The Assignment

We have to design a relational schema and code an accompanying application for a cookbook. The original assignment said to use Java for the application, but I got permission to implement that as a PHP website. Live version of this project is available on http://sdu.projectjj.com/DM26/live/. Additionally, I worked alone on the project.

The Design & Relational Schema

I did not start by doing an ER model of the schema, but instead created the required relations directly from the assignment. So the ER model found in on the last page was created at the end of the process. It is also globbed with all the fields in a single oval form since the multiple ovals got very chaotic. I generally find it easier to go straight to the relational model, even for much more complex schemas.

Thus I will go straight to presenting the relational model, building the schema from the ground up. Ironically, it would be benificial to have the ER model on the side for reference. I'll use the dumps from pgAdmin III, so the column types are those of PostgreSQL and not always of standard SQL.

The first two tables, `ingredient_types` and `ingredient_units`, are not really important, but they do form the foundation for the rest of the relations. They are practically identical, and only serve as domains. They could have been defined as actual domains, but that would have made them more complicated to modify.

```
CREATE TABLE ingredient_types (
      ingredient_type_id serial NOT NULL,
      ingredient_type_type text NOT NULL,
      CONSTRAINT ingredient_types_pkey PRIMARY KEY (ingredient_type_id)
)
CREATE TABLE ingredient_units (
      ingredient_unit_id serial NOT NULL,
      ingredient_unit_unit text NOT NULL,
      CONSTRAINT ingredient_units_pkey PRIMARY KEY (ingredient_unit_id)
)
```

From these tables it is also clear to see my naming scheme. A table is plural denoting it can contain any number of records, whereas the columns are singular since they at most contains their own value. I do not use any arrays or sets, as this would violate BCNF... Ok, so my real reason is that it's

simpler and cleaner to use actual tables; that it complies with BCNF is a side-bonus. Columns inherit the table name, and appends the field name to that. It does create rather long names, but with a good editor you won't have to type them that often anyways. Readability is preferable. It does also create names such as `ingredient_type_type` which some would probably want as `ingredient_type_name`, but I prefer this way.

The `ingredients` table is the center of the model, in that it provides the information about ingredients required by both recipes and inventories. The type and unit of an ingredient are weak foreign keys, in that if the user decides to delete a type or unit it won't also delete the ingredient. The application deals with missing type or unit by outputting [unknown] for them, thus alerting the user to problematic entries.

```
CREATE TABLE ingredients (
	ingredient_id serial NOT NULL,
	ingredient_name text NOT NULL,
	ingredient_type int4,
	ingredient_unit int4,
	ingredient_price float4,
	CONSTRAINT ingredients_pkey PRIMARY KEY (ingredient_id),
	CONSTRAINT "$1" FOREIGN KEY (ingredient_type)
		REFERENCES ingredient_types (ingredient_type_id)
		ON UPDATE CASCADE ON DELETE SET NULL,
	CONSTRAINT "$2" FOREIGN KEY (ingredient_unit)
		REFERENCES ingredient_units (ingredient_unit_id)
		ON UPDATE CASCADE ON DELETE SET NULL,
	CONSTRAINT ingredients_ingredient_name_key UNIQUE (ingredient_name)
)
```

In fact, the entire design relies mostly on weak foreign keys, since these are more friendly to the end-user. The person who designed the database would know that deleting an ingredient type would also delete all ingredients of that type, but the end-user might want to split a type into two or more subtypes. So the end-user deletes the old type, and creates the new...not realizing he just wiped his ingredients. This problem can be dealt with at the application level by showing a warning about what will happen, and explain that the end-user must first create the new types, then assign the ingredients to the new types, and then delete the old type. Oh, and if he forgot to change all would-be-affected ingredients he'll get the warning again... Much better to use weak foreign keys and maybe show a list of cascade affected ingredients for reference (but, I do have some strong keys, as will be shown in a bit).

```
CREATE TABLE inventories (
	inventory_id serial NOT NULL,
	inventory_name text NOT NULL,
	inventory_begin date,
	inventory_end date,
	CONSTRAINT inventories_pkey PRIMARY KEY (inventory_id)
)
```

The `inventories` table is used to define both current and various minimum inventories. The current inventory is an application-level special case where `inventory_id` is 0. I also thought it would be nice to define a timeframe for minimum inventories, hence the _begin and _end columns. In the application this could be used for showing notices about upcoming events. The actual relation to ingredients is defined in the `inventory_ingredients` table as:

```
CREATE TABLE inventory_ingredients (
	inventory_id int4 NOT NULL,
	ingredient_id int4 NOT NULL,
	ingredient_units float4,
	CONSTRAINT inventory_ingredients_pkey PRIMARY KEY (inventory_id, ingredient_id),
	CONSTRAINT "$1" FOREIGN KEY (inventory_id)
		REFERENCES inventories (inventory_id)
		ON UPDATE CASCADE ON DELETE CASCADE,
	CONSTRAINT "$2" FOREIGN KEY (ingredient_id)
		REFERENCES ingredients (ingredient_id)
		ON UPDATE CASCADE ON DELETE SET NULL
)
```

This relation also keeps track of how many units of a given ingredient is in a given inventory.

Here it should be noted that I have not made special constraints to limit the use of negative amounts or values in the design, and that many values can be fractional. Even at the application-level I do not limit such values. This is again to be friendly to the end-user.

Also, here the foreign key to inventories is strong, while the one to ingredients is weak. The end-user will expect that deleting an inventory will also clear out its amounts. However, in this case deleting an ingredient would not actually wipe out the entire inventory so the foreign key to ingredients could be strong. I have chosen a weak key for the same reason as before, though: I'd much rather be told [unknown] when listing the inventory and be aware of a potential problem with my data set (more on this later).

Ok, so far so good. Now we can create ingredients, stock the current inventory, and manage other minimum inventories. Onwards to recipes and menus...

```
CREATE TABLE recipe_types (
      recipe_type_id serial NOT NULL,
      recipe_type_type text NOT NULL,
      CONSTRAINT recipe_types_pkey PRIMARY KEY (recipe_type_id)
)
CREATE TABLE recipes (
      recipe_id serial NOT NULL,
      recipe_name text NOT NULL,
      recipe_persons int2 NOT NULL DEFAULT 1,
      recipe_prepare_time interval,
      recipe_work_time interval,
      recipe_picture text,
      recipe_comment text,
      recipe_type int4,
      recipe_vegetarian bool NOT NULL DEFAULT false,
      recipe_instructions text,
      CONSTRAINT recipes_pkey PRIMARY KEY (recipe_id),
      CONSTRAINT "$1" FOREIGN KEY (recipe_type)
            REFERENCES recipe_types (recipe_type_id)
            ON UPDATE CASCADE ON DELETE SET NULL
)
```

Another basic domain table, and the actual `recipes` table. Two important points with this table:

First, the picture is stored as text because the picture is not actually stored in the database at all. Instead, information on where to find the picture is stored in the database in the form of a URI (more specifically, here it's a URL). Storing binary data that in itself can be rendered or used outside the application, which JPEG pictures can, is strongly discouraged. If it was data that only this application knew how to decode, that would have been fine. Central storage is usually not enough reason to want to put everything in a blob.

Second, how do we know if a recipe is vegetarian? Showing vegetarian recipes can be handled in two ways: Either you traverse the entire ingredient list where every ingredient has a boolean IsVegetarian value, or you set that boolean in the recipe itself. The first way is far more dynamic, but much slower. The second way is incredibly fast, but requires more application-level awareness. Setting it in the recipe itself is the only sane way to handle it, though, or so I believe.
If you want to bring it up to menu level, then the dynamic way would require traversing every ingredient for every recipe in the menu, whereas the faster way would only require traversing recipes. You could go so far as to have a copy of the IsVegetarian defined in the menu and then update it whenever a recipe is altered, which would be fast and simple. It can be done in the application or directly in the database using triggers.

It would of course be possible to have the IsVegetarian value in all three levels (ingredient, recipe, menu) and update the recipe and menu when an ingredient is altered...but that'd be quite a lot of overhead and indirection.

```
CREATE TABLE recipe_ingredients (
	recipe_id int4 NOT NULL,
	ingredient_id int4 NOT NULL,
	ingredient_units float4 NOT NULL,
	CONSTRAINT recipe_ingredients_pkey PRIMARY KEY (recipe_id, ingredient_id),
	CONSTRAINT "$1" FOREIGN KEY (recipe_id)
		REFERENCES recipes (recipe_id)
		ON UPDATE CASCADE ON DELETE CASCADE,
	CONSTRAINT "$2" FOREIGN KEY (ingredient_id)
		REFERENCES ingredients (ingredient_id)
		ON UPDATE CASCADE ON DELETE SET NULL
)
```

This table probably looks familiar at first glance, and indeed `recipe_ingredients` is very similar to `inventory_ingredients`. But it is with this relation that I want to express why I prefer weak foreign keys where possible.

Let's say we have the recipe for Meatballs that require 750 g beef, 2 pcs egg, and 10 g flour. Then we delete the ingredient 'beef'. Had the relation used only strong keys, Meatballs would now require 2 pcs egg and 10 g flour. Since I am using weak keys, my Meatballs would list 750 [unknown] [unknown], 2 pcs egg, 10 g flour. In this case it is obvious that something is wrong with the recipe for Meatballs that require no meat, but one could imagine less obvious cases where data set errors would not be noticed for a long time, especially in a multi-user environment.

The last two tables are about menus:

```
CREATE TABLE menus (
	menu_id serial NOT NULL,
	menu_name text NOT NULL,
	menu_comment text,
	CONSTRAINT menus_pkey PRIMARY KEY (menu_id)
)
```

```
CREATE TABLE menu_recipes (
	menu_id int4 NOT NULL,
	recipe_id int4 NOT NULL,
	recipe_order int2 NOT NULL,
	CONSTRAINT menu_recipes_pkey PRIMARY KEY (menu_id, recipe_id),
	CONSTRAINT "$1" FOREIGN KEY (menu_id)
		REFERENCES menus (menu_id)
		ON UPDATE CASCADE ON DELETE CASCADE,
	CONSTRAINT "$2" FOREIGN KEY (recipe_id)
		REFERENCES recipes (recipe_id)
		ON UPDATE CASCADE ON DELETE SET NULL
)
```

Only new or unseen part is that I store an order for the recipes, to be able to tell which recipe belongs where in the course of the menu.

The Implementation

As I mentioned in the beginning, I implemented the project as a PHP website. This has the effect that I don't really prepare queries so much as build them dynamically. Supposedly there is a performance hit, but I haven't spotted it so far. Average query execution time is 0.003 seconds.

On a side note, BCNF and modelling is a good start in making efficient queries and designs, but the real power lies in the Explain SQL command. Good use of Explain will spot most bottlenecks in the design, but it is of course RDBMS specific.

I will take some of the application end-user functionality requirements and explain (in English, not SQL) the query of how I solve that with regards to my schema.

Requirement: Print recipe scaled to a given number of persons.
Requirement: Write out list of things to buy to be able to make a given recipe, given the current inventory and a number of persons to serve.
Done in: /live/recipes.php
Queries:

```
SELECT
r.recipe_id, r.recipe_name, r.recipe_persons, r.recipe_prepare_time,
r.recipe_work_time, r.recipe_picture, r.recipe_comment,
rt.recipe_type_type, r.recipe_vegetarian, r.recipe_instructions
FROM recipes AS r
LEFT JOIN recipe_types AS rt ON (r.recipe_type=rt.recipe_type_id)
WHERE recipe_id={$_REQUEST['id']}
```

```
SELECT
ri.ingredient_id, (ri.ingredient_units*{$scale}) as required_units,
i.ingredient_name, i.ingredient_price, iu.ingredient_unit_unit,
ii.ingredient_units as current_units
FROM recipe_ingredients AS ri
LEFT JOIN ingredients AS i USING (ingredient_id)
LEFT JOIN ingredient_units as iu ON (i.ingredient_unit=iu.ingredient_unit_id)
LEFT JOIN inventory_ingredients AS ii
      ON (ii.inventory_id=0 AND ri.ingredient_id=ii.ingredient_id)
WHERE ri.recipe_id={$_REQUEST['id']}
ORDER BY i.ingredient_name ASC
```

A quick note on the mysterious `{$_REQUEST['id']}` and `{$scale}` that appear in the queries. These are inline PHP variables, parsed directly into the query.

I use LEFT JOIN since I use weak foreign keys, whereas a straight JOIN would negate the nice effect of the weak keys. I use aliases with the initials of the table, so `ingredient_units` becomes `iu`. Technically, those two queries could be joined to a single query, but the amount of redundant information in the result set would be horrible.

Requirement: Show recipes containing a given set of ingredients.
Requirement: Show recipes possible to make with the current inventory (when serving a given number of persons).
Requirement: Show recipes possible to make within a given time limit.
Requirement: Show recipes possible to make within a given price limit (when serving a given number of persons).
Requirement: Show recipies that are vegetarian.
Done in: /live/recipes.php
Query:

```
(
      SELECT r.recipe_id
      FROM recipes AS r
      WHERE true
      AND r.recipe_prepare_time>='{$_REQUEST['query']['min_time']}'::interval
      AND r.recipe_prepare_time<='{$_REQUEST['query']['max_time']}'::interval
      AND r.recipe_vegetarian IS TRUE
      INTERSECT (
            SELECT price.recipe_id FROM (
                  SELECT ri.recipe_id, sum(ri.ingredient_units *({$_REQUEST['query']['persons']}/(
                        SELECT sub.recipe_persons
                        FROM recipes as sub
                        WHERE sub.recipe_id = ri.recipe_id))*i.ingredient_price)
            as recipe_price
            FROM recipe_ingredients AS ri
            LEFT JOIN ingredients AS i USING (ingredient_id)
```

```
            GROUP BY ri.recipe_id
            ) as price
            WHERE true
            AND price.recipe_price>={$_REQUEST['query']['min_price']}
            AND price.recipe_price<={$_REQUEST['query']['max_price']}
      )
)
      EXCEPT (
            SELECT DISTINCT ri.recipe_id
            FROM recipe_ingredients AS ri
            LEFT JOIN inventory_ingredients AS ii
                  ON (ii.inventory_id=0 AND ri.ingredient_id=ii.ingredient_id)
            WHERE (
                  (ri.ingredient_units *({$_REQUEST['query']['persons']}/(
                  SELECT sub.recipe_persons FROM recipes as sub
                  WHERE sub.recipe_id=ri.recipe_id)))>ii.ingredient_units
                  OR ii.ingredient_units IS NULL
            )
      )
      INTERSECT (
            SELECT r.recipe_id FROM recipes as r
            WHERE _int_contains(ARRAY(
                  SELECT ri.ingredient_id FROM recipe_ingredients as ri
                  WHERE ri.recipe_id=r.recipe_id), '{{$ingredients}}')
      )
```

Don't expect to be able to run that as it is there. I created a Recipe Search Tool in /live/recipe.php where you set the parameters by which the above query might be generated. It is broken into sections that limit the overall result set by their own criteria. This results in some seemingly redundant subqueries. It is by far not the most elegant solution, but it was more to show that 5 of the requirements could be done from a single expanded search form. This returns is a set of recipe IDs that can then be fed into a query to get all the other information about them.

The last intersect is the “recipes containing a given set of ingredients” requirement. To do this I use a PostgreSQL specific C extension from pgsql/contrib/_int.sql that can determine if a given integer array is a subset of another integer array.

Requirement: Write out list of things to buy to ensure a given minimum inventory.

Done in: /live/inventory.php

Queries:

```
SELECT inventory_id, inventory_name
FROM inventories
WHERE inventory_id!=0
ORDER BY inventory_name
```

```
SELECT ii.ingredient_id, i.ingredient_name, ii.ingredient_units as required_units,
      iu.ingredient_unit_unit, (ii.ingredient_units-ic.ingredient_units) as missing_units,
      ((ii.ingredient_units-ic.ingredient_units) * i.ingredient_price) as missing_price
FROM inventory_ingredients AS ii
LEFT JOIN inventory_ingredients AS ic
      ON (ic.inventory_id=0 AND ii.ingredient_id=ic.ingredient_id)
LEFT JOIN ingredients as i ON (ii.ingredient_id=i.ingredient_id)
LEFT JOIN ingredient_units AS iu ON (i.ingredient_unit=iu.ingredient_unit_id)
WHERE ii.inventory_id={$row['inventory_id']} AND ii.ingredient_units!=0
ORDER BY ingredient_name ASC
```

Requirement: Update inventory based on recipe usage.

Requirement: Update single inventory entry.

Requirement: Update inventory based on shopping lists (recipe, menu, minimal inventories).

Done in: /live/inventory.php

Executed from: /live/inventory.php or /live/recipes.php or /live/menus.php

Queries:

```
UPDATE inventory_ingredients
SET ingredient_units=(ingredient_units+({$ingredient['units']}))
WHERE inventory_id=0 AND ingredient_id={$id}

INSERT INTO inventory_ingredients (inventory_id, ingredient_id, ingredient_units)
VALUES (0,{$id},{$ingredient['units']})
```

Updating the inventory takes at least 1 query per addition or update, but may take 2 if the ingredient being added to doesn't exist in the current inventory.

Requirement: Show total preparation time and price for menu.

Requirement: Write out list of things to buy to be able to make a given menu, given the current inventory and a number of persons to serve.

Done in: /live/menus.php

Queries:

```
SELECT
sum(r.recipe_prepare_time) as prepare_time
FROM recipes AS r
LEFT JOIN menu_recipes AS mr USING (recipe_id)
WHERE mr.menu_id={$_REQUEST['id']}

SELECT
ri.ingredient_id, (ri.ingredient_units*{$scale}) as required_units,
i.ingredient_name, i.ingredient_price, iu.ingredient_unit_unit,
ii.ingredient_units as current_units
FROM recipe_ingredients AS ri
LEFT JOIN menu_recipes AS mr USING (recipe_id)
LEFT JOIN ingredients AS i USING (ingredient_id)
LEFT JOIN ingredient_units as iu ON (i.ingredient_unit=iu.ingredient_unit_id)
```

```
LEFT JOIN inventory_ingredients AS ii
      ON (ii.inventory_id=0 AND ri.ingredient_id=ii.ingredient_id)
WHERE mr.menu_id={$_REQUEST['id']}
```

These are basically just the ones from showing recipes, but with the added in LEFT JOIN menu_recipe to get the multiple recipes.

As a final note on these queries, I want to draw attention to the fact that I never use `SELECT * FROM`. The reasoning behind this is that it introduces possible flaws when modifying or extending the database in the future. If you use *, you have no control over the column output order, and as such you commit yourself to the current column order. Suppose I wanted to re-order the columns or add in new columns in the beginning, then I would have to make sure my result set processing knew about this. But with named columns, I know exactly what columns I am selecting and in what order I'll get them. An even better example: Suppose I added a 2MB blob field to each record. * would happily retrive this new data, thus drowning out the database connection, while named columns would stick with what it was told. Very simply never use *, except in count(*).

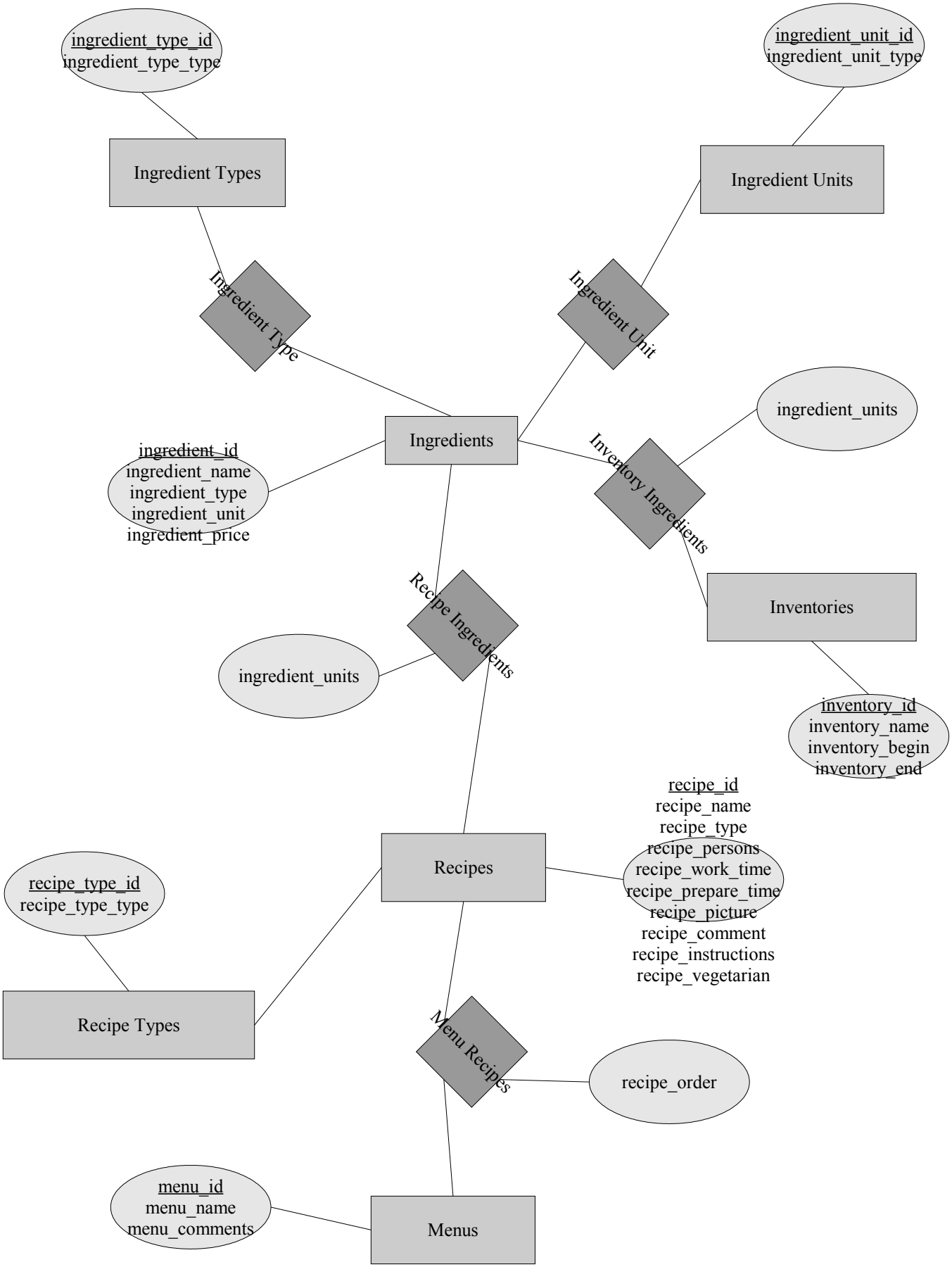
The End Note

Live version: http://sdu.projectjj.com/DM26/live/
tar.gz archive: http://sdu.projectjj.com/DM26/DM26.tar.gz
tar.gz md5sum: 2caa5d8602de5a724606e7477b9ceec4
zip archive: http://sdu.projectjj.com/DM26/DM26.zip
zip md5sum: b8d665618ce4ab04c5a096a4d848e9a3

The database and program does not run IMADAs machines. At least PHP 4.3.4 and PostgreSQL 7.4.6 are required, plus a HTTP daemon.

Tino Didriksen, tidid02

ER Model


ingredient_type_id
ingredient_type_type
Ingredient Types
ingredient_unit_id
ingredient_unit_type
Ingredient Units
Ingredient Type
Ingredient Unit
Ingredients
ingredient_id
ingredient_name
ingredient_type
ingredient_unit
ingredient_price
ingredient_units
Inventory Ingredients
Recipe Ingredients
Inventories
ingredient_units
inventory_id
inventory_name
inventory_begin
inventory_end
recipe_id
recipe_name
recipe_type
recipe_persons
recipe_work_time
recipe_prepare_time
recipe_picture
recipe_comment
recipe_instructions
recipe_vegetarian
Recipes
recipe_type_id
recipe_type_type
Recipe Types
Menu Recipes
recipe_order
menu_id
menu_name
menu_comments
Menus