limited to authorities derived from tables (and table-like structures), functions, and roles.

`auth.`**`auth_diff`**(*source*, *dest*)

Parameters

- **`source`** – The base role to compare authorizations against
- **`dest`** – The target role to test for similar authorities

This table function is effectively a set subtraction function. It takes the set of authorities from the *source* role and subtracts from them the set of authorities that apply to the *target* role (both derived by calling `role_auths()`). The result is returned as a table with the same structure as that returned by `role_auths()`.

Note that if *source* holds SELECT WITH GRANT OPTION on a table, while *target* holds SELECT (with no GRANT option), then this function will consider those different "levels" of the grant and the result will include SELECT WITH GRANT OPTION.

`auth.`**`copy_role_auths`**(*source*, *dest*)

Parameters

- **`source`** – The role to copy authorities from
- **`dest`** – The role to copy authorities to

This function determines the GRANTs that need to be execute in order for *dest* to have the same rights to all objects as *source* (this is done with the `auth_diff()` function documented above). It then attempts to execute all such GRANTs; the calling user must have the authority to do this, therefore the use of this function is typically restricted to super users.

`auth.`**`remove_role_auths`**(*auth_name*)

Parameters **`auth_name`** – The role to remove authorities from

This function attempts to REVOKE all authorities from the specified role *auth_name*. This is not a great deal of use on PostgreSQL where it is simpler to just delete the role, but it is used by `move_role_auths()` below.

Warning: This will not remove authorities derived from ownership of an object.

`auth.`**`move_role_auths`**(*source*, *dest*)

Parameters

- **`source`** – The role to remove authorities from
- **`dest`** – The role to transfer authorities to

This function attempts to transfer all authorities from the *source* role to the *dest* role with a combination of `copy_role_auths()` and `remove_role_auths()`.

As in the case of *copy_role_auths()*, the calling user must have the authority to execute all necessary GRANTs and REVOKEs, therefore the use of this function is typically restricted to super users.

Warning: This will not remove authorities derived from ownership of objects from *source*. See REASSIGN OWNED for a method of accomplishing this.

auth.**store_table_auths**(*aschema*, *atable*)
auth.**store_table_auths**(*atable*)

Parameters

- **aschema** – The schema containing the table to read authorizations for
- **atable** – The table to read authorizations for

This function writes all authorities that apply to the table *atable* (in schema *aschema* or the current schema if this is omitted) to the `stored_table_auths` table which has the following structure:

Column	Type	Description
table_schema	name	The schema of the table
table_name	name	The name of the table the privilege applies to
grantee	name	The role the privilege is granted to
privilege_type	varchar	The name of the privilege, e.g. SELECT, UPDATE, etc.
is_grantable	boolean	If the privilege was granted WITH GRANT OPTION, then this is true

The table is keyed by table_schema, table_name, grantee, and privilege_type.

No errors will be raised if rows already exist in `stored_table_auths` violating this key; they will be updated instead. In other words, it is not an error to run this procedure multiple times in a row for the same table. However, the similar *restore_table_auths()* removes rows from this table, therefore usual practice is to perform the two functions within the same transaction effectively leaving the `stored_table_auths` table unchanged after.

auth.**restore_table_auths**(*aschema*, *table*)
auth.**restore_table_auths**(*atable*)

Parameters

- **aschema** – The schema containing the table to write authorizations to
- **atable** – The table to write authorizations to

This function removes rows from the `stored_table_auths` table (documented above for *store_table_auths()* function) and attempts to execute the GRANT represented by each row. Updating the `stored_table_auths` table between calls to *store_table_auths()* and this function permits various effects, including copying authorizations from one table to another, manipulating the list of authorities to be copied, and so on.

2.7 The `merge` Extension

The merge extension was created to simplify bulk transfers of data between similarly structured tables. It should be stressed up front that it does *not*, and is not intended to solve the UPSERT problem.

2.7.1 Usage

To transfer all rows from one table to another one would traditionally use SQL similar to the following:

```
INSERT INTO dest
    SELECT * FROM source;
```

However, when the source and destination are similar but not *exactly* the same one has to tediously specify all columns involved. The *auto_insert()* function eases this process by taking column names from the destination table and matching them to columns from the source table by name (regardless of position). In the following examples we will assume these table definitions:

```
CREATE TABLE contracts_clean (
    customer_id     integer NOT NULL,
    contract_id     integer NOT NULL,
    title           varchar(20) NOT NULL,
    plan_cost       decimal(18, 2) DEFAULT NULL,
    plan_revenue    decimal(18, 2) DEFAULT NULL,
    last_updated    timestamp NOT NULL,
    last_updated_by name NOT NULL
);

CREATE TABLE contracts (
    contract_id     integer NOT NULL,
    customer_id     integer NOT NULL,
    title           varchar(20) NOT NULL,
    plan_cost       decimal(18, 2) DEFAULT NULL,
    plan_revenue    decimal(18, 2) DEFAULT NULL,
    actual_cost     decimal(18, 2) DEFAULT 0.0 NOT NULL,
    actual_revenue  decimal(18, 2) DEFAULT 0.0 NOT NULL,

    PRIMARY KEY (customer_id, contract_id)
);
```

The *auto_insert()* function can be used with these definitions like so:

```
SELECT auto_insert('contracts_clean', 'contracts');
```

This is equivalent to executing the following SQL:

```
INSERT INTO contracts (
    customer_id,
    contract_id,
    title,
    plan_cost,
    plan_revenue
)
SELECT
    customer_id,
    contract_id,
    title,
    plan_cost,
    plan_revenue
FROM contracts_clean;
```

Note that columns are matched by name so that even though the order of `customer_id` and `contract_id` differs between the two tables, they are ordered correctly in the generated statement. Furthermore, columns that are not present in both tables are excluded, so `last_updated` and `last_updated_by` from the source table are ignored while `actual_cost` and `actual_revenue` will use their default values in the target table.

The similar function *auto_merge()* can be used to perform an "upsert" (a combination of INSERT or UPDATE as appropriate) between two tables. The function can be used with our example relations like so:

```
SELECT auto_merge('contracts_clean', 'contracts');
```

This is equivalent to executing the following SQL:

```
WITH upsert AS (
    UPDATE contracts AS dest SET
        plan_cost = src.plan_cost,
        plan_revenue = src.plan_revenue,
        title = src.title
    FROM contracts_clean AS src
    WHERE
        src.contract_id = dest.contract_id
        AND src.customer_id = dest.customer_id
    RETURNING
        src.contract_id,
        src.customer_id
)
INSERT INTO contracts (
    contract_id,
    customer_id,
    plan_cost,
    plan_revenue,
    title
)
SELECT
    contract_id,
    customer_id,
    plan_cost,
    plan_revenue,
    title
FROM contracts_clean
WHERE ROW (contract_id, customer_id) NOT IN (
    SELECT contract_id, customer_id
    FROM upsert
);
```

As you can discern from reading the above, this will attempt to execute updates with each row from source against the target table and, if it fails to find a matching row (according to the primary key of the target table, by default) it attempts insertion instead.

Finally, the *auto_delete()* function can be used to automatically delete rows that exist in the target table, that do not exist in the source table:

```
SELECT auto_delete('contracts_clean', 'contracts');
```

This is equivalent to executing the following statement:

```
DELETE FROM contracts WHERE ROW (contract_id, customer_id) IN (
    SELECT contract_id, customer_id FROM contracts
    EXCEPT
    SELECT contract_id, customer_id FROM contracts_clean
)
```

2.7.2 Use-cases

These routines are designed for use in a database environment in which cleansing of incoming data is handled by views within the database. The process is intended to work as follows:

1. Data is copied into a set of tables which replicate the structures of their source, without any constraints or restrictions. The lack of constraints is important to ensure that the source data is represented accurately, imperfections and all. However, non-unique indexes can be created on these tables to ensure performance in the next stages.
2. On top of the source tables, views are created to handle cleaning the data. Bear in mind that transformation of data (by INSERT, UPDATE, or DELETE operations) can be accomplished via queries. For example:
    - If you need to INSERT records into the source material, simply UNION ALL the source table with the new records (generated via a VALUES statement)

   - If you need to DELETE records from the source material, simply filter them out in the WHERE clause (or with a JOIN)
   - If you need to UPDATE records in the source material, change the values with transformations in the SELECT clause

3. Finally, the reporting tables are created with the same structure as the output of the cleaning views from the step above.

To give a concrete example of this method, consider the examples from above. Let us assume that the source of the contracts data is a CSV file periodically refreshed by some process (this probably sounds awful, and it is, but I've seen worse in practice). We would represent this source data with a table like so:

```
CREATE TABLE contracts_raw (
    customer_id     text NOT NULL,
    contract_id     text NOT NULL,
    title           text NOT NULL,
    plan_cost       text NOT NULL,
    plan_revenue    text NOT NULL,
    last_updated    text NOT NULL,
    last_updated_by text NOT NULL
);
```

Note the use of text fields as we've no guarantee that any of the CSV data is actually well structured and we want to ensure that it is loaded successfully (even if subsequent cleaning fails) so that we have a copy of the source data to debug within the database (this is much easier than relying on external files for debugging).

Now we'd construct the `contracts_clean` table as a view on top of this:

```
DROP TABLE contracts_clean;
CREATE VIEW contracts_clean AS
    SELECT
        b.customer_id::int,
        b.contract_id::int,
        b.title::varchar(20),
        CASE
            WHEN b.plan_cost    ~ '^[0-9]{0,16}\.[0-9]{2}$' THEN b.plan_cost
        END::decimal(18, 2) AS plan_cost,
        CASE
            WHEN b.plan_revenue ~ '^[0-9]{0,16}\.[0-9]{2}$' THEN b.plan_revenue
        END::decimal(18, 2) AS plan_revenue,
        b.last_updated::timestamp,
        b.last_updated_by::name
    FROM
        contracts_base b
        JOIN customers c
            ON b.customer_id::int = c.customer_id
    WHERE b.contract_id::int > 0;
```

The view performs the following operations:

- Casts are used to convert the text from the CSV into the correct data-type. In certain cases, CASE expressions with regexes are used to guard against "known bad" data, but in others an error will occur if the source data is incorrect (this is deliberate, under the theory that it is better to fail loudly than silently produced incorrect results).
- A JOIN on a customers table ensures any rows with invalid customer numbers are excluded; if we wished to include them we could use an OUTER JOIN, and make up an "invalid customer" customer to substitute in such cases (if `customer_id` weren't part of the primary key we could simply use the OUTER JOIN and accept the NULL in cases of invalid customer numbers).
- A WHERE clause excludes any negative or zero contract IDs (presumably these occur in the source and are not wanted).

Now we can load data into our final `contracts` table, with all data cleaning performed in SQL as follows:

```
COPY contracts_raw FROM 'contracts.csv' WITH (FORMAT csv);
SELECT auto_merge('contracts_clean', 'contracts');
SELECT auto_delete('contracts_clean', 'contracts');
```

Why are the merge and delete functions provided separately? Consider the case where our contracts table has a foreign key to the customers table we referenced above:

```
CREATE TABLE contracts (
    contract_id     integer NOT NULL,
    customer_id     integer NOT NULL,
    title           varchar(20) NOT NULL,
    plan_cost       decimal(18, 2) DEFAULT NULL,
    plan_revenue    decimal(18, 2) DEFAULT NULL,
    actual_cost     decimal(18, 2) DEFAULT 0.0 NOT NULL,
    actual_revenue  decimal(18, 2) DEFAULT 0.0 NOT NULL,

    PRIMARY KEY (customer_id, contract_id),
    FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
        ON DELETE RESTRICT
);
```

In this case we have to ensure that new customers are inserted before contracts is updated (in case any contracts reference the new customers) but we must also ensure that old customers are only deleted *after* contracts has been updated (in case any existing contracts reference the old customers). Assuming customers had a similar setup (a table to hold the raw source data, a view to clean the raw data, and a final table to contain the cleaned data), in this case our loading script might look something like this:

```
COPY contracts_raw FROM 'contracts.csv' WITH (FORMAT csv);
COPY customers_raw FROM 'customers.csv' WITH (FORMAT csv);

SELECT auto_merge('customers_clean', 'customers');
SELECT auto_merge('contracts_clean', 'contracts');
SELECT auto_delete('contracts_clean', 'contracts');
SELECT auto_delete('customers_clean', 'customers');
```

As a general rule, given a hierarchy of tables with foreign keys between them, merge from the top of the hierarchy down to the bottom, then delete from the bottom of the hierarchy back up to the top.

Why bother with a merge function at all? Why not truncate and re-write the target table each time? In the case of small to medium sized tables this may be a perfectly realistic option in terms of performance (it may even lead to better performance in some circumstances). In the case of large tables, obviously it pays to do as little IO as possible and therefore merging is usually preferable (on the assumption that most of the data doesn't change that much).

However, there is another more subtle reason to consider. By merging we are accurately telling the database engine what happened to each record: whether it was inserted, updated or deleted at the source. If we truncated and re-wrote the whole table such information would be lost. In turn this allows us to accurately use the *history* extension to keep a history of our customers and contracts tables. We could simply execute the following statements:

```
SELECT create_history_table('customers', 'day');
SELECT create_history_table('contracts', 'day');
SELECT create_history_triggers('customers', 'day');
SELECT create_history_triggers('contracts', 'day');
```

Now every time the customers and contracts tables are loaded with our script above, the history is updated too and we can show the state of these tables for any day in the past.

2.7.3 API

merge.**auto_insert**(*source_schema*, *source_table*, *dest_schema*, *dest_table*) `merge.`**`auto_insert`**`(`*source_table*, *dest_table*`)`

Parameters

- **`source_schema`** – The schema containing the source table. Defaults to the current schema if omitted.
- **`source_table`** – The source table from which to read data.
- **`dest_schema`** – The schema containing the destination table. Defaults to the current schema if omitted.
- **`dest_table`** – The destination table into which data will be inserted.

Inserts rows from the table named by *source_schema* and *source_table* into the table named by *target_schema* and *target_table*. The schema parameters can be omitted in which case they will default to the current schema.

Columns of the two tables will be matched by name, *not* by position. Any columns that do not occur in both tables will be omitted (if said columns occur in the target table, the defaults of those columns will be used on insertion). The source table may also be a view.

`merge.`**`auto_merge`**`(`*source_schema*, *source_table*, *dest_schema*, *dest_table*, *dest_key*`)`
`merge.`**`auto_merge`**`(`*source_schema*, *source_table*, *dest_schema*, *dest_table*`)`
`merge.`**`auto_merge`**`(`*source_table*, *dest_table*, *dest_key*`)`
`merge.`**`auto_merge`**`(`*source_table*, *dest_table*`)`

Parameters

- **`source_schema`** – The schema containing the source table. Defaults to the current schema if omitted.
- **`source_table`** – The source table from which to read data.
- **`dest_schema`** – The schema containing the destination table. Defaults to the current schema if omitted.
- **`dest_table`** – The destination table into which data will be merge.
- **`dest_key`** – The primary or unique key on the destination table which will be used for matching existing records. Defaults to the primary key if omitted.

Merges rows from the table identified by *source_schema* and *source_table* into the table identified by *target_schema* and *target_table*, based on the primary or unique key of the target table named by *dest_key*. If the schema parameters are omitted they default to the current schema. If the *dest_key* parameter is omitted it defaults to the name of the primary key of the target table.

Columns of the two tables will be matched by name, *not* by position. Any columns that do not occur in both tables will be omitted from updates or inserts. However, all columns specified in *dest_key* must also exist in the source table.

If a row from the source table already exists in the target table, it will be updated with the non-key attributes of that row in the source table. Otherwise, it will be inserted into the target table.

> **Warning:** This function is intended for bulk transfer between similarly structured relations. It does not solve the concurrency issues required by those looking for atomic upsert functionality.

`merge.`**`auto_delete`**`(`*source_schema*, *source_table*, *dest_schema*, *dest_table*, *dest_key*`)`
`merge.`**`auto_delete`**`(`*source_schema*, *source_table*, *dest_schema*, *dest_table*`)`
`merge.`**`auto_delete`**`(`*source_table*, *dest_table*, *dest_key*`)`
`merge.`**`auto_delete`**`(`*source_table*, *dest_table*`)`

Parameters

- **`source_schema`** – The schema containing the source table. Defaults to the current schema if omitted.

- **`source_table`** – The source table from which to read data.
- **`dest_schema`** – The schema containing the destination table. Defaults to the current schema if omitted.
- **`dest_table`** – The destination table from which data will be deleted.
- **`dest_key`** – The primary or unique key on the destination table which will be used for matching existing records. Defaults to the primary key if omitted.

Removes rows from the table identified by *target_schema* and *target_table* if those rows do not also exist in the table identified by *source_schema* and *source_table*, based on the primary or unique key of the target table named by *dest_key*. If the schema parameters are omitted they default to the current schema. If the *dest_key* parameter is omitted it defaults to the primary key of the target table.

Columns of the two tables will be matched by name, *not* by position. All columns specified in *dest_key* must exist in the source table.

2.8 The `history` Extension

The history extension was created to ease the construction and maintenance of temporal tables; that is tables which track the state of another table over time. It can be installed and removed in the standard manner:

```
CREATE EXTENSION history;
DROP EXTENSION history;
```

It is a relocatable, pure SQL extension which therefore requires no external libraries or compilation, and consists entirely of user-callable functions.

2.8.1 Setup

To create a history table corresponding to an existing table, first set up your base tables as normal. The examples in this section will be using the following table definitions:

```
CREATE TABLE departments (
    dept_id         char(4) NOT NULL PRIMARY KEY,
    name            varchar(100) NOT NULL
);

CREATE TABLE employees (
    emp_id          integer NOT NULL PRIMARY KEY,
    name            varchar(100) NOT NULL,
    dob             date NOT NULL,
    dept_id         char(4) NOT NULL REFERENCES departments(dept_id),
    is_manager      boolean DEFAULT false NOT NULL,
    salary          numeric(8) NOT NULL CHECK (salary >= 0)
);

COMMENT ON TABLE employees IS 'The set of people currently employed by the company
↪';
COMMENT ON COLUMN employees.emp_id     IS 'The unique identifier of the employee';
COMMENT ON COLUMN employees.name       IS 'The full name of the employee';
COMMENT ON COLUMN employees.dob        IS 'The date of birth of the employee';
COMMENT ON COLUMN employees.dept_id    IS 'The department the employee belongs to';
COMMENT ON COLUMN employees.is_manager IS 'True if the employee manages others';
COMMENT ON COLUMN employees.salary     IS 'The base annual salary of the employee␣
↪in US dollars';

GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO dir_manager;
GRANT SELECT ON employees TO dir_web_intf;
```

Then, use the *create_history_table()* function like so:

```
db=# select create_history_table('employees', 'day');
 create_history_table
----------------------

(1 row)
```

The procedure will create a new table called `employees_history` (you can specify a different name with one of the overloaded variants of the procedure). The new table will have a structure equivalent to executing the following statements:

```
CREATE TABLE employees_history (
    effective       date NOT NULL DEFAULT current_date,
    expiry          date NOT NULL DEFAULT '9999-12-31'::date,
    emp_id          integer NOT NULL,
    name            varchar(100) NOT NULL,
    dob             date NOT NULL,
    dept            char(4) NOT NULL,
    is_manager      boolean DEFAULT false NOT NULL,
    salary          numeric(8) NOT NULL CHECK (salary >= 0)

    PRIMARY KEY (emp_id, effective),
    UNIQUE (emp_id, expiry),
    CHECK (effective <= expiry)
);

CREATE INDEX employees_history_ix2 ON
    employees_history (effective, expiry);

COMMENT ON TABLE employees_history IS 'History table which tracks the content of␣
↪@public.employees';
COMMENT ON COLUMN employees_history.effective  IS 'The date/timestamp from which␣
↪this row was present in the source table';
COMMENT ON COLUMN employees_history.expiry     IS 'The date/timestamp until which␣
↪this row was present in the source table (rows with 9999-12-31 currently exist␣
↪in the source table)';
COMMENT ON COLUMN employees_history.emp_id     IS 'The unique identifier of the␣
↪employee';
COMMENT ON COLUMN employees_history.name       IS 'The full name of the employee';
COMMENT ON COLUMN employees_history.dob        IS 'The date of birth of the␣
↪employee';
COMMENT ON COLUMN employees_history.dept       IS 'The department the employee␣
↪belongs to';
COMMENT ON COLUMN employees_history.is_manager IS 'True if the employee manages␣
↪others';
COMMENT ON COLUMN employees_history.salary     IS 'The base annual salary of the␣
↪employee in US dollars';

GRANT SELECT ON employees_history TO dir_manager;
GRANT SELECT ON employees_history TO dir_web_intf;

INSERT INTO employees_history
    (emp_id, name, dob, dept, is_manager, salary)
    SELECT emp_id, name, dob, dept, is_manager, salary
    FROM employees;
```

The structure of the new table is the same as the old with the following differences:

- Two columns, `effective` and `expiry` have been inserted at the front of the table.
- These two columns have the type `date` (this is partially derived from the `'day'` resolution parameter which will be explained further below)

- All NOT NULL constraints have been copied to the new table.
- All CHECK and EXCLUDE constraints are copied but for reasons explored below, FOREIGN KEY constraints are *not* copied.
- The primary key of the new table is the same as the old table with the addition of the new `effective` column.
- An additional unique constraint has been created which is equivalent to the primary key but with the `expiry` column instead.
- An additional CHECK constraint has been created to ensure that `effective` is always less than or equal to `expiry`.
- An additional index is created covering just `effective` and `expiry` for performance purposes.
- All column comments from the base table are copied to the history table, and appropriate comments are set for the table itself (referencing the base table), and for the new `effective` and `expiry` columns.
- SELECT authorizations for the base table are copied to the history table. INSERT, UPDATE, DELETE and TRUNCATE authorizations are *not* copied for reasons explained below.
- Finally, data is copied from the original table into the new history table. The defaults of the excluded `effective` and `expiry` columns will set those fields appropriately during this operation.

This completes the first step in creating a functional history table. The reason that FOREIGN KEY constraints are excluded from duplication on the history table is that there is no good way to enforce them upon history rows. Consider the scenario where an employee used to be a member of a department which is removed. The history table must represent that the employee used to belong to this department, but the parent row no longer exists in the departments table.

Even if we also applied a history to the departments table, a simple equality lookup (which is all that foreign keys support) is insufficient to find the parent row; as demonstrated in the *Querying* section below an, inequality is required.

The second, and final, step is to create the triggers that link the base table to the history table. This is performed separately for reasons that will be explained below. The procedure to create these triggers is called as follows:

```
db=# select create_history_triggers('employees', 'day');
 create_history_triggers
-------------------------

(1 row)
```

This creates four triggers (and their corresponding functions):

- `employees_insert` which is triggered upon INSERT operations against the `employees` table, which inserts new rows into `employees_history`.
- `employees_update` which is triggered upon UPDATE operations against the `employees` table. This expires the current history row (by changing its date from 9999-12-31 to yesterday's date), and inserts a new one with the newly updated values (which will have an effective date of today, and an expiry date of 9999-12-31).
- `employees_delete` which is triggered upon DELETE operations against the `employees` table. This simply expires the current history row as detailed above.
- `employees_truncate` which is triggered upon TRUNCATE operations against the `employees` table. This expires all current history rows as detailed above.
- `employees_keychg` which is triggered upon UPDATE of key columns in the `employees` table. This simply raises an exception; i.e. updates of the primary key columns are not permitted in tables which have their history tracked (to update the primary key columns you must DELETE the row and re-INSERT it with the new key).

The trigger functions are defined as SECURITY DEFINER. Combined with the exclusion of INSERT, UPDATE, DELETE, and TRUNCATE authorizations (see action list above) this ensures that the only way (regular) users can update the history table is via the trigger responding to manipulations of the base table.

If you have existing history records that you wish to load into the history table, this should be done before the creation of history triggers. See below for more information on the structure and behaviour of the history table.

Resolution

The last parameter when creating both the table and triggers is the resolution to use in the resulting structures. So far, we have used `'day'` but any of the following resolutions are valid:

- `'microsecond'`
- `'millisecond'`
- `'second'`
- `'minute'`
- `'hour'`
- `'day'`
- `'week'`
- `'month'`
- `'quarter'`
- `'year'`
- `'decade'`
- `'century'`
- `'millennium'`

The resolution affects how many changes are kept in the history table. With the `'day'` resolution, only the final state of a record in a given day will be stored in the history table. For example, if a row is inserted into the base table, it will also appear in the history table:

```
db=# insert into departments values ('SR01', 'Slate Rock and Gravel dept 01');
INSERT 0 1
db=# insert into employees values (1, 'Fred Flintstone', '1960-07-05', 'SR01',␣
↪false, 10000.0);
INSERT 0 1
db=# select * from employees_history;
 effective  |   expiry   | emp_id |      name       |    dob     | dept_id | is_
↪manager | salary
------------+------------+--------+-----------------+------------+---------+-------
↪-----+--------
 2015-04-23 | 9999-12-31 |      1 | Fred Flintstone | 1960-07-05 | SR01    | f     ␣
↪      |  10000
(1 row)
```

Now, if we update the row (on the same day that we inserted it), the history row is also updated:

```
db=# update employees set salary = 20000.0 where emp_id = 1;
UPDATE 1
db=# select * from employees_history;
 effective  |   expiry   | emp_id |      name       |    dob     | dept_id | is_
↪manager | salary
------------+------------+--------+-----------------+------------+---------+-------
↪-----+--------
```

```
 2015-04-23 | 9999-12-31 |      1 | Fred Flintstone | 1960-07-05 | SR01     | f      ␣
↪     |  20000
(1 row)
```

Finally, if we delete the row (again, on the same day), the history row is removed. In each case, the history table is showing the *final* state of the row on the given day:

```
db=# delete from employees where emp_id = 1;
DELETE 1
db=# select * from employees_history ;
 effective | expiry | emp_id | name | dob | dept_id | is_manager | salary
-----------+--------+--------+------+-----+---------+------------+--------
(0 rows)
```

However, if we insert the row again, then "cheat" and tweak the history table so it appears as if it were inserted yesterday (we can do this because we are the table owner but ordinary users would not have the necessary UPDATE privilege), a subsequent UPDATE of the row will expire the old history row and insert a new one:

```
db=# insert into employees values (1, 'Fred Flintstone', '1960-07-05', 'SR01',␣
↪false, 10000.0);
INSERT 0 1
db=# update employees_history set effective = effective - interval '1 day' where␣
↪emp_id = 1;
UPDATE 1
db=# update employees set salary = 20000.0 where emp_id = 1;
UPDATE 1
db=# select * from employees_history;
 effective  |   expiry   | emp_id |      name       |    dob     | dept_id | is_
↪manager | salary
------------+------------+--------+-----------------+------------+---------+-------
↪-----+--------
 2015-04-22 | 2015-04-22 |      1 | Fred Flintstone | 1960-07-05 | SR01    | f      ␣
↪     |  10000
 2015-04-23 | 9999-12-31 |      1 | Fred Flintstone | 1960-07-05 | SR01    | f      ␣
↪     |  20000
(2 rows)
```

Usually the first reaction of users of the history framework is "I'll just use microsecond resolution because I want to keep all changes". I would caution against this for several reasons:

- Firstly, there is no guarantee that all changes will be kept (although at the time of writing the author has never seen a setup that was capable of making two separate changes to the same record within the same microsecond, so this is a rather theoretical objection).
- Secondly, this implies that someone is attempting to use the extension as an auditing solution. For reasons discussed in the *Design* section below, this is not a good idea.

If you are not attempting to build an auditing setup, consider carefully whether you *really* need every single change. As an example, in one case the author is aware of a company kept a record of every change to its employees table. After a few years, the employees history table was over 6 million rows long and caused significant performance problems in joins.

An analysis of the history table showed that over 90% of the rows had effective ranges lasting less than a minute; the result of people making changes, then correcting mistakes, or just making many changes as individual transactions.

For most data analysis or business intelligence purposes that the author has been engaged in, day or sometimes even week resolution has proved sufficient for all analytical purposes.

Finally, an offset can also be applied to all timestamp calculations undertaken by the history triggers. This facility was primarily designed for dealing with sources which significantly delay the delivery of their data, but for which a history with accurate dates is still desired. See the API documentation for the `create_history_triggers()` function for further information.

Limitations

It is worth noting that there are a few limitations on which tables can be used as the basis for a history table:

- Base tables *must* have a primary key.
- The primary key of a base table must be immutable (you may have noticed that this will be enforced through the `keychg` trigger above).

It is still possible to update the primary key of a base table with a history table but it must be done via a DELETE and INSERT operation rather than UPDATE (this is how such an operation would be represented by the history in either case, hence why this restriction is enforced).

2.8.2 Querying

The structure of the history table can be understood as follows:

- For each row that currently exists in the base table, an equivalent row will exist in the history table with the expiry date set to 9999-12-31 (i.e. in the future because it is an extant row).
- For each row that historically existed in the base table, an equivalent row will exist in the history table with the effective and expiry dates indicating the range of dates between which that row existed in the base table.

Therefore, to query the state of the base table at date 2014-01-01 we can simply use the following query:

```
SELECT emp_id, name, dob, dept, is_manager, salary
FROM employees_history
WHERE '2014-01-01' BETWEEN effective AND expiry;
```

In general, to retrieve the state of a base table at a given timestamp from a history table, one uses a query of this format:

```
SELECT fields, of, the, base, table
FROM history_table
WHERE required_timestamp BETWEEN effective AND expiry;
```

If you have a join to the base table, you can join to the history table in the same way: just include the criteria above to select the state of the table at a particular time. For example, assume there exists a table which tracks any bonuses awarded to employees. We can calculate the amount that the company has spent on bonuses like so:

```
CREATE TABLE bonuses (
    emp_id          integer NOT NULL,
    awarded_on      date NOT NULL,
    bonus_percent   numeric(4, 1) NOT NULL,

    PRIMARY KEY (emp_id, awarded_on),
    CHECK (bonus_percent BETWEEN 0 AND 100)
);

SELECT
    extract(year from b.awarded_on)          AS year,
    sum(e.salary * (b.bonus_percent / 100)) AS annual_bonus_spend
FROM
    employees_history e
    JOIN bonuses
        ON e.emp_id = b.emp_id
        AND b.awarded_on BETWEEN e.effective AND e.expiry
GROUP BY
    extract(year from b.awarded_on);
```

It should be noted that the design of the `bonuses` table in the example above demonstrates an alternative structure for storage of temporal data. This, and a few other designs will be discussed in the *Design* section below.

While it is easy to query the state of the base table at a given timestamp, it is harder to see how one could query changes within the history. For example, which employees have received a salary increase? Usually for this, it is necessary to self-join the history table so that one can see before and after states for changes. Creation of such views is automated with the `create_history_changes()` function. We can simply execute:

```
db=# select create_history_changes('employees_history');
 create_history_changes
------------------------

(1 row)
```

This will create a view named `employees_changes` with the following attributes:

- The first column will be named `changed` and will contain the timestamp of the change that occurred.
- The second column will be named `change` and will contain the string INSERT, UPDATE, or DELETE indicating which operation was performed.
- The remaining columns are defined as follows: for each column in the base table there will be two columns in the view, prefixed with "old_" and "new_"

In our example above, the view would be defined with the following SQL:

```
CREATE VIEW employees_changes AS
SELECT
    COALESCE(
        new.effective, old.expiry + '1 day'::interval) AS changed,
    CASE
        WHEN old.emp_id IS NULL AND new.emp_id IS NOT NULL THEN 'INSERT'
        WHEN old.emp_id IS NOT NULL AND new.emp_id IS NOT NULL THEN 'UPDATE'
        WHEN old.emp_id IS NOT NULL AND new.emp_id IS NULL THEN 'DELETE'
        ELSE 'ERROR'
    END AS change,
    old.emp_id AS old_emp_id,
    new.emp_id AS new_emp_id,
    old.name AS old_name,
    new.name AS new_name,
    old.dob AS old_dob,
    new.dob AS new_dob,
    old.dept AS old_dept,
    new.dept AS new_dept,
    old.is_manager AS old_is_manager,
    new.is_manager AS new_is_manager,
    old.salary AS old_salary,
    new.salary AS new_salary
FROM (
    SELECT *
    FROM employees_history
    WHERE employees_history.expiry < '9999-12-31'
    ) AS old
    FULL JOIN employees_history AS new
        ON (new.effective - interval '1 day'::interval) BETWEEN old.effective AND␣
↪old.expiry
        AND old.emp_id = new.emp_id;
```

With this view it is now a simple matter to determine which employees have received a salary increase:

```
db=# select new_emp_id, new_name, old_salary, new_salary
db-# from employees_changes where change = 'UPDATE' and new_salary > old_salary;
 new_emp_id |    new_name     | old_salary | new_salary
------------+-----------------+------------+------------
          1 | Fred Flintstone |      10000 |      20000
(1 row)
```

Or we can find out who joined and who left during the last year:

```
db=# select coalesce(old_emp_id, new_emp_id) as emp_id, coalesce(old_name, new_
↪name) as name
db-# from employees_changes where change in ('INSERT', 'DELETE') and changed >=␣
↪current_date - interval '1 year';
 emp_id |       name
--------+-----------------
      1 | Fred Flintstone
(1 row)
```

Another common use case of history tables is to see the changes in data over time via regular snapshots. This is also easily accomplished with the *create_history_snapshots()* function which takes the history table and a resolution (which must be greater than the history table's resolution). For example, to view the employees table as a series of monthly snapshots:

```
db=# select create_history_snapshots('employees_history', 'month');
 create_history_snapshots
--------------------------

(1 row)
```

This is equivalent to executing the following SQL:

```
CREATE VIEW employees_by_month AS
WITH RECURSIVE range(at) AS (
    SELECT min(employees_history.effective) AS min
    FROM employees_history

    UNION ALL

    SELECT range.at + interval '1 month'
    FROM range
    WHERE range.at + interval '1 month' <= current_date
    )
SELECT
    date_trunc('month', r.at) + interval '1 month' - interval '1 day' AS snapshot,
    h.emp_id,
    h.name,
    h.dob,
    h.dept,
    h.is_manager,
    h.salary
FROM
    range r
    JOIN employees_history h
        ON r.at BETWEEN h.effective AND h.expiry;
```

The resulting view has the same structure as the base table, but with one extra column at the start: `snapshot` which in the case above will contain a date running from the lowest date in the history to the current date in monthly increments. If we wished for an employee head-count by month we could simply use the following query:

```
db=# select snapshot, count(*) as head_count
db-# from employees_by_month
db-# group by snapshot;
      snapshot        | head_count
----------------------+------------
 2015-04-30 00:00:00 |          1
(1 row)
```

Or we could find out the employee headcount and salary costs broken down by month and managerial status:

```
SELECT
    snapshot,
    is_manager,
    count(*) AS head_count,
    sum(salary) AS salary_costs
FROM employees_by_month
GROUP BY snapshot, is_manager;
```

Note that because this view relies on a recursive CTE its performance may suffer with large date ranges. In such cases you may wish to materialise the view and index relevant columns.

2.8.3 Maintenance

For the most part, the history table should maintain itself. The same goes for any changes or snapshots views which you create (the latter automatically uses the minimum effective date in the underlying history table, and today's current date as its range).

However, there are circumstances under which it is necessary to perform manual maintenance of the history structures which are detailed in the sections below.

Structural Changes

You may find yourself needing to change the structure of the base table feeding the history table. Naturally, this is more complicated than simply altering a regular table. The first thing to determine is whether the modification you wish to make is capable of being made in a way that does not damage the history.

For example, if you are adding a column, the history table may also need that column added in which case you will either need to make the column nullable or come up with some suitable default (similar to adding a column to any non-empty table). The procedure is as follows:

1. Destroy the existing history triggers
2. Alter the base table
3. Alter the history table
4. Re-creating the history triggers

Thankfully, PostgreSQL supports transactional DDL which means we can accomplish all this in a single transaction with inconsistent states invisible to other active transactions, as demonstrated below:

```
db=# begin;
BEGIN
db=# select drop_history_triggers('employees');
 drop_history_triggers
-----------------------

(1 row)

db=# alter table employees add column full_time boolean default true not null;
ALTER TABLE
db=# alter table employees_history add column full_time boolean default true not␣
↪null;
ALTER TABLE
db=# select create_history_triggers('employees', 'day');
 create_history_triggers
-------------------------

(1 row)

db=# commit;
COMMIT
```

Note that the history table need not have all the attributes of the base table. This is specifically to support the use-case where certain attributes of the base table should not be tracked (in this case one can create the history table with `create_history_table()`, drop certain attributes from the newly created table with ALTER TABLE and then create the triggers). However, all primary key columns from the base table must exist in the history table.

Removing a column is a similar process, provided it's not a key column. Remember that history triggers *require* a primary key on the base table and that the history tables also require that key plus the effective column. Therefore, unless you are sure that removing a key column leaves a unique key sufficient to identify each row in the base *and* history tables (when combined with the effective date), you cannot remove it.

Altering columns is also a similar process: just remember to alter the history table in the same way as the base table in between destroying and recreating the triggers.

Archiving

In the case that you wish to make an alteration to the base table that cannot also be made in the history table, you may wish to store the current history table as an archive, then create a new one starting from the current point in time. The procedure is as follows:

1. Destroy the history triggers
2. Alter the base table
3. Expire all rows in the history table (this requires calculating the prior date or timestamp for the selected resolution)
4. Rename the history table
5. Create a new history table (remember that this will copy data from the base table)
6. Recreate the history triggers

For example, consider the case where we want to store employee's salary in local currency instead of US dollars. Firstly this will entail adding a field to store the currency, and then updating all the salaries accordingly. Let us assume that we do not have a historical record of currency exchange rates and thus the decision is made to leave the current history as US dollars and start a new history.

```
db=# begin;
BEGIN
db=# select drop_history_triggers('employees');
 drop_history_triggers
-----------------------

(1 row)

db=# create table currencies (cur_id char(3) not null primary key, usd_to_lcl␣
↪decimal(12, 4) not null);
CREATE TABLE
db=# insert into currencies values ('USD', 1.0), ('GBP', 0.66), ('EUR', 0.92), (
↪'CNY', 6.20), ('JPY', 119.47);
INSERT 0 5
db=# alter table employees add cur_id char(3) default 'EUR' not null references␣
↪currencies (cur_id);
ALTER TABLE
db=# update employees e set salary = salary * usd_to_lcl from currencies c where e.
↪cur_id = c.cur_id;
UPDATE 1
db=# update employees_history set expiry = current_date - interval '1 day' where␣
↪expiry = '9999-12-31';
UPDATE 0
db=# alter table employees_history rename to employees_history_old;
ALTER TABLE
db=# alter index employees_history_pkey rename to employees_history_old_pkey;
ALTER INDEX
```

```
db=# alter index employees_history_ix1 rename to employees_history_old_ix1;
ALTER INDEX
db=# alter index employees_history_ix2 rename to employees_history_old_ix2;
ALTER INDEX
db=# select create_history_table('employees', 'day');
 create_history_table
----------------------

(1 row)

db=# select create_history_triggers('employees', 'day');
 create_history_triggers
-------------------------

(1 row)

db=# commit;
COMMIT
```

Obviously this requires users querying the history to bridge the discontinuity themselves (for example, by unioning compatible transforms or subsets of the two histories). You may wish to provide a convenience view in such cases.

The same process can be used in the case you simply wish to archive the existing history for performance reasons. Obviously in this case you would not alter the base table, but it would still be necessary to disable and re-create the history triggers.

Finally, in certain rare circumstances you may find that you need to alter the content of the history without affecting the base table. In this case you can simply disable the existing triggers, make your alterations and re-enable them. However, you must be extremely careful to ensure that you do not create overlapping history ranges, or contradict the current state of the base table (by affecting rows with expiry date 9999-12-31).

For example, to double all the historical salaries (without affecting current ones):

```
db=# begin;
BEGIN
db=# alter table employees disable trigger all;
ALTER TABLE
db=# update employees_history set salary = salary * 2.0 where expiry < '9999-12-31
↪';
UPDATE 0
db=# alter table employees enable trigger all;
ALTER TABLE
db=# commit;
COMMIT
```

2.8.4 Design

This section discusses the various ways in which one can represent temporal data and attempts to justify the design that this particular extension uses. The first naïve attempts to track the history of a table typically look like this (assuming the structure of the `employees` table from the usage section above):

```
CREATE TABLE employees (
    changed         date NOT NULL,
    emp_id          integer NOT NULL,
    name            varchar(100) NOT NULL,
    dob             date NOT NULL,
    dept            char(4) NOT NULL,
    is_manager      boolean DEFAULT false NOT NULL,
    salary          numeric(8) NOT NULL CHECK (salary >= 0),

    PRIMARY KEY (changed, emp_id)
);
```

Now let's place some sample data in here; the addition of three employees sometime in 2007:

```
INSERT INTO employees VALUES
    ('2007-07-06', 1, 'Tom',   '1976-01-01', 'D001', false, 40000),
    ('2007-07-07', 2, 'Dick',  '1980-03-31', 'D001', true,  80000),
    ('2007-07-01', 3, 'Harry', '1977-12-25', 'D002', false, 35000);
```

Now later in 2007, Harry gets a promotion to manager, and Dick changes his name to Richard:

```
INSERT INTO employees VALUES
    ('2007-10-01', 3, 'Harry',   '1977-12-25', 'D002', true, 70000),
    ('2007-10-01', 2, 'Richard', '1980-03-31', 'D001', true, 80000);
```

At this point we can see that the table is tracking the history of the employees, and we can write relatively simple queries to answer questions about the data. For example, when did Harry get his promotion?

```
SELECT min(changed)
FROM employees
WHERE emp_id = 3
AND salary = 80000;
```

However, other questions are more difficult to answer with this structure. What was Harry's salary immediately before his promotion?

```
SELECT salary
FROM employees e1
WHERE emp_id = 3
AND changed = (
    SELECT max(changed)
    FROM employees e2
    WHERE e1.emp_id = e2.emp_id
    AND e2.salary <> 80000
    );
```

Furthermore, some questions are impossible to answer because one particular operation is not represented in this structure: deletion. Because there's no specific representation for deletion we can't tell the difference between an update and a deletion followed by later re-insertion (with the same key).

This is why *two* dates are required in the history table (or more precisely a date or timestamp *range*). Alternatively we could do something similar to the view produced by `create_history_snapshots()` and place a copy of all the data in the table for every single day that passes. That way the absence of a key on a given day would indicate deletion. Obviously this method is extremely wasteful of space, and thus very slow in practice.

Another alternative, similar to the view produced by `create_history_changes()` is to add another field indicating the change that occurred, e.g.:

```
CREATE TABLE employees (
    changed         date NOT NULL,
    change          char(6) NOT NULL,
    emp_id          integer NOT NULL,
    name            varchar(100) NOT NULL,
    dob             date NOT NULL,
    dept            char(4) NOT NULL,
    is_manager      boolean DEFAULT false NOT NULL,
    salary          numeric(8) NOT NULL CHECK (salary >= 0),

    PRIMARY KEY (changed, emp_id),
    CHECK (change IN ('INSERT', 'UPDATE', 'DELETE'))
);
```

Note that without the duplication of fields for before and after values, this makes the structure more space efficient but actually makes querying it very difficult for certain questions. Furthermore, it's quite difficult to transform this structure into the date-range structure required to answer the question "what did the table look like at time X?".

Hopefully the above exploration of alternate structures has convinced you that the simplest, most flexible, and most space efficient representation of temporal data is the date-range structure used by the functions in this extension. It is worth noting that in all implementations of temporal data storage that the author is aware of (DB2's time travel queries, Teradata's T-SQL2 implementation, and Oracle's flashback queries) date ranges are used in the underlying storage.

The following sections summarize the advantages and disadvantages of the design of this particular temporal data implementation.

Advantages

- Simplicity: because the base table is not altered in any way, no operations against that table need to change. Nor do any views that rely on that table, or any APIs that reference it.
- Security: as a separate table is used to store the history, and that table is not directly manipulable by users, the history can be "trusted" to a greater degree than a system which relies upon a single table or one in which the users can directly manipulate the history table.
- Performance and space: the date-range representation of temporal data is minimal compared to other designs, although not perfectly minimal (see next section).
- Performance and space: this system provides a wide variety of resolutions for the history table and triggers. In the case that every single update does not need to be kept (and generally this is not a requirement for many reporting databases) this permits one to keep a minimal history to maintain performance.

The above qualities make this extension useful for data science and business intelligence purposes, especially where data gathering is a long term exercise. It can also be used for general purpose temporal data storage, although it does not provide all the facilities provided by the aforementioned implementations bundled with the major commercial engines.

Disadvantages

- Performance: naturally all operations against the base table will take longer with the triggers and history table in place (simply because more work is being done for each operation). Furthermore, performance degradation will gradually increase the larger the history table gets (as each operation will involve a lookup in a larger and larger index). Administrators are encouraged to keep an eye on operational performance over time and implement archiving when necessary.
- Space: the history table is not a perfectly minimal representation of the history. Certain combinations of operations, in particular removing and inserting the same set of rows from the base table repeatedly, result in an extremely bloated history table (containing many contiguous rows representing the same state). Furthermore, it can be argued that the current row in the history table is a redundant duplicate of the equivalent row in the base table, which also wastes space. Whilst this is true, the alternative (performing a union of the base and history tables each time a temporal query is required) introduces considerable complexity.

This particular extension is unsuitable for creating audit mechanisms. Firstly it does not keep track of which user made which inserts or updates. Secondly, even if such functionality were added to the base table there would be no way of representing who performed a deletion (as there's no row in the history table representing deletions; they are represented by a shortened effective range). Thirdly, and crucially for an audit system, it provides no means of storing details of operations that *failed*.

2.8.5 API

`history.`**`create_history_table`**`(`*source_schema*, *source_table*, *dest_schema*, *dest_table*, *dest_tbspace*, *resolution*`)`
`history.`**`create_history_table`**`(`*source_table*, *dest_table*, *dest_tbspace*, *resolution*`)`

history.**create_history_table** (*source_table*, *dest_table*, *resolution*)
history.**create_history_table** (*source_table*, *resolution*)

Parameters

- **source_schema** – The schema containing the base table. Defaults to the current schema if omitted.
- **source_table** – The table to use as a basis for the history table.
- **dest_schema** – The schema that the history table is to be created in. Defaults to the current schema if omitted.
- **dest_table** – The name of the history table. Defaults to the name of the source table with the suffix `_history` if omitted.
- **dest_tbspace** – The tablespace in which to create the history table. Defaults to the tablespace of the source table if omitted.
- **resolution** – The resolution of the history that is to be stored, e.g. 'day', 'microsecond', 'hour', 'week', etc.

history.**create_history_triggers** (*source_schema*, *source_table*, *dest_schema*, *dest_table*, *resolution*, *offset*)
history.**create_history_triggers** (*source_table*, *dest_table*, *resolution*, *offset*)
history.**create_history_triggers** (*source_table*, *resolution*, *offset*)
history.**create_history_triggers** (*source_table*, *resolution*)

Parameters

- **source_schema** – The schema containing the base table. Defaults to the current schema if omitted.
- **source_table** – The table to use as a basis for the history table.
- **dest_schema** – The schema that the history table is to be created in. Defaults to the current schema if omitted.
- **dest_table** – The name of the history table. Defaults to the name of the source table with the suffix `_history` if omitted.
- **resolution** – The resolution of the history that is to be stored, e.g. 'day', 'microsecond', 'hour', 'week', etc.
- **offset** – An interval which specifies an offset to apply to all timestamps recorded in the history table. Defaults to no offset if omitted.

history.**drop_history_triggers** (*source_schema*, *source_table*)
history.**drop_history_triggers** (*source_table*)

Parameters

- **source_schema** – The schema containing the base table. Defaults to the current schema if omitted.
- **source_table** – The table to use as a basis for the history table.

history.**create_history_changes** (*source_schema*, *source_table*, *dest_schema*, *dest_view*)
history.**create_history_changes** (*source_table*, *dest_view*)
history.**create_history_changes** (*source_table*)

Parameters

- **source_schema** – The schema containing the history table. Defaults to the current schema if omitted.
- **source_table** – The history table on which to base the changes view.
- **dest_schema** – The schema in which to create the changes view. Defaults to the current schema if omitted.

- **`dest_view`** – The name of the new changes view. Defaults to the history table's name with `_history` replaced with `_changes`.

`history.`**`create_history_snapshots`**(*source_schema*, *source_table*, *dest_schema*, *dest_view*, *resolution*)
`history.`**`create_history_snapshots`**(*source_table*, *dest_view*, *resolution*)
`history.`**`create_history_snapshots`**(*source_table*, *resolution*)

Parameters

- **`source_schema`** – The schema containing the history table. Defaults to the current schema if omitted.
- **`source_table`** – The history table on which to base the snapshots view.
- **`dest_schema`** – The schema in which to create the snapshots view. Defaults to the current schema if omitted.
- **`dest_view`** – The name of the new snapshots view. Defaults to the history table's name with `_history` replaced with `_by_` and the resolution.
- **`resolution`** – The resolution of the snapshots to be generated in the view. This must be longer than the resolution of the history table.

2.9 The MIT License (MIT)

CHAPTER 3

Indices and Tables

- genindex
- search

Python Module Index

Index