TECHNICAL UNIVERSITY OF DENMARK

02170 - DATABASE SYSTEMS

Recipe database

Mandatory Group Project

Group number 4

Professor:

Anne Haxthausen

Teaching Assistant:

Thomas Løye Skafte

Groupmembers:	*Student ID:*
Alexander Lykke Lademann Østergaard	s164424
Casper Skjærris	s164429
Imre Nagy	s172182

Hand in date:
April 16th 2018

DTU Compute
Department of Applied Mathematics and Computer Science

1. Statement of requirements

Our project is about a database managing *recipes*. For each recipe we store the name, the amount (how many persons is it enough for) and the required preparation time.

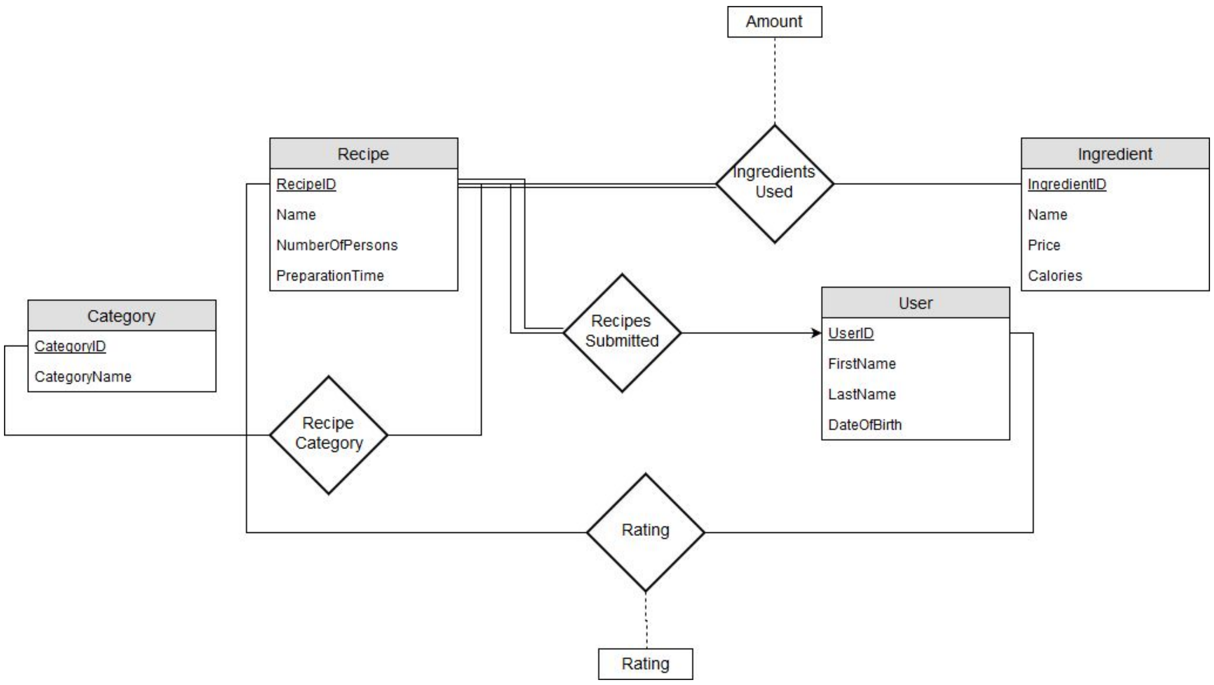
All of the recipes are divided into *categories*. These categories allow us to query all recipes in one subcategory, therefore providing the users an easier filtering.

The recipes have been submitted by *users*. Each user has a name, and a birthdate. For each recipe we store who submitted it.

Each recipe consists of *ingredients*. To provide important and informative statistics about the dishes, we decided to store the price and the calories of each ingredient next to its name. When matching the ingredient with the recipe, we store the amount as well.

Users have a possibility to *rate* recipes. Each user can only rate one recipe once. This way providing us a way of listing the best meals based on our user's reviews.

2. Conceptual design



Entity Relationship

In our conceptual design the entity sets are:

- Recipe

        - Consisting of recipes. These recipes represent a single recipe. They have the following attributes:
            - RecipeID
            - Name
            - NumberOfPersons
            - PreparationTime
- Ingredient
    - Consisting of ingredients. These represent a single ingredient to be used for a recipe. They have the following attributes:
        - IngredientID
        - Name
        - Price
            - We store the price scaled to 100 gramms.
        - Calories
            - We store the calorie scaled to 100 gramms.
- User
    - Consisting of users. These represent a single user in our system. All of the users have the following attributes:
        - UserID
        - FirstName
        - LastName
        - DateOfBirth
- Category
    - Consisting of categories. This set contains all of our categories we use to classify the recipes. They have the following attributes:
        - CategoryID
        - CategoryName

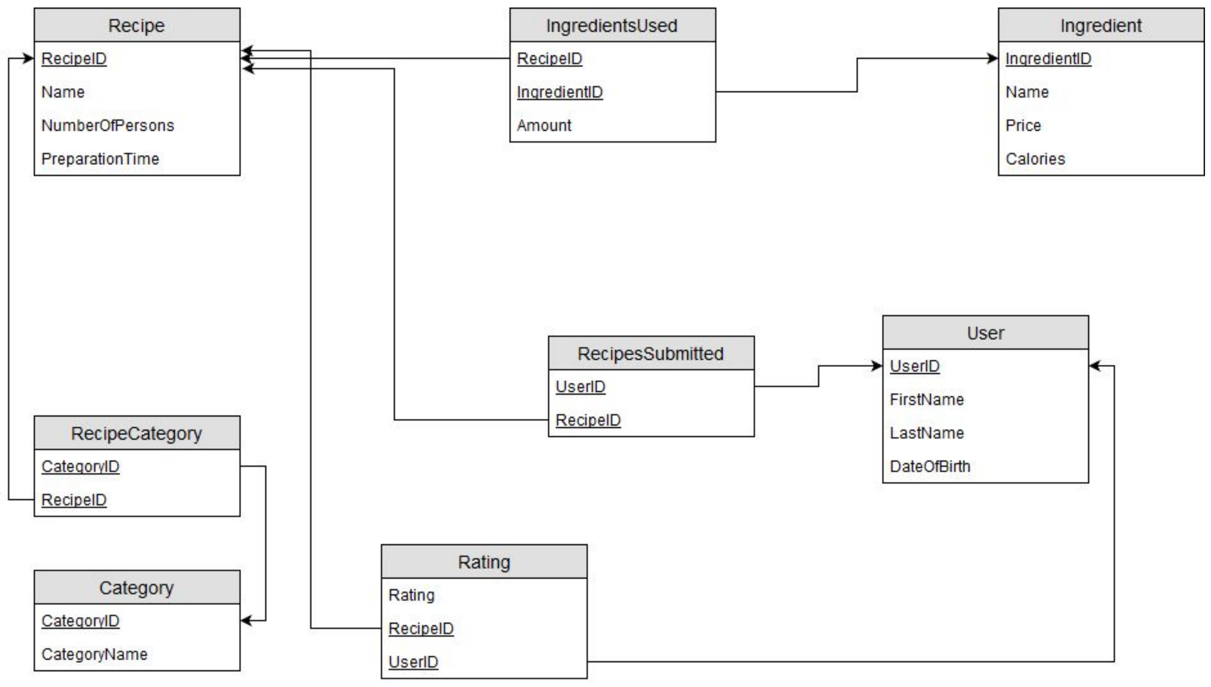
And our relationships are:

- Ingredients Used
    - Which connects the Recipe with the Ingredient. Due to the fact that each ingredient can be used for several recipes, and that each recipe can have several ingredient, the cardinality of this relationship is **many-to-many**.
    - There can be ingredients in our planned database which haven't been used in any recipes, but all of the recipes have to have at least one ingredient. Therefore **Recipe** has **total participation**, while **Ingredient** only **partial participation** in the relationship.
    - This relationship also has the following attribute:
        - Amount
            - Meaning how many times a 100 gramms it contains.
- Recipes Submitted
    - Which connects the User set with the Recipe set. No recipe can have multiple authors, but each user can submit multiple recipes. Therefore the cardinality of this relationship is **one-to-many**.

    - Each recipe has to be submitted by a user, but there can be users who haven't submitted any recipes yet. Therefore **Recipe** has a **total participation**, while **User** has only **partial participation** in the relationship.
    - This relationship has no attributes.
- Recipe Category
    - Which connects Category with Recipe. Each recipe can have multiple categories and each category can have multiple recipes. Therefore this relationhip is **many-to-many**.
    - There can be a category which has no recipes, as well as a recipe which has no category. Therefore the **participation both** sides are **partial**.
    - This relationship has no attributes.
- Rating
    - Connecting the User set with the Recipe set. Each recipe can have multiple ratings, and each user can make multiple ratings leading to a cardinality of **many-to-many**.
    - Not all of the recipes have to have rating, and not all of the users have to submit ratings. Therefore the participation in this relationship is **both sides partial**.
    - This relationship has an attribute:
        - Rating

3. Logical design

After converting the entity-relationship diagram to a relation schemas, we get the following: (shown as a diagram)



2. Relation schema

4. Normalization

- Recipe
    - 1NF because all of the attributes are atomic. The primary key consists of one single column, therefore 2NF is guaranteed if 1NF is guaranteed, and due to the fact that all of the attributes depend directly on the primary key, it is in 3NF.
- IngredientsUsed
    - There are two foreign keys here, and together they make the primary key for this table. The other attribute depends on both of them and is atomic. Therefore this table is in 3NF as well.
- Rating
    - The two foreign keys together make the primary key for the table. There is only one atomic attribute left, which depends on both of them, due to the fact that it describes one particular user's rating on one particular recipe. Therefore the table is in 3NF.
- Ingredient
    - The attributes are atomic, and unrelated to each other. The primary key consists of one single attribute therefore the 3NF is guaranteed.
- User
    - This table also has only one attribute as a primary key. The other attributes are atomic, meaning that they are all representing one single attribute, neither of them can be duplicated for a single user. There are no transitional dependencies either therefore the table is in 3rd normal form.
- Categories
    - 3NF. Only one attribute as primary key, atomic values and no transitional dependencies.

5. Implementation

We implemented the database with the following script:

```sql
DROP DATABASE IF EXISTS databasesys;
CREATE DATABASE databasesys;
USE databasesys;

-- Dropping existing tables
SET foreign_key_checks = 0;

    DROP TABLE IF EXISTS `Recipe`;

    DROP TABLE IF EXISTS `IngredientsUsed`;

    DROP TABLE IF EXISTS `Ingredient`;

    DROP TABLE IF EXISTS `RecipeCategory`;

    DROP TABLE IF EXISTS `Category`;

    DROP TABLE IF EXISTS `User`;

    DROP TABLE IF EXISTS `RecipesSubmitted`;

    DROP TABLE IF EXISTS `Rating`;

SET foreign_key_checks = 1;

-- Creating all tables
CREATE TABLE `Recipe`(
    `RecipeId` int NOT NULL AUTO_INCREMENT UNIQUE,
    `Name` varchar(255) NOT NULL,
    `NumberOfPersons` int NOT NULL,
    `PreparationTime` int NOT NULL);

ALTER TABLE `Recipe` ADD PRIMARY KEY (`RecipeId`);

CREATE TABLE `IngredientsUsed`(
    `Amount` int NOT NULL,
    `IngredientId` int NOT NULL,
    `RecipeId` int NOT NULL);

ALTER TABLE `IngredientsUsed` ADD PRIMARY KEY (`RecipeId`, `IngredientId`);

CREATE TABLE `Ingredient`(
    `IngredientId` int NOT NULL AUTO_INCREMENT UNIQUE,
    `Name` varchar(255) NOT NULL,
    `Price` int NOT NULL,
    `Calories` int NOT NULL);

ALTER TABLE `Ingredient` ADD PRIMARY KEY (`IngredientId`);

CREATE TABLE `RecipeCategory`(
    `CategoryId` int NOT NULL,
    `RecipeId` int NOT NULL);

ALTER TABLE `RecipeCategory` ADD PRIMARY KEY (`CategoryId`, `RecipeId`);

CREATE TABLE `User`(
    `UserId` int NOT NULL AUTO_INCREMENT UNIQUE,
    `FirstName` varchar(255) NOT NULL,
    `LastName` varchar(255) NOT NULL,
    `DateOfBirth` datetime NOT NULL);

ALTER TABLE `User` ADD PRIMARY KEY (`UserId`);
```

```sql

CREATE TABLE `RecipesSubmitted`(
    `UserId` int NOT NULL,
    `RecipeId` int NOT NULL);

ALTER TABLE `RecipesSubmitted` ADD PRIMARY KEY (`UserId`, `RecipeId`);

CREATE TABLE `Rating`(
    `RatingNum` int NOT NULL,
    `UserId` int NOT NULL,
    `RecipeId` int NOT NULL);

ALTER TABLE `Rating` ADD PRIMARY KEY (`UserId`, `RecipeId`);

CREATE TABLE `Category`(
    `CategoryId` int NOT NULL AUTO_INCREMENT UNIQUE,
    `CategoryName` varchar(255) NOT NULL);

ALTER TABLE `Category` ADD PRIMARY KEY (`CategoryId`,`CategoryName`);


-- Creating all FOREIGN KEY constraints
ALTER TABLE `IngredientsUsed`
ADD CONSTRAINT `FK_IngredientIngredientsUsed`
    FOREIGN KEY (`IngredientId`)
    REFERENCES `Ingredient`
        (`IngredientId`)
    ON DELETE NO ACTION ON UPDATE NO ACTION;

CREATE INDEX `IX_FK_IngredientIngredientsUsed`
    ON `IngredientsUsed`
    (`IngredientId`);

    ALTER TABLE `IngredientsUsed`
    ADD CONSTRAINT `FK_RecipeIngredientsUsed`
        FOREIGN KEY (`RecipeId`)
        REFERENCES `Recipe`
            (`RecipeId`)
        ON DELETE NO ACTION ON UPDATE NO ACTION;

    ALTER TABLE `RecipesSubmitted`
    ADD CONSTRAINT `FK_UserRecipesSubmitted`
        FOREIGN KEY (`UserId`)
        REFERENCES `User`
            (`UserId`)
        ON DELETE NO ACTION ON UPDATE NO ACTION;

    ALTER TABLE `Rating`
    ADD CONSTRAINT `FK_RatingUser`
        FOREIGN KEY (`UserId`)
        REFERENCES `User`
            (`UserId`)
        ON DELETE NO ACTION ON UPDATE NO ACTION;

```

```sql
ALTER TABLE `Rating`
ADD CONSTRAINT `FK_RatingRecipe`
    FOREIGN KEY (`RecipeId`)
    REFERENCES `Recipe`
        (`RecipeId`)
    ON DELETE NO ACTION ON UPDATE NO ACTION;

CREATE INDEX `IX_FK_RatingRecipe`
    ON `Rating`
    (`RecipeId`);

ALTER TABLE `RecipesSubmitted`
ADD CONSTRAINT `FK_RecipesSubmittedRecipe`
    FOREIGN KEY (`RecipeId`)
    REFERENCES `Recipe`
        (`RecipeId`)
    ON DELETE NO ACTION ON UPDATE NO ACTION;

CREATE INDEX `IX_FK_RecipesSubmittedRecipe`
    ON `RecipesSubmitted`
    (`RecipeId`);

ALTER TABLE `RecipeCategory`
ADD CONSTRAINT `FK_CategoryRecipeCategory`
    FOREIGN KEY (`CategoryId`)
    REFERENCES `Category`
        (`CategoryId`)
    ON DELETE NO ACTION ON UPDATE NO ACTION;

ALTER TABLE `RecipeCategory`
ADD CONSTRAINT `FK_RecipeCategoryRecipe`
    FOREIGN KEY (`RecipeId`)
    REFERENCES `Recipe`
        (`RecipeId`)
    ON DELETE NO ACTION ON UPDATE NO ACTION;

CREATE INDEX `IX_FK_RecipeCategoryRecipe`
    ON `RecipeCategory`
    (`RecipeId`);
```

6. Implementation

We filled the database with sample data with the following script:

```sql
USE databasesys;

-- Sample data for category
INSERT INTO category (CategoryName) VALUES
    ("Vegan"),
    ("Main"),
    ("Soup"),
    ("Dessert");

-- Sample data for user
INSERT INTO user (FirstName, LastName, DateOfBirth) VALUES
    ("Alexander", "Østergaard", TIMESTAMP('1996-06-01')),
    ("Casper", "Skjærris", TIMESTAMP('1996-12-14')),
    ("Imre", "Nagy", TIMESTAMP('1994-11-19'));

-- Sample data for ingredient
INSERT INTO ingredient (Name, Price, Calories) VALUES
    ("Potato", "100", "200"),
    ("Rice", "200", "210"),
    ("Pasta", "150", "170"),
    ("Cucumber", "130", "50"),
    ("Tomato", "140", "30"),
    ("Chicken breast", "400", "190"),
    ("Avocado", "250", "235");

-- Sample data for recipe
INSERT INTO recipe (Name, NumberOfPersons, PreparationTime) VALUES
    ("Cucumber chicken", 4, 60),
    ("Avocado stuffed potato", 2, 45),
    ("Pasta'la vista", 2, 25);

-- Sample data for recipesubmitted
INSERT INTO recipessubmitted (UserId, RecipeId) VALUES
    (1, 1),
    (3, 2),
    (3, 3);

-- Sample data for ingrediendsused
INSERT INTO ingredientsused (RecipeId, IngredientId, Amount) VALUES
    (1, 4, 4),
    (1, 6, 2),
    (2, 7, 3),
    (2, 1, 3),
    (3, 3, 2),
    (3, 5, 1);

-- Sample data for recipecategory
INSERT INTO recipecategory (CategoryId, RecipeId) VALUES
    (2, 1),
    (1, 2),
    (2, 2);

-- Sample data for rating
INSERT INTO rating (RatingNum, UserId, RecipeId) VALUES
    (5, 1, 1),
    (4, 1, 2),
    (3, 2, 1),
    (3, 2, 2),
    (2, 2, 3);
```

Our database is made from an administrators perspective to prevent duplicate data. This however, doesn't make it very user friendly for anyone wanting to look at the recipes. We are dealing with this by creating a view for the users called RecipeUserView. The view is defined in the following way:

```
CREATE VIEW RecipeUserView AS
SELECT r.Name, r.NumberOfPersons, r.PreparationTime, CaloriesInRecipe(r.RecipeId)
AS 'Calories pr 100g', RecipeRating(r.RecipeId) AS Rating
FROM Recipe r;
```

Views are selected the same way as standard queries to tables, so we see what's in the view by calling:

```
SELECT * FROM RecipeUserView;
```

7. SQL Data Queries

ORDER BY

Imagine a scenario when one of our users enter the search area with no filtering applied. Our default listing would display all the recipes in alphabetic order. Therefore our task is to query all the recipes ordered by the alphabet:

```
SELECT * FROM recipe ORDER BY Name ASC;
```

GROUP BY

Group by is used when aggregate functions are called. Given the case we want to find the most frequently used ingredient, we use the aggregate function SUM on amount from the table IngredientsUsed. Thus we can calculate how many time each ingredient are used in all of the recipes

```
SELECT i.Name, SUM(iu.Amount) AS NumberOfTimesUsed
FROM Ingredient i INNER JOIN  IngredientsUsed iu INNER JOIN Recipe r
WHERE i.IngredientId = iu.IngredientId AND r.RecipeId = iu.RecipeId
GROUP BY i.Name;
```

OUTER JOIN

Outer joins are great for getting all the data on one table, and only getting related rows in a different table.

E.g. if we want to know which Recipes are in which category, but we also want to show categories which doesn't have any recipes, we can do the following Outer join:

```
SELECT CategoryName, Name FROM Category
LEFT OUTER JOIN RecipeCategory as rc ON Category.CategoryId = rc.CategoryId
LEFT OUTER JOIN Recipe as r ON r.RecipeId = rc.RecipeId;
```

8. SQL Table Modifications

INSERT

We might want to add a new category to be more specific. To do this we use an insert. Category has the Schema (CategoryID INT, CategoryName VARCHAR(25), however the ID auto increments so when inserting we only need to provide the name

```
INSERT INTO Category(CategoryName) VALUES ('Vegetarian');
```

UPDATE

Another scenario would be when someone decides to change his rate on a specific recipe. Maybe at first it wasn't that delicious, but on a second try it ended up being really good. So the user decides to adjust his or her rate on the recipe. Maybe from 3 to 5.

```
UPDATE rating SET RatingNum = 5 WHERE UserId = 2 AND RecipeId = 2;
```

9. SQL Programming

FUNCTION

Some values doesn't make sense to store directly, instead should these be calculated when needed. We use functions for these values. One of the values we've decided to calculate using a function is the calories of a recipe. The reason for this is to be more user friendly, so the users don't have to know how many calories are in a recipe, instead we will calculate it from the amount of each ingredient and the calories of the ingredient

```
DROP FUNCTION IF EXISTS CaloriesInRecipe;
DELIMITER //
CREATE FUNCTION CaloriesInRecipe(vRecipeID INT) RETURNS INT
BEGIN
    DECLARE vCalories INT;
    SELECT SUM(i.Calories*iu.Amount) INTO vCalories
    FROM Recipe r INNER JOIN IngredientsUsed iu
    INNER JOIN Ingredient i
    WHERE r.RecipeId = vRecipeID AND r.RecipeID = iu.RecipeID
    AND iu.IngredientId = i.IngredientID;
    RETURN vCalories;
END; //
DELIMITER ;
```

A different use of this function is to only show recipes where the calories pr 100g are less than 1000

```
SELECT Name, CaloriesInRecipe(Name) AS Calories
FROM Recipe
WHERE CaloriesInRecipe(Name) < 1000;
```

TRANSACTION

Imagine a scenario where the management decides that our current rating system is too restrictive. Our users are begging for a finer scale. The solution would be to, instead of a 1 to 5 scale, we allow users to rate on a scale of 1 to 10. But for that to take effect, we first have to convert all of our existing rates by multiplying them with 2. It is really important that these changes should be applied once to all of the database because afterwards there is no way of determining which ones we did not change. In this case a transaction could ensure the state of the database.

```
START TRANSACTION;
UPDATE rating SET RatingNum = RatingNum * 2;
COMMIT;
```

TRIGGER

Imagine that every time a user adds a new recipe we would like to check whether there already exists a recipe with the same name and if it exists, we append a number to make it unique

```
delimiter $$
CREATE TRIGGER exists_aready BEFORE INSERT ON recipe
FOR EACH ROW
BEGIN
    DECLARE c INT;
    SET c = (SELECT COUNT(*) FROM recipe WHERE Name LIKE CONCAT(NEW.Name, '%'));
    IF c >= 0 THEN SET NEW.Name = CONCAT(NEW.Name, c); END IF;
END$$
delimiter ;
```

EVENT

Since we don't have unlimited storage we don't want to store all recipes. We want to create an event that runs every month and delete recipes with a rating between 0 and 1, however we want to keep recipes without any rating (null), since these are probably new and haven't been tried yet.
To start this out we need to create a function to calculate the rating of a recipe. This is done by:

```
DROP FUNCTION IF EXISTS RecipeRating;
DELIMITER //
CREATE FUNCTION RecipeRating(vID INT) RETURNS INT
BEGIN
    DECLARE vRating INT;
    SELECT AVG(ra.RatingNum) INTO vRating
    FROM Recipe r INNER JOIN Rating ra ON r.RecipeId = ra.RecipeId
    WHERE r.RecipeId = vID;
    RETURN vRating;
END; //
DELIMITER ;
```

Now we need to create a procedure that the event can call:

```
DROP PROCEDURE IF EXISTS DeleteBadRecipes;
DELIMITER //
CREATE PROCEDURE DeleteBadRecipes()
BEGIN
    SET @badRecipeIDs = (
    SELECT RecipeId
    FROM Recipe
    WHERE RecipeRating(RecipeId) BETWEEN 0 AND 1);
    DELETE FROM RecipeCategory WHERE RecipeId IN (@badRecipeIDs);
    DELETE FROM RecipesSubmitted WHERE RecipeID IN (@badRecipeIDs);
    DELETE FROM IngredientsUsed WHERE RecipeID IN (@badRecipeIDs);
    DELETE FROM Rating WHERE RecipeId IN (@badRecipeIDs);
    DELETE FROM Recipe WHERE RecipeId IN (@badRecipeIDs);
END//
DELIMITER ;
```

Finally we can create the event. This is done by:

```
DROP EVENT IF EXISTS DeleteBadRecipesEvent;
CREATE EVENT DeleteBadRecipesEvent
ON SCHEDULE EVERY 1 MONTH
STARTS '2018-05-01 00:00:01'
```

```
DO CALL DeleteBadRecipes();
```

Now that we have an event, we want to look for and turn on events created in our database:

```
SET GLOBAL event_scheduler = 1;
```

BACKUP PROCEDURE

We've created a small backup procedure that will back up the recipes and the ingredients used, so that we can restore that part of the database to an earlier state. This is set to run every month, 1 minute before the DeleteBadRecipesEvent, so that if a wrong recipe gets deleted, it is posible to restore it, though with out the rating at the moment.
We've included a BackupTime column on each of the tables, and added it to the primary key and index, so that we have different states of the database each month, instead of deleting the old backups. In case we wanted to delete the old backups, we can simply do a DELETE FROM on the tables with the 'Old' Suffix in the procedure, and that way we'll only have one months data in the backups at any given time.

```
USE databasesys;


-- CLEANUP
DROP TABLE IF EXISTS RecipeOld;
DROP TABLE IF EXISTS IngredientsOld;
DROP TABLE IF EXISTS IngredientsUsedOld;
DROP PROCEDURE IF EXISTS RecipeBackup;
DROP EVENT IF EXISTS BackupEvent;

-- Backup Procedure example

-- Creating backup table.
CREATE TABLE RecipeOld LIKE Recipe;
ALTER TABLE RecipeOld
    ADD BackupTime TIMESTAMP(6),
    DROP PRIMARY KEY,
    DROP INDEX RecipeId,
    ADD PRIMARY KEY(RecipeId, BackupTime),
    ADD UNIQUE (RecipeId, BackupTime);

CREATE TABLE IngredientOld LIKE Ingredient;
ALTER TABLE IngredientOld
    ADD BackupTime TIMESTAMP(6),
    DROP PRIMARY KEY,
    DROP INDEX IngredientId,
    ADD PRIMARY KEY(IngredientId, BackupTime),
    ADD UNIQUE (IngredientId, BackupTime);
```

```
CREATE TABLE IngredientsUsedOld LIKE IngredientsUsed;
ALTER TABLE IngredientsUsedOld
    ADD BackupTime TIMESTAMP(6),
    DROP PRIMARY KEY,
    ADD PRIMARY KEY(RecipeId, IngredientId, BackupTime),
    ADD UNIQUE (RecipeId,IngredientId, BackupTime);


-- Creating backup procedure for RecipeTable
DELIMITER //
CREATE PROCEDURE RecipeBackup()
BEGIN
    DECLARE vSQLSTATE CHAR(5) DEFAULT '00000';
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
     BEGIN
         GET DIAGNOSTICS CONDITION 1
         vSQLSTATE = RETURNED_SQLSTATE;
     END;
    START TRANSACTION;
    SET @t = NOW(6);
    INSERT INTO RecipeOld SELECT *,@t FROM Recipe;
    INSERT INTO IngredientOld SELECT *,@t FROM Ingredient;
    INSERT INTO IngredientsUsedOld SELECT *,@t FROM IngredientsUsed;
    IF vSQLSTATE = '00000' THEN COMMIT;
     ELSE ROLLBACK;
    END IF;
END;//
DELIMITER ;

CREATE EVENT BackupEvent
ON SCHEDULE EVERY 1 MONTH
STARTS '2018-05-01 00:00:00'
DO CALL RecipeBackup();
```