**Untitled MMO Database Design Proof of Concept**



**Designed by Phil Picinic**

**Table of Contents**

Executive Summary 3

Entity Relationship Diagram 3

Tables: 3

Players 3

Classes 4

Races 4

Levels 5

Skills 5

Characters 6

ClassSkills 7

RaceSkills 7

CharacterSkills 8

Items 9

EquipItems 10

Inventories 10

InventoryContains 11

Equipments 12

Views: 13

CharacterItemView 13

CharacterSkillView 14

PlayerCharactersView 14

Reports and their Queries: 15

Character and race skills 15

Character and class skills 15

Equip able items 15

Stored Procedures: 16

addExp() 16

affectHealth() 17

affectMana() 17

Triggers 18

Security 19

Implementation Notes / Known Problems / Future Enhancement 19

**Executive Summary**

This document represents the database design and implementation for the basics of an mmorpg game. The purpose of this database is to cover a part of database needed for building an mmo. This database includes players, their characters, as well as the characters’ stats, class, race, skills, and items. This database acts as a proof of concept to best see how the data of players and application data can be best designed for an mmo in order to communicate between client and server with minimal effort. Thus the database is designed as needs for only minimally needed data. The goal is to design a normalized database capable of handling interactions of data quickly and safely.

**Entity Relationship Diagram**



**Tables**

**Players Table**

The Players table lists all registered players and login information.

CREATE TABLE players(

playerId SERIAL NOT NULL,

email VARCHAR(70) NOT NULL UNIQUE,

password VARCHAR(24) NOT NULL, -- password in plain text for this PoC

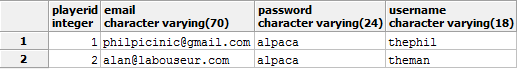
username VARCHAR(18) NOT NULL UNIQUE,

PRIMARY KEY (playerId)

);

Functional Dependencies: playerId -> email, password, username

Sample Data:



**Classes Table**

The classes table lists all possible classes of a character.

CREATE TABLE classes(

classId SERIAL NOT NULL,

name VARCHAR(16) NOT NULL UNIQUE,

description TEXT NOT NULL,

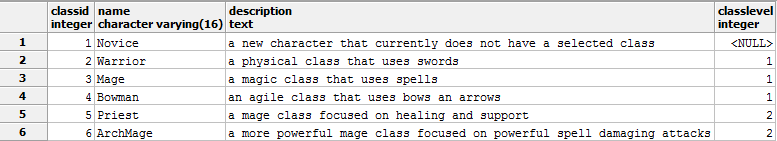
classLevel INTEGER,

PRIMARY KEY (classId)

);

Functional Dependencies: classId -> name, description, classLevel

Sample Data:



**Races Table**

The races table lists all the possible races of a character.

CREATE TABLE races(

raceId SERIAL NOT NULL,

name VARCHAR(16),

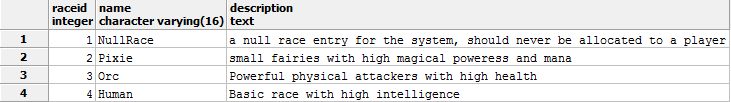
description TEXT,

PRIMARY KEY(raceId)

);

Functional Dependencies: raceId -> name, description

Sample Data:



**Levels Table**

The levels table lists all levels and required experience.

CREATE TABLE levels(

level SERIAL NOT NULL,

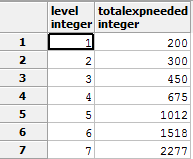
totalExpNeeded INTEGER NOT NULL,

PRIMARY KEY(level)

);

Functional Dependencies: level -> totalExpNeeded

Sample Data:



**Skills Table**

The skills table lists all the skills a character can have.

CREATE TABLE skills(

skillId SERIAL NOT NULL,

name VARCHAR(16) NOT NULL,

description TEXT NOT NULL,

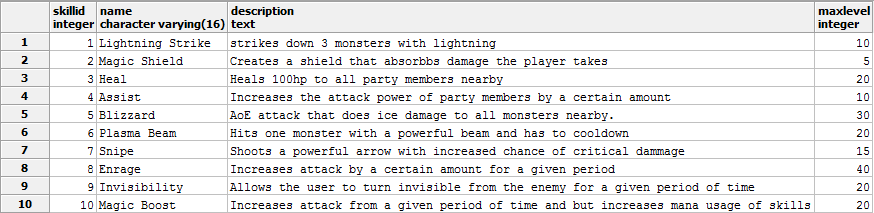
maxLevel INTEGER NOT NULL,

PRIMARY KEY (skillId)

);

Functional Dependencies: skillId -> name, description, maxLevel

Sample Data:



**Characters Table**

The characters table lists all playable characters belonging to players, their stats, and characteristics.

CREATE TABLE characters(

characterId SERIAL NOT NULL,

playerId INTEGER NOT NULL REFERENCES players(playerId),

raceId INTEGER NOT NULL REFERENCES races(raceId) check(raceId != 1),

classId INTEGER NOT NULL REFERENCES classes(classId),

level INTEGER NOT NULL REFERENCES levels(level),

characterName VARCHAR(12) NOT NULL UNIQUE,

experience INTEGER NOT NULL,

maxHealth INTEGER NOT NULL,

maxMana INTEGER NOT NULL,

health INTEGER NOT NULL check(health >= 0),

mana INTEGER NOT NULL check(mana >= 0),

strength INTEGER NOT NULL,

intelligence INTEGER NOT NULL,

endurance INTEGER NOT NULL,

speed INTEGER NOT NULL,

PRIMARY KEY (characterId),

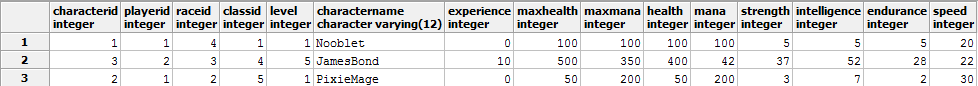
check(health <= maxHealth),

check(mana <= maxMana)

);

Functional Dependencies: characterId -> playerId, raceId, classId, level, characterName, experience, maxHealth, maxMana, health, mana, strength, intelligence, endurance, speed

Sample Data:



**ClassSkills Table**

Lists all skills a class can learn.

CREATE TABLE classSkills(

skillId INTEGER NOT NULL REFERENCES skills(skillId),

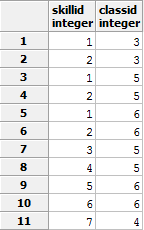
classId INTEGER NOT NULL REFERENCES classes(classId),

PRIMARY KEY (skillId, classId)

);

Functional Dependencies: skillId, classId ->

Sample Data:



**RaceSkills Table**

Lists all skills a race can learn.

CREATE TABLE raceSkills(

skillId INTEGER NOT NULL REFERENCES skills(skillId),

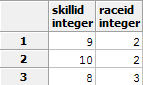
raceId INTEGER NOT NULL REFERENCES races(raceId),

PRIMARY KEY (skillId, raceId)

);

Functional Dependencies: skillId, raceId ->

Sample Data:



**CharacterSkills Table**

Lists all skills a characters knows and the level of the skill.

CREATE OR REPLACE FUNCTION checkSkillLevel(cid integer, sid integer, le integer) RETURNS boolean AS $res$

DECLARE

maxL integer;

BEGIN

maxL = (SELECT maxLevel FROM skills WHERE skillId = sid);

IF le > maxL THEN

RETURN FALSE;

END IF;

RETURN TRUE;

END;

$res$ LANGUAGE plpgsql;

CREATE TABLE characterSkills(

characterId INTEGER NOT NULL REFERENCES characters(characterId),

skillId INTEGER NOT NULL REFERENCES skills(skillId),

level INTEGER NOT NULL DEFAULT 0,

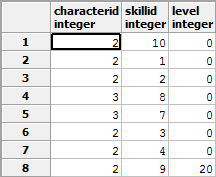
PRIMARY KEY (characterId, skillId),

check(checkSkillLevel(characterId, skillId, level))

);

Functional Dependencies: characterId, skillId -> level

Sample Data:



**Items Table**

Items table lists all items in the game.

CREATE TABLE items(

itemId SERIAL NOT NULL,

name VARCHAR(16) NOT NULL UNIQUE,

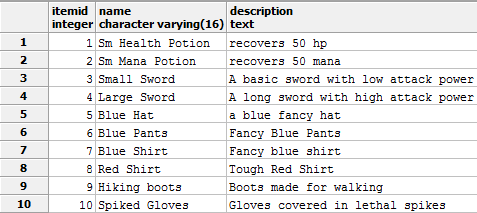
description TEXT NOT NULL,

PRIMARY KEY (itemId)

);

Functional Dependencies: itemId -> name, description

Sample Data:



**EquipItems Table**

The EquipItems Table lists all equip able items in the game.

CREATE TYPE EQUIPTYPE as ENUM('hat', 'shirt', 'pants', 'shoes', 'gloves', 'weapon');

CREATE TABLE equipItems(

itemId INTEGER NOT NULL REFERENCES items(itemId),

type EQUIPTYPE,

attack INTEGER NOT NULL DEFAULT 0 check(attack >= 0),

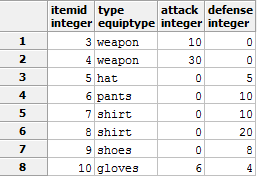
defense INTEGER NOT NULL DEFAULT 0 check(defense >= 0),

PRIMARY KEY(itemId)

);

Functional Dependencies: itemId -> type, attack, defense

Sample Data:



**Inventories table**

Lists all characters’ inventories and capacities.

CREATE TABLE inventories(

characterId INTEGER NOT NULL REFERENCES characters(characterId),

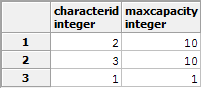
maxCapacity INTEGER DEFAULT 10,

PRIMARY KEY (characterId)

);

Functional dependencies: characterId -> maxCapacity

Sample Data:



**InventoryContains Table**

Lists all the items existing in a character’s inventory

CREATE OR REPLACE FUNCTION isBeyondMax(id integer, iid integer) RETURNS boolean AS $beyond$

DECLARE

amt integer;

maxCap integer;

BEGIN

amt = (SELECT count(characterId) FROM inventoryContains WHERE characterId = id);

maxCap = (SELECT maxCapacity FROM inventories WHERE characterId = id);

IF NOT EXISTS (SELECT itemId FROM inventoryContains WHERE characterId = id AND itemId = iid) THEN

IF amt >= maxCap THEN

RETURN FALSE;

END IF;

END IF;

RETURN TRUE;

END

$beyond$ LANGUAGE plpgsql;

CREATE TABLE inventoryContains(

characterId INTEGER NOT NULL REFERENCES characters(characterId),

itemId INTEGER NOT NULL REFERENCES items(itemId),

amount INTEGER NOT NULL DEFAULT 1,

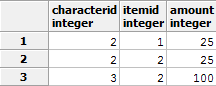
PRIMARY KEY (characterId, itemId),

check(isBeyondMax(characterId, itemId))

);

Functional Dependencies: characterId, itemId -> amount

Sample Data:



**Equipments Table**

Lists all the equipped items on characters.

CREATE OR REPLACE FUNCTION isProperType(id integer, matchType EQUIPTYPE) RETURNS boolean AS $properType$

DECLARE

actualType EQUIPTYPE;

BEGIN

IF id IS NULL THEN

RETURN TRUE;

END IF;

actualType = (SELECT type FROM equipItems WHERE itemId = id);

IF actualType = matchType THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

RETURN TRUE;

END

$properType$ LANGUAGE plpgsql;

CREATE TABLE equipments(

characterId INTEGER NOT NULL REFERENCES characters(characterId),

hatId INTEGER REFERENCES equipItems(itemId),

shirtId INTEGER REFERENCES equipItems(itemId),

pantsId INTEGER REFERENCES equipItems(itemId),

shoesId INTEGER REFERENCES equipItems(itemId),

glovesId INTEGER REFERENCES equipItems(itemId),

weaponId INTEGER REFERENCES equipItems(itemId),

PRIMARY KEY (characterId),

check(isProperType(hatId, 'hat')),

check(isProperType(shirtId, 'shirt')),

check(isProperType(pantsId, 'pants')),

check(isProperType(shoesId, 'shoes')),

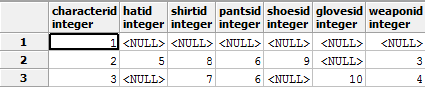
check(isProperType(glovesId, 'gloves')),

check(isProperType(weaponId, 'weapon'))

);

Functional Dependencies: characterId -> hatId, shirtId, pantsId, shoesId, glovesId, weaponId

Sample Data:



**Views**

**CharacterItemView**

Displays all the items a character has equipped and the total attack and defense.

CREATE OR REPLACE VIEW characterItemView AS

SELECT c.characterId, characterName, hi.name AS hat, si.name AS shirt, pi.name AS pants, shi.name AS shoes, gi.name AS gloves, wi.name AS weapon,

(coalesce(h.attack, 0) + coalesce(w.attack, 0) + coalesce(s.attack, 0) + coalesce(p.attack,0) + coalesce(sh.attack,0)

+ coalesce(g.attack,0)) AS totalAttack,

(coalesce(h.defense, 0) + coalesce(w.defense, 0) + coalesce(s.defense, 0) + coalesce(p.defense,0) + coalesce(sh.defense,0)

+ coalesce(g.defense,0)) AS totalDefense

FROM characters AS c

INNER JOIN equipments AS e ON c.characterId = e.characterId

LEFT JOIN equipItems AS h ON e.hatId = h.itemId

LEFT JOIN items AS hi ON h.itemId = hi.itemId

LEFT JOIN equipItems AS w On e.weaponId = w.itemId

LEFT JOIN items AS wi ON w.itemId = wi.itemId

LEFT JOIN equipItems AS s ON e.shirtId = s.itemId

LEFT JOIN items AS si ON s.itemId = si.itemId

LEFT JOIN equipItems AS p ON e.pantsId = p.itemId

LEFT JOIN items AS pi ON p.itemId = pi.itemId

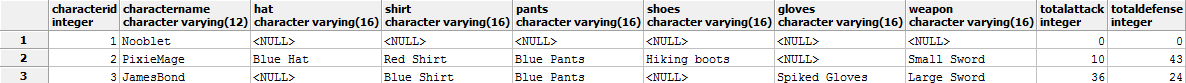
LEFT JOIN equipItems AS sh ON e.shoesId = sh.itemId

LEFT JOIN items AS shi ON sh.itemId = shi.itemId

LEFT JOIN equipItems AS g ON e.glovesId = g.itemId

LEFT JOIN items AS gi ON g.itemId = gi.itemId;

Sample Data:



**CharacterSkillView**

Displays all the skills a character knows and its level.

CREATE OR REPLACE VIEW characterSkillView AS

SELECT c.characterId, c.characterName, c.level, r.name AS race, cl.name AS class, s.name AS skill, cs.level AS skilllevel

FROM characters AS c

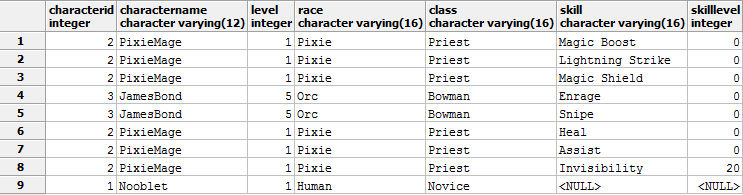
LEFT JOIN races AS r ON c.raceId = r.raceId

LEFT JOIN classes AS cl ON c.classId = cl.classId

LEFT JOIN characterSkills AS cs ON c.characterId = cs.characterId

LEFT JOIN skills AS s ON cs.skillId = s.skillId;

Sample Data:



**PlayerCharactersView**

Displays all the characters belonging to a player’s account.

CREATE OR REPLACE VIEW playerCharactersView AS

SELECT p.playerId, email, username, characterName, level, r.name AS race, cl.name AS class, experience,

maxhealth, maxmana, strength, intelligence, endurance, speed

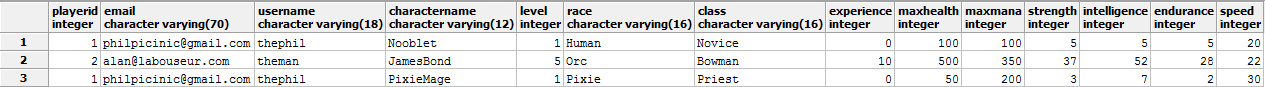
FROM players AS p

LEFT JOIN characters AS c ON p.playerId = c.playerId

LEFT JOIN classes AS cl ON c.classId = cl.classId

LEFT JOIN races AS r ON c.raceId = r.raceId;

Sample Data:



**Reports and their Queries**

**Characters and race skills**

Displays all characters with all of their race skills only.

SELECT c.characterName, r.name AS race, s.name AS skill, cs.level

FROM characters AS c

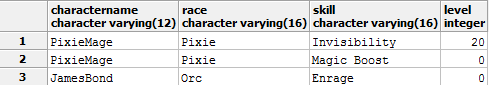
INNER JOIN races AS r ON c.raceId = r.raceId

INNER JOIN raceSkills AS rs ON r.raceId = rs.raceId

INNER JOIN skills AS s ON rs.skillId = s.skillId

INNER JOIN characterSkills AS cs ON cs.characterId = c.characterId AND cs.skillId = s.skillId;

Sample Data:



**Characters and class skills**

Displays all characters with all of their class skills only.

SELECT c.characterName, cl.name AS class, s.name AS skill, cs.level

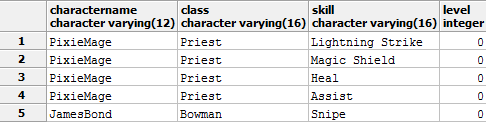
FROM characters AS c

INNER JOIN classes AS cl ON c.classId = cl.classId

INNER JOIN classSkills AS cls ON cl.classId = cls.classId

INNER JOIN skills AS s ON cls.skillId = s.skillId

INNER JOIN characterSkills AS cs ON cs.characterId = c.characterId AND cs.skillId = s.skillId;



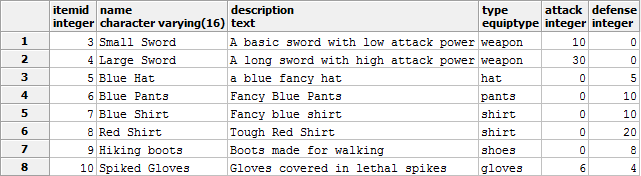
**Equip able Items**

Gives a full display of equip able items.

SELECT i.itemid, i.name, i.description, e.type, e.attack, e.defense

FROM equipItems AS e

INNER JOIN items AS i ON e.itemId = i.itemId;



**Stored Procedures**

**addExp()**

Adds experience to a character and levels them up if need be. Will also auto generate new levels when the level cap is reached.

CREATE OR REPLACE FUNCTION addExp(expAmt integer, id integer) RETURNS void AS $$

DECLARE

expNeeded int;

expHas int;

nextLevel int;

maxLevel int;

nextExp int;

BEGIN

expNeeded = (SELECT totalExpNeeded FROM levels WHERE level = (SELECT level FROM characters WHERE characterId = id));

expHas = (SELECT experience FROM characters WHERE characterId = id);

IF expHas + expAmt >= expNeeded THEN

nextLevel = (SELECT level FROM characters WHERE characterId = id) + 1;

UPDATE characters SET level = nextLevel WHERE characterId = id;

UPDATE characters SET experience = 0 WHERE characterId = id;

maxLevel = (SELECT level FROM levels ORDER BY level DESC LIMIT 1);

IF nextLevel = maxLevel THEN

nextExp = (((SELECT totalExpNeeded FROM levels WHERE level = maxLevel) \* 3) / 2);

INSERT INTO levels (totalExpNeeded) VALUES (nextExp);

END IF;

PERFORM addExp( ((expHas + expAmt) - expNeeded), id);

ELSE

UPDATE characters SET experience = (expHas + expAmt) WHERE characterId = id;

END IF;

END;

$$ LANGUAGE plpgsql;

**affectHealth()**

Increases or decreases a character’s health but will not go above maxHealth or below 0.

CREATE OR REPLACE FUNCTION affectHealth(hpAmt integer, id integer) RETURNS void AS $$

DECLARE

maxHp int;

currHealth int;

BEGIN

maxHp = (SELECT maxHealth FROM characters WHERE characterId = id);

currHealth = (SELECT health FROM characters WHERE characterId = id);

IF currHealth + hpAmt > maxHp THEN

UPDATE characters SET health = maxHp WHERE characterId = id;

ELSIF currHealth + hpAmt <= 0 THEN

UPDATE characters SET health = 0 WHERE characterId = id;

ELSE

UPDATE characters SET health = (currHealth + hpAmt) WHERE characterId = id;

END IF;

END;

$$ LANGUAGE plpgsql;

**affectMana()**

Increases or decreases a character’s mana, but will not go above maxMana or below 0.

CREATE OR REPLACE FUNCTION affectMana(manaAmt integer, id integer) RETURNS void AS $$

DECLARE

maxMn int;

currMana int;

BEGIN

maxMn = (SELECT maxMana FROM characters WHERE characterId = id);

currMana = (SELECT mana FROM characters WHERE characterId = id);

IF currMana + manaAmt > maxMn THEN

UPDATE characters SET mana = maxMn WHERE characterId = id;

ELSIF currMana + manaAmt <= 0 THEN

UPDATE characters SET mana = 0 WHERE characterId = id;

ELSE

UPDATE characters SET mana = (currMana + manaAmt) WHERE characterId = id;

END IF;

END;

$$ LANGUAGE plpgsql;

**Triggers**

There are three triggers for filling in data when a character is created or its class or race is changed. The results can be seen by the sample data from the tables characterSkills, equipments, inventories, some or all of which may have been filled in by a trigger.

CREATE OR REPLACE FUNCTION characterFill() RETURNS trigger AS $characterFill$

DECLARE

r integer;

c integer;

BEGIN

FOR r IN SELECT skillId FROM raceSkills WHERE raceId = NEW.raceId

LOOP

INSERT INTO characterSkills (characterId, skillId) VALUES (NEW.characterId, r);

END LOOP;

FOR c IN SELECT skillId FROM classSkills WHERE classId = NEW.classId

LOOP

INSERT INTO characterSkills (characterId, skillId) VALUES (NEW.characterId, c);

END LOOP;

INSERT INTO inventories (characterId) VALUES (NEW.characterId);

INSERT INTO equipments (characterId) VALUES (NEW.characterId);

RETURN NULL;

END;

$characterFill$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION skillFill() RETURNS trigger AS $sFill$

DECLARE

r integer;

c integer;

BEGIN

FOR r IN SELECT skillId FROM raceSkills WHERE raceId = NEW.raceId AND skillId NOT IN (SELECT skillId FROM characterSkills WHERE characterId = NEW.characterId)

LOOP

INSERT INTO characterSkills (characterId, skillId) VALUES (NEW.characterId, r);

END LOOP;

FOR c IN SELECT skillId FROM classSkills WHERE classId = NEW.classId AND skillId NOT IN (SELECT skillId FROM characterSkills WHERE characterId = NEW.characterId)

LOOP

INSERT INTO characterSkills (characterId, skillId) VALUES (NEW.characterId, c);

END LOOP;

RETURN NULL;

END;

$sFill$ LANGUAGE plpgsql;

CREATE TRIGGER characterFill

AFTER INSERT ON characters FOR ROW EXECUTE PROCEDURE characterFill();

CREATE TRIGGER classSkillUpdate

AFTER UPDATE OF classId ON characters FOR ROW EXECUTE PROCEDURE skillFill();

CREATE TRIGGER raceSkillUpdate

AFTER UPDATE OF raceId ON characters FOR ROW EXECUTE PROCEDURE skillFill();

**Security**

This database design currently has two users. The first is “application” which is for the server accessing it with full access. The second is “sysadmin” for viewing player accounts.

-- users

CREATE USER sysadmin WITH PASSWORD 'alpaca';

CREATE USER application WITH PASSWORD 'alpaca';

-- permissions (grant / revoke)

REVOKE ALL PRIVILEGES ON characters FROM sysadmin;

REVOKE ALL PRIVILEGES ON classes FROM sysadmin;

REVOKE ALL PRIVILEGES ON races FROM sysadmin;

REVOKE ALL PRIVILEGES ON characterSkills FROM sysadmin;

REVOKE ALL PRIVILEGES ON skills FROM sysadmin;

REVOKE ALL PRIVILEGES ON classSkills FROM sysadmin;

REVOKE ALL PRIVILEGES ON raceSkills FROM sysadmin;

REVOKE ALL PRIVILEGES ON equipments FROM sysadmin;

REVOKE ALL PRIVILEGES ON inventories FROM sysadmin;

REVOKE ALL PRIVILEGES ON levels FROM sysadmin;

REVOKE ALL PRIVILEGES ON inventoryContains FROM sysadmin;

REVOKE ALL PRIVILEGES ON items FROM sysadmin;

REVOKE ALL PRIVILEGES ON equipItems FROM sysadmin;

**Implementation Notes / Known Problems / Future Enhancements**

The implementation went very well. One of the difficulties was how to do certain things without blowing the scale out of proportion. For example equip able items could be completely separate for items and then they can separate inventories or complicatedly combined inventories. For simplicity sake I chose to just have equip able items subtype items. There is a similar compromise with classes where in a class can branch off to a higher more powerful class, such as Mage to Priest or Archmage. I left in class levels such that Priest at level 2 is higher than mage at level 1, but the correlation that a Mage can become a Priest is not there. For a future enhancement, this could probably be done with an associative table that maps one classId, to a classId it can become and possibly at the character level that change can happen.

The biggest challenge were some of the more complex check constraint that call user functions. The problem here is there are queries that are made, but since the data is checked before being inserted some of the data is missing. To fix this with the skill level constraint I passed in the new skill level into the function as well. However for the inventory capacity check, the problem is more complicated as I can’t pass in a count. Rather what I did was pass in both the characterId and itemId and checked for it in the associative table. If the entry didn’t exist, that would mean it must either be a new item being inserted or a previous item being updated to a new one. At this point regardless of action, it checks if the inventory is already full and fails the constraint if it is. It works if you insert a new item and inventory is already full. However it creates a side effect where if the inventory is full and you update one item to another item it fails even though you aren’t going past capacity. This is completely bad because this should happen anyway as a Red Potion doesn’t magically become a Large Sword. If anything the two items might be traded, but then the red potion should be removed before the large sword added.

One thing that I did not like and might consider a problem was the use of serial, an auto incrementing integer as primary keys. It made inserting test data difficult as if there were errors the sequence would be out of order and dropping and recreating tables caused issues reinserting data. Because of this I had to be very careful when recreating the whole schema and test data. It would also be a huge problem when the database becomes used in clusters and there is data replication among many database servers. After a little research I came along the UUID type, which would be much better used in some cases as a primary key. Other enhancements could be adding or removing needed attributes and just generally growth of the database to cover more features like monsters, maps, guilds, parties, etc.