

---

# **pimdb**

***Release 0.2.3***

**Thomas Aglassinger**

**May 02, 2020**



## TABLE OF CONTENTS

<b>1</b>	<b>License</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Usage</b>	<b>7</b>
3.1	Downloading datasets . . . . .	7
3.2	Transferring datasets into tables . . . . .	7
3.3	Querying tables . . . . .	7
3.4	Databases other than SQLite . . . . .	8
3.5	Building normalized tables . . . . .	8
3.6	Querying normalized tables . . . . .	8
3.7	Reference . . . . .	9
<b>4</b>	<b>Data model</b>	<b>11</b>
4.1	Dataset tables . . . . .	11
4.2	Normalized tables . . . . .	12
<b>5</b>	<b>Contributing</b>	<b>19</b>
5.1	Project setup . . . . .	19
5.2	Testing . . . . .	19
5.3	Test run with PostgreSQL . . . . .	20
5.4	Documentation . . . . .	20
5.5	Coding guidelines . . . . .	20
<b>6</b>	<b>Changes</b>	<b>21</b>
<b>7</b>	<b>Indices and tables</b>	<b>23</b>
	<b>Index</b>	<b>25</b>



Pimdb is a python package and command line utility to maintain a local copy of the essential parts of the [Internet Movie Database](#) (IMDb) based in the TSV files available from [IMDb datasets](#).



## **LICENSE**

The [IMDb datasets](#) are only available for personal and non-commercial use. For details refer to the previous link.

Pimdb is open source and distributed under the [BSD license](#). The source code is available from <https://github.com/roskakori/pimdb>.





## INSTALLATION

Pimdb is available from [PyPI](#) and can be installed using:

```
$ pip install pimdb
```



## 3.1 Downloading datasets

To download the current IMDb datasets to the current folder, run:

```
pimdb download all
```

(This downloads about 1 GB of data and might take a couple of minutes).

## 3.2 Transferring datasets into tables

To import them in a local SQLite database `pimdb.db` located in the current folder, run:

```
pimdb transfer all
```

(This will take a while. On a reasonably modern laptop with a local database you can expect about 2 hours).

The resulting database contains one tables for each dataset. The table names are PascalCase variants of the dataset name. For example, the data from the dataset `title.basics` are stored in the table `TitleBasics`. The column names in the table match the names from the datasets, for example `TitleBasics.primaryTitle`. A short description of all the datasets and columns can be found at the download page for the [IMDb datasets](#).

## 3.3 Querying tables

To query the tables, you can use any database tool that supports SQLite, for example the freely available and platform independent community edition of [DBeaver](#) or the [command line shell for SQLite](#).

For simple queries you can also use **pimdb**'s built-in query command, for example:

```
pimdb query "select count(1) from TitleBasics"
```

The result is shown on the standard output and can be redirected to a file, for example:

```
pimdb query "select primaryTitle, startYear from TitleBasics limit 10" >some.tsv
```

You can also store a query in a text file and specify the path:

```
pimdb query --file some-select.sql >some.tsv
```

## 3.4 Databases other than SQLite

`--database DATABASE`

Optionally you can specify a different database using the `--database` option with an [SQLAlchemy engine configuration](#), which generally uses the template “dialect+driver://username:password@host:port/database”. SQLAlchemy supports several SQL [dialects](#) out of the box, and there are external dialects available for other SQL databases and other forms of tabular data.

Here’s an example for using a [PostgreSQL](#) database:

```
pimdb transfer --database "postgresql://user:password@localhost:5432/mydatabase" all
```

## 3.5 Building normalized tables

The tables so far are almost verbatim copies of the IMDb datasets with the exception that possible duplicate rows have been removed. This means that `NameBasics.nconst` and `TitleBasics.tconst` are unique, which sadly is not always (but still sometimes) the case for the datasets in the `.tsv.gz` files.

This data model already allows to perform several kinds of queries quite easily and efficiently.

However, the IMDb datasets do not offer a simple way to query N:M relations. For example, the column `NameBasics.knownForTitles` contains a comma separated list of `tconsts` like “tt2076794,tt0116514,tt0118577,tt0086491”.

To perform such queries efficiently you can build strictly normalized tables derived from the dataset tables by running:

```
pimdb build
```

If you did specify a `--database` for the `transfer` command before, you have to specify the same value for `build` in order to find the source data. These tables generally use snake\_case names for both tables and columns, for example `title_allias.is_original`.

## 3.6 Querying normalized tables

N:M relations are stored in tables using the naming template `some_to_other`, for example `name_to_known_for_title`. These relation tables contain only the numeric ID’s to the respective actual data and a numeric column ordering to remember the sort order of the comma separated list in the IMDb dataset column.

For example, here is an SQL query to list the titles Alan Smithee is known for:

```
select
    title.primary_title,
    title.start_year
from
    name_to_known_for_title
join name on
    name.id = name_to_known_for_title.name_id
join title on
    title.id = name_to_known_for_title.title_id
where
    name.primary_name = 'Alan Smithee'
```

To list all movies and actors that played a character named “James Bond”:

Listing 1: Movies with a character named “James Bond” and the respective actor

```
select
  title.primary_title as "Title",
  title.start_year as "Year",
  name.primary_name as "Actor",
  "character".name as "Character"
from
  "character"
join participation_to_character on
  participation_to_character.character_id = "character".id
join participation on
  participation.id = participation_to_character.participation_id
join name on
  name.id = participation.name_id
join title on
  title.id = participation.title_id
join title_type on
  title_type.id = title.title_type_id
where
  "character".name = 'James Bond'
  and title_type.name = 'movie'
order by
  title.start_year,
  name.primary_name,
  title.primary_title
```

## 3.7 Reference

To get an overview of general command line options and available commands run:

```
pimdb --help
```

To learn the available command line options for a specific command run for example:

```
pimdb transfer --help
```



## DATA MODEL

The tables created by `pimdb` are part of two different data models:

1. The dataset model as created by `pimdb transfer`.
2. The normalized model as created by `pimdb build` and derived from the tables of the dataset model.

The main difference is that the tables dataset model are basically a copy of the flat datasets, while the normalized model has normalized relations and several data quality and naming issues cleaned up.

This chapter describes both kinds of tables and gives examples on how to query them.

### 4.1 Dataset tables

The data are transferred “as is” except that **duplicates** are skipped. Most of the time the datasets do not include duplicates, but every once in a while they do, especially `names.basic.tsv.gz`.

You can find a short description of the datasets and the available fields at the page about the [IMDb datasets](#).

Dataset tables have their names in PascalCase, for example `TitleBasics` while the field names preserve the original camelCase, for example `runtimeMinutes`.

Typically queries will start from either `NameBasics` or `TitleBasics` and from there join into other tables. Data about names are connected with the field `nconst` while data about titles use `tconst`. The only exception is `TitleAkas` which has its `tconst` stored in `titleId`.

The tables have **no foreign key relations** to other tables for the following reasons:

1. At the time of this writing, the datasets include minor inconsistencies that would break the data import due to unresolvable foreign key relations.
2. Without foreign key relations is easily possible to `pimdb transfer` multiple tables in parallel.

Here is an example query that lists all the titles directed by Alan Smithee:

Listing 1: Example query: titles directed by Alan Smithee

```
select
  TitleBasics.primaryTitle,
  TitleBasics.startYear
from
  TitleBasics
join TitlePrincipals on
  TitlePrincipals.tconst = TitleBasics.tconst
join NameBasics on
  NameBasics.nconst = TitlePrincipals.nconst
```

(continues on next page)

(continued from previous page)

```

where
  NameBasics.primaryName = 'Alan Smithee'
  and TitlePrincipals.category = 'director'

```

The tables have very few indexes, typically only the key fields `nconst` and `tconst` are indexed. So SQL joins on these fields should be reasonably fast while for example SQL `where` conditions on name fields are pretty slow.

You can add your own indexes at any time but be aware that too many indexes might slow down future runs of **pimdb transfer**. Also they take more space. And finally, if you use the command line option `--drop`, they are removed and you will have to create them again.

## 4.2 Normalized tables

While the dataset table are already sufficient for many simple queries, they have several issues that normalized tables created with **pimdb build** solve:

1. 1:N relations are stored in relation tables instead of comma separated `varchar` fields. For exaple, compare `NameBasics.knownForTitles` with `name_to_known_for_title`.
2. Inconsistent or non-existent references are removed and replaced by clean foreign key relationships.
3. SQL joins can be performed more efficiently using integer `id` fields instead of the `varchar` fields `nconst` and `tconst`. If needed, the latter are still available from `name.nconst` and `title.tconst`.
4. Ratings from `TitleRatings` (dataset: `title.ratings`) have been merged with `TitleBasics` into `title.average_rating` and `title.rating_count`. For titles that have no rating, these values are both 0.

Normalized tables are named using `snake_case`, for example `title_alias`, and so are fields in these tables, for example `primary_title`. This makes it easy to know whether a table is a dataset or normalized.

Let's take a look at these tables and how they are related to each other.

First, there is `name` which contains information about persons that one way or another contributed to a title:


📁 name	
123  id	NOT NULL
ABC nconst	NOT NULL
ABC primary_name	NOT NULL
123 birth_year	
123 death_year	
ABC primary_professions	

Fig. 1: The essential data model for table `name`

Next, there is `title` which is somewhat similar to `name` but contains details about titles. It also has relation to `title_type`, which is a **key table** containing all available title types, for example: “movie”, “tvEpisode” and so on.



Apparently there are only a few different such types, so this table is pretty small. Similarly there is a table `genre` with names like “Action”, “Sci-Fi”, “Documentary” and so on. Because a title can have multiple genres, they are connected in a **relation table** `title_to_genre` in order to represent this M:N relation.

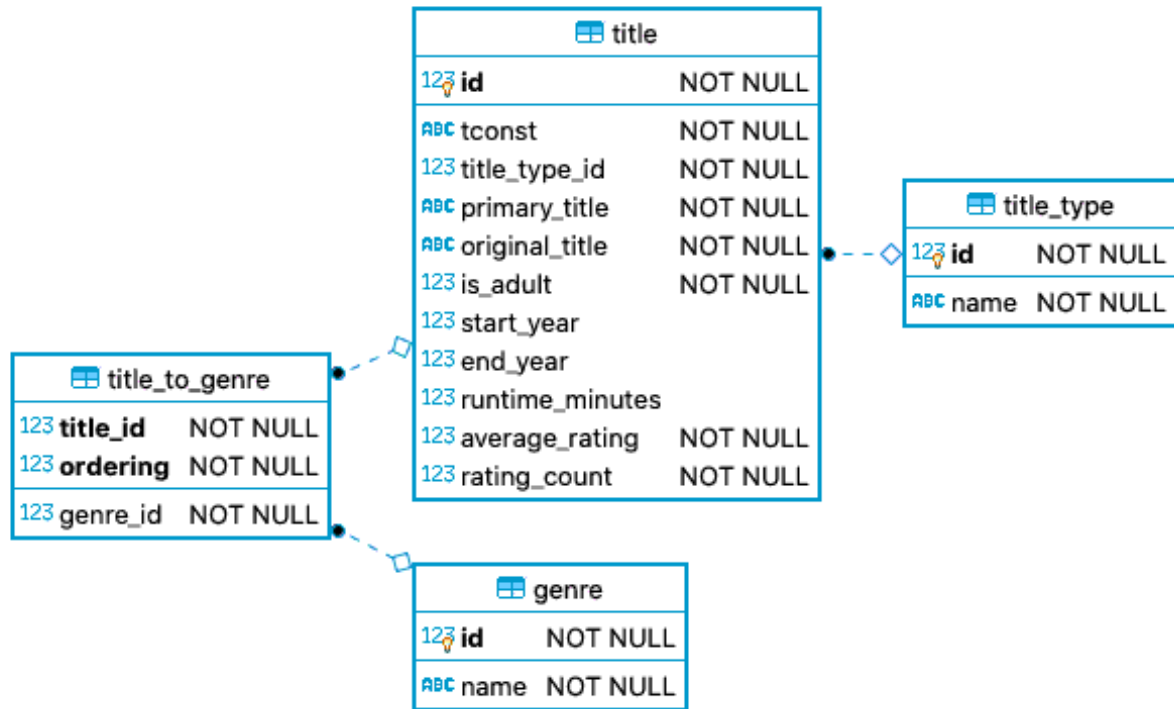


Fig. 2: The essential data model for table `title`

As example here is a query to list the genres of a certain title:

Listing 2: Example query: genres of “Wyrnwood: Road if the Dead”

```

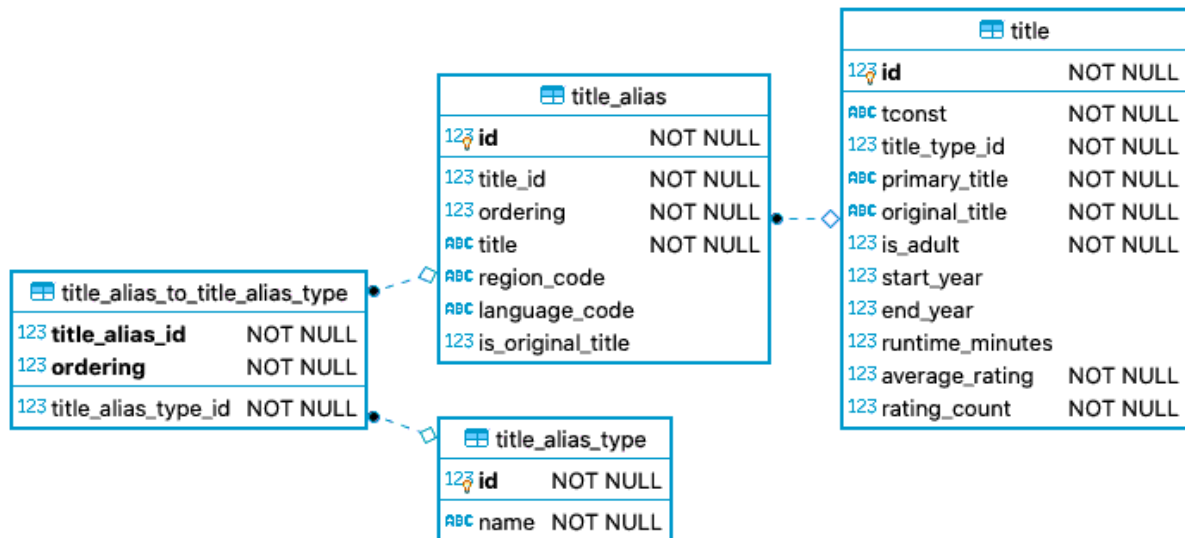
select
  title.tconst,
  title.primary_title,
  genre.name as "genre.name"
from
  title
  join title_to_genre on
    title_to_genre.title_id = title.id
  join genre on
    genre.id = title_to_genre.genre_id
where
  title.tconst = 'tt2535470' -- "Wyrnwood: Road of the Dead"
order by
  title.tconst,
  title_to_genre.ordering

```

The output would be:

tconst	primary_title	genre.name
tt2535470	Wyrnwood: Road of the Dead	Action
tt2535470	Wyrnwood: Road of the Dead	Comedy
tt2535470	Wyrnwood: Road of the Dead	Horror

Similarly, a title be known under different names, for example depending on the country or media released on. A `title_alias` is related to exactly one title and can have multiple aliases. They are connect with the relation table `title_to_alias` and have a certain `title_alias_type` like “dvd”, “tv” or “festival”.

Fig. 3: The essential data model for table `title_alias`

And finally names and titles can be related to each other. A simple variant are the titles a person is known for:

As example, here is a query that lists the titles Alan Smithee is known for:

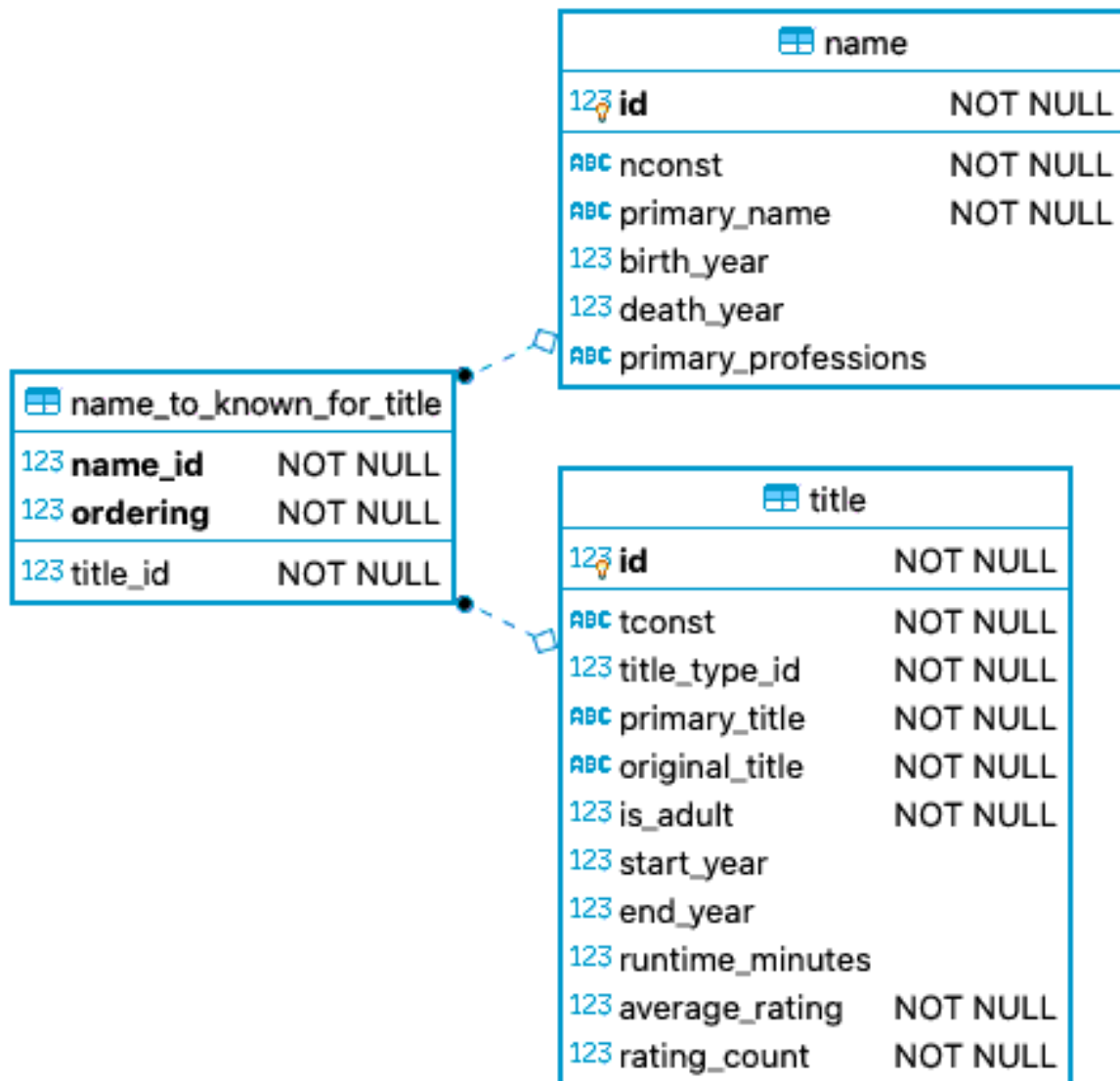


Fig. 4: The data model for known titles of a name.

Listing 3: Example query: titles Alan Smithee is known for

```

select
  title.primary_title,
  title.start_year
from
  name_to_known_for_title
join name on
  name.id = name_to_known_for_title.name_id
join title on
  title.id = name_to_known_for_title.title_id
where
  name.primary_name = 'Alan Smithee'

```

More details on how a person contributed in the making of a title are available via the `participation` table, which connects names and titles with a profession like “actress” or “director”. For professions like “actor” and “actress” there also is information on which character(s) they played in a certain title using the relation table `participation_to_character` and the key table `character`. Unlike most other key table that have only a couple of entries, `character` has about two million.

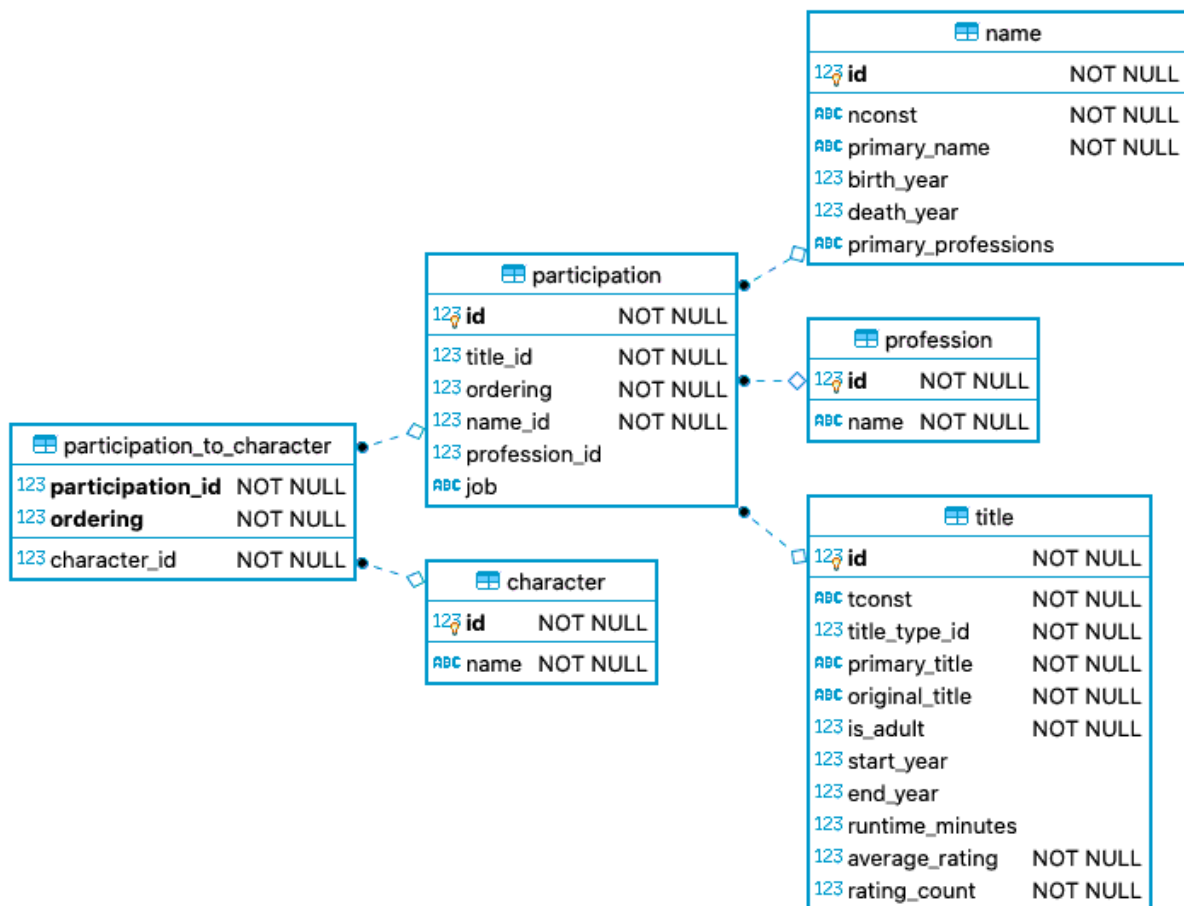


Fig. 5: The data model about who participated in which title and possibly played what character.

Note that not everyone actually played a character as a title typically has many supporting roles. Depending on the

goals of your query you might have to use a `left join` for `participation_to_character`.

Listing 4: Example query: movies with a character named “James Bond”  
and the respective actor

```
select
  title.primary_title as "Title",
  title.start_year as "Year",
  name.primary_name as "Actor",
  "character".name as "Character"
from
  "character"
join participation_to_character on
  participation_to_character.character_id = "character".id
join participation on
  participation.id = participation_to_character.participation_id
join name on
  name.id = participation.name_id
join title on
  title.id = participation.title_id
join title_type on
  title_type.id = title.title_type_id
where
  "character".name = 'James Bond'
  and title_type.name = 'movie'
order by
  title.start_year,
  name.primary_name,
  title.primary_title
```



## CONTRIBUTING

### 5.1 Project setup

In case you want to play with the source code or contribute changes proceed as follows:

1. Check out the project from GitHub:

```
$ git clone https://github.com/roskakori/pimdb.git
$ cd pimdb
```

2. Create and activate a virtual environment:

```
$ python -m venv venv
$ . venv/bin/activate
```

3. Install the required packages:

```
$ pip install --upgrade pip
$ pip install -r requirements.txt
```

4. Install the pre-commit hook:

```
$ pre-commit install
```

### 5.2 Testing

To run the test suite:

```
$ pytest
```

To build and browse the coverage report in HTML format:

```
$ pytest --cov-report=html
$ open htmlcov/index.html # macOS only
```

#### **PIMDB\_TEST\_DATABASE**

By default, all database related tests run on SQLite. Some tests can run on different databases in order to test that everything works across a wide range. To use a specific database, set the respective engine in the environment variable `PIMDB_TEST_DATABASE`. For example:

```
export PIMDB_TEST_DATABASE="postgresql+psycpg2://postgres@localhost:5439/pimdb_test"
```

### PIMDB\_TEST\_FULL\_DATABASE

Some tests require a database built with actual full datasets instead of just small test datasets. Use the environment variable `PIMDB_TEST_FULL_DATABASE` to set it. For example:

```
export PIMDB_FULL_TEST_DATABASE="sqlite:///Users/me/Development/pimdb/pimdb.db"
```

## 5.3 Test run with PostgreSQL

While the test suite uses SQLite, you can test run **pimdb** on a PostgreSQL database in a docker container:

1. Install [Docker Desktop](#)
2. Run the postgres container in port 5439 (possibly using **sudo**):

```
docker-compose --file tests/docker-compose.yml up postgres
```

3. Create the database (possibly using **sudo**):

```
docker exec -it pimdb_postgres psql --username postgres --command "create_  
↪database pimdb"
```

If you want a separate database for the unit tests:

```
docker exec -it pimdb_postgres psql --username postgres --command "create database pimdb_test"
```

4. Run **pimdb**:

```
pimdb transfer --dataset-folder tests/data --database postgresql+psycpg2://  
↪postgres@localhost:5439/pimdb all
```

## 5.4 Documentation

To build the documentation in HTML format:

```
$ make -C docs html  
$ open docs/_build/html/index.html # macOS only
```

## 5.5 Coding guidelines

The code throughout uses a natural naming schema avoiding abbreviations, even for local variables and parameters.

Many coding guidelines are automatically enforced (and some even fixed automatically) by the pre-commit hook. If you want to check and clean up the code without performing a commit, run:

```
$ pre-commit run --all-files
```

In particular, this applies [black](#), [flake8](#) and [isort](#).



## CHANGES

### Version 0.2.3, 2020-05-02

- Fixed `ForeignKeyViolation` when building normalized temporary table `characters_to_character`.
- Fixed `ValueError` when no command was specified for the **pimdb** command line client.

### Version 0.2.2, 2020-04-26

- Fixed `AssertionError` when command line option `--bulk` was less than 1.
- Added `NAME normalized` as option for **pimdb transfer** to transfer only the datasets needed by **pimdb build**.
- Removed redundant normalized tables `title_to(director|writer)`. Use relation `participation`. `profession_id` to limit query results to certain professions.
- Added documentation chapter explaining the *Data model* including example SQL queries and overview ER diagrams.
- Added automatic removal of temporary tables only needed to build the normalized tables.

### Version 0.2.1, 2020-04-18

- Improved performance of command **build** for PostgreSQL by changing `bulk insert` to `copy from`.

### Version 0.2.0, 2020-04-16

- Fixed command **build** for PostgreSQL (issue #25).:
  - Index names now have at most 63 characters under PostgreSQL. Proper limits should also be in place for MS SQL and Oracle but have yet to be tested. SQLite always worked because it has a very large limit.
  - The PostgreSQL docker container for the test run now has more shared memory in order to allow “insert ... from select ...” with millions of rows. Performance still has a lot of room for improvement.
- Added TV episodes (tables `TitleEpisode` resp. `episode`).
- Cleaned up logging for `transfer` and `build` to consistently log the time and rows per second for each table.

### Version 0.1.2, 2020-04-14

- Fixed remaining “value to long” errors (issue #14).
- Fixed `TypeError` when command line option `-bulk` was specified.
- Added instructions on how to test run **pimdb** on a PostgreSQL docker container, see *Test run with PostgreSQL*.

### Version 0.1.1, 2020-04-13

- Fixed “value to long” for `NameBasics.knowForTitles` (issue #13).

- Added option to omit “sqlite:///” prefix from `--database` and specify only the path to the database file.
- Moved documentation to [ReadTheDocs](#).
- Improved performance of SQL inserts by using bulk inserts consistently and changing loops to SQL `insert ... from select ...` (where possible).

Version 0.1.0, 2020-04-11

- Initial public release.

## INDICES AND TABLES

- `genindex`



## Symbols

--database DATABASE  
pimdb command line option, 8

## E

environment variable  
PIMDB\_TEST\_DATABASE, 19  
PIMDB\_TEST\_FULL\_DATABASE, 20

## P

pimdb command line option  
--database DATABASE, 8  
PIMDB\_TEST\_DATABASE, 19  
PIMDB\_TEST\_FULL\_DATABASE, 20