abridger Documentation

Release 0.1.1

Will Angenent

July 24, 2016

Contents

CHAPTER 1

About abridger

Abridger is a tool for creating a referentially intact copy of a subset of a database.

Contents:

1.1 Overview

The objective is to create a new database with a referentially intact subset of data from another database. The schema of the database is identical, but the data is different. Creating a new database with the same schema as an old database is easy, however copying just some of the data can be tricky due to the database's relational nature.

It all comes down to defining rules on what to extract. If the rules are too strict, then not enough data is copied. If the rules are not strict enough, too much data is copied. Furthermore, for highly complex databases, it can become quite a task to define the rules and combine them in a sensible way.

The rules are defined in an Extraction Model which is configured in one or more yaml files.

1.1.1 Concepts

Extraction model A collection of rules describing what to extract.

Subject An extraction model has one or more subjects. A subject is a collection of tables and relations. See *Subjects* for more information.

Table A subject has one or more tables. A table can either be an extraction of all rows in the table, or filtered rows using a column and list of values. A table can also be set as top-level and be applied to all subjects. See *Tables* for more information.

Relation A relation is a reference to a database foreign key. A relation is either *outgoing* or *incoming* from the perspective of a subject. Relations can be *disabled* or made *sticky*. See *Relations* for more information.

Default relations By default, any row found in a table in the extraction model is fetched in its entirety. This will pull in rows required to satisfy any foreign key constraints on the row's table. Rows in other tables referencing the source table aren't fetched by default. These defaults can be overridden. See *Defaults* for more information.

Not null columns When populating the destination database or generating SQL, nullable columns can be treated as not null so that they are included in the insert statements. This is useful if check constraints are used. See *Not Null Columns* for more information.

1.2 Getting Started

1.2.1 Installation

The code is hosted on GitHub. Abridger should be installed with python's pip installer.

If you don't have pip installed, run:

```
$ sudo easy_install pip
```

Root installation

Installation using pip

```
$ sudo pip install abridger
```

Install from github

```
$ git clone https://github.com/freewilll/abridger
$ cd abridger
$ sudo python setup.py install
```

Or alternatively, you can do it in one step:

```
$ sudo pip install git+https://github.com/freewilll/abridger
```

If you wish to use postgresql, install the psycopg2 package:

```
$ sudo pip install psycopg2
```

Non-root installation

If you would rather not install it as root, you can use `virtualenv` to install a local copy

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install abridger

If you want to use postgresql
$ pip install psycopg2
```

1.2.2 Quick start

In the following example, a test sqlite3 database will be created with some tables and some data. An extraction is shown using all relations as a default.

Create a test database

Create a file called `test-input.sql` and put the following in it:

```
CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department_id INTEGER NOT NULL REFERENCES departments
);

INSERT INTO departments (id, name) VALUES
    (1, 'Research'),
    (2, 'Accounting');

INSERT INTO employees (id, name, department_id) VALUES
    (1, 'John', 1),
    (2, 'Jane', 1),
    (3, 'Janet', 2);
```

Load `test-input.sql` into an sqlite3 database called `test-db.sqlite3`

```
$ sqlite3 test-db.sqlite3 < test-input.sql
```

The contents of the test database

```
$ sqlite3 -header -column test-db.sqlite3 'SELECT e.*, d.name as department_name FROM employees e joi

id          name        department_id  department_name
----------  ----------  -------------  ---------------
1           John        1              Research
2           Jane        1              Research
3           Janet       2              Accounting
```

Create an extraction config file

In this example, we'll fetch the `Research` department, which will also fetch all employees in it. Create a file called `getting-started-config.yaml` and put the following in it:

```
- relations:
  - { defaults: everything}
- subject:
  - tables:
    - {table: departments, column: name, values: Research}
```

Run abridger

```
$ abridge-db getting-started-config.yaml sqlite:///test-db.sqlite3 -f test-output.sql

Connecting to sqlite:///test-db.sqlite3
Querying...
Extraction completed: fetched rows=4, tables=2, queries=3, depth=2, duration=0.0 seconds
Writing SQL for 3 inserts and 0 updates in 2 tables...
Done
```

Results

```
$ cat test-output.sql

BEGIN;
INSERT INTO departments (id, name) VALUES(1, 'Research');
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
COMMIT;
```

1.2.3 Running abridger

Usage: `abridge-db [-h] [-u URL] [-f FILE] [-e] [-q] [-v] CONFIG_PATH SRC_URL`

positional arguments:

CONFIG_PATH	path to extraction config file
SRC_URL	source database url

optional arguments:

-h, --help	show this help message and exit
-u URL, --url URL	destination database url
-f FILE, --file FILE	destination database file. Use - for stdout
-e, --explain	explain where rows are coming from
-q, --quiet	don't output anything
-v, --verbose	verbose output

Unless `--explain` is being used, exactly one of `--file` and `--url` must be specified. Use `--file -` to output the SQL results to stdout.

Note that using `--explain` is very inefficient since the extractor will do one query for each row.

Examples

Extract data from a postgresql database and add it to another

```
abridge-db config.yaml postgresql://user@localhost/test -u postgresql://user@localhost/abridged_test
```

Extract data from a postgresql database and write an sql file

```
abridge-db config.yaml postgresql://user@localhost/test -f test-postgresql.sql
```

Extract data from a sqlite3 database and output SQL to stdout

```
abridge-db config.yaml sqlite:///test-db.sqlite3 -q -f -
```

1.3 Extraction Model

1.3.1 Subjects

An extraction model consists of one more more *subjects*. Each subject has its own set of *tables* and *relations*. Relations can also be global, which allows setting of global defaults that can be overridden by subjects. By *default*, any row

found in a table in the extraction model is fetched in its entirety.

1.3.2 Tables

A table on a subject consists of the following:

table The name of the table

column The name of the column to extract values out of, when used together with `values`

values A single number of string *or* an array of numbers or strings

Examples:

Example	Description
All departments	A table entry with just a table name will fetch all rows for that table
One department	A table entry with a single column/value will fetch one row
Two departments	A table entry with multiple column/values will fetch multiple rows
Two tables	A table entry with multiple column/values will fetch multiple rows
Two subjects	Two subjects with one table each

1.3.3 Relations

A relation enables or disables the processing of a foreign key in the database schema. A relationship is `incoming` or `outgoing` as seen from the perspective of a subject. All `outgoing` not null foreign keys *must* be processed to satisfy the foreign key constraint. This type of relationship is therefore always enabled and cannot be disabled,

A relation can be applied globally or to a subject. A global relation is always included in all subjects. Relations in a subject are only processed on rows related to the subject. See Extraction for more information.

defaults Add all relations from a couple of selected types. See *defaults* for more details.

table A foreign key constraint is identified by specifying a `table` and `column` in a relation. The first foreign key relationship to match the table and column is used. Compound foreign keys are fully supported, but can only be identified by a specifying a single column.

column Must be specified when using `table` to identify a foreign key.

type One of `incoming` or `outgoing`, with `incoming` the default. This identifies the direction of a relationship from the perspective of a subject.

name Optional and purely for informational purposes.

disabled Foreign key relations can be disabled. This is useful in the blacklisting approach where the `everything` default is used and then relations disabled.

sticky Sticky relations can be used to keep track of which rows are directly connected to the subject. See *sticky relations* for more details.

Compound keys are also supported, see e.g. *Compound Foreign Keys*

A relationship is uniquely identified by its `table`, `column`, `type` and `name`. Identical relationships are processed in order and merged according to the following rules:

- If any relation is disabled, then the relation is disabled and not processed.
- If any relation is sticky, then the relation is sticky.

Examples:

Example	Description
Default relations for a department	Default relations for a department
Incoming Relation	Incoming Relation
Relation for two departments 1	Default relations for two departments
Relation for two departments 2	Alternative default relations for two departments
Relation for two departments 3	Another alternative default relations for two departments
Relation for an employee	All relations
Outgoing relation	Outgoing relation
Disabled incoming relation	Blacklisting approach with a disabled incoming relation
Disabled outgoing relation	Blacklisting approach with a disabled outgoing relation

1.3.4 Defaults

Default relations can be set by using the `relations default` key. There are four default settings that can be combined in an additive way:

Setting	Default	Meaning
all-outgoing-not-null	yes	Always satisfy not null foreign key constraints
all-outgoing-nullable	yes	Ensures that complete rows are fetched
all-incoming	no	Process incoming foreign keys
everything	no	All of the above

If no defaults are specified, a single relation of type `all-outgoing-nullable` is used. The `all-outgoing-not-null` default is always present. The combination of these two ensures that whenever a row is encountered, all outgoing foreign keys are processed. This causes rows referenced by the foreign key to be fetched.

This is the default setting:

```
- relations:
  - {defaults: all-outgoing-not-null}
  - {defaults: all-outgoing-nullable}
```

To add all incoming relations to the default, use:

```
- relations:
 - {defaults: everything}
```

Since `all-outgoing-not-null` is always included implicitly, the above is equivalent to:

```
- relations:
  - {defaults: all-outgoing-nullable}
  - {defaults: all-incoming}
```

Use this to disable all relations except the minimal required `all-outgoing-not-null`:

```
- relations:
  - {defaults: all-outgoing-not-null}
```

Setting default relations is useful when using the blacklisting approach. See *Disabled incoming relation* and *Disabled outgoing relation*.

Examples:

Example	Description
all-outgoing-not-null	all-outgoing-not-null
all-outgoing-nullable	all-outgoing-nullable
all-incoming	all-incoming
all-incoming and all-outgoing-nullable	all-incoming and all-outgoing-nullable
everything	everything

1.3.5 Includes

Yaml files can be included in each other using the `include` directive. For example having this in a top level file:

```
- include basic-tables.yaml
- subject:
  - tables:
    - {table: departments}
```

and this in another file called `basic-tables.yaml`

```
- subject:
  - tables:
    - {table: building_types}
    - {table: something_essential}
```

is equivalent to:

```
- subject:
  - tables:
    - {table: building_types}
    - {table: something_essential}
- subject:
  - tables:
    - {table: departments}
```

Includes can be done anywhere, so that e.g. a common file of relations can be defined and then used in several subjects like so:

```
- subject:
  - tables:
    - {table: employees, column: name, values: ['John', 'Jane']}
    - include employee-relations.yaml
- subject:
  - tables:
    - {table: departments, column: name, values: ['Research']}
    - include employee-relations.yaml
```

This is useful in complex databases where several combinations of relations and tables could get combined in several subjects.

1.3.6 Sticky relations

What can quickly happen when doing an extraction with a complicated database schema is an explosion of data. In many of these cases, just enabling a foreign key relationship can pull in lots of unwanted data. An easy solution to prevent this is to make use of the `sticky` relations. When this flag is set on a relation, then the relation is *only* processed if there is a direct graph of sticky relations back to a subject. The rules of transmitting stickiness are:

- Subject's table's rows start off being sticky

- Non-sticky relations are always processed, however the fetched rows aren't marked sticky. This is the default behavior.
- A sticky relationship is only processed if the row is sticky
- Stickiness is only transmitted if a) the row is sticky and b) the relationship is sticky

This behavior can be summarized in a table:

Fetched row sticky	Relationship sticky	Relationship is processed	Processed row sticky
No	No	Yes	No
Yes	No	Yes	No
No	Yes	No	-
Yes	Yes	Yes	Yes

See *Sticky Relations* for an example.

1.4 Extraction

Extraction is done by keeping track of a queue of *work items* for each subject. The work items queue starts with the subjects and grows as rows in new tables are added by processing relations. The procedure is complete as soon as the queue is empty.

1.4.1 The procedure

The procedure is as follows

1. Add all subject tables/columns/values to the work item queue
2. Fetch an item from the queue
3. Skip the item if the table, column, subject and values have already been processed
4. Query for the table/column/values
5. For each row, process the subject's relationships
6. For each row, null any nullable foreign keys that didn't have their relationship processed
7. If a row has been seen in a previous iteration, merge in any not null values for columns that may have been made null. This ensures that if a row is seen twice and the second time is processed with more relationships, then the final row contains foreign key values for the new relationships.
8. Repeat step 2 if the queue isn't empty

1.4.2 Identical rows

Identical rows for a table are processed by using an *effective primary key*. This is:

1. The table's primary key, if available
2. Otherwise, if available, the first discovered most restrictive unique index
3. Otherwise, the entire row is treated as unique, but duplicate rows are allowed. Duplicates are counted and the row will be inserted the correct amount of times.

1.4.3 Using explain

When running from the command line, use the `--explain` option to get a detailed view of the extraction procedure. The output of the script will have details about each query and processed relationships.

When running with explain, a query is done for each individual row instead of batching them using SQL `IN` statements. This makes the procedure much slower, but this is needed to be able to identify exactly where a row is coming from. The Examples all contain the output of `--explain`.

1.5 SQL Generation

SQL generation uses the fetched and processed rows from the extraction and converts them into `INSERT` and `UPDATE` statements. The insert statements are done in order so that not null foreign keys are respected.

1.5.1 Not Null Columns

If an insert statement cannot be done without violating foreign key constraints due to nullable foreign keys, then it is split into an insert and update statement. If a nullable foreign key cannot be made null due to e.g. a `CHECK` constraint, then a simple rule can be added which tells the SQL generator to treat that column as if it were not null.

Examples:

Example	Description
Not Null Columns	Not Null Columns
Not Null Columns Switched	Not Null Columns Switched

1.6 Databases

Two databases are supported: sqlite and postgresql. The database URLs follow the django url convention. The following features are supported in both databases:

- Schema parsing of tables, columns, primary keys, foreign keys and unique indexes
- Compound primary and foreign keys
- SQL generation

1.6.1 Sqlite

Use the `sqlite:///` prefix in front of the path name.

For a relative path use e.g.

```
sqlite:///test-db.sqlite3
```

For an absolute path use e.g.

```
sqlite:////var/lib/databases/test-db.sqlite3
```

1.6.2 Postgresql

A full postgresql URL is something like:

```
postgresql://user:password@host:port/dbname
```

`host` and `dbname` are required and `password` and `port` are optional. This is e.g. a valid url

```
postgresql://test_user@localhost/test_database
```

The generated SQL always starts with a `BEGIN`, ends with a `COMMIT` and has an extra `\set ON_ERROR_STOP` for convenience, so that a full SQL result looks something like:

```
BEGIN;
\set ON_ERROR_STOP
INSERT INTO ...
COMMIT;
```

1.7 Examples

1.7.1 Subjects

Schema

```
CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT
);

INSERT INTO departments (id, name) VALUES
    (1, 'Research'),
    (2, 'Accounting'),
    (3, 'Finance');
```

All departments

A table entry with just a table name will fetch all rows.

Config

```
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
```

Results

```
INSERT INTO departments (id, name) VALUES(1, 'Research');
INSERT INTO departments (id, name) VALUES(2, 'Accounting');
INSERT INTO departments (id, name) VALUES(3, 'Finance');
```

One department

A table entry with a single column/value will fetch one row.

Config

```
- subject:
  - tables:
    - {column: name, table: departments, values: Research}
```

Explain output

```
departments.name=Research*
```

Results

```
INSERT INTO departments (id, name) VALUES(1, 'Research');
```

Two departments

A table entry with multiple column/values will fetch multiple rows.

Config

```
- subject:
  - tables:
    - column: name
      table: departments
      values: [Research, Accounting]
```

Explain output

```
departments.name=Research*
```

Results

```
INSERT INTO departments (id, name) VALUES(1, 'Research');
INSERT INTO departments (id, name) VALUES(2, 'Accounting');
```

Two tables

A subject can have multiple tables.

Config

```
- subject:
  - tables:
    - {column: name, table: departments, values: Accounting}
    - {column: name, table: departments, values: Research}
```

Explain output

```
departments.name=Accounting*
departments.name=Research*
```

Results

```
INSERT INTO departments (id, name) VALUES(1, 'Research');
INSERT INTO departments (id, name) VALUES(2, 'Accounting');
```

Two subjects

Subjects can have different tables and relations. This example is a bit silly since both departments can be done in the same subject, it just illustrates that things can be broken down.

Config

```
- subject:
  - tables:
    - {column: name, table: departments, values: Accounting}
- subject:
  - tables:
    - {column: name, table: departments, values: Research}
```

Explain output

```
departments.name=Accounting*
departments.name=Research*
```

Results

```
INSERT INTO departments (id, name) VALUES(1, 'Research');
INSERT INTO departments (id, name) VALUES(2, 'Accounting');
```

1.7.2 Defaults

Schema

```
CREATE TABLE buildings (
    id INTEGER PRIMARY KEY,
    name TEXT
);

CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT,
    building_id INTEGER REFERENCES buildings
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department_id INTEGER NOT NULL REFERENCES departments
);

INSERT INTO buildings (id, name) VALUES
    (1, 'London'),
    (2, 'Paris');

INSERT INTO departments (id, name, building_id) VALUES
    (1, 'Research', 1),
    (2, 'Accounting', NULL);

INSERT INTO employees (id, name, department_id) VALUES
    (1, 'John', 1),
```

```
    (2, 'Jane', 1),
    (3, 'Janet', 2);
```

all-outgoing-not-null

`all-outgoing-not-null` is the minimum required relation. Nullable outgoing foreign keys are ignored, as well as incoming foreign keys. Fetching all `departments` will make the `building_id` foreign key `null`. Also, since no incoming relations are in the defaults, no rows in `employees` are fetched. If rows in `buildings` are required, they can be enabled by adding an `outgoing` relation from `departments` to `buildings`. This will also make the `null` go away in the research department. See *Outgoing relation*.

Config

```
- relations:
  - {defaults: all-outgoing-not-null}
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
```

Results

```
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', NULL);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
```

all-outgoing-nullable

`all-outgoing-nullable` ensures that all foreign keys are processed. In this example it means that the `buildings` row with id=1 must be fetched to satisfy the `building_id` foreign key constraint on the `departments` table. Also, since no incoming relations are in the defaults, no rows in `employees` are fetched.

Config

```
- relations:
  - {defaults: all-outgoing-nullable}
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
departments* -> departments.id=1 -> buildings.id=1
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
```

all-incoming

`all-incoming` ensures that for any row that is fetched all *referencing* foreign keys are processed in other tables. In this example it means that that all `employees` with `department_id` in the fetched departments are fetched. Note how no rows in `buildings` are fetched, since `all-outgoing-nullable` wasn't enabled.

Config

```
- relations:
  - {defaults: all-incoming}
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
departments* -> departments.id=1 -> employees.department_id=1
departments* -> departments.id=2 -> employees.department_id=2
departments* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> departments.id=1
departments* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> departments.id=2
```

Results

```
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', NULL);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

all-incoming and all-outgoing-nullable

The combination of `all-outgoing-nullable` and `all-incoming`, which is equivalent to *everything*, fetches everything in `buildings` and `employees` related to all the `departments`.

Config

```
- relations:
  - {defaults: all-outgoing-nullable}
  - {defaults: all-incoming}
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
departments* -> departments.id=1 -> buildings.id=1
departments* -> departments.id=1 -> employees.department_id=1
departments* -> departments.id=2 -> employees.department_id=2
departments* -> departments.id=1 -> buildings.id=1 -> departments.building_id=1
departments* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> departments.id=1
departments* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> departments.id=2
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
```

```
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

everything

This is equivalent to the above *all-incoming and all-outgoing-nullable*

Config

```
- relations:
  - {defaults: everything}
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
departments* -> departments.id=1 -> buildings.id=1
departments* -> departments.id=1 -> employees.department_id=1
departments* -> departments.id=2 -> employees.department_id=2
departments* -> departments.id=1 -> buildings.id=1 -> departments.building_id=1
departments* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> departments.id=1
departments* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> departments.id=2
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

1.7.3 Relations

Schema

```
CREATE TABLE buildings (
    id INTEGER PRIMARY KEY,
    name TEXT
);

CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT,
    building_id INTEGER REFERENCES buildings
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department_id INTEGER NOT NULL REFERENCES departments
```

```
);

INSERT INTO buildings (id, name) VALUES
    (1, 'London'),
    (2, 'Paris');

INSERT INTO departments (id, name, building_id) VALUES
    (1, 'Research', 1),
    (2, 'Accounting', NULL);

INSERT INTO employees (id, name, department_id) VALUES
    (1, 'John', 1),
    (2, 'Jane', 1),
    (3, 'Janet', 2);
```

Default relations for a department

By default, whenever a row is seen, everything is done to ensure the row is complete. Since the `departments` table contains a foreign key on `building_id`, all buildings referenced from departments will be also fetched. However rows referencing the department aren't fetched.

Config

```
- subject:
  - tables:
    - {column: name, table: departments, values: Research}
```

Explain output

```
departments.name=Research*
departments.name=Research* -> departments.id=1 -> buildings.id=1
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
```

Incoming Relation

This does an extraction with a relation from `employees` to `departments`. This will include both employees in the research department. Note how rows in `buildings` are also included since by the default all outgoing foreign keys are fetched. See *Defaults* for more details. The `type` of `incoming` doesn't have to be included in the config since this is the default.

Config

```
- subject:
  - tables:
    - {column: name, table: departments, values: Research}
  - relations:
    - {column: department_id, table: employees}
```

Explain output

```
departments.name=Research*
departments.name=Research* -> departments.id=1 -> buildings.id=1
```

```
departments.name=Research* -> departments.id=1 -> employees.department_id=1
departments.name=Research* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> depa
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
```

Relation for two departments 1

This does an extraction with the above relation, but with both departments. This ends up fetching all employees.

Config

```
- subject:
  - tables:
    - column: name
      table: departments
      values: [Research, Accounting]
  - relations:
    - {column: department_id, table: employees}
```

Explain output

```
departments.name=Research*
departments.name=Research* -> departments.id=1 -> buildings.id=1
departments.name=Research* -> departments.id=1 -> employees.department_id=1
departments.name=Research* -> departments.id=2 -> employees.department_id=2
departments.name=Research* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> depa
departments.name=Research* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> depa
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

Relation for two departments 2

This is equivalent to *Relation for two departments 1*, except it used two separate table sections.

Config

```
- relations:
  - {column: department_id, table: employees}
- subject:
  - tables:
    - {column: id, table: departments, values: 1}
    - {column: id, table: departments, values: 2}
```

Explain output

```
departments.id=1*
departments.id=2*
departments.id=1* -> departments.id=1 -> employees.department_id=1
departments.id=1* -> departments.id=1 -> buildings.id=1
departments.id=2* -> departments.id=2 -> employees.department_id=2
departments.id=1* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> departments.
departments.id=2* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> departments.
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

Relation for two departments 3

This is equivalent to *Relation for two departments 1*, except it used two separate subjects.

Config

```
- relations:
  - {column: department_id, table: employees}
- subject:
  - tables:
    - {column: id, table: departments, values: 1}
- subject:
  - tables:
    - {column: name, table: departments, values: Accounting}
```

Explain output

```
departments.id=1*
departments.name=Accounting*
departments.id=1* -> departments.id=1 -> employees.department_id=1
departments.id=1* -> departments.id=1 -> buildings.id=1
departments.name=Accounting* -> departments.id=2 -> employees.department_id=2
departments.id=1* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> departments.
departments.name=Accounting* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> de
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

Relation for an employee

A subject to fetch the John employee with the `everything` default leads to all employees in the research department being fetched since:

- John belongs to the research department

- All employees in the research department are fetched, which pulls in Jane

Config

```
- relations:
  - {defaults: everything}
- subject:
  - tables:
    - {column: name, table: employees, values: John}
```

Explain output

```
employees.name=John*
employees.name=John* -> employees.id=1 -> departments.id=1
employees.name=John* -> employees.id=1 -> departments.id=1 -> buildings.id=1
employees.name=John* -> employees.id=1 -> departments.id=1 -> employees.department_id=1
employees.name=John* -> employees.id=1 -> departments.id=1 -> buildings.id=1 -> departments.building_
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
```

Outgoing relation

This shows the explicit enabling of an outgoing nullable relation when using the minimal defaults of `all-outgoing-not-null`. Without the relation, no rows in the `buildings` table would be fetched, since the default rules don't including following nullable foreign keys as described on *all-outgoing-not-null*. In this example, the relation is enabled, resulting in building being included.

Config

```
- relations:
  - {defaults: all-outgoing-not-null}
- subject:
  - tables:
    - column: name
      table: departments
      values: [Research, Accounting]
  - relations:
    - {column: building_id, table: departments, type: outgoing}
```

Explain output

```
departments.name=Research*
departments.name=Research* -> departments.id=1 -> buildings.id=1
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
```

Disabled incoming relation

This demonstrates the blacklisting approach. All relations are enabled by default, however the relation from `employees` to `departments` is disabled. Fetching a department will therefore not fetch any of the employees.

This is an `incoming` relationship type from the perspective of the `employees` table. The `type` key doesn't have to be included since the default type is `incoming`. Relations can be disabled globally or per subject.

Config

```
- relations:
  - {defaults: everything}
- subject:
  - tables:
    - column: name
      table: departments
      values: [Research, Accounting]
  - relations:
    - {column: department_id, disabled: true, table: employees}
```

Explain output

```
departments.name=Research*
departments.name=Research* -> departments.id=1 -> buildings.id=1
departments.name=Research* -> departments.id=1 -> buildings.id=1 -> departments.building_id=1
```

Results

```
INSERT INTO buildings (id, name) VALUES(1, 'London');
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', 1);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
```

Disabled outgoing relation

This is another example of the blacklisting approach. All relations are enabled by default, however the relation from `departments` to `buildings` is disabled. Fetching a department will therefore not fetch any of the buildings. This is an `outgoing` relationship type from the perspective of the `departments` table due to the `building_id` foreign key. A side effect of disabling this relation is that `building_id` becomes `null` for the "Research" department, even though the "Research" department is associated with the "London" building.

Config

```
- relations:
  - {defaults: everything}
- subject:
  - tables:
    - column: name
      table: departments
      values: [Research, Accounting]
  - relations:
    - {column: building_id, disabled: true, table: departments, type: outgoing}
```

Explain output

```
departments.name=Research*
departments.name=Research* -> departments.id=1 -> employees.department_id=1
departments.name=Research* -> departments.id=2 -> employees.department_id=2
departments.name=Research* -> departments.id=1 -> employees.department_id=1 -> employees.id=1 -> depa
departments.name=Research* -> departments.id=2 -> employees.department_id=2 -> employees.id=3 -> depa
```

Results

```
INSERT INTO departments (id, name, building_id) VALUES(1, 'Research', NULL);
INSERT INTO departments (id, name, building_id) VALUES(2, 'Accounting', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 1);
INSERT INTO employees (id, name, department_id) VALUES(3, 'Janet', 2);
```

1.7.4 Compound Foreign Keys

Schema

```
CREATE TABLE buildings (
    id1 INTEGER,
    id2 INTEGER,
    name TEXT,
    PRIMARY KEY(id1, id2)
);

CREATE TABLE departments (
    id1 INTEGER,
    id2 INTEGER,
    name TEXT,
    building1_id INTEGER,
    building2_id INTEGER,
    FOREIGN KEY(building1_id, building2_id) REFERENCES buildings,
    UNIQUE(id1, id2)
);

CREATE TABLE employees (
    id1 INTEGER,
    id2 INTEGER,
    name TEXT,
    department1_id INTEGER NOT NULL,
    department2_id INTEGER NOT NULL,
    PRIMARY KEY(id1, id2),
    FOREIGN KEY(department1_id, department2_id) REFERENCES departments(id1, id2)
);

INSERT INTO buildings (id1, id2, name) VALUES
    (1, 1, 'London'),
    (2, 2, 'Paris');


INSERT INTO departments (id1, id2, name, building1_id, building2_id) VALUES
    (1, 1, 'Research', 1, 1),
    (2, 2, 'Accounting', NULL, NULL);

INSERT INTO employees (id1, id2, name, department1_id, department2_id) VALUES
    (1, 1, 'John', 1, 1 ),
    (2, 2, 'Jane', 1, 1),
    (3, 3, 'Janet', 2, 2);
```

Compound Foreign Keys

This extremely convoluted example shows that compound key support is built in. Relations can also be used, but only one column is matched in the foreign key.

Config

```
- relations:
  - {defaults: everything}
- subject:
  - tables:
    - {table: departments}
```

Explain output

```
departments*
departments* -> departments.id1,id2=1,1 -> buildings.(id1,id2)=(1,1)
departments* -> departments.id1,id2=1,1 -> employees.(department1_id,department2_id)=(1,1)
departments* -> departments.id1,id2=2,2 -> employees.(department1_id,department2_id)=(2,2)
departments* -> departments.id1,id2=1,1 -> buildings.(id1,id2)=(1,1) -> buildings.id1,id2=1,1 -> depa
departments* -> departments.id1,id2=1,1 -> employees.(department1_id,department2_id)=(1,1) -> employe
departments* -> departments.id1,id2=2,2 -> employees.(department1_id,department2_id)=(2,2) -> employe
```

Results

```
INSERT INTO buildings (id1, id2, name) VALUES(1, 1, 'London');
INSERT INTO departments (id1, id2, name, building1_id, building2_id) VALUES(1, 1, 'Research', 1, 1);
INSERT INTO departments (id1, id2, name, building1_id, building2_id) VALUES(2, 2, 'Accounting', NULL,
INSERT INTO employees (id1, id2, name, department1_id, department2_id) VALUES(1, 1, 'John', 1, 1);
INSERT INTO employees (id1, id2, name, department1_id, department2_id) VALUES(2, 2, 'Jane', 1, 1);
INSERT INTO employees (id1, id2, name, department1_id, department2_id) VALUES(3, 3, 'Janet', 2, 2);
```

1.7.5 Sticky Relations

Schema

```
CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department_id INTEGER NOT NULL REFERENCES departments
);

ALTER TABLE employees ADD COLUMN boss_id INTEGER REFERENCES employees;

CREATE TABLE addresses (
    id INTEGER PRIMARY KEY,
    employee_id INTEGER NOT NULL REFERENCES employees,
    details TEXT
);

INSERT INTO departments (id, name) VALUES
```

```
    (1, 'Managers'),
    (2, 'Engineers');

INSERT INTO employees (id, name, department_id, boss_id) VALUES
    (1, 'John', 1, NULL),
    (2, 'Jane', 2, 1),
    (3, 'Janet', 2, 2);

INSERT INTO addresses (id, employee_id, details) VALUES
    (1, 1, 'John''s address'),
    (2, 2, 'Jane''s adddress'),
    (3, 3, 'Janet''s first address'),
    (4, 3, 'Janet''s second address');
```

Sticky Relations

Let's say we want to have a database with all engineers in the engineering department and include all engineer's addresses. We want to specifically *not* include manager's addresses. If we were to simply add relations from `employees` to `departments` and `addresses` to `employees`, then any employee's boss would trigger a fetch from the management department, which would lead to all employee's managers being fetched, which would lead to all manager's addresses being fetched.

By making relations *sticky*, they are only processed if there is a sticky trail all the way back to a subject. If we set the sticky flag on the `department_id` and `employee_id` foreign keys, then these relationships are *only* processed if there is a direct trail back to a subject. When the `boss_id` foreign key is processed, the sticky flag is dropped. Therefore, when the "John" employee is processed, the sticky flag has been lost and the sticky relationships aren't used, resulting in no addresses being fetched for John. This can be seen in the explain output in the lines where the employee to boss relationship is processed: the asterisks, indicating stickiness, disappear.

Config

```
- subject:
  - tables:
    - {column: name, table: departments, values: Engineers}
  - relations:
    - {column: department_id, sticky: true, table: employees}
    - {column: employee_id, sticky: true, table: addresses}
```

Explain output

```
departments.name=Engineers*
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2*
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=2* ->
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=3* ->
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=2 -> e
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=3 -> e
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=2 -> d
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=3* ->
departments.name=Engineers* -> departments.id=2* -> employees.department_id=2* -> employees.id=2 -> e
```

Results

```
INSERT INTO departments (id, name) VALUES(1, 'Managers');
INSERT INTO departments (id, name) VALUES(2, 'Engineers');
INSERT INTO employees (id, name, department_id, boss_id) VALUES(1, 'John', 1, NULL);
INSERT INTO employees (id, name, department_id, boss_id) VALUES(2, 'Jane', 2, NULL);
INSERT INTO employees (id, name, department_id, boss_id) VALUES(3, 'Janet', 2, NULL);
INSERT INTO addresses (id, employee_id, details) VALUES(2, 2, 'Jane''s adddress');
```

```
INSERT INTO addresses (id, employee_id, details) VALUES(3, 3, 'Janet''s first address');
INSERT INTO addresses (id, employee_id, details) VALUES(4, 3, 'Janet''s second address');
UPDATE employees SET boss_id=1 WHERE id=2;
UPDATE employees SET boss_id=2 WHERE id=3;
```

1.7.6 Not Null Columns

Schema

```
CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department_id INTEGER REFERENCES departments
);

ALTER TABLE departments ADD COLUMN primary_employee_id INTEGER REFERENCES employees;

INSERT INTO departments (id, name) VALUES
    (1, 'Managers'),
    (2, 'Engineers');

INSERT INTO employees (id, name, department_id) VALUES
    (1, 'John', 1),
    (2, 'Jane', 2),
    (3, 'Janet', NULL);

UPDATE departments SET primary_employee_id=1 WHERE id=1;
```

Not Null Columns

In this example, two tables, `departments` and `employees` both reference each other with nullable foreign keys. It makes no difference to the SQL generation which table comes first when generating the `INSERT` statements, so they are processed in alphabetical order with the `departments` table getting insert statements generated first. The value for `primary_employee_id` can't be added until the rows have been inserted in to the `employees` table so an `UPDATE` statement for `departments` is needed after the `employees` rows have been inserted.

Config

```
- subject:
  - tables:
    - {table: departments}
  - relations:
    - {column: department_id, table: employees}
```

Explain output

```
departments*
departments* -> departments.id=1 -> employees.id=1
departments* -> departments.id=1 -> employees.department_id=1
```

```
departments* -> departments.id=2 -> employees.department_id=2
departments* -> departments.id=1 -> employees.id=1 -> departments.id=1
departments* -> departments.id=2 -> employees.department_id=2 -> employees.id=2 -> departments.id=2
```

Results

```
INSERT INTO departments (id, name, primary_employee_id) VALUES(1, 'Managers', NULL);
INSERT INTO departments (id, name, primary_employee_id) VALUES(2, 'Engineers', NULL);
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', 1);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', 2);
UPDATE departments SET primary_employee_id=1 WHERE id=1;
```

Not Null Columns Switched

Let's pretend there is some reason why a row can't be inserted into `departments` without `primary_employee_id` being set due to a `CHECK` constraint. The SQL generation engine can be given a hint by using a `not-null-columns` rule. This results in the tables being processed in reverse order with `employees` getting inserts first, then `departments`. This results in the rows in the `departments` table having `primary_employee_id` set in the `INSERT` statement. The consequence of this is that the `department_id` on `employees` must be set late with an `UPDATE` statement.

Config

```
- not-null-columns:
  - {column: primary_employee_id, table: departments}
- subject:
  - tables:
    - {table: departments}
  - relations:
    - {column: department_id, table: employees}
```

Explain output

```
departments*
departments* -> departments.id=1 -> employees.id=1
departments* -> departments.id=1 -> employees.department_id=1
departments* -> departments.id=2 -> employees.department_id=2
departments* -> departments.id=1 -> employees.id=1 -> departments.id=1
departments* -> departments.id=2 -> employees.department_id=2 -> employees.id=2 -> departments.id=2
```

Results

```
INSERT INTO employees (id, name, department_id) VALUES(1, 'John', NULL);
INSERT INTO employees (id, name, department_id) VALUES(2, 'Jane', NULL);
INSERT INTO departments (id, name, primary_employee_id) VALUES(1, 'Managers', 1);
INSERT INTO departments (id, name, primary_employee_id) VALUES(2, 'Engineers', NULL);
UPDATE employees SET department_id=1 WHERE id=1;
UPDATE employees SET department_id=2 WHERE id=2;
```