An Adventure Game with a Database Back-End

Mårten Forsmark
c00mfk@cs.umu.se

Martin Berglund
c00mbe@cs.umu.se

Sandra Taha
taha@cs.umu.se

John Nilsson
c00jnn@cs.umu.se

Abstract
This article describes a prototype multi-user adventure game and the possibility of implementing such a game almost entirely using a database, leaving only minimal parsing and output formatting to a database client.

The prototype demonstrates that advanced database features are well suited for the creation of a multi-user adventure game and ensures stability and allows otherwise hard to write functionality.

1 Introduction
Those who have played text-based multi-user dungeon games (MUD’s) have most likely encountered some of the problems involved with this type of games. They have insufficient locking and concurrency control, and suffer from data corruption in connection with stability problems. Since a relational database does not suffer from these problems, and the MUD concept maps well into it, a text-based MUD with a relational database as backend might be a great improvement. Since the two map together so well, the game content and logic might be possible to implement in the database, just leaving a tiny parser as the database client. Some of the advantages in this system is that it is easy to maintain data in a safe way, everything is logged so retrieval of facts and data analysis is a simple task plus that a powerful query language should make game-logic easy to implement.

2 Approach
2.1 Database Design
The designed schema has four basic types of tables.

- **Event tables** Things that happen is placed in these tables. Examples are `sayEvents`, `moveEvents` and `itemActionEvents`. The names are hopefully self-explanatory. These hold all the information needed to recreate a sequence of actions that a player have made, for example when a player says something a new row is added to the `sayEvents` table. The new row contains the name of the player, the room, the time and the actual text said. Movement tables have a valid start-time and valid end-time to keep track of how long the item or player remained at the location moved to.
- **Game content tables** For example items, rooms and exits. Rooms and items have names and a some strings that are used for different messages (description and the suffix for when item lies in a room). Exits relate rooms to each other by naming a path between.

- **Game logic tables** For example takeables, readables and exits. Exits is featured again here because they can have a restriction query that prevents players from using the exit based on the return value of an arbitrary query. The form of these tables will be discussed in the next section.

- **Game object tables** For example users, characters and itemInstances. These contain the actual object visible in the game, the player characters and the instances of items. These also contain materialized views of the current location of items and characters to speed up the most common queries (updated by triggers on the movement event tables).

### 2.2 Requirement Queries

In the initial design it was attempted to represent game logic as rows in tables to a much greater extent than in the final implementation. The current solution is so that for an item to be possible to pick up (when the player uses the *get* command), the item must be in the takeables table and the requirements associated with the row has to succeed. The takeables table has four columns, the name of an item, the name of a requirement set that has to be fulfilled for the item to be takeable by the player and the strings printed upon success or failure of the *get* command. The requirement set is just a string name, in another table this name is used to look up a set of requirement queries to check. A typical requirement query could look like “SELECT okTakeItem($$nick$$, $$itemid$$) AS pass” and is just a regular string in SQL. The $$nick$$ and $$itemid$$ is filled in by the trigger to the users name and items identification number (from the event record being inserted) using simple string manipulation.

### 2.3 Asynchronous Notification

To keep the client-side application from knowing about other instances of the application a method to inform the application of new events arriving in the database was needed. Luckily PostgreSQL has a mechanism to do just that in the asynchronous notification system. This system allows an application to subscribe to a channel (simply a 60-character string) and it will then get a notice whenever someone does a notification on the same channel name. This allows the database to implement push-semantics, waking up clients when new data to be processed is available.

This system is used to wake up clients and make them fetch the new data from the event-tables whenever something happens. To aid performance the channels used in the asynchronous notification is named by room names. The clients then subscribe to the room they are currently in and the triggers only notify on the rooms for which events matter (in cases like say-events only the room in which it happens, for move-events in both the room left and the room entered). The subscription is actually completely transparent to the client application since it is done when a
move-event is inserted (the previous room subscription is cancelled and the new room is added to the subscription).

Clients also subscribe to a channel named the same as the character they represent, to allow direct notifications.

2.4 The Client Application

A small Python application implements the parser and keeps the database-connection. It is only about 300 lines long (mostly text formatting code). The interesting thing to know about this client application is how little it knows about the state of the MUD. It knows exactly two things, the name of the character it is playing and the time of the last event it saw. The name of the client is all the database needs to know to deduce everything else about the character, like current location and inventory. The time of the last event is needed to tell which events in the event tables that have already been shown to the user.

2.5 Content and Support Code

In addition to this base implementation a lot of content and code related to the content have been implemented. Several quests and items that shows the possibilities of adventure games with a database backend. A dynamically generated homepage with statistics and an administration client that allows replaying of all events in a room at a given time has also been implemented.

3 Results

3.1 Performance

Performance was initially thought to be one of the greater obstacles for the project but it appears that performance is adequate. Delays are low and no great scalability problems have been found in the basic implementation. With materialized views of the current location data it seems likely that the common queries (walking around and showing room descriptions) will mostly use data in disk cache. More complex queries relating to item and exit restrictions will no doubt be costly but would instead be more or less impossible in the classic MUD environment.

3.2 Examples

Examples of features that have easily been implemented in this model that would be extremely hard in a classic MUD codebase are shown here.

The “Orb of Tracking” is an item that when read views all the movement in the room over the last fifteen minutes. This does not require that the orb has been in the room when the movement occurred. To implement this, keeping a log of the movement is needed for all rooms. In the database model the orb could check back 24 hours just as well without any trouble. Obviously there is no way to implement this object other than by keeping a full log. See the test run below (playing as “mbe”):
Sky-walk to TFE

From the sky-walk you have a good view of the meadows outside. You sense pure evil to the east. It is really hot in here.

Obvious exits: east west

Stareater enters the room coming from Third floor
Stareater says: Hi!
Stareater leaves the room by going east

You are holding:
  Keycard
  Orb of Tracking
  Spyorb

You read orb of tracking

You read orb of tracking
Stareater went east 0 minutes and 4 seconds ago.
Stareater came from Third floor 0 minutes and 14 seconds ago.

Another example of such an object that needs a full log to work well is the “Spyorb”, which can be set to a room, whenever it is read after that it will list all the things that have been said in that room since the orb was activated. This query obviously also needs a full log. It should be noted that the push actually does nothing other than getting logged and printing the message. When the orb is read a query checks where it was when it was last pushed. A very nice and simple solutions thanks to the database.

Fourth floor

Stareater is standing here
mb is standing here

[User input] push spyorb
NOTICE: The orb whirs and clicks as the complex mechanism of wheels and springs inside tune themselves to track this room
[User input] go MA426
NOTICE: The door says "bleep" as you unlock it with your keycard.

MA426

[User input (some time has passed)] read spyorb

You read spyorb
Orb tuned to room Fourth floor:
At 15:36:33 Stareater said: "Now I shall gossip about mbe!".
At 15:37:11 Stareater said: "I know he uses the database to cheat!".
At 15:37:30 Stareater said: "He might spy on us even now..."

The readables and booleanRequirementQueries tables have row-embedded queries that handles text generation and enforce restrictions respectively.

In addition to this a number of quests are implemented. A quest is just a named series of entries in the quests table, it is filled in by triggers on exits and items.
Some quests exploit the database in great ways, allowing extremely simple implementation of quests that would need specialized data-structures and similar in a typical MUD. Unfortunately they require much too much explanation to reproduce in this report. An example of a simpler quest is the cabbage-goat-wolf problem where three items should be transported across a stream with restrictions on which can be left alone on the banks. It is easily implemented as item manipulation and exit restrictions.

4 Discussion

A number of problems have turned up during the work on this solution. The worst ones are mentioned here.

While the asynchronous notification system in PostgreSQL has been very helpful it is unfortunately implemented in a notoriously inefficient manner. It is implemented as a table with three columns, name of channel, pid of backend and number of notices pending. This table is not indexed! This means for each notification two sequential scans over this table is performed and the pending-counter is updated both times. This table is luckily fairly small but this might be the worst scalability problem.

Implementing the system this way is not meaningful since there is no reason to keep this data persistent. If the system would crash the clients would get disconnected anyway.

Another problem is the query strings, doing embedded queries this way is ugly and inefficient. Having anonymous functions as a primitive type in the database would have been a great feature, allowing the functions to be inserted directly. As an additional advantage this would lessen the clutter of the global name space that is inherent in this solution. This would also allow proper type-checking. The strings executed must always return the same type (exactly, no coercions provided because the type is used by the query planner) but it is left to the user to make sure it is so.

The database schema should be redesigned to some extent. A lot more things that are currently done by embedded queries could be implemented by tables instead, making the design more structured.

5 Conclusion

Overall this project succeeded in its goals. It was not assumed all along that everything that was planned were going to be possible to implement since the design was fairly radical. As hoped the client is completely unaware of most of what is going on in the game and most features are nicely mapped into the database.

Stability is naturally great since even if bugs are found they only get the client that found it disconnected (separate database-processes).

Some advanced quests are extremely easy to construct compared to a typical MUD. Simpler quests are a bit cumbersome however but not really hard.

All the basic features of an adventure game has been implemented, a lot of them not mentioned in this report due to lack of space.