Database model implementation of RPGGame

Adrián Barňák

April 29, 2025

Introduction

This is documentation of implementing a database design for RPG game system.

1 Logical-physical model mapping

Entity	**Atribute**	**Type, constraints**
		Classes
class_id	PK	SERIAL
class_name		VARCHAR(45), NOT NULL
action_points_modifier		NUMERIC, NOT NULL, ≥ 0
damage_modifier		NUMERIC, NOT NULL, ≥ 0
inventory_bonus		NUMERIC, NOT NULL, ≥ 0
		Character
character_id	PK	SERIAL
character_name		VARCHAR(45), NOT NULL
action_points		NUMERIC, ≥ 0
armor_class		NUMERIC, ≥ 0
class_id	FK	INT, NOT NULL (**Classes**)
strength, dexterity, constitution, intelligence		NUMERIC, ≥ 0
max_health, health		NUMERIC, ≥ 0
inventory_capacity		NUMERIC, ≥ 0
inventory_capacity_reached		NUMERIC, ≥ 0 AND ≤ 0 invent
		Spells
spell_id	PK	SERIAL
spell_name		VARCHAR(45), NOT NULL
damage, action_points_cost		NUMERIC, ≥ 0
required_strength, dexterity, constitution, intelligence		NUMERIC, ≥ 0
modifying_attribute		VARCHAR(45), IN ('strength', 'c
		Items
item_id	PK	SERIAL
item_name		VARCHAR(45), NOT NULL
damage, weight, action_points_cost		NUMERIC, ≥ 0
		Character_has_It

Entity	Atribute	Type, constraints
character_id	PK, FK	INT, (**Character**)
item_id	PK, FK	INT, (**Items**)
		Combats
combat_id	PK	SERIAL
char1_id, char2_id, winner_id	FK	INT, (**Character**)
		Arena_items_has_
items_id, combat_id	PK, FK	INT, (**Items**), INT, (**Combats**)
		Combat_log
log_id	PK	SERIAL
combat_id	FK	INT, (**Combats**)
round_num, event_num		INT, ≥ 0
event_type		VARCHAR(45), IN ('enter_comba
char1_id, char2_id	FK	INT, (**Character**)
char1_ap, char2_ap, char1_health, char2_health		NUMERIC, ≥ 0
d20_value		INT, BETWEEN 1 AND 20
attack_damage		NUMERIC, ≥ 0
item_id	FK	INT, (**Items**)
spell_id	FK	INT, (**Spells**)
char1_ap_after, char2_ap_after		NUMERIC, ≥ 0
char1_health_after, char2_health_after		NUMERIC, ≥ 0
		Item_Ownership_H
history_id	PK	SERIAL
item_id, character_id, log_id	FK	INT, (**Items**), INT, (**Character**)
		Arena_Items_His
arena_history_id	PK	SERIAL
item_id, log_id	FK	INT, (**Items**), INT, (**Combat_lo**

Purpose of the Tables

- **Classes** – Stores information about character classes and their modifiers.
- **Character** – Characters that participate in battles.
- **Spells** – Spells usable by characters.
- **Items** – Items available to characters or in the arena.
- **Character_has_Items** – Many-to-Many relationship between Characters and Items.
- **Combats** – Records of battles between characters.
- **Arena_items_has_Items** – Items available in the arena during battles.
- **Combat_log** – Ongoing battle events.
- **Item_Ownership_History** – Who owned which item and when.
- **Arena_Items_History** – Historical composition of the arena.
- **Character** $\rightarrow$ **Classes**: Each character belongs to one class.

- **Character_has_Items**: Many-to-Many relationship between **Character** and **Items**. One item belongs to at most one character.
- **Combats**: Each battle is between two characters.
- **Arena_items_has_Items**: Many-to-Many relationship between **Items** and **Combats**.
- **Combat_log**: Each battle record contains foreign keys to **Combats**, **Character**, **Items**, **Spells**.
- **Item_Ownership_History**: Records the history of item ownership.
- **Arena_Items_History**: Historical records of items in the arena.

Constraints

The tables include several constraints to ensure data consistency and integrity:

- All numerical values like `action_points`, `damage`, `health`, `weight` must be greater than or equal to zero.
- The attribute `inventory_capacity_reached` in the **Character** table must not exceed `inventory_capacity`.
- **Spells** are constrained such that `modifying_attribute` can only have values: 'strength', 'dexterity', 'constitution', 'intelligence'.
- In the **Combat_log** table, the attribute `event_type` must be one of: 'enter_combat', 'spell_attack', 'loot_item', 'rest_character', 'reset_round', 'throw_item', 'attack_item'.
- The attribute `d20_value` must be between 1 and 20.
- All foreign keys (FK) ensure entity relationships, preventing cases like a character owning a non-existing item or a combat record referencing a non-existing character.

2 Functions

Sp_enter_combat

This function is used to initialize a new battle between two characters. The sequence of processing and conditions are as follows:

- **Conflict Check:** The function checks if either character (`p_char1_id`, `p_char2_id`) is already involved in an ongoing battle (i.e., a battle without a set `winner_id`).
- **Insert Combat:** If no conflicts are found, a new record is created in the **Combats** table with the given `combat_id`.
- **Load Action Points and Health:** From the **Character** table, the `action_points`, `health`, and `inventory_capacity` values are retrieved for both players.

- **Calculate Capacity Bonus:** From the **Classes** table, the `inventory_bonus` is obtained and added to the original `inventory_capacity`:

$$\texttt{inventory\_capacity} := \texttt{inventory\_capacity} + \texttt{inventory\_bonus}$$

- **Record Entry into Combat:** A record is inserted into **Combat_log** with the type `enter_combat`, documenting the initial state of both characters (AP, health).
- **Select 3 Random Items:** The function selects 3 items from the **Items** table that are:
  - Not owned by any character (**Character_has_Items**),
  - Not already placed in another arena for an ongoing battle.

  These items are inserted into **Arena_items_has_Items** for this combat.
- **Log Arena and Inventories:**
  - The current state of the arena is recorded into **Arena_Items_History**.
  - All items owned by both fighters are recorded into **Item_Ownership_History**.

Execution Conditions:

- Both fighters must not be involved in any other active battle.
- There must be enough free items available outside other arenas.

Updated Data:

- `Combats` – new battle record.
- `Character` – updated inventory capacity.
- `Combat_log` – record of the start of the battle.
- `Arena_items_has_Items`, `Arena_Items_History`, `Item_Ownership_History` – item state and history.

Sp_rest_character

This function ends a battle between two characters, sets the winner, restores the health and action points of the characters according to their attributes and class. It also clears the arena of items and logs these events into the appropriate logs.

Execution Conditions:

- There must exist a battle between `p_character1_id` and `p_character2_id` that is not yet finished (`winner_id IS NULL`).

Calculation of new AP:

$$\texttt{new\_AP} = (\texttt{dexterity} + \texttt{intelligence}) \times \texttt{action\_points\_modifier}$$

Process:

- Find the current `combat_id` for the given characters.
- Load the last record from `combat_log` and prepare for a new event (`event_num + 1`).
- Load the current health of both characters.
- Determine the winner based on who has positive health:
  - If `health > 0`, the character is the winner.
  - If both have `health <= 0`, winner = 0 (none).
- Update the `Combats` table - setting the `winner_id`.
- Calculate the new action points for both players according to the formula above.
- Restore the health of characters to their `max_health`.
- Remove all arena items for this combat (`DELETE FROM Arena_items_has_Items`).
- Log the event of type `rest_character` into `combat_log` with the new AP and health values.
- Update `Item_Ownership_History` for both characters - recording current item ownerships.

Result: The battle ends, characters are regenerated, and the arena is cleared.

Sp_reset_round

This function is called when none of the characters have enough action points (AP) to perform another spell. It serves to restore the characters' action points and continue the battle in a new round.

Process Flow:

- First, the existence of the given battle is verified using `p_combat_id`. If the battle does not exist, an exception is raised.
- It is checked whether the battle is not already finished (i.e., `winner_id` is not NULL). If so, the function ends with an exception.
- The number of the last round is loaded and incremented by 1 (`v_round_num + 1`).
- For each character, the attributes `dexterity`, `intelligence`, and the `action_points_modifier` are loaded.
- New action points are calculated for both characters according to the formula:

$$\texttt{new\_AP} = (\texttt{dexterity} + \texttt{intelligence}) \times \texttt{action\_points\_modifier}$$

- These values are updated in the **Character** table for each character.
- The current health of both characters is retrieved to be recorded in the log.

- A new entry is inserted into the **Combat_log** table with the event type `reset_round`, recording the new action points and health.
- All current items in the arena for this battle are logged into **Arena_Items_History**.
- The current item ownerships for both characters are saved into **Item_Ownership_History**.

Result: Characters gain new action points, the battle continues in the next round, and all changes are recorded in the logs and history.

sp_cast_spell

This function performs a magical attack between two characters in an ongoing battle.

Execution Conditions:

- A battle between characters `p_caster_id` and `p_target_id` must be ongoing (`winner_id IS NULL`).
- The caster must have sufficient attributes (`strength`, `dexterity`, `intelligence`, `constitution`) to use the spell.
- The caster must have enough action points (AP) after considering the effective cost of the spell.

Formulas:

- **Effective spell cost:**

$$effective\_cost = spell\_ap\_cost \times \left(1 - \frac{class\_ap\_modifier + intelligence}{100}\right)$$

- **Caster's AP after spell:**

$$caster\_ap\_after = caster\_ap - effective\_cost$$

- **Attack damage:**

$$attack\_damage = \{\ spell\_damage \times \left(1 + \frac{class\_damage\_modifier + strength + modifying\_attrib}{20}\right.$$

- **Target's new health:**

$$target\_health\_after = \max(target\_health - attack\_damage, 0)$$

Process:

- Verify the battle and the caster's attributes.
- Calculate the effective spell cost based on intelligence and class modifier.
- If the caster has enough AP, roll a 20-sided die (`random()`).
- If the roll exceeds the target's `armor_class`, calculate the attack damage.
- Update the target's health and the caster's action points.
- Log the event into **Combat_log**.
- If the target dies, call `sp_rest_character`.
- If neither character has enough AP for a spell, call `sp_reset_round`.

sp_loot_item

This function allows a character to obtain an item from the arena during a battle.

Execution Conditions:

- The battle (`p_combat_id`) must exist and must not be finished.
- The character (`p_character_id`) must be a participant of the battle.
- The item (`p_item_id`) must be present in the arena of the given battle.
- The character's inventory must have sufficient capacity to accommodate the item's weight.

Formulas:

- **Capacity Check:**

$$inventory\_capacity\_reached + item\_weight \leq inventory\_capacity$$

- **Capacity Update:**

$$inventory\_capacity\_reached\_new = inventory\_capacity\_reached + item\_weight$$

Process:

- Verify the existence of the battle by `p_combat_id`.
- Verify if the character is a participant in the battle and if the battle is still ongoing.
- Check if the item is present in the arena for the given battle.
- Retrieve the current inventory capacity and item weight.
- If there is enough space in the inventory, update `inventory_capacity_reached` and add the item to **Character_has_Items**.
- Remove the item from **Arena_items_has_Items**.
- Insert a record of looting the item into **Combat_log**.
- Update the **Arena_Items_History** and **Item_Ownership_History** tables.

f_effective_spell_cost

This auxiliary function is used to calculate the effective cost of a spell for a given character. The result is influenced by the base spell cost, the character class's action points modifier, and the character's intelligence.

Formula:

$$effective\_cost = spell\_ap\_cost \times \left(1 - \frac{class\_ap\_modifier + intelligence}{100}\right)$$

Conditions:

- The values of `class_ap_modifier` and `intelligence` are set using the `COALESCE` function to 0 if they are `NULL`.

Process:

- The base spell cost is retrieved from the **Spells** table.
- The class action points modifier and the character's intelligence are retrieved.
- The effective cost of the spell is calculated using the formula above.
- The function returns the effective cost as a **NUMERIC** type value.

3 Description of Indexes

To optimize query speed and ensure efficient access to frequently queried data, the following indexes have been created:

Combat_log

- **idx_combatlog_combat_id** – speeds up the retrieval of records with the same combat identifier, used when filtering or obtaining combat history.
- **idx_combatlog_char1_id**, **idx_combatlog_char2_id** – quick access to all events related to a specific character.
- **idx_combatlog_item_id**, **idx_combatlog_spell_id** – optimizes queries tracking the use of a specific item or spell.
- **idx_combatlog_event_type** – speeds up filtering events by their type.
- **idx_combatlog_combat_round** – quickly retrieves the last round of a combat.

Combats

- **idx_combats_char1_id**, **idx_combats_char2_id**, **idx_combats_winner_id** – speeds up finding active or finished battles by participant or winner.

Character_has_Items

- **idx_character_items** – speeds up retrieval of all items belonging to a character.
- **idx_items_character** – optimizes access to an item's owner.

Arena_items_has_Items

- **idx_arena_items_combat** – efficient access to items assigned to a specific battle.
- **idx_arena_items_id** – speeds up finding arena items by their identifier.

Item_Ownership_History

- **idx_item_ownership_item_id**, **idx_item_ownership_character_id**, **idx_item_ownership_log** – ensures quick access to the ownership history of items for a specific character or event.

Arena_Items_History

- **idx_arena_history_item_id**, **idx_arena_history_log_id** – efficiently filters historical records of arena items by item or event.

Spells

- **idx_spells_mod_attr** – speeds up the retrieval of spells based on their modifying attribute (e.g., strength or intelligence spells).

Character

- **idx_character_class_id** – speeds up queries searching for characters by their class.

4 Instructions for Loading Sample Data and Acceptance Tests

Basic rules for loading data:

- Maintain consistency and foreign keys between tables.
- Before inserting data, it is recommended to clean the tables using `TRUNCATE` and reset sequences using `ALTER SEQUENCE`.
- Identifiers (e.g., `class_id`, `character_id`) must correspond with referenced values.
- All values must meet the validation constraints defined in the tables (e.g., numerical limits).

Order and constraints for data insertion:

- **Classes** – Must be inserted first.
- **Items** – Can be inserted independently.
- **Spells** – Must contain valid values including `modifying_attribute`.
- **Character** – Can be inserted after **Classes**.
- **Character_has_Items** – Only after **Character** and **Items**.
- Functions are inserted afterward.

Function testing:

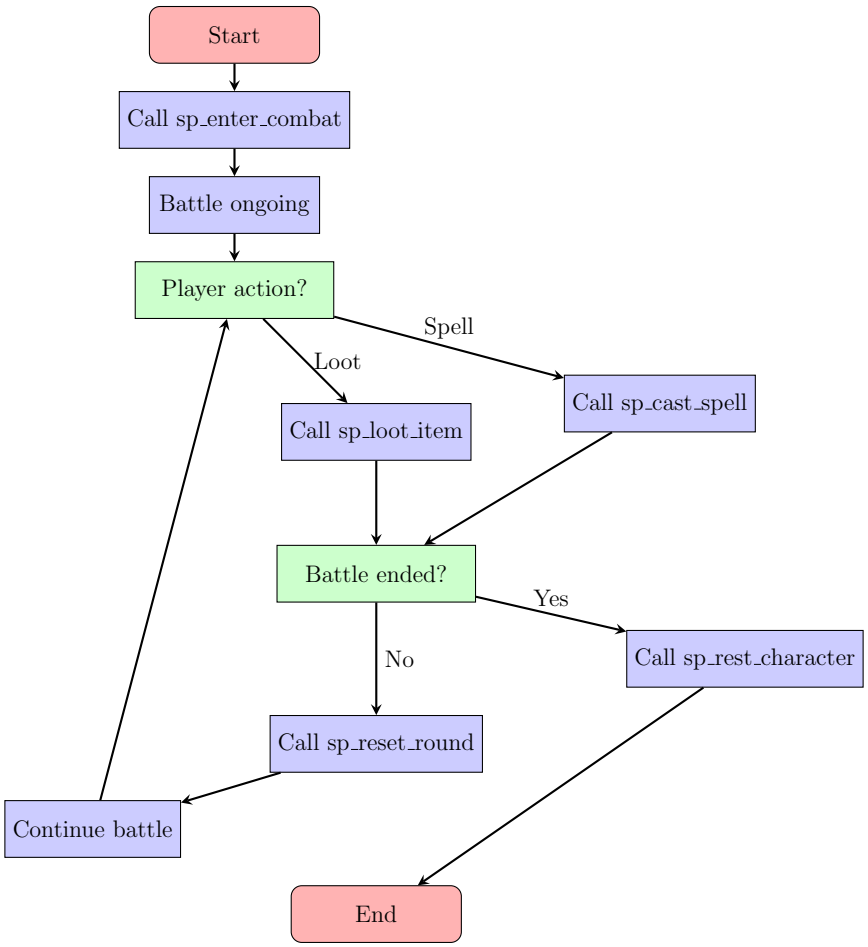
- `sp_enter_combat` – Verify that the characters are not already in a battle and that enough free items are available.
- `sp_cast_spell` – Verify sufficient attributes and AP. Check the formula for spell cost.
- `sp_loot_item` – Verify the character's inventory capacity and item availability in the arena.
- `sp_reset_round`, `sp_rest_character` – Verify battle end or battle reset conditions.

Database maintenance:

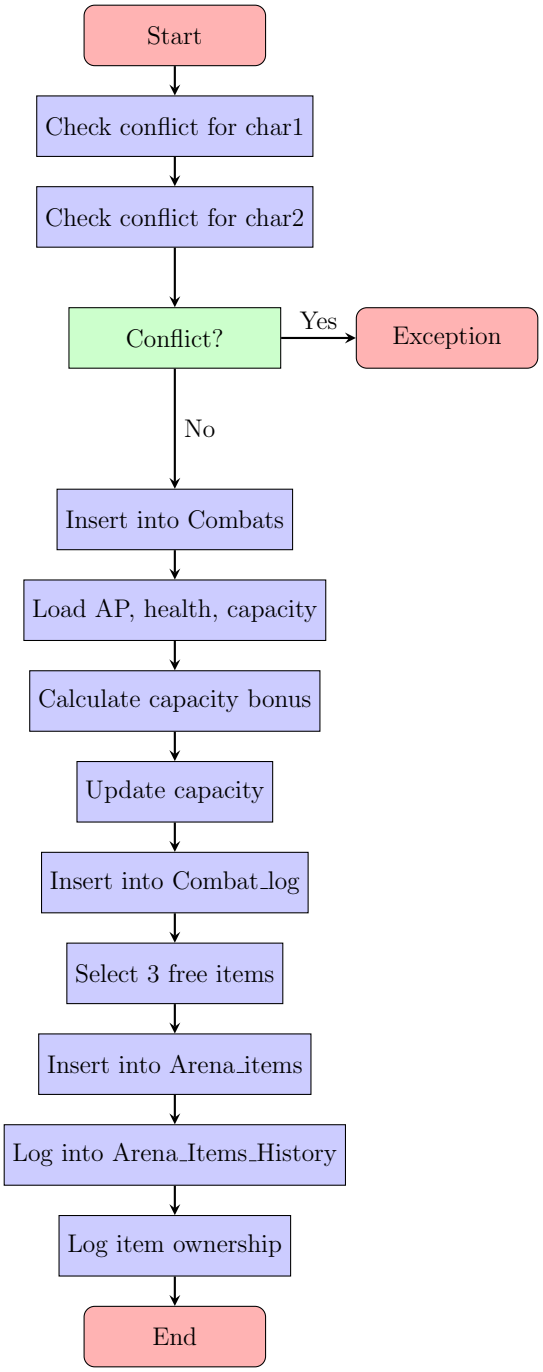
- Use `TRUNCATE TABLE` to clear the tables and `RESTART IDENTITY CASCADE` to reset IDs.
- Restore sequences using `ALTER SEQUENCE`.

5 Activity Diagrams

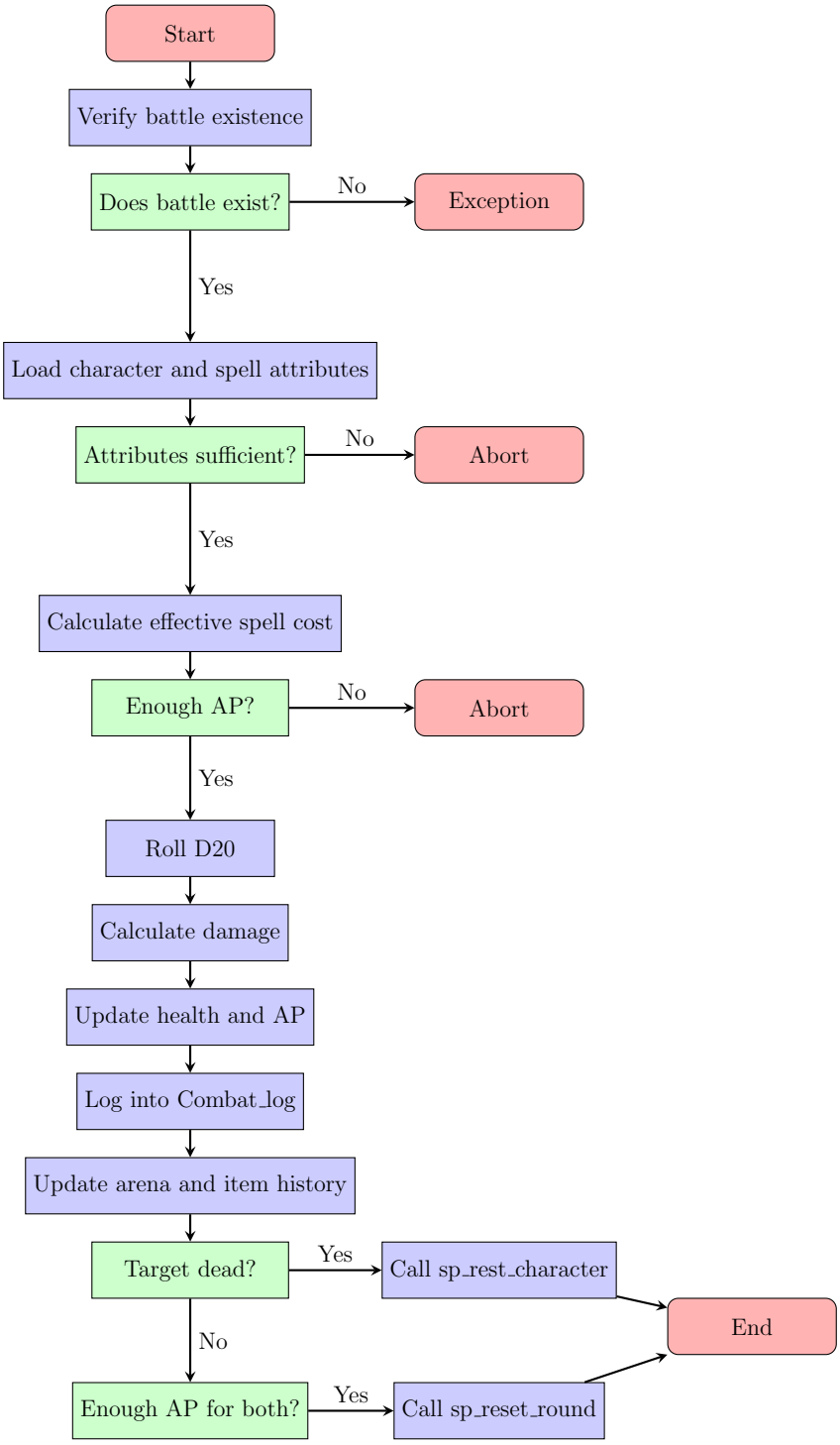
Overall Battle Process Diagram



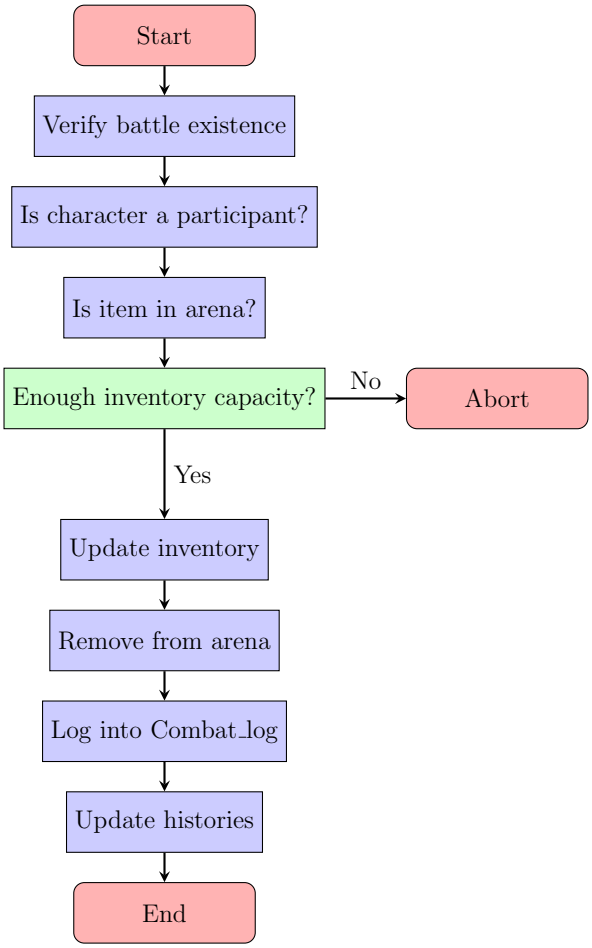
sp_enter_combat



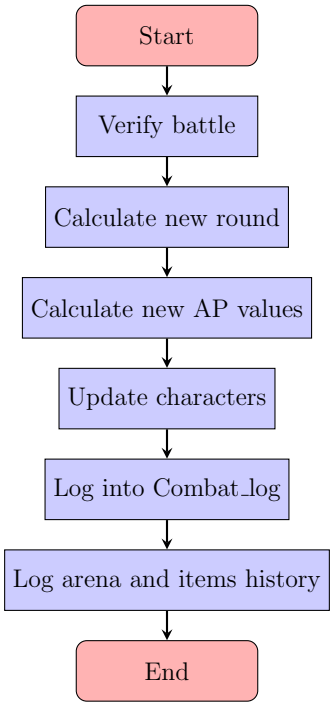
sp_cast_spell



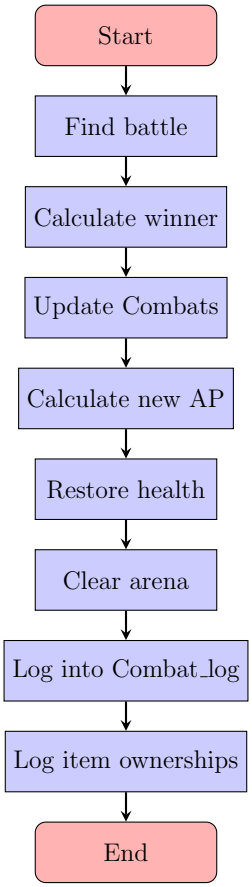
sp_loot_item



sp_reset_round



sp_rest_character



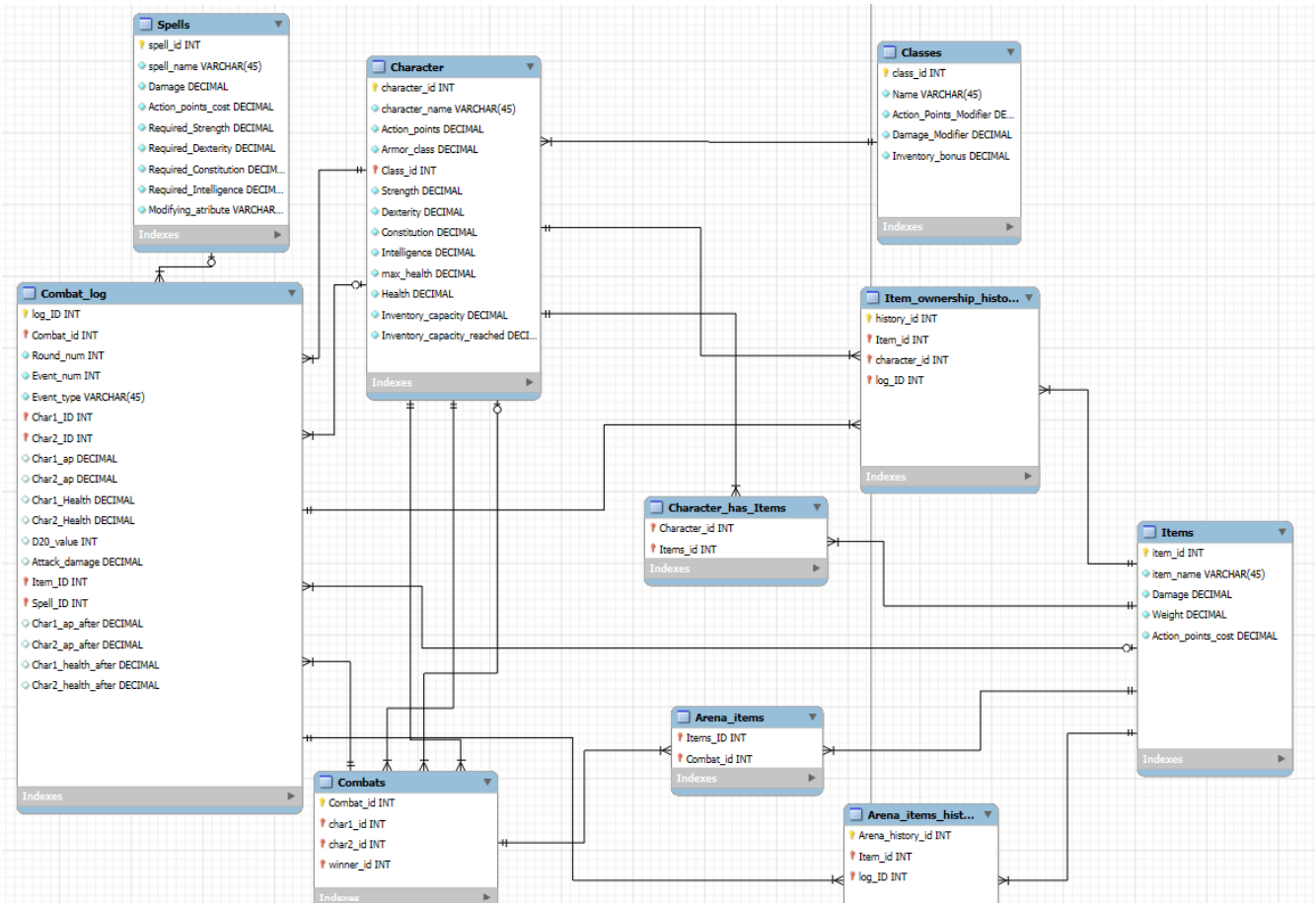


Figure 1: RPG database diagram

6 Summary

The implemented model and game system represent a solution for managing an RPG battle system that is based on consistent logic, clearly defined rules, and a high degree of data consistency. Thanks to well-designed procedures and detailed tracking of all game actions, the system is ready for further expansion and adaptation for more complex game scenarios.

This system is ready for deployment in both simulated and real environments and provides a solid foundation for the development of an RPG game.