SteamChecklist

Database Design Proposal

Jason Wong 2014

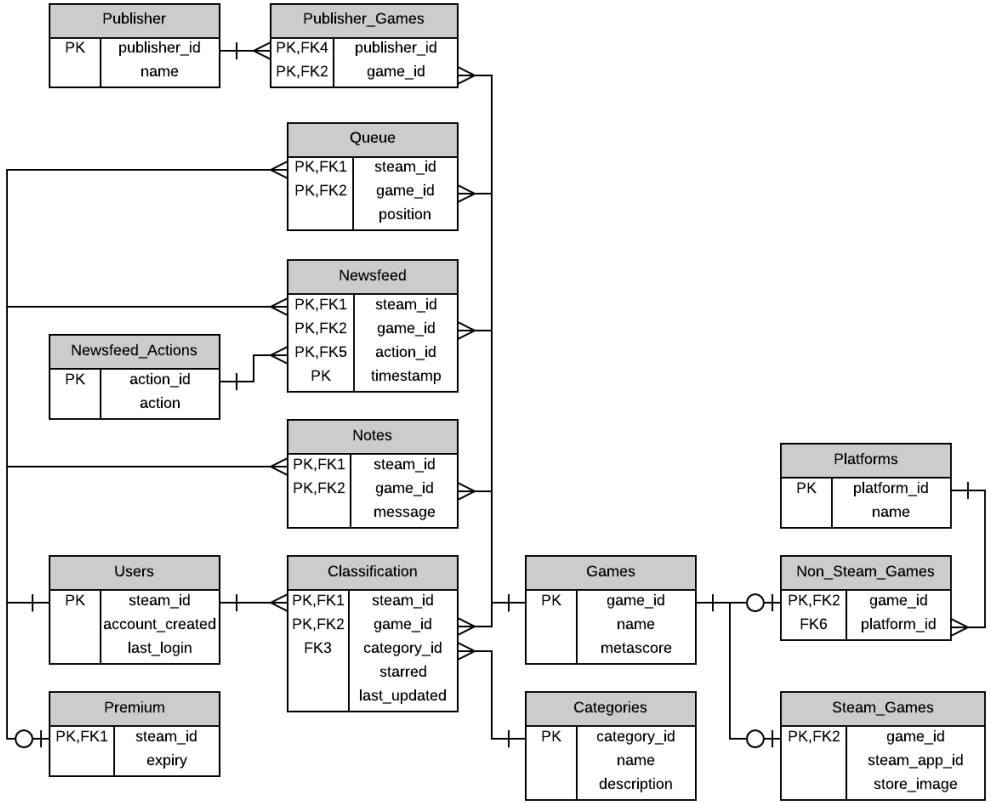
Table of Contents

Executive Summary

SteamChecklist is a website that allows gamers to create a checklist of their Steam games. When the site was first built, its owner was unaware of database design rules and best practices. Consequently, the current database design violates many of them, the foremost of which is storing user data using JSON. Not only does this prevent the database from being in first normal form, this also mandates excessively complicated application-side logic for updating the data. The database needs to be redesigned and expanded to allow for additional website features.

The following slides will outline a well-designed database schema to replace the current one. An E/R diagram will be presented as an overview of the database, followed by a detailed description of its tables. Subsequently, examples of various views, reports, stored procedures, triggers, and user permissions will be presented to show how the database would be used. Finally, some suggestions for implementing the database, known problems, and future enhancements will be presented.

Entity-Relationship Diagram



Tables: Users

The Users table stores all the users of the website, identifying them by their Steam ID.

Functional dependency: steam_id → account_created, last_login

```
drop table if exists users;
create table if not exists users(
  steam_id int not null,
  account_created date not null default CURRENT_DATE,
  last_login date not null default CURRENT_DATE,
  primary key(steam_id)
);
```

	steam_id numeric	account_created date	last_login date
1	76561198087255940	2013-02-11	2014-01-11
2	76561198090369121	2014-01-31	2014-01-21
3	76561198034669301	2014-01-22	2014-02-02
4	76561197976448645	2012-03-04	2014-03-03
5	76561197982989396	2013-09-01	2014-04-19
6	76561197991871927	2013-01-15	2014-04-03
7	76561198024992258	2014-03-22	2014-02-25
8	76561197976906767	2013-12-19	2014-04-17
9	76561197971248393	2013-12-12	2014-01-20
10	76561197970394103	2012-06-15	2014-02-07

Tables: Premium

The Premium table stores the users of the website that have purchased a premium membership.

Functional dependency: steam_id → expiry

```
drop table if exists premium;
create table if not exists premium(
  steam_id numeric not null references users(steam_id),
  expiry date not null default CURRENT_DATE + interval '1 month',
  primary key(steam_id)
);
```

	steam_id numeric	expiry date
1	76561197971248393	2014-05-08
2	76561197991871927	2013-02-16
3	76561197976448645	2013-11-24

Tables: Games

The Games table stores all the games present in the database.

Functional dependency: game_id → name, metascore

```
drop table if exists games;
create table if not exists games(
  game_id serial,
  name text not null,
  metascore int,
  primary key(game_id),
  check (metascore > -1)
);
```

	game_id integer	name text	metascore integer
1	1	FTL	84
2	2	Assassin's Creed	79
3	3	Skyrim	94
4	4	Borderlands 2	89
5	5	Titanfall	86
6	6	The Last of Us	95

Tables: Steam_Games

The Steam_Games table stores additional information specific to games that are on Steam.

Functional dependency: game_id → steam_app_id, store_image

```
drop table if exists steam_games;
create table if not exists steam_games(
  game_id int not null references games(game_id),
  steam_app_id numeric not null,
  store_image text not null,
  primary key(game_id)
);
```

	game_id integer	steam_app_id numeric	store_image text
1	1	212680	http://media.steampowered.com/ste
2	2	15100	http://media.steampowered.com/ste
3	3	72850	http://media.steampowered.com/ste
4	4	49520	http://media.steampowered.com/ste

Tables: Platforms

The Platforms table stores the platforms of the non-Steam games present in the database.

Functional dependency: platform_id → name

```
drop table if exists platforms;
create table if not exists platforms(
  platform_id serial,
  name text not null,
  primary key(platform_id),
  unique(name)
);
```

	platform_id integer	name text
1	1	Origin
2	2	PS3

Tables: Non_Steam_Games

The Non_Steam_Games table stores additional information specific to games that are not on Steam.

Functional dependency: game_id → platform_id

```
drop table if exists non_steam_games;
create table if not exists non_steam_games(
  game_id int not null references games(game_id),
  platform_id int not null references platforms(platform_id),
  primary key(game_id)
);
```

	game_id integer	platform_id integer
1	5	1
2	6	2

Tables: Categories

The Categories table stores the categories a game can fall under.

Functional dependency: category_id → name, description

```
drop table if exists categories;
create table if not exists categories(
  category_id serial,
  name text not null,
  description text not null,
  primary key(category_id),
  unique name
);
```

	category_id integer	name text	description text
1	1	Not started	Games you own but have not started playing yet.
2	2	Started	Games you have started playing.
3	3	Finished	Games you have finished playing.

Tables: Classification

The Classification table associates users with their games as well as with the category that their games fall under.

Functional dependency: steam_id, game_id, category_id → starred, last_updated

```
drop table if exists classification;
create table if not exists classification(
  steam_id numeric not null references users(steam_id),
  game_id int not null references games(game_id),
  category_id int not null references categories(category_id),
  starred boolean not null default 'false',
  last_updated date not null default CURRENT_DATE,
  primary key(steam_id,game_id)
);
```

	steam_id numeric	game_id integer	category_id integer	starred boolean	last_updated date
1	76561198087255940	1	3	t	2013-10-23
2	76561198087255940	3	2	t	2013-08-14
3	76561198087255940	4	1	f	2013-02-11
4	76561197970394103	1	3	f	2014-02-07
5	76561197982989396	1	2	t	2014-04-12
6	76561197971248393	6	3	f	2014-01-01

Tables: Notes

The Notes table stores any notes the user has created for their games.

Functional dependency: steam_id, game_id → message

```
drop table if exists notes;
create table if not exists notes(
  steam_id numeric not null references users(steam_id),
  game_id int not null references games(game_id),
  message text not null,
  primary key(steam_id,game_id)
);
```

	steam_id numeric	game_id integer	message text
1	76561198087255940	1	Best game ever!
2	76561197982989396	1	I'll come back to this later

Tables: Newsfeed_Actions

The Newsfeed_Actions table stores the possible actions a user can perform on their checklist.

Functional dependency: action_id → action

```
drop table if exists newsfeed_actions;
create table if not exists newsfeed_actions(
  action_id int not null,
  action text not null,
  primary key(action_id)
);
```

	action_id integer	action text
1	1	added a new game
2	2	completed a game
3	3	starred a game
4	4	added a game note

Tables: Newsfeed

The Newsfeed table stores a history of updates users have made to their checklist.

Functional dependency: steam_id, game_id, action_id, timestamp →

```
drop table if exists newsfeed;
create table if not exists newsfeed(
  steam_id numeric not null references users(steam_id),
  game_id int not null references games(game_id),
  action_id int not null references newsfeed_actions(action_id),
  time_stamp date not null default CURRENT_DATE,
  primary key(steam_id,game_id,action_id)
);
```

	steam_id numeric	game_id integer	action_id integer	timestamp date
1	76561198087255940	1	2	2013-10-17
2	76561198087255940	1	3	2013-10-23
3	76561198087255940	1	1	2013-02-11
4	76561198087255940	1	4	2013-10-23
5	76561197982989396	1	4	2014-04-12

Tables: Queue

The Queue table stores users' queues of games to play next.

Functional dependency: steam_id, game_id → position

```
drop table if exists queue;
create table if not exists queue(
  steam_id numeric not null references users(steam_id),
  game_id int not null references games(game_id),
  position int not null,
  primary key(steam_id,game_id),
  unique(steam_id,game_id,position),
  check (position > 0)
);
```

	steam_id numeric	game_id integer	position integer
1	76561198087255940	3	1
2	76561198087255940	4	2
3	76561197982989396	1	1

Tables: Publisher

The Publisher table stores the publishers of games in the database.

Functional dependency: publisher_id → name

```
drop table if exists publisher;
create table if not exists publisher(
  publisher_id serial,
  name text not null,
  primary key(publisher_id)
);
```

	publisher_id integer	name text
1	1	Subset Games
2	2	Ubisoft
3	3	Bethesda Softworks
4	4	2K Games
5	5	EA Games
6	6	Sony Computer Entertainment

Tables: Publisher_Games

The Publisher_Games table associates the publishers with the game(s) they have published.

Functional dependency: publisher_id, game_id →

```
drop table if exists publisher_games;
create table if not exists publisher_games(
  publisher_id int not null references publisher(publisher_id),
  game_id int not null references games(game_id),
  primary key(publisher_id,game_id)
);
```

	publisher_id integer	game_id integer
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6

Views: End-User

A view that only contains checklist-related information. Useful for a bare-bones checklist display.

```
CREATE VIEW EndUser AS
  SELECT
    classification.steam_id AS "User_ID",
    games.name AS "Game",
    categories.name AS "Status",
    categories.description AS "Status Description",
    classification.starred AS "Starred"
  FROM
    public.classification,
    public.games,
    public.categories
  WHERE
    classification.category_id = categories.category_id AND
    classification.game_id = games.game_id;
```

Views: End-User

Sample output

```
SELECT * FROM EndUser;
```

	User_ID numeric	Game text	Status text	Status Description text	Starred boolean
1	76561198087255940	FTL	Finished	Games you have finished playing.	t
2	76561198087255940	Skyrim	Started	Games you have started playing.	t
3	76561198087255940	Borderlands 2	Not started	Games you own but have not started playing yet	f
4	76561197970394103	FTL	Finished	Games you have finished playing.	f
5	76561197982989396	FTL	Started	Games you have started playing.	t
6	76561197971248393	The Last of Us	Finished	Games you have finished playing.	f

Views: Newsfeed

A view that only contains newsfeed information.

```
CREATE VIEW Display_Newsfeed AS
  SELECT newsfeed.timestamp as "At", newsfeed.steam_id as "User",
  newsfeed_actions.action as "Did", games.name as "With_Game"
  FROM newsfeed
    LEFT OUTER JOIN newsfeed_actions on (newsfeed.action_id = newsfeed_actions.action_id)
    LEFT OUTER JOIN games on (newsfeed.game_id = games.game_id)
  ORDER BY newsfeed.timestamp ASC;
```

Views: Newsfeed

Sample output

```
SELECT * FROM Display_Newsfeed;
```

	At date	User numeric	Did text	With_Game text
1	2013-02-11	76561198087255940	added a new game	FTL
2	2013-10-17	76561198087255940	completed a game	FTL
3	2013-10-23	76561198087255940	starred a game	FTL
4	2013-10-23	76561198087255940	added a game note	FTL
5	2014-04-12	76561197982989396	added a game note	FTL

Reports: Game Completion

Shows how many people own each game and how many of those people have finished them.

```
select games.name, count(c.game_id) as "Num_Owned", coalesce(q2."Finished", 0) as "Num_Finished"
from games, classification as c
left outer join(
  select c2.game_id, count(c2.game_id) as "Finished"
  from classification as c2
  where c2.category_id  in(
    select category_id
    from categories
    where name = 'Finished'
  )
  group by c2.game_id) as q2
on c.game_id = q2.game_id
where games.game_id = c.game_id
group by c.game_id, q2.game_id, q2."Finished",
games.name
order by c.game_id asc;
```

Reports: Game Completion

Sample output

	name text	Num_Owned bigint	Num_Finished bigint
1	FTL	3	2
2	Skyrim	1	0
3	Borderlands 2	1	0
4	The Last of Us	1	1

Reports: Percentage Completion

Shows the percentage of games a user owns that they have completed.

```
select c.steam_id, coalesce(round(q2."Finished" / count(c.game_id)::numeric * 100), 0) || '%' as
"Percent_Completed"
from classification as c
left outer join(
  select c2.steam_id, count(c2.steam_id) as "Finished"
  from classification as c2
  where c2.category_id in(
    select category_id
    from categories
    where name = 'Finished'
  )
  group by c2.steam_id) as q2
on c.steam_id = q2.steam_id
group by c.steam_id, q2."Finished"
order by steam_id asc;
```

Reports: Percentage Completion

Sample output

	steam_id numeric	Percent_Completed text
1	76561197970394103	100%
2	76561197971248393	100%
3	76561197982989396	0%
4	76561198087255940	33%

Stored Procedures: Clean Premium

Cleans out expired users from the Premium table.

```
create or replace function CleanPremium() returns boolean as
$$
begin
  delete from premium where expiry < CURRENT_DATE;
  return 'true';
end;
$$
language 'plpgsql';
```

Stored Procedures: Clean Premium

Sample output

```
SELECT CleanPremium();
```

	cleanpremium boolean
1	t

```
SELECT * FROM Premium; -- recall that there were previously two expired entries in the Premium table
```

	steam_id numeric	expiry date
1	76561197971248393	2014-05-08

Stored Procedures: Delete User

Deletes an user.

```
create or replace function DeleteUser(numeric) returns boolean as
$$
declare
  user_to_delete numeric := $1;
begin
  delete from classification where steam_id = user_to_delete;
  delete from queue where steam_id = user_to_delete;
  delete from newsfeed where steam_id = user_to_delete;
  delete from notes where steam_id = user_to_delete;
  delete from premium where steam_id = user_to_delete;
  delete from users where steam_id = user_to_delete;
  return 'true';
  end;
$$
language 'plpgsql';
```

Stored Procedures: Clean Premium

Sample output

```
SELECT DeleteUser(76561197970394103);
```

	deleteuser boolean
1	t

```
SELECT steam_id FROM users; -- recall that there were previously ten entries in the Users table
```

	steam_id numeric
1	76561197976906767
2	76561198087255940
3	76561197971248393
4	76561198034669301
5	76561198024992258
6	76561197982989396
7	76561197991871927
8	76561198090369121
9	76561197976448645

Triggers: Update last_updated

Updates the last updated date field in the Classification table when an update is made.

```
create or replace function update_last_updated()
returns trigger as
$$
begin
  new.last_updated = CURRENT_DATE;
  return new;
end;
$$
language 'plpgsql';
```

```
create trigger update_timestamp before update
on classification for each row execute procedure
update_last_updated();
```

Triggers: Update last_updated

Sample output

Recall that last_updated for the selected row was previously '2014-01-01'

```
update classification set starred='true' where steam_id=76561197971248393 and game_id=6;
select * from classification where steam_id=76561197971248393;
```

	steam_id numeric	game_id integer	category_id integer	starred boolean	last_updated date
1	76561197971248393	6	3	t	2014-04-27

Security

There are three access levels that need to be set. The database administrator has all privileges. Logged in users interact with a PHP script that has read and write access to all tables. Users that are not logged in interact with a PHP script that has read access to checklist-related tables only. (The website allows users to share their checklist with others.)

```
create role database_admin;
grant all privileges
on all tables in schema public
to database_admin;
```

```
create role logged_in;
grant select, insert, update
on all tables in schema public
to logged_in;
```

```
create role logged_out;
grant select
on classification, games, platforms,
non_steam_games, steam_games, categories
to logged_out;
```

Implementation Notes

Implementers should take note of the following points:

- The implementation software should have an entry in the Categories table with name = "Finished. The reports will not work otherwise.
- The implementation software is expected to use Steam's OpenID as an authentication mechanism, which returns the end user's Steam ID. This ensures that the end user actually owns the Steam ID associated with their Users entry, and explains why the steam_id field is a numeric field rather than a serial (auto-incrementing) field.

Known Problems

The database design has the following known problems:

- The Steam_Games table technically violates BCNF, since steam_app_id is the unique ID Steam assigns to each game.
- Performance may be degraded because steam_id is numeric (much longer than it has to be). DBAs may recommend creating an artificial serial key in the Users table to replace steam_id as a primary and foreign key, although this will violate BCNF.

Future Enhancements

Ways the database design could be enhanced:

- Allowing multiple notes to be set per game
- Views to allow publishers to see the completion rates of their games
- Triggers for automatically managing the position field in the Queue table (needs to account for both insertion and deletion)
- Using a trigger to enforce uniqueness of additions to the Games table (needs to account for titles being on multiple platforms)
- Newsfeed tables could be enhanced to allow for more detail in the generated newsfeed
- A friends table could be added to allow for Facebook-style aggregated friends' newsfeeds