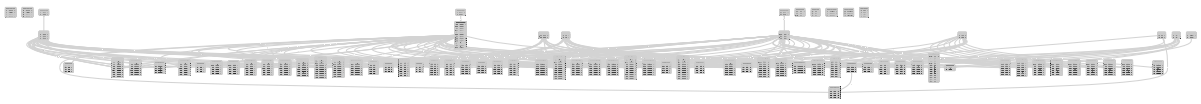
Database Construction

Contents

Input datasets in Temoa are stored in a relational database management system. For those unfamiliar with databases, you can think of them as collections of tables. Within each table, a ‘primary key’ uniquely identifies each row. A ‘foreign key’ is a column in one table that references the primary key of another table, thereby establishing relationships between tables and ensuring data consistency across the database.

The following Entity-Relationship (ER) diagram provides a visual overview of the Temoa v4 database schema and the relationships between its various tables:



Temoa uses sqlite, a widely used, self-contained database system. Building a database first requires constructing a sql file, which is simply a text file that defines the structure of different database tables and includes the input data. The snippet below is from the `time_period` table used to define the `test_system` dataset:

```
1   CREATE TABLE time_period
2   (
3     sequence INTEGER UNIQUE,
4     period   INTEGER
5         PRIMARY KEY,
6     flag     TEXT
7         REFERENCES time_period_type (label)
8   );
9   INSERT INTO "time_period" VALUES(1,2015,'e');
10  INSERT INTO "time_period" VALUES(2,2020,'f');
11  INSERT INTO "time_period" VALUES(3,2025,'f');
12  INSERT INTO "time_period" VALUES(4,2030,'f');
13  INSERT INTO "time_period" VALUES(5,2035,'f');
```

The first line creates the table. **Lines 3-7** define the columns within this table. Note that `period` is the primary key. Therefore, the same time period cannot be entered twice; each name must be unique. **Line 7**, which helps define the `flag` column, declares a foreign key

reference to the `label` column of the `time_period_type` table. As a result, if the user tries to enter a label in this table that does not exist in the `time_period_type` table, it will fail with an error. This foreign key reference ensures that the modeler doesn't accidentally type the wrong label in this table. For context, there are two basic types of time periods in Temoa, `e`, which defines pre-existing periods, and `f`, which defines future time periods that are to be optimized.

This enforcement of names across tables using foreign keys helps immediately catch typos. As you can imagine, typos happen in plain text files and Excel when defining thousands of rows of data. Another big advantage of using databases is that the model run outputs are stored in separate database output tables. The outputs by model run are indexed by a scenario name, which makes it possible to perform thousands of runs, programmatically store all the results, and execute arbitrary queries that instantaneously return the requested data.

Because some database table elements serve as foreign keys in other tables, we recommend that you populate input tables in the following order:

Group 1: labels used for internal database processing

- commodity_type: Need to identify which type of commodity. Do NOT change these abbreviations.
- technology_type: Need to identify which type of technology. Do NOT change these abbreviations. Categorizing and sub-categorizing can be done in the Technology table itself.
- time_period_type: Used to distinguish which time periods are simply used to specify pre-existing vintages and which represent future optimization periods.

Group 2: sets used within Temoa

- commodity: list of commodities used within the database
- technology: list of technologies used within the database
- time_period: list of both past and future time periods considered in the database
- time_season: seasons modeled in the database (also contains segment fractions)
- time_of_day: time of day segments modeled in the database

Group 3: parameters used to define processes within Temoa

- metadata_real (global_discount_rate)
- demand
- demand_specific_distribution
- efficiency
- existing_capacity
- capacity_factor_tech

- capacity_factor_process (only if CF varies by vintage; overwrites capacity_factor_tech)
- capacity_to_activity
- cost_fixed
- cost_invest
- cost_variable
- emission_activity
- lifetime_tech
- lifetime_process (only if LT varies by vintage; overwrites lifetime_tech)

Group 4: parameters used to define constraints within Temoa (non-exhaustive)

- limit_activity
- limit_capacity
- limit_emission
- limit_growth_capacity
- limit_new_capacity
- limit_resource
- limit_tech_input_split
- limit_tech_output_split

Group Region and Technology Constraints

Some constraint tables support summation over groups of regions or technologies. Note that each row in these tables will still only create one constraint, but that constraint will be a summation over the defined group. For example, the `limit_capacity` table will limit the total summed capacity of all technologies in the technology group (if used) and over all the regions in the region group (if used). Consider behaviour carefully. For example, the `limit_annual_capacity_factor` table will constrain the total summed capacity factor of the group, which would allow for varying capacity factors of processes within that group as long as the limit is met in aggregate.

Group Regions:

For the supported tables, the `region` column can be populated with either a single region (e.g., `"east"`), a subset of regions delineated with a `+` (e.g., `"east+west"`), or `"global"` to indicate summation over all model regions.

Important

When `+` delineated region summations are used with exchange technologies, exchange flows are only counted if the relevant exchange-technology region pairs are explicitly included in the region string used by the constraint row. Exchange-technology regions are directional and use a hyphen to separate the two regions (e.g., `east-west` and `west-east` are distinct keys). `"global"` *does* include all exchange flows by default.

For example, to constrain activity across the `east` and `west` regions including all exchange flows between them, the region string should be:

```
east+east-west+west-east+west
```

If exchange keys are omitted (e.g., using only `east+west`), flows through exchange technologies between those regions will *not* be included in the constraint summation.

The only exception to this rule is reserve margin constraints, `planning_reserve_margin` and `operating_reserve_margin`, which automatically include all exchange flows into and out of the region or region group due to their specific logic requiring this behaviour.

Supported tables:

- limit_annual_capacity_factor
- limit_seasonal_capacity_factor
- limit_resource
- limit_activity
- limit_capacity
- limit_new_capacity
- limit_activity_share
- limit_capacity_share
- limit_new_capacity_share
- limit_emission
- planning_reserve_margin
- operating_reserve_margin

Technology Groups:

For the supported tables, the following columns accept either technologies or technology groups, over which the constraint is summed. Technology groups are defined in the `tech_group` and `tech_group_member` tables.

Technology group columns:

- tech_or_group
- sub_group
- super_group

Supported tables:

- limit_annual_capacity_factor
- limit_seasonal_capacity_factor
- limit_resource
- limit_activity
- limit_capacity
- limit_new_capacity
- limit_activity_share
- limit_capacity_share
- limit_new_capacity_share
- planning_reserve_margin
- operating_reserve_margin

For help getting started, consider using the `temoa tutorial` command to generate a template project or inspect the example SQL file at `temoa/tutorial_assets/utopia.sql`. To begin building your own database file, use `temoa/db_schema/temoa_schema_v4.sql`, which is a database file with the requisite structure but no data added. We recommend leaving the database structure intact, and simply adding data to the schema file, or constructing an empty database from the schema file and then using a script or database editor to import data. For existing databases from older versions, use the `temoa migrate` command to safely transition your data to the V4 schema. Once the sql file is complete, you can convert it into a binary sqlite file by installing sqlite3 and executing the following command:

```
$ sqlite3 my_database.sqlite < my_database.sql
```

Note

The command above using the `<` operator for input redirection is supported in Linux, macOS, and the Windows Command Prompt. Note that PowerShell does not support the Unix-style `<` redirection operator. In PowerShell, you can achieve the same result by piping the file content into `sqlite3`:

```
Get-Content my_database.sql | sqlite3 my_database.sqlite
```

Alternatively, you can load a SQL file from within the `sqlite3` interactive interface using the `.read` command:

```
sqlite3 my_database.sqlite
sqlite> .read my_database.sql
```

When running on Windows, be sure to use the correct path separators (`\`) if you provide absolute paths to your files.

Now you can specify this database as the source for both input and output data in the config file.

Both of the checks below can be run to QA data by using the `temoa validate` command (which runs the model in `BUILD_ONLY` mode) and inspecting the log file. During validation runs, no solve is attempted on the model.

The “price checker” reviews cost data in the 3 cost tables and considers technology lifetime. It screens for possible inconsistencies that would corrupt output quality. Larger models may have well over 100K cost entries and an overlooked investment cost for a particular vintage tech in a particular region could easily be overlooked. Price checks performed/reported:

1. **Missing Costs (Check 0)**: This check looks for technologies that have no fixed/invest/variable costs at all. Other checks are more discriminating, so this check is only reported when Temoa is run in *debug* mode by using the *-d* flag on the run command.
2. **Missing Fixed/Investment Costs (Check 1a)**: This check identifies technologies that are *not* flagged as *uncapacitated* with neither a fixed or investment cost associated. These *might* be problematic for solve because the model minimizes cost, so capacity in these technologies would be free. *uncapacitated* technologies have no capacity measure, so fixed/investment costs are prohibited for them and that is checked elsewhere.

3. **Inconsistent Fixed/Investment Cost (Check 1b)**: This check looks for inconsistent application of fixed or base costs in the "base" or vintage year across all vintages and regions. So, if a tech has a fixed cost in some particular region and vintage year, but not in all, it will be flagged as a likely omission.
4. **Inconsistent Fixed & Variable Costs (Check 2)**: This check identifies techs that have inconsistencies in the application of fixed - variable costs. Techs that have *any* fixed cost for a particular [region, tech, vintage] process, but do not have entries that match the variable cost entries for the same process are flagged, and vice-versa. This would hopefully identify an accidental omission of some of the fixed/var costs for processes that have at least 1 entry for either.
5. **Lifetime Costing (Check 3)**: This check identifies costs that fall short or are missing during the process's lifetime. If a process has a variable cost in *any* year during the lifetime, but not all years, it is flagged. Same for fixed cost.
6. **Uncapacitated Tech Costs**: Any technology flagged as *uncapacitated* will trigger warnings here if it has any fixed/invest costs.

Temoa works backwards from demands to identify chains of technologies required to meet the demand. Source Tracing is designed to ensure that this backward tracing from demands describes a proper commodity network without gaps that might allow intermediate commodities to be treated as a free "source" commodity. Further description of possible network problems is included in the Commodity Network section.

Source Tracing pre-builds the entire commodity network in each region-period contained in the data and analyzes it for "orphans" which likely represent gaps in the network that would lead to erroneous output data. The operation is enabled by tagging foundational commodities for which there are no predecessors as "source" commodities in the *Commodity* database table with an *s* tag. Orphans (or chains of orphans) on either the demand or supply side are reported and *suppressed* in the data to prevent network corruption. Additionally, Temoa performs cycle detection on the commodity network to identify circular dependencies that could lead to non-convergence or erroneous results. Users can configure the cycle detection behavior using the following settings:

- **cycle_count_limit**: Limits the number of cycles reported in the log. A value of *-1* allows unbounded detection, *0* causes the system to log an error on the first detected cycle and then suppresses further cycle reports for the remainder of the run (without terminating execution), and a positive integer sets a specific limit. Default is 100.
- **cycle_length_limit**: Minimum length of cycles to report. This can be used to filter out small, expected circularities if necessary. Default is 1. The length limit is inclusive, so a cycle of length 1 is a self-loop, and a cycle of length *n* has *n* unique nodes.

Note that the myopic mode *requires* the use of Source Tracing to ensure accuracy as some orphans may be produced by endogenous decisions in myopic runs.

For large-scale models or long-running simulation modes (such as myopic or MGA), database I/O can become a performance bottleneck. Temoa allows you to tune the SQLite connection parameters in your configuration file under the `[sqlite]` section.

The following settings are available:

- **journal_mode**: Sets the SQLite journaling mode. Default is `WAL` (Write-Ahead Logging), which provides better performance and concurrency. Note that this will create temporary `-wal` and `-shm` files alongside your database during execution.
- **synchronous**: Controls how frequently SQLite flushes data to disk. Default is `NORMAL`, which provides a good balance between speed and safety.
- **mmap_size**: The maximum number of bytes for memory-mapped I/O. Default is 8GB (`8589934592`). This allows SQLite to access the database file directly from memory, significantly speeding up reads for large databases.
- **cache_size**: The number of pages or the size in KiB for the SQLite page cache. If negative, it specifies size in KiB. Default is 500MiB (`-512000`).

These settings are especially impactful in **myopic mode**, where Temoa frequently updates and queries the database between period iterations. By default, Temoa also disables the per-period `VACUUM` operation in myopic runs to avoid redundant and expensive full-database rewrites.