

Project Report

Logan Gill, Brielle Hoff

2025-12-10

1 Overview

For our final project, we decided to build, analyze, and visualize a tool to facilitate easier user interaction with the crafting recipes in [Minecraft](#). In order to accomplish this, we created a PostgreSQL script that processes the recipe data in the game's built-in format (JSON) and produces a normalized relational database. Using this database, we built an application with which a user may create a plan for a Minecraft project and determine which resources are needed to create all necessary items. Additionally, we created a visualization in the form of a graph which contains the relationships between every recipe¹ and their relevant items.

All relevant project files are included either in [our project repository](#) or are generated by scripts in the repository from an official copy of Minecraft.

2 Dataset — Interrelated Tables

To construct our database, our script is run against one's official copy of Minecraft. This assumes the user has Minecraft installed at its standard location (`7.minecraft`). To acquire a copy, visit www.minecraft.net. To the extent required by law, all copyrighted data is copyright Mojang AB. The visualization and application presented in this report were produced with data from Minecraft 1.21.10, but our script works with some earlier versions.

The dataset that we constructed contains, in addition to several mostly-trivial tables used for data integrity purposes, three substantially interesting tables: `recipe`, `recipe_ingredient`, and `item_tag`. The `recipe` table (1422 rows) represents each recipe in the game; the `recipe_ingredient` table (5083 rows) represents each item in each recipe, including metadata about the item's usage (such as placement location); the `item_tag` table (2828 rows) represents the game's ability to use classes of items instead of individual items in recipes. We believe this is a sufficiently complex and interesting dataset for this project.

3 Application — Software Development

After constructing the dataset, we created a TUI application with which the recipe tree may be explored interactively. The application may be found in the `browser/` folder of our project repository. It is written using `psycpg`, a Python PostgreSQL database adapter, and uses `curses`,

¹There are some pathological cases which are too strange to include.

a built-in Python library for writing terminal applications (and a wrapper around the C library of the same name). It can connect to either the course database or a local PostgreSQL server containing the extracted data.

Using this application, the user may create and edit “plans” containing a set of Minecraft items and quantities, and can then explore the space of ways to craft that set of items. For example, one might create a plan for an in-game house, which would consist of all of the block types present in the build (wood, brick, etc.) along with the total number of each one used. The application would then present a tree structure with a node for each item. Any item may be left as-is, if the user already possesses enough of that item, or expanded so that a recipe (if available) to produce that item may be selected. Upon selection, the recipe would then expand to its constituent items. This process may be repeated, hierarchically, as needed. Required total quantities are tracked throughout. Items may also be split into multiple identical nodes whose count sum to that of the original, which is useful if the user wants to be able to acquire an item through a combination of recipes. After each change to the tree, the application updates a list of the base items (corresponding to leaf nodes of the tree), as well as items that will be left over after crafting the set. This could be very useful to Minecraft players who need to know what resources to gather in what quantities in order to successfully build their ideas.

Once created, user plans can be saved to the database for further exploration at a later date.

4 Outputs

4.1 Data Loading and Cleaning

4.1.1 Source Data

Minecraft stores its recipes as a large collection of JSON files. Each file represents a single recipe in the game, and may come in one of a variety of formats. Recipes typically include an input section and an output section. The output section takes on a mostly consistent format (including the number and identity of the produced item), but the input section differs dramatically depending on the “type” of the recipe.

The most well-known recipe type is the “shaped crafting” recipe. Here, input items are placed in particular cells of a grid. Recipe files of this type contain a **key** attribute, which maps ASCII characters to items or tags, and a **pattern** attribute, which contains an array of strings forming a keyed representation of the recipe in ASCII art.

Many other recipes are “shapeless”, meaning items may be placed in any order. The input format is correspondingly simple, providing a flat array of ingredients.

Our database includes 6 other recipe types for things like cooking or breaking down items at various workstations. Recipes of these types map a single input to a single output, but with slightly different formats.

A few additional recipe types exist, but mostly represent special cases in the game’s code, and we do not include these in this project.

4.1.2 Data Cleaning

We start by putting all JSON records into two files in CSV format (using `src/mk-csv.sh`), with one CSV row per JSON file. This is the extent of processing done outside SQL. From here, at the request of our instructor, we proceed with data processing entirely in SQL. The CSV files are loaded into staging tables and SQL JSON operators are used to gradually munge the data into a usable format (using `src/load.sql`).

While a few tables are trivial to build, some require more complex processing. A recursive CTE was required to expand tags (tags may contain other tags hierarchically with arbitrary depth). Additionally, some recipes eschew the tag system and store ingredients as arrays of items where an item or tag would normally be expected. We address this problem by introducing synthetic tags, which we fill with the relevant items, and updating troublesome records accordingly. Finally, converting the ASCII-art format used in shaped recipes (which is likely used to make recipes easily human-editable) to our internal format required some lateral thinking.

4.2 Database Schema Design

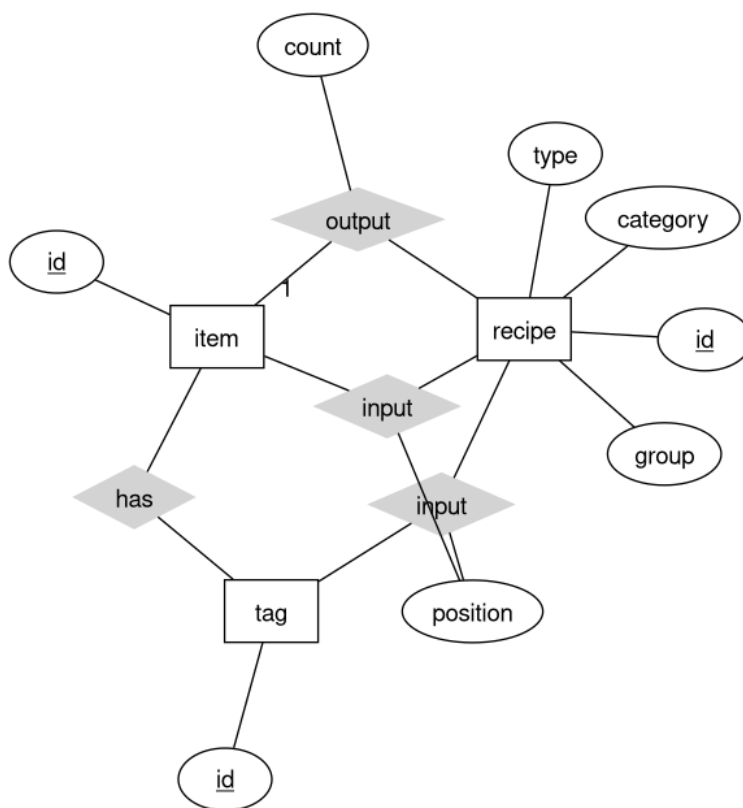


Fig 1. Entity-Relationship Diagram

One of our first tasks of the project was planning our database schema. There are three primary entity types here, as illustrated above: items, recipes, and tags. These relationships are many-to-many in general, so the edge labels are omitted for brevity. Items are related to tags (a tag can contain any number of items and an item can belong to any number of tags); recipes always have a single output item (with a quantity) and a number of positioned inputs which can be items or

tags; and recipes have various attributes which are used by the game and shown for informational purposes in the application (category, type, and group). We generate a few additional single-column tables which serve as enumerations for certain attributes. For example, there is a `recipe_group` table which the `recipe_group` attribute on the `recipe` table references as a foreign key. The same applies to recipe categories. Most other key constraints are fairly self-explanatory given the entity relationships.

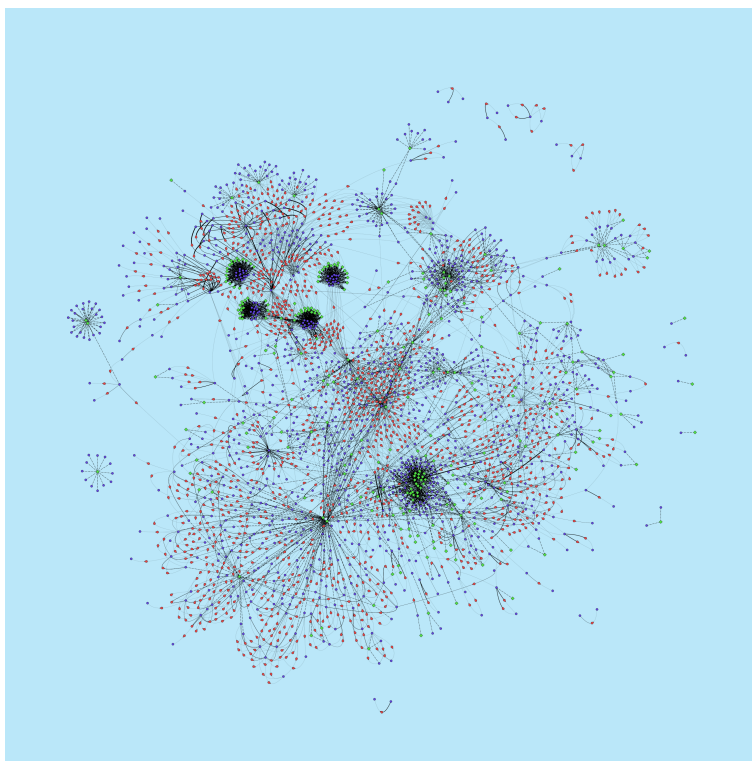
We chose to do quite a bit of normalization here. This increased the complexity of the data loading step, but made future processing and application development much easier. For example, native tags are allowed to reference other tags, but we chose to flatten these relationships while building the table. This has practical implications, as applications can find all items in a tag with one query; without such flattening, the tag entity would have a relationship to itself and applications would likely be doing a lot of repeated work without much benefit.

4.3 Data Visualization

In order to visualize this recipe data we wrote a program which performs a physics-like simulation to untangle a graph of the data into recognizable structures (see `visualizer/`).

To use it, first run `export.sql` to pull the data out of the database into a couple of flat JSON files (still in our normalized format) spin up a simple http server (e.g. with `python3 -m http.server`) and then navigate to `/visualize.html` (Requires a fairly powerful computer and a relatively modern browser).

The output of that program looks something like this: (N.B. the randomness in the program means the exact graph is different each time)



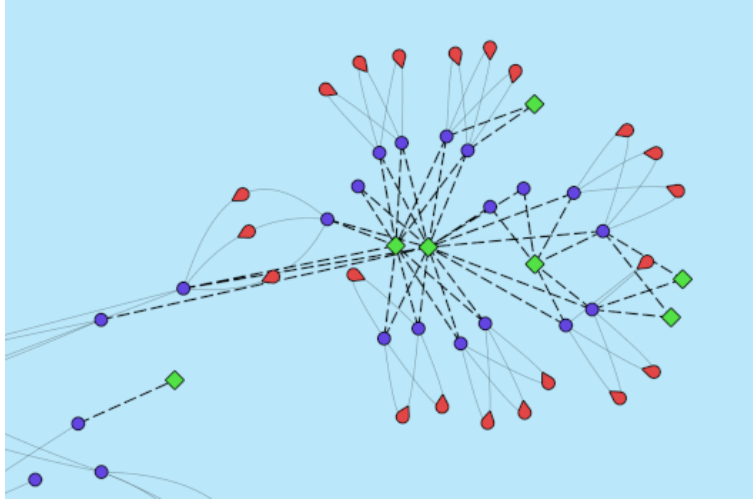


Fig 2. The complete visualization and a cropped portion to show detail.

In this graph, blue circle nodes represent items, red teardrop nodes recipes (with the direction pointing at the recipe's output), and green diamond nodes tags. The thickness of edges is proportional to the number of items involved.

4.4 Software Development

As mentioned earlier in this report, the source code for our application can be found in our [project repository](#).

```

    < minecraft:bamboo_block >
      4 (4) minecraft:bamboo_block
      [minecraft:bamboo_block via minecraft:crafting_shapeless]
    36 (36) minecraft:bamboo
73 (1 stack and 9) minecraft:cobblestone
1 (1) minecraft:crafting_table
[minecraft:crafting_table via minecraft:crafting_shaped]
  4 (4) #minecraft:planks
  1 (1) #minecraft:planks
  < minecraft:bamboo_planks >
    1 (1) minecraft:bamboo_planks
    [minecraft:bamboo_planks via minecraft:crafting_shapeless]
    1 (1) #minecraft:bamboo_blocks
    < minecraft:bamboo_block >
      1 (1) minecraft:bamboo_block
      [minecraft:bamboo_block via minecraft:crafting_shapeless]
    3 (3) #minecraft:planks
    < minecraft:cherry_planks >

Items for Cobblestone House
Required items
  45 (45) minecraft:bamboo
  3 (3) minecraft:cherry_planks
  73 (1 stack and 9) minecraft:cobblestone
  1 (1) minecraft:furnace
Leftover items
  1 (1) minecraft:bamboo_planks

```

Fig 3. A demo of the TUI application in action. This user already possesses 3 `minecraft:cherry_planks`, so they created a split node, and has now discovered that they need to farm 45 `minecraft:bamboo` in order to complete this build.

Most tree nodes make database queries when expanded for the first time. Making queries lazily like this ensures that the application only stores what it needs to and is reactive as possible.

For example, the following code is used when a node representing a recipe is expanded to show its input items.

```
self.cur.execute('''
    SELECT item, tag
    FROM recipe_ingredient
    WHERE recipe = %s
    ORDER BY position
''', (id,))
self.ingredients = self.cur.fetchall()
```

Similar code is used in other tree nodes to fetch the required data as needed. Additionally, the “planning” functionality uses similarly simple SQL commands to insert, modify, and query the tables used to store user-created item lists. Parameterization is used throughout the program to reduce the risk of SQL injection.

5 Conclusion

This was a really fun project that involved a lot of clever tricks, interesting problem solving, and a good level of complexity. The dataset is small enough to meaningfully internalize the entire thing (as many players do) but large and interlinked enough to provide numerous interesting patterns. The data processing required to convert the data in its native format into a format usable for application and analysis posed a fascinating challenge and we’ve produced a program that may prove to have genuine utility.