Bot!Battle! Database

Now listen up here guys, 'cause this part is important.

- All integers are being stored in the database as INT(11).
- Since the version of MySQL we're using doesn't support JSON objects, all JSON objects are being stored as TEXT variables. Many times when you see a data type of TEXT, you may as well think 'oh, that's likely a JSON object'.
  - Discussed with Dr. Blum. The beauty of the JSON type is that you can read/write to a specified path inside of a stored JSON object. The performance gains from this ability will be very significant in the case of test arena matches/turn by turn computation.
  - He agrees and has made it a priority to give me access to a server (possibly the actual IBM virtual machine) where I will set up and manage a mysql server for you guys to use. This will hopefully be happening next week.
  - Dr. Blum has made “a complete mess”(his words) of things on the current DB server in use by the website team. Thats why he doesn't even want to try to update it, but rather wants to start from scratch.
- While not set-in-stone, the database is very close to being finalized, so bring up any concerns you have with it like really really really soon.
- PhpMyAdmin
  - http://hbgwebfe.hbg.psu.edu/phpmyadmin/
  - u: bbweb
  - p: sun16event
- You may access the mysql database to add or remove records over SSH.
  - The host name is hbgwebfe.hbg.psu.edu, the username is 'bbweb', and the port number is 22. The password is 'sun16event'.
  - Log on to mysql by entering 'mysql -u bbweb -p', typing the password 'sun16event', and then 'use bbweb'.
  - ***DO NOT DROP THE TABLES DO NOT DROP THE TABLES DO NOT DROP THE TABLES DO NOT DROP THE TABLES DO NOT DROP THE TABLES!!!!!!!!!!!***
- Tables which the game evaluation team really need to look at are preceded with the blue tag **[GAME EVALUATION TEAM]**.
- Tables which the graphical display team really need to look at are preceded with the red tag **[GRAPHICS TEAM]**.

Comment on Changes:

- I made a lot of them. I am pretty confident that this design will work and that if all teams follow this integration will go smoothly. However, I may have made mistakes that will be much more obvious to your eyes than mine, so please read this carefully and make sure it makes sense.
- Furthermore, all of this is open to discussion. If you don't like a change I made, don't know why I made it, or think something can be accomplished in a cleaner or more efficient way, I am all ears.
- The semester is going to be over before we all know it, and ironing out the database design needs to be made top priority by all teams. I'm sorry I couldn't get this done much earlier.

Other Thoughts:

- Contests === Tournaments and we are migrating towards calling them tournaments but website team doesn't want to change the tables to be called tournaments instead of contests at this time.

1. Website team needs to provide an HTTP API for the display team to get a list of bots available for use in a particular challenge by the currently logged in user. Also a separate call to get a list of available public bots for that challenge. Like we discussed for now, it would be fine just to give them the bot_ids and not a version number. (Theres no point in them writing code to do this because the website team will have to write something nearly identical to support their bot management page/ allow users to play challenges.)
2. WEBSITE-TEAM: When a challenge is added to the system, game developers will expect to be able to upload, then later access any static assets (currently only images but could be extended in the future) that they may need to display the game.
    1. Suggestion: Store the assets in the local file system at [SomeCommonPlace]/assets/challengeId. Then use /assets/[challenge_id]/ as the base uri for a challenge's static assets.

[WEBSITE TEAM]
Table: users
uid: integer → Primary Key
account: varchar(30) → Unique
display: varchar(80)
email: varchar(80)
password: char(86)
admin: boolean
reg_date: timestamp
SALT: char(86)

Description: The users table is for storing the basic user data. We are using SHA512 to encode the password, which is salted with the string in the SALT field. Currently password and SALT are set to varchar(1024), but it should be changed to the final length of the hashed string in both cases. If we continue to use SHA512 in base64, then it would be 86 characters. Since the unique SALT must be stored and is generated at account creation and password change, we store it in the users table.

[WEBSITE TEAM]
Table: category
category_id: integer → Primary Key
name: varchar(32) → Unique

[WEBSITE TEAM]
Table: difficulty
difficulty_id: integer → Primary Key
name: varchar(32) → Unique

Description: Since MySQL does not support predefined enums, this seems to be the best way to accomplish the same task.

[GAME EVALUATION TEAM] [GRAPHICS TEAM]
Table: languages
language_id: int(11) → Primary Key
name: varchar(32)

Description: Used as foreign key in the tables dealing with code that may be in one of several supported languages. Though not strictly necessary, the foreign key constraint will be a nice check to protect against typos and such. Changed my mind about primary key. In the future I think its possible additional information could be tied to a language for various purposes such as a list of allowed file extensions for upload, an expected file/class name, the name/path of the compiler on the server, etc. Therefore for future compatibility its best to just use an id as the primary key as it is more idomatic.

[WEBSITE TEAM] [GAME EVALUATION TEAM]
Table: challenges
challenge_id: integer → Primary Key
creator_uid: integer → Foreign Key (users.uid)
difficulty: integer → Foreign Key (difficulty.difficulty_id)
category: integer → Foreign Key (category.category_id)
name: varchar(80)
description: mediumtext
creation_date: timestamp
player_count: integer
active: boolean

Description: The challenges table stores the basic information about challenges, allowing them to be referenced by other tables and easily displayed in a list format as necessary within the web application. Relevant lessons are mapped to challenges using the challenges_lessons_map table.

Field Descriptions:

- The game evaluation team will need to read these player counts from this table in order to verify that they have been given a valid number of bots to run a match. Otherwise this table is fully managed by the website team.

[WEBSITE TEAM] [GAME EVALUATION TEAM]
Table: challenges_code
challenge_id: integer → Primary Key, Foreign Key (challenges.challenge_id)
new_version: bit(1)
main_class_name: varchar(20)
compiled_code_archive: longblob

Description: Separate table because reads/writes will be expensive due to tuple size in this table, which all other fields in challenges table will be very small in size. Main idea is game designers should be

allowed to replace their source code with a new source file without having to delete and recreate the entire challenge. I envision a create challenge webform where admin enters all the information for the challenges table and uploads a zip archive with all the class files necessary to evaluate a bot for this challenge. Then another simple update form that allows you to upload a new zip file to replace the old one (as well as change the description/difficulty/category etc).

The new_version field indicates to the Game Evaluation team that the compiled_code_archive has been updated. It is set to true when the website team allows a user to upload new code. Once the Game Evaluation Team takes note of this, they will set the new_version flag to false.

Limitations:

1. No source code. Apparently Dr. Blum specified he wanted game designers to upload byte code directly. Presumably this is because he doesn't want you guys to worry about checking for compiler errors and dealing with all that. In the future, I think there should be a game designer arena comparable to the test arena for bots. But thats easily a whole other semesters worth of work.
2. No revisions. Historical bot rankings meaningless if replaced with an easier/harder challenge?

[WEBSITE TEAM]

Table: lessons

lid: integer → Primary Key
uid: integer → Foreign Key (users.uid)
difficulty: integer → Foreign Key (difficulty.difficulty_id)
category: integer → Foreign Key (category.category_id)
name: varchar(80)
description: mediumtext
video: varchar(1024)

Description: The lessons table stores the basic information about lessons, allowing them to be referenced by other tables and easily displayed in a list format as necessary within the web application. Video is currently stored as a longblob, but it may be better off hosted elsewhere (e.g. YouTube) and instead embedded. In this case, a varchar of appropriate length would suffice. It may also be better to instead have a relative file path to the video here.

[WEBSITE TEAM]

Table: challenge_lessons_map

lesson_id: integer → Primary Key, Foreign Key (lessons.lid)
challenge_id: integer → Primary Key, Foreign Key (challenges.challenge_id)

Indexes:

- Extra index on challenge_id that way given a challenge, you can lookup all related lessons efficiently. The primary implies there will be an index on <lid, challenge_id>, so any queries for lid will be efficient (you can find all related challenges efficiently, (and obviously the combination of both would be efficient but that wouldn't make sense for this table since theres no other attributes)Given a lesson, you can lookup all related challenges efficiently.

- Note the order of these attributes in the primary key is important because you need the extra index on whichever attribute will be the second attribute in the primary key. This may not make sense if you don't have experience with indexes/have taken the advanced DB class. I can explain how indexes work much better in person, so feel free to ask.

Description: Provides bidirectional efficient lookup of lessons related to a challenge and challenges related to a lesson.

[WEBSITE TEAM] [GAME EVALUATION TEAM]

Table: contests

contest_id: integer → Primary Key
challenge_id: integer → Foreign Key (challenges.challenge_id)
admin_uid: integer → Foreign Key (users.uid)
category_id: integer → Foreign Key (category.category_id)
name: varchar(80)
description: mediumtext
matches_per_user_in_initial_round: integer
users_in_bracket: integer
creation_date: timestamp
run_date: datetime
finished: boolean

Description:
The contests table stores the basic information about the contests. Each contest has an administrator, category, and associated challenge (game).

New Fields:

- **matches_per_user_in_initial_round:** The tournament will consist of an initial round where all bots play this number of matches (wont be exact because thats an NP-hard problem to solve in number of bots is allowed to be any value, but will be close). They will be scored based on their performance and the top ranked will enter the main single-elimination bracket.
- **users_in_bracket:** This must be a power of challenges.player_count. E.g. for 3 player games, this must be 3^k, where k is an integer greater than 0.
- **finished:** Will be set By the Game Evaluation Team after tournament is complete. Read by the website team when a user requests the view tournament page, nothing will be displayed if entire tournament has not been computed yet. (e.g. a true value for this means that for each match in the tournament, there exists a tuple in the contest_mapping table which points to a match in the matche_turns table which contains JSON object containing all data needed by the Display module team in order to display the match.

Team Responsibilites:

- Website:
  - There will be a webform that will allow admins to add challenges to the system and will add a tuple to this table.

    - Dr. Blum has indicated that he would like a tournament bracket type display, with the administrator allowing each round to become publicly visible all at once
        - So I think a table view should be fine for now, can extend to a bracket view in the future if truly desired.
        - I will add this to my list of things to ask him about explicitly. And revise this if he declares he really wants a bracket.
        - I'm trying to save you guys the pain of having to draw the bracket. However all the information will still be available such that a bracket could be constructed
- Game Evaluation:
    - Just sets the finished field to true, see description of finished field above.

[WEBSITE TEAM] [GAME EVALUATION TEAM]

Table: contests_registration

contest_id: integer	→ Primary Key, Foreign Key (contests.contest_id)
uid: integer	→ Primary Key, Foreign Key (users.uid)
bot_id: integer	→ Foreign Key (bot_versions.bot_id)
bot_version: integer	→ Foreign Key (bot_versions.version_id)

Limitations:

1. It seems this design implies that all contest will be open for registration by all users. That is there is no notion of a “private” contest where an admin invites a select group of users. I think that is fine for now, but wanted to make note of this because I think a complete system should definitely have this feature.
2. Theres also the notion of a contest be open to registration by all, but the results only being accessible to users who have registered and completed in the tournament. Again this design doesn't support this use case currently, but it shouldn't be an earth-shattering change to add support for this later.

Description: The **contests_registration** table tracks the user who have signed up for a given contest, as well as the bot that they have registered to use within the contest. The bot must already exist within the system.

New Fields:

- bot_version: Allows users to use old versions of bots in tournaments if they'd like.

Team Responsibilities:

1. Website: Will allow users to register through a web page for the tournament which results in a new tuple in this table.
2. Game Evaluation: When a contest task is dequeued from pending_contests, grab a list of all user/bots in this table for the contest. Then grab all the bot code from the bot_versions table.

[WEBSITE TEAM]

Table: user_bots

bot_id: integer → Primary Key
uid: integer → Foreign Key (users.uid)
challenge_id: integer → Foreign Key (challenges.challenge_id)
default_version: integer → Foreign Key (bot_versions.version_id)
name: varchar(80)
creation_date: timestamp
description: varchar(255)
public: bool

Description: This table holds basic metadata about the bots that users have uploaded. The default_version field can be used to lookup a bot in the bot_versions table in the event that only a bot_id is provided and no explicit version has been specified. In the initial version of the system, the website team will ensure that the default version is equal to the latest version available. This table should only be needed by the website team. However the website team should provide a http route that the display module team can use to request a list of name/bot_id/version mappings for the currently logged in user (you will need this for your challenge page anyway) If you want you can just always return that there is only the default_version available for now, but the test arena team should have a version number so they can use it tell the game evaluation team which version to use to play the game without having to have logic to read from this table to get the default.

[WEBSITE TEAM] [GRAPHICS TEAM] [GAME EVALUATION TEAM]

Table: user_bots_versions

bot_id: integer → Foreign Key (user_bots.bot_id), Primary Key
version: integer → Primary Key
creation_date: timestamp
source_code: mediumtext
comments: varchar (255)

language_id: int(11) → Foreign Key (langauges.language_id)
errors: integer
error_messages: text
warnings: integer
warning_messages: text

Description:

Dr. Blum wants us to keep track of different versions of bots, perhaps with a limitation to keep the database from exploding (his words). Any such limitation should be able to be changed by an

Administrator, but would, for example, give the user a warning which will require them to remove a previous version, or upon confirmation remove the oldest version (as an example). This would allow the user to download or view the old code before submitting the new code.

bot_id ties to the user_bots table. Version is used with bot_id to create a primary key.

Version will have to be be handled with queries (as per current).

[WEBSITE TEAM] [GAME EVALUATION TEAM]
Table: contest_map
contest_id: integer → Primary Key, Foreign Key
round_id: integer → Primary Key
match_id: integer → Primary Key, Foreign Key

Description:
This table is used to map contest rounds to matches that were played. After a match has been played during a contest, it must be added to the match_turns table, and an entry to it with the contest's and round's id must be added to this table.

[WEBSITE TEAM]
Table: challenge_map
challenge_id: integer → Primary Key, Foreign Key
uid: integer → Primary Key, Foreign Key
match_id: integer → Primary Key, Foreign Key
date: datetime

Description:
This table is used to map users who requested to play a challenge to the matches that were calculated. Since the website team needs to redirect the user immediately to a playback page before the game evaluation team has actually finished computing the match, the match must be created before writing the job to the **pending_challenge_matches** table. Thus the website team will first insert a black entry in the **matches** table to reserve a match_id, then insert a tuple into this table so that the website can later generate a list of old challenges for a user. Finally, the website team will insert a tuple into the **pending_challenge_matches** table and redirect the user to the playback page where the user will eventually be able to see the results calculated by the game evaluation team.

[WEBSITE TEAM] [GRAPHICS TEAM] [GAME EVALUATION TEAM]
Table: matches
match_id: integer → Primary Key, Auto Increment

winner: varchar(80)
game_initialization_message: JSON → DEFAULT: NULL
turns: JSON → DEFAULT: NULL
ready_for_playback: boolean → DEFAULT: FALSE

Description: All challenge matches and tournament matches will be placed in this table. A key point is that for challenges the website team must insert a blank tuple into this table as a placeholder. But for tournaments, the game evaluation team will insert the tuples. I lay this out in detail below.

Field Descriptions:

- match_id: Must be unique across all matches whether they originate from a tournament, or a challenge match. This allows the display module team to display a match the same way, whether its in a tournament or just a challenge match.
- game_initialization_message: See Dr. Blum's test arena design
- turns: See Dr. Blum's test arena design
- ready_for_playback: This is a synchronization point between the game evaluation and game display module teams.
  - False value means that the game evaluation team is still computing data for this match and that the display module team should not try to display it
  - True value means that the game evaluation team has finished with this match and has no intention of modifying this tuple ever again. Therefore, the game display team can read the data and animate the match on the playback page.
- Winner: Some nice human readable text that the website team can use to display a summary of the match. For challenge matches this will likely be something like “player 1”, in the contests you could use the username and bot name to be more descriptive.

Team Responsibilities:

- **Website Team:**
  - **Challenges:**
    - User requests to run a match for a particular challenge_id and set of bot_ids.
    - Website team inserts a blank record into **matches** table to create a placeholder for the match.
    - The auto generated match id assigned to this new record is placed in the **pending_challenge_matches** table (along with challenge_id and bot_ids)
    - The match_id used to redirect the user to /playback?match_id=[id].
  - **Tournaments:** Website team does not create any matches for tournament, the game evaluation team is responsible for that. All you do is place a tournament job in the **pending_contests** table when the tournament has reached its scheduled start time.
- **Game Display Team: (Read Only)**
  - **Playback Page:** You're given a match_id in the url, just read from the matches table to get the JSON objects you need and send them to the client.
  - **Test Arena:** This table is not relevant, you will use the **test_arena_matches** table for the analogous purpose.

- **Game Evaluation Team**
  - **Tournaments: (Inserts)** In the case of tournaments, it is your responsibility to construct the matches. So when you have assigned user bots to play in a tournament match, and have computed the game initialization and complete turns array for that match, you simply insert that complete tuple into the matches table.
    - In addition you MUST add a tuple to the **contest_map** table which ties that match_id to the contest_id and a round_id.
  - **Challenges: (Updates)** Each tuple in the pending_challenges table has an associated match_id. That match_id indicates the tuple in the matches table that must be updated with the JSON game initialization and turns objects that are generated by the bot evaluation program for the specified challenge. Once you have finished with the challenge match, you MUST set the ready_for_playback flag to true.

[WEBSITE TEAM] [GAME EVALUATION TEAM]
Table: pending_challenge_matches
challenge_match_id: → Primary Key, Auto Increment
challenge_id: integer → Foreign Key
match_id: integer → Foreign Key
bots: JSON

Field Descriptions:

- pending_challenge_id: The game evaluation team will always process the tuple with MIN(pending_challenge_id) next.
- match_id: References the tuple in the matches table where the game evaluation team will write the JSON data for the game display module team to later read from.
- bots: A JSON array of bot_id's. All elements of the array must be valid bot_id's in the user_bots table. And the length of the array is equal to challenges.player_count.

Description:
Since no one but the game evaluation team ever reads from this table, its really up to them when they want to delete tuples from it, but at the very least a processed tuple should be deleted before reading the next one.

[WEBSITE TEAM] [GAME EVALUATION TEAM]

Table: pending_contests

pending_contest_id: → Primary Key, Auto Increment
contest_id: integer

Description:
Game evaluation team will always read the tuple with MIN(pending_contest_id) next.

Since no one but the game evaluation team ever reads from this table, its really up to them when they want to delete tuples from it, but at the very least a processed tuple should be deleted before reading the next one.

This table may look a bit slim to the Game Evaluation team. But they will get list of users/bots from **contests_registration table.** Will get number of players in a game from the **challenges** table**,** number of matches per player for the initial round, and the number of players to be seeded into the single elimination bracket from the **contests** table**.** All of which is accessible using just a contest_id.

[WEBSITE TEAM] [GAME EVALUATION TEAM]
Table: test_arena_template_bots
challenge_id: integer → Primary Key, Foreign Key (challenges.challenge_id)
language_id: integer → Primary Key, Foreign Key (languages.language_id)
source_code: mediumtext

Description: This table holds the template code that should be pre-populated into the on-screen IDE for a given challenge and language. I tagged website team because there would need to be a field on the create challenge webform that would allow the game designer to upload a template bot for each supported language. Honestly, this is more a nicety than a required high priority feature. For now it would be fine if these were just blank.

[GRAPHICS TEAM] [GAME EVALUATION TEAM]
Table: test_arena_bots
uid: integer → Primary Key, Foreign Key (users.uid)
challenge_id: integer → Primary Key, Foreign Key (challenges.challenge_id)
language_id: integer → Primary Key, Foreign Key (languages.name)
source_code: mediumtext
errors: integer
error_messages: text
warnings: integer
warning_messages: text
needs_compiled → DEFAULT: TRUE

Description: The needs_compiled field is a synchronization point between the display module and game evaluation teams. When the user changes the code and requests a turn to be played, set the needs_compiled flag to rue after updating the source_code field. You MUST NOT change the code if

there is already a tuple in **pending_test_arena_turns** for the specified uid and challenge_id. Doing so may result in a situation where the game evaluation team would use the new bot for an old pending turn.

[GRAPHICS TEAM] [GAME EVALUATION TEAM]

Table: pending_test_arena_turns

pending_turn_id	→ Auto Increment, Unique
uid: integer	→ Primary Key, Foreign Key (users.uid)
challenge_id: integer	→ Primary Key, Foreign Key (challenges.challenge_id)
bot_type: varchar(10)	
language_id: int(11)	→ Foreign Key (languages.language_id), NULLABLE
bot_id: integer	→ Foreign Key (bot_versions.bot_id), NULLABLE
bot_version: integer	→ Foreign Key (bot_versions.version), NULLABLE
player: integer	
last_turn_index: integer	

Field Descriptions:

- **pending_turn_id** is used by the game evaluation team to dequeue from this table in FIFO order. That is, they will always read the tuple with MIN(pending_turn_id) next.
- **uid, challenge_id:** The primary key for this table guarantees that there can only be one pending test arena turn per user per challenge at any given time. This is by design, if this is not guaranteed then some very challenging concurrency problems arise since the user could potentially click buttons on the client much faster than the game evaluation team would be dequeuing tasks, completing them, and having them read by the display team and sent back to the client. I tried to come up with a nice efficient design to support concurrent requests for turns by the user in the test arena but ultimately failed to come up with anything that would be safe without having to duplicate the bot code in every single tuple of this table, which just seems to be a terrible waste of resources and would actually likely result in even worse performance by the game evaluation team which would have to read that source code from the DB (potentially over the network), write it to a file, compile it, and run it in a new sandbox for every single turn of every single user in the test arena. If you guys come up with something better, I'm all ears. I just don't have the time to think about it further right now so settled for something that will work.
- **bot_type:** Is either “user” or “test_arena”.
  - If it is “user”, then then language will be null and both bot_id and bot_version will be non-null and will reference a valid tuple in the bot_versions table.
  - If it is “test_arena” then language will be non-null and both bot_id and bot_version will be null. In this case use <uid, challenge_id, language> to grab the code from the test_arena_bots table.
- **Player:** Whose turn is it? The client side test arena will need to know this so that they know which bot to use for the upcoming turn.
- **last_turn_index:** will be used as an index into the test_arena_matches.turns JSON array for the game evaluation team to retrieve the game state immediately prior to the turn that will be computed. Therefore it is the responsibility of the game display team to ensure that the last_turn references a valid turn object in that JSON array.

I think this should change. I don't think the client side should have to worry about whose turn it is at all. Instead, the should be something limited to between the Game evaluation team and the bot evaluation programs. That is given a state, the bot evaluation program should be able to tell you who's turn it is. Then, knowing whos turn it is, the game evaluation team should be able to go to the database and get the correct bot. So instead what we need is a table mapping a player number to a bot. I'll think about this more and propose a schema very soon

[GRAPHICS TEAM] [GAME EVALUATION TEAM]

Table: test_arena_matches

uid: integer	→ Primary Key, Foreign Key (users.uid)
challenge_id: integer	→ Primary Key, Foreign Key (challenges.challenge_id)
game_initialization_message: JSON	→ DEFAULT: NULL
turns: JSON	→ DEFAULT: NULL
last_turn_status: varchar(10)	→ DEFAULT: “DISPLAYED”

Description: This table is very similar to the matches table, however the test arena use case is different enough that its better suited to a different table.

Field Descriptions:

- uid, challenge_id: Instead of a match_id we use the combination of these because for each user and each challenge, we will only track one match. In other words anything done in the test arena is overwritten as soon as the user plays a new game. Furthermore if a user backtracks to turn number 5, then selects a new bot and plays forward, then turns 6...10 will be overwritten but turns 1...5 will remain as they were.
- game_initialization_message: See Dr. Blum's test arena design
- turns: See Dr. Blum's test arena design.
- last_turn_status: **THIS IS IMPORTANT THAT EVERYONE AGREES ON** Will synchronize the Game display module and game evaluation teams with regards to test arena moves.
  - PENDING: The user requested a turn be executed, the display module team saw the previous state was DISPLAYED, therefore they changed the state to PENDING, then they added a tuple to the pending_test_arena_turns table.
  - READY: The Game evaluation team has grabbed the turn request from the pending_test_arena_turns table, has finished executing it, and has written the data to the correct position in the turns JSON array in this table. (The game display module team can now display it)
  - DISPLAYED: The Game Display Module team saw that the state was READY, so they read the turn data, changed the state to DISPLAYED, then sent the data to the client to be displayed.