DND Character Management System

12/3/20

CSDS 341

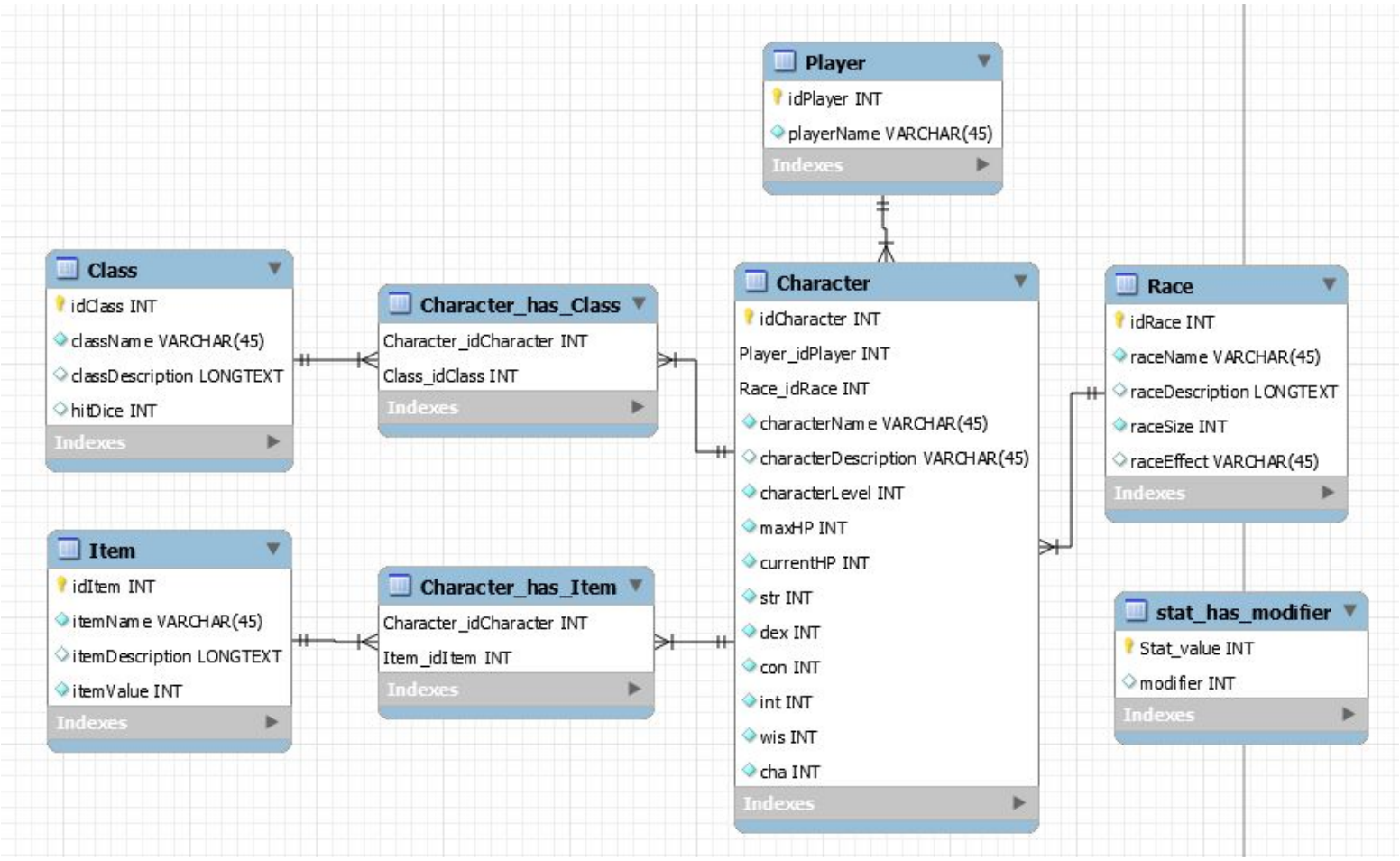
Eytan Kaplan and Nainoa Faulkner-Jackson

1. Application Background

Our project is built around the role playing game Dungeons and Dragons. One of the most tedious and difficult parts of DND is creating, tracking, and ensuring consistency of a given player's character. This gets even more difficult if a player decides to take on multiple characters. In stock DND, character information is stored on a character sheet, which is effectively a spreadsheet with blank fields in which a player fills in their given character's information. This is a broad swath of a mixture of text and numeric data, some of which will remain static, but the majority of which will change over the career of the character in question. The system works fine in theory, but speaking from experience, it's always a good idea to have your character's information backed up in another format.

In order to do this, we have created a database which tracks character information, and relates it to a given player. While no new information is actually being added, the goal here is to make tracking these vast amounts of data easier, and make the information contained within more resilient to being damaged, lost or otherwise sabotaged. (It wouldn't be the first time a player deliberately changed their character sheet to their advantage.) At the moment some of the character data is simplified, and some has been omitted entirely. In the long term, our goal would be to fully implement a full depth character as designated by the players handbook, and having a user interface through which players could alter their characters in real time without having to write SQL code themselves.

2. ER Dragon


Player
idPlayer INT
playerName VARCHAR(45)
Indexes
Class
idClass INT
className VARCHAR(45)
classDescription LONGTEXT
hitDice INT
Indexes
Character_has_Class
Character_idCharacter INT
Class_idClass INT
Indexes
Character
idCharacter INT
Player_idPlayer INT
Race_idRace INT
characterName VARCHAR(45)
characterDescription VARCHAR(45)
characterLevel INT
maxHP INT
currentHP INT
str INT
dex INT
con INT
int INT
wis INT
cha INT
Indexes
Race
idRace INT
raceName VARCHAR(45)
raceDescription LONGTEXT
raceSize INT
raceEffect VARCHAR(45)
Indexes
Item
idItem INT
itemName VARCHAR(45)
itemDescription LONGTEXT
itemValue INT
Indexes
Character_has_Item
Character_idCharacter INT
Item_idItem INT
Indexes
stat_has_modifier
Stat_value INT
modifier INT
Indexes

3. Data Description

The DND Character Management System has five traditional entities, one entity that is present only for utility, and two relations.

Player holds the information for the actual individual people. There is very little actual data stored here, as information about the players themselves isn't the priority of this database. All that is needed is enough information to match a player to their respective characters.

Character is the heart of this database, storing a vast amount of character information. Most if this information is simply stats, other important bits are the Character's Classes, of which there can be one or many, and their items, which represents their inventory.

Class represents the role or roles a character plays in a given DND campaign. This entity also has information which alters stats in the Character entity. Because a Character can have an arbitrarily large number of classes (though more than two isn't advised,) this information is stored in the Character_Has_Class relation.

Race represents the species of a given character, and any special traits that would come with said species.

Item represents an individual item a character can have in their possession. By being many to one with respect to Character, this entity can effectively represent a Character's inventory. This relation is stored in the Character_Has_Item relation.

The final entity, Stat_Has_Modifier, is present only as a utility. Data here must be pre populated inorder to convert between character stat numbers (between 0 and 20) to

their respective modifiers (between -5 to +5). This data contained within this entity is used to populate the stat fields in the Character entity.

We have used foreign key, NOT NULL and UNIQUE constraints. Foreign keys have been used to connect the Character entity to the respective tables with which it has one to many relationships, these being Item, and Class. Not Null has been used to ensure all necessary data for DND play is present and populated within a given entity table, such as names of players, characters, and stat information without which a character would not be usable. Unique is used used in order to make sure all tables which could have accidental crossover do not, we mainly did this by including unique fields for each individual instance of an entity.

4. Entity Schemas

4.1 Player

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Player` (
  `idPlayer` INT NOT NULL AUTO_INCREMENT,
  `playerName` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`idPlayer`))
```

The table holds the basic information about each individual player. Each player can have an arbitrary number of characters. idPlayer is a unique, arbitrarily assigned number to track individual players. playerName holds the name of the player.

4.2 Character

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Character` (
  `idCharacter` INT NOT NULL,
  `Player_idPlayer` INT NOT NULL,
  `Race_idRace` INT NOT NULL,
  `characterName` VARCHAR(45) NOT NULL,
  `characterDescription` VARCHAR(45) NULL,
  `characterLevel` INT NOT NULL,
```

```
  `maxHP` INT NOT NULL,
  `currentHP` INT NOT NULL,
  `str` INT NOT NULL,
  `dex` INT NOT NULL,
  `con` INT NOT NULL,
  `int` INT NOT NULL,
  `wis` INT NOT NULL,
  `cha` INT NOT NULL,
  PRIMARY KEY (`idCharacter`, `Race_idRace`, `Player_idPlayer`),
  INDEX `fk_Character_Player1_idx` (`Player_idPlayer` ASC),
  INDEX `fk_Character_Race1_idx` (`Race_idRace` ASC),
  CONSTRAINT `fk_Character_Player1`
    FOREIGN KEY (`Player_idPlayer`)
    REFERENCES `SQLDnD`.`Player` (`idPlayer`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Character_Race1`
    FOREIGN KEY (`Race_idRace`)
    REFERENCES `SQLDnD`.`Race` (`idRace`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

This table represents the base of a character, not including specialized information such as race, class, or inventory. Base stats of a character are tracked here (str, dex, con, int, wis, cha), as well as maxHP, which is a derived stat relying on multiple values, including CON. Because a character can have at most one player and one race, these are referenced here via foreign keys.

4.3 Class

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Class` (
  `idClass` INT NOT NULL AUTO_INCREMENT,
  `className` VARCHAR(45) NOT NULL,
  `classDescription` LONGTEXT NULL,
  `hitDice` INT NULL,
  PRIMARY KEY (`idClass`),
  UNIQUE INDEX `idClass_UNIQUE` (`idClass` ASC),
  UNIQUE INDEX `className_UNIQUE` (`className` ASC))
```

A class represents unique in game attributes about a character, which largely dictate how the character will play. The main thing of importance in terms of SQL is hitDice, which will indirectly affect the maxHP stat from the Character table over time. All classes will be premade and cannot be altered by the user, only selected. One Character can have an arbitrary number of classes (Though more than two is generally considered a bad idea.)

4.4 Race

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Race` (
  `idRace` INT NOT NULL,
  `raceName` VARCHAR(45) NOT NULL,
  `raceDescription` LONGTEXT NULL,
  `raceSize` INT NOT NULL,
  `raceEffect` VARCHAR(45) NULL,
  PRIMARY KEY (`idRace`))
```

Race represents the species of a given character. Each race has a few unique traits that influence playstyle. These can include being different sizes (raceSize), or having unique attributes such as night vision or immunity to poison (raceEffect). All races will be premade and cannot be altered by the user, only selected. One character can have at most one race.

4.5 Item

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Item` (
  `idItem` INT NOT NULL AUTO_INCREMENT,
  `itemName` VARCHAR(45) NOT NULL,
  `itemDescription` LONGTEXT NULL,
  `itemValue` INT NOT NULL,
  PRIMARY KEY (`idItem`))
```

Items are a general catchall for anything a character can have on their person. This can include weapons, loot, quest items, etc. A character can have an arbitrary number of items.

4.6 Stat_has_Modifier

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`stat_has_modifier` (
  `Stat_value` INT NOT NULL,
  `modifier` INT NULL,
  PRIMARY KEY (`Stat_value`))
```

This table is disconnected from all other tables, and represents no actual data needed by a specific table. Instead, this is effectively a key value pair which assists in the conversion from base character stats (str, dex, con, int, wis, cha) which can be anywhere between 0 to 20, to their respective modifiers which will be a value between -5 to +5.

5. Relation Schemas

5.1 Character_has_Class

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Character_has_Class` (
  `Character_idCharacter` INT NOT NULL,
  `Class_idClass` INT NOT NULL,
  PRIMARY KEY (`Character_idCharacter`, `Class_idClass`),
  INDEX `fk_Character_has_Class_Class1_idx` (`Class_idClass` ASC),
  INDEX `fk_Character_has_Class_Character_idx` (`Character_idCharacter` ASC),
  CONSTRAINT `fk_Character_has_Class_Character`
    FOREIGN KEY (`Character_idCharacter`)
    REFERENCES `SQLDnD`.`Character` (`idCharacter`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Character_has_Class_Class1`
    FOREIGN KEY (`Class_idClass`)
    REFERENCES `SQLDnD`.`Class` (`idClass`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

This relation tracks class to character relations. One character can have an arbitrary number of classes (though again more than 2 is not advised), and every class can have an infinite number of characters. This relation is many to many.

5.2 Character_has_Item

```
CREATE TABLE IF NOT EXISTS `SQLDnD`.`Character_has_Item` (
  `Character_idCharacter` INT NOT NULL,
  `Item_idItem` INT NOT NULL,
  PRIMARY KEY (`Character_idCharacter`, `Item_idItem`),
  INDEX `fk_Character_has_Item_Item1_idx` (`Item_idItem` ASC),
  INDEX `fk_Character_has_Item_Character1_idx` (`Character_idCharacter` ASC),
  CONSTRAINT `fk_Character_has_Item_Character1`
    FOREIGN KEY (`Character_idCharacter`)
    REFERENCES `SQLDnD`.`Character` (`idCharacter`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Character_has_Item_Item1`
    FOREIGN KEY (`Item_idItem`)
    REFERENCES `SQLDnD`.`Item` (`idItem`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

This represents a character's inventory. A character can have an arbitrary number of items, but an item can only be possessed by a single character at any given time. This relation is one (character) to many (items).

6. Example Queries

6.1 Get unique player names with Elven characters

```
SELECT DISTINCT
SQLDND.Player.playerName

FROM SQLDND.Player
JOIN SQLDND.Character ON SQLDND.Character.Player_idPlayer = SQLDND.Player.idPlayer
WHERE SQLDND.Character.Race_idRace = 1
;
```

Expressed as Tuple Relational Calculus

$$\{t|(\exists p)(Player(p) \wedge t[playerName]=p[playerName] \wedge (\exists c)(Character(c) \wedge c[Player\_idPlayer]=p[idPlayer] \wedge c[Race\_idRace]=1))\}$$

Expressed as Tuple Relational Algebra:

$$\square_{Player.playerName}(\sigma_{Character.Race\_idRace\ =\ 1}(Character \bowtie_{Character.Player\_idPlayer\ =\ Player.idPlayer} Player))$$

6.2 Get a list of character names and their races

```
SELECT
SQLDND.Character.characterName,
SQLDND.Race.raceName

FROM SQLDND.Character
JOIN SQLDND.Race ON SQLDND.Character.Race_idRace = SQLDND.Race.idRace
;
```

Expressed as Tuple Relational Calculus:

$$\{t^{(2)} \mid (\exists c)(\exists r)(Character(c) \wedge Race(r) \wedge t^1[characterName]=c[characterName] \wedge t^2[raceName]=r[raceName] \wedge c[Race\_idRace]=r[idRace])\}$$

Expressed as Tuple Relational Algebra:

$$\square_{Character.characterName,Race.raceName}(Race \bowtie_{Race.idRace=Character.Race\_idRace} Character)$$

6.3 Get a list of item names and descriptions belonging to characters played by Eytan

```
SELECT

SQLDnD.Player.playerName,

SQLDnD.Item.itemName,

SQLDnD.Item.itemDescription

FROM SQLDnD.Player

JOIN SQLDnD.Character ON SQLDnD.Player.idPlayer = SQLDnD.Character.Player_idPlayer

JOIN SQLDnD.Character_has_Item ON SQLDnD.Character.idCharacter =

SQLDnD.Character_has_Item.Character_idCharacter

JOIN SQLDnD.Item ON SQLDnD.Character_has_Item.Item_idItem = SQLDnD.Item.idItem

WHERE Player.playerName = "Eytan"

;
```

Expressed as Tuple Relational Calculus:

$$\{t^{(3)} \mid (\exists p)(\exists i)(Player(p) \wedge Item(i) \wedge t^1[playerName]=p[playerName] \wedge t^2[itemName]=i[itemName] \wedge t^3[itemDescription]=i[itemDescription] \wedge (\exists c)(Character(c) \wedge c[Player.idPlayer]=p[idPlayer]$$

^(∃h)(Character_has_item(h)

^c[idCharacter]=h[Character_idCharacter]

^i[idItem]=h[Item_idItem]

^p[playerName]=”Eytan”))))

Expressed as Tuple Relational Algebra:

$$\square_{\text{Player.playerName,Item.itemName,Item.itemDescription}}(\sigma_{\text{Player.playerName=”Eytan”}}($$
$$\text{Item} \bowtie_{\text{Character\_has\_Item.Item\_idItem=Item.idItem}} ($$
$$\text{Character\_has\_Item} \bowtie_{\text{Character.idCharacter=Character\_has\_Item.Character\_idCharacter}} ($$
$$\text{Character} \bowtie_{\text{Player.playerId=Character.Player\_playerId}} \text{Player})))))$$

7. Functional Dependencies

Each of these functional dependencies is in BCNF because of the way we set up the schemas. Each entry in each table is given a unique id (`idPlayer`, `idCharacter`, etc.), which means that the id alone is enough to identify the specific entry assigned to it. For example, if you know that a specific character has `idCharacter` = 3, then you can determine the rest of that character’s stats from that information alone.

7.1 Player

`idPlayer → {playerName}`

7.2 Character

`idCharacter → {Player_idPlayer, Race_idRace, characterName, …, cha} (All the attributes of Character)`

7.3 Class

`idClass → {className, classDescription, hitDice}`

7.4 Race

`idRace → {raceName, raceDescription, raceSize, raceEffect}`

7.5 Item

`idItem → {itemName, itemDescription, itemValue}`

7.6 Stat_has_modifier

```
Stat_value → {modifier}
```

Each of the following functional dependencies represent a many-many relationship, and thus, it is possible for each attribute to show up multiple times. That means that the superkey of each of these tables is just the two attributes, and thus, they are all in BCNF.

7.7 Character_has_Class

```
{Character_idCharacter, Class_idClass} → {}
```

7.8 Character_has_Item

```
{Character_idCharacter, Item_idItem} → {}
```

8. Roles of Members

Eytan Kaplan

- ER Diagram
- Entity and Relational Schemas
- Functional Dependencies
- Document formatting and editing
- Presentation slides
- Sample queries

Nainoa Faulkner-Jackson

- Application Background
- Data Description
- Entity and Relational Schema descriptions
- Created sample data for testing purposes
- Presentation slides
- Sample queries