

**Trabalho Prático**

**(Fase 1)**

|  |  |  |
| --- | --- | --- |
| Alunos: | Guilherme Cepeda | 47531 |
|  | Rafael Coelho | 47578 |
|  | Tiago Martinho | 48256 |

|  |  |
| --- | --- |
| Professor: | Nuno Leite |

Relatório final da 1ª Fase realizado no âmbito de Sistemas de Informação,  
do curso de licenciatura em Engenharia Informática e de Computadores  
Semestre de Verão 2022/2023

Maio de 2023

<< Esta página foi intencionalmente deixada em branco >>

# Resumo

A empresa **“GameOn”**, pretende desenvolver um sistema para a gestão de jogos, jogadores e as partidas que estes efetuam.

O sistema deve registar os jogadores e cada um destes pertence a uma determinada região.

Cada vez que um jogo é jogado é criada uma partida automaticamente com a data de começo e após terminar a data de fim da partida. Existem partidas para apenas um jogador e para múltiplos jogadores estando sempre associadas aos jogadores da região que as jogam.

Os jogadores podem ter recompensas (crachás) do jogo após cada partida caso cheguem a um determinado número de pontos.

Tanto os jogadores como os jogos têm tabelas de estatística associadas a estes, com informação do número de partidas, número de jogos diferentes que jogou e total de pontos no caso da tabela de jogador e o número de partidas, número de jogadores e total de pontos no caso do jogo.

Os jogadores podem também ter outros jogadores como amigos e ter conversas entre vários jogadores com várias mensagens em cada conversa.

# Abstract

The company **“GameOn”** intends to develop a system for managing games, players and the matches they play.

The system must register the players and each one of them belongs to a certain region.

Each time a game is played, a game is automatically created with the start date and the end date of the game.

There are games for just one player and for multiple players and these are always associated with the players in the region that play them.

Players can have in-game rewards (badges) after each match if they reach a certain number of points.

Both players and games have statistics tables associated with them, with information on the number of matches, number of different games played and total points in the case of the player table and the number of matches, number of players and total points in the case of the game table.

Players can also have other players as friends and have multiplayer conversations with multiple messages in each conversation.

**Índice**

[Resumo iii](#_Toc134192543)

[Abstract iv](#_Toc134192544)

[1. Introdução 1](#_Toc134192545)

[2. Modelo Relacional 2](#_Toc134192546)

[3. Restrições de Integridade 5](#_Toc134192547)

[4. Explicação da solução apresentada para cada exercício 7](#_Toc134192548)

[a. Criar o modelo físico 7](#_Toc134192549)

[b. Remover o modelo físico 7](#_Toc134192550)

[c. Preenchimento inicial da base de dados 7](#_Toc134192551)

[d. Mecanismos para criar, desativar e banir o jogador 8](#_Toc134192552)

[e. Criar a função para obter o total de pontos por jogador 8](#_Toc134192553)

[f. Criar a função para obter o total de jogos por jogador 8](#_Toc134192554)

[g. Criar a função para obter o total de pontos num jogo por jogador 9](#_Toc134192555)

[h. Criar o procedimento armazenado para associar um crachá 9](#_Toc134192556)

[i. Criar o procedimento armazenado para iniciar uma conversa 10](#_Toc134192557)

[j. Criar o procedimento armazenado para juntar um jogador a uma conversa 10](#_Toc134192558)

[k. Criar o procedimento armazenado para enviar uma mensagem 11](#_Toc134192559)

[l. Criar a vista para aceder à informação total de um jogador 11](#_Toc134192560)

[m. Criar os mecanismos necessários para atribuir crachás de forma automática quando uma partida termina 12](#_Toc134192561)

[n. Criar os mecanismos necessários para banir os jogadores que constem na vista “jogadorTotalInfo” 12](#_Toc134192562)

[o. Testes das funcionalidades 12](#_Toc134192563)

[5. Conclusões 14](#_Toc134192564)

[Referências 15](#_Toc134192565)

# Introdução

Nesta primeira fase do trabalho foi pedido para implementar um sistema de informação para a gestão de jogos, jogadores e as partidas que estes efetuam para a empresa **“GameOn”**.

Para isto começamos por desenvolver um modelo de dados, conceptual e relacional, com base num texto facultado pelo cliente (**“GameOn”**), que possuía a informação necessária para compreender o que iria ser requerido durante o funcionamento deste SI, e que também possuía algumas notas sobre a sua eventual estrutura.

A partir dai procedemos ao desenvolvimento de um modelo físico, com código para o criar, apagar, inserção de dados, e algumas *queries* que vão ser necessárias para o funcionamento deste SI durante o seu tempo útil, criadas sobre a forma de *stored procedures*. Adicionalmente para o bom funcionamento desta base de dados são requeridos alguns *triggers* e funções que estão também implementadas na base dados, para ter a certeza que é dado a resposta a todos os pedidos dos clientes.

# Modelo Relacional

**REGION**

**REGION**(r\_name[PK]).

**PK** → r\_name → Permite a identificação no enquadramento da relação, sendo que r\_name é da região em questão

**PLAYER**

**PLAYER**( id[PK], email[AK], username[AK], activity\_state, region\_name[PK]).

**PK** → id → Atributo que permite fácil identificação no enquadramento da relação sendo que representa o jogador em questão.

**AK** → email, username → Não existem emails e usernames iguais.

**FK** → region\_name ref. REGION.r\_name → Cada jogador tem associado uma região.

**FRIENDSHIP**

**FRIENDSHIP**(player1\_id[PK, FK], player2\_id[PK, FK]).

**PK** → player1\_id, player2\_id → Ambos os atributos fazem a chave da relação representando uma amizade entre 2 jogadores.

**FK** → player1\_id ref. PLAYER.id , player2\_id ref. PLAYER.ID → Cada amizade tem associado os players que a constituem.

**CHAT**

**CHAT**(id[PK],c\_name).

**PK** → id → É atribuído a cada chat um id único.

**CHAT\_LOOKUP**

**CHAT\_LOOKUP**( chat\_id[PK,FK] , player\_id[PK,FK]).

**PK** → chat\_id, player\_id → Permite fácil identificação no enquadramento da relação sendo ambos fazem a chave primária da relação

**FK** → chat\_id ref, CHAT.id, player\_id ref, PLAYER\_id → Cada chat\_lookup tem um chat e um player associado.

**MESSAGE**

**MESSAGE**(n\_order[PK], chat\_id[PK, FK], player\_id[FK], m\_time, m\_text).

**PK** → n\_order, chat\_id→ É atribuido a cada message ambos os atributos.

**FK**  → chat\_id ref. CHAT.id, player\_id ref. PLAYER.id → Cada message precisa de ter o player que a enviou e o chat para qual enviou

**GAME**

**GAME** ( id [PK], g\_name[AK], url)

**PK** → id → Permite fácil identificação no enquadramento da relação sendo que é um identificador único para cada game.

**AK**  → g\_name → Não existem nomes de games iguais

**PURCHASE**

**PURCHASE**( player\_id[PK,FK], game\_id[PK,FK], p\_date, price).

**PK** → player\_id, game\_id → Ambos os atributos identificam uma purchase.

**FK** → player\_id ref. PLAYER.id, game\_id rerf. GAME.id→ Cada purchase tem um player id e um game id.

**MATCH**

**MATCH**(number[PK], game\_id[PK,FK], dt\_start, dt\_end).

**PK** → number, game\_id→ Ambos os atributos identificam uma match.

**FK** → game\_id ref. GAME.id→ Cada match tem um game.

**MATCH\_NORMAL**

**MATCH\_NORMAL**(match\_number[PK,FK],game\_id[PK,FK], difficulty\_level).

**PK** → match\_number, game\_id→ Ambos os atributos identificam uma match\_normal.

**FK** → match\_number ref. MATCH.number, game\_id ref MATCH.game\_id→ Cada match\_normal precisa do number da match e do game id.

**MATCH\_MULTIPLAYER**

**MATCH\_MULTIPLAYER**(match\_number[PK,FK],game\_id[PK,FK], state).

**PK** → match\_number, game\_id → Ambos os atributos identificam uma match\_multiplayer.

**FK** → match\_number ref. MATCH.number, game\_id ref MATCH.game\_id→ Cada match\_multiplayer precisa do number da match e do game id.

**PLAYER\_SCORE**

**PLAYER\_SCORE**(player\_id[PK,FK], match\_number[PK, FK], game\_id[PK,FK], score).

**PK** → player\_id, match\_number, game\_id→ Os 3 atributos identificam um player\_score.

**FK** → player\_id ref. PLAYER.id, match\_number ref. MATCH.number, game\_id ref MATCH.game\_id→ Cada player\_score precisa de saber qual o player, a match em que está e qual o jogo.

**BADGE**

**BADGE**(b\_name[PK], game\_id[PK,FK], points\_limit, url).

**PK** → b\_name, game\_id→ Ambos os atributos juntos representam uma badge.

**FK** → game\_id ref. GAME.id → Cada badge tem associada o jogo a qual pertence.

**PLAYER\_BADGE**

**PLAYER\_BADGE**(player\_id[PK,FK],b\_name[PK], game\_id[PK,FK]).

**PK** → player\_id, b\_name, game\_id→ Os atributos juntos representam uma player\_badge.

**FK** → b\_name ref. BADGE.b\_name, game\_id ref. BADGE.game\_id, player\_id ref. PLAYER.id → Cada player\_badge tem associada a badge a qual pertence e o player que a tem.

**STATISTIC\_PLAYER**

**STATISTIC\_PLAYER**(player\_id[PK,FK],matches\_played, total\_points, games\_played).

**PK** → player\_id → Permite fácil identificação no enquadramento da relação sendo que é um identificador único para cada statistic\_player.

**FK** → player\_id ref. PLAYER.id → Cada statistic\_player tem associada o player de qual se irá fazer a estatistica.

**STATISTIC\_GAME**

**STATISTIC\_GAME**(game\_id[PK,FK],matches\_played, total\_points, games\_played).

**PK** → game\_id → Permite fácil identificação no enquadramento da relação sendo que é um identificador único para cada statistic\_game.

**FK** → game\_id ref. GAME.id → Cada statistic\_game tem associada o game de qual se irá fazer a estatistica.

# Restrições de Integridade

**id**→ player,chat →serial→ numero inteiro positivo crescente

**email**→ player → varchar→ segue o formato “%@%.%”

**activity\_state**→ player→ varchar→ valores possíveis “Active”, “Inactive” e “”Banned”

**r\_name**→ region →varchar→ Segue o formato [a-z] ou [A-Z] e 20 dígitos

**url**→ game, badge → varchar→ segue o formato “https://%”

**m\_time**→ message→ timestamp→ data com valores até aos segundos

**p\_date**→ purchase→ timestamp→ data com valores até aos segundos

**dt\_start**→ match→ timestamp→ data com valores até aos segundos

**dt\_end**→match→ timestamp→ data com valores até aos segundos

**difficulty\_level**→ match\_normal→ integer→ com valores entre 1 e 5

**state**→ match\_multiplayer→ varchar→ valores possíveis “To start”, “Waiting for Players”, “Ongoing” e “Finished”

**Regras de Negócio:**

* Um

Regras de Negócio são as informações complementares fornecidas pelo cliente(empresa “GameOn”) que não se conseguem representar pelo modelo EA e o modelo lógico mas são necessárias para clarificar os aspetos do domínio da aplicação

# Explicação da solução apresentada para cada exercício

# Criar o modelo físico

Nesta primeira fase foi criado o modelo físico do sistema contemplando todas as restrições de integridade passíveis de ser garantidas declarativamente, assim como a atomicidade nas operações.

Para a realização do script **createTable.sql**, começamos por criar as tabelas de acordo com as **FK**.

Os atributos de cada relação são enumerados a seguir à instrução **CREATE TABLE** e o seu tipo é definido a seguir. Para os atributos que têm de cumprir determinada condição, utilizamos a instrução **CHECK (condição)**. Se os tuplos só podem ter certos valores utilizamos a instrução **CHECK (atributo IN (valores))**.

Os atributos que são chave primária da relação são representados com **PRIMARY KEY** em frente ao tipo ou no fim da declaração dos atributos.

A representação das chaves candidatas é feita com a instrução **UNIQUE (atributo)**.

Para as chaves estrangeiras a sintaxe utilizada é **FOREIGN KEY (tuplo) REFERENCES <tabela> (tuplo)**.

Para podermos realizar restrições de alguns atributos que se relacionam entre si, utilizamos a instrução **CONSTRAINT nome CHECK (condição)**.

# Remover o modelo físico

Para a concretização do script **removeTable.sql** utilizámos a instrução **DROP TABLE nome**.

# Preenchimento inicial da base de dados

Para a realização do script **insertTables.sql**, que tem como função preencher as tabelas criadas anteriormente com valores que permitam testar as diferentes interrogações que irão ser feitas à base de dados, utilizamos a instrução **INSERT INTO <relação> (atributos)** seguido de **VALUES** e os valores com os quais desejamos preencher as tabelas. Foi necessário colocar valores iguais para as **FK** das diferentes relações.

# Mecanismos para criar, desativar e banir o jogador

Foi realizada a inserção de dados na tabela **PLAYER** com o comando insert dados o email, região, username e estado de atividade do jogador. E como no enunciado era pedido para desativar e banir um jogador, foi realizado um update para alterar o estado de atividade de jogador para **“Inactive”** no caso da operação de desativar e para **“Banned”** no caso da operação de banir.

# Criar a função para obter o total de pontos por jogador

Foi criada a função **totalPontosJogador** que recebe um identificador de utilizador e retorna a pontuação total de todas as partidas jogadas pelo utilizador com o identificador pretendido. No caso de o utilizador não existir, o valor retornado é nulo.

Para cumprir o objetivo, fazemos uso da tabela única onde guardamos as pontuações de cada partida, selecionamos as partidas do utilizador e fazemos uso da operação SUM para somar as pontuações das partidas encontradas.

# Criar a função para obter o total de jogos por jogador

Foi criada a função **totalJogosJogador** que recebe um parâmetro p\_id (que representa o id de um jogador) e retorna um valor inteiro que representa o número total de jogos diferentes em que o jogador com o id p\_id já jogou.

A função começa por verificar se o jogador com o id fornecido existe, chamando a função check\_player\_exists. Se o jogador não existir, a função retorna null.

Caso o jogador exista, a função faz uma consulta à tabela PLAYER\_SCORE para contar o número de jogos diferentes em que o jogador já jogou. A contagem é feita usando a função de agregação COUNT, que conta o número de valores distintos da coluna game\_id da tabela PLAYER\_SCORE onde o player\_id é igual ao valor passado como argumento. O resultado da contagem é armazenado na variável games\_count, que é retornada pela função no final.

# Criar a função para obter o total de pontos num jogo por jogador

Na alínea g foi criada a função **pontosJogoPorJogador** que devolve uma tabela com 2 colunas com o identificador de jogador e o seu total de pontos.

Esta função começa por verificar se o id do jogo recebido por parâmetro, é ou não null, caso não seja verificamos se o id é valido e se existe e caso exista fazemos um ciclo onde iremos buscar o player id e a soma dos seus pontos naquele jogo específico e apresentamos em uma tabela.

# Criar o procedimento armazenado para associar um crachá

Foi criado um procedimento **associarCrachá** cujo propósito é associar a um utilizador a um devido crachá, se este tiver pontos suficientes para tal. O procedimento recebe então o utilizador, o id do jogo e o nome do crachá.

Primeira consideração a ter neste processo é que se o utilizador já obteve este crachá, não se deve tentar reatribuir o mesmo. Para tal faz-se uso da operação PERFORM de plpgsql para efetuar uma query sem se obter o valor de retorno. A query realizada é feita à tabela PLAYER\_BADGE que contém as atribuições de crachás a jogadores, procurando pelos 3 argumentos recebidos. Se o utilizador já tiver sido atribuído ao crachá, então lança-se um aviso ao cliente e termina-se o processamento, caso contrário continuamos.

Realizamos o processo de seleção, onde, em vez de apenas limitarmos as partidas do utilizador, limitamos também às partidas realizadas no jogo pretendido, fazendo uso da operação SUM para obter a soma das pontuações. Uma vez conhecida a pontuação total do utilizador neste jogo, verifica-se se este tem de facto pontos suficientes. A pontuação necessária está associada ao crachá na tabela BADGE, no atributo "points\_limit". Se o utilizador não tiver pontos suficientes lança-se um aviso ao cliente. Se o utilizador tiver pontos suficientes, insere-se na tabela PLAYER\_BADGE o tuplo que sugere o mesmo.

# Criar o procedimento armazenado para iniciar uma conversa

Com o intuito de criar o procedimento **iniciarConversa**, foi criado um gatilho, uma função e o próprio procedimento armazenado.

A função set\_message\_id() é responsável por atribuir o valor n\_order a uma mensagem no momento da sua criação. Isto é necessário porque este valor tem de ser único e sequencial para cada conversa, servindo de identificador juntamente com o seu chat\_id. A função realiza uma subconsulta que retorna o maior número de n\_order já existente nessa conversa onde a nova mensagem será inserida, e adiciona 1 a esse valor para atribuí-lo ao n\_order da nova mensagem. Retorna a nova mensagem com o n\_order atualizado.

O gatilho set\_message\_id\_trigger é chamado antes da inserção de uma nova mensagem na tabela MESSAGE. Este chama a função set\_message\_id() para definir o valor do n\_order da nova mensagem a ser inserida. Se o valor do n\_order já foi definido, o gatilho não faz nada.

O procedimento **iniciarConversa** tem o objetivo de iniciar uma nova conversa. Este recebe o p\_id do jogador que inicia a conversa e o nome da nova conversa, retornando o id desta no final. A *procedure* verifica se o jogador existe e se o nome da conversa não está vazio. Em seguida, insere uma nova entrada na tabela CHAT e obtém o id da nova conversa retornado pelo comando *returning*. Insere também um registo na tabela CHAT\_LOOKUP para associar o jogador à conversa criada. Obtém o nome do jogador a partir do id deste e insere uma mensagem a indicar que o jogador iniciou a conversa na tabela MESSAGE. Esta inserção da mensagem é detetada pelo gatilho set\_message\_id\_trigger que atribuirá o valor correto para o campo n\_order da mensagem.

# Criar o procedimento armazenado para juntar um jogador a uma conversa

Na alínea j foi criado o procedimento **juntarConversa** que junta um jogador específico a uma conversa.

Para fazer este procedimento foi verificado se o id do jogador e o id da conversa não são nulos, caso não sejam verificamos se estes existem nas respetivas tabelas, após este passo fazemos uma última verificação onde vamos a tabela **CHAT\_LOOKUP**  verificar se se o player já se encontra na conversa indicada, caso não se encontre inserimos na tabela **CHAT\_LOOKUP**  o id do player com o respetivo chat id.

# Criar o procedimento armazenado para enviar uma mensagem

Foi criado o procedimento que permite o envio de uma mensagem para uma conversa, sendo esta representada por um tuplo na tabela MESSAGE. O procedimento recebe 3 parâmetros, o identificador do utilizador, o identificador da conversa e a mensagem que se pretende enviar.

Começa-se por verificar que o utilizador está associado à conversa, usando o mecanismo PERFORM para fazer uma *query* à tabela CHAT\_LOOKUP, procurando por um tuplo que contenha o id da conversa e do utilizador. Se este não for encontrado, avisa-se o servidor de que o utilizador não tem acesso à conversa e termina-se o processamento.

Caso o utilizador tenha permissões, insere-se na tabela MESSAGE o tuplo que representa a mensagem atual vinda do utilizador e para a conversa que se pretende, adicionando o timestamp atual para o "quando" da mensagem ter sido enviada. Este procedimento não se preocupa em atribuir o valor da ordem da mensagem uma vez que um gatilho foi realizado para tratar deste assunto.

# Criar a vista para aceder à informação total de um jogador

A vista **jogadorTotalInfo** é criada a partir de uma consulta que faz várias junções entre as tabelas PLAYER, PLAYER\_SCORE, MATCH e GAME. O objetivo desta vista é mostrar todas as informações de um jogador que não esteja banido, como o seu identificador, estado, email, username, número total de jogos em que participou, número total de partidas em que participou e número total de pontos que já obteve.

A cláusula LEFT JOIN é usada para garantir que todas as linhas da tabela PLAYER sejam incluídas na consulta, mesmo que não haja correspondência nas outras tabelas. A cláusula WHERE é usada para filtrar jogadores cujo estado de atividade não seja 'Banned'. A cláusula GROUP BY é usada para agrupar os resultados por id do jogador.

# Criar os mecanismos necessários para atribuir crachás de forma automática quando uma partida termina

Para a execução desta alínea foi criado um gatilho **AFTER UPDATE** que é ativado após um update na tabela **MATCH\_MULTIPLAYER** nomeadamente após update do campo state de **“Ongoing”** para **“Finished”,** verificando através de NEW.state o novo estado e OLD.state o estado anterior**.**

Após o update é chamada a função **trigger\_exM** que verifica se o novo estado é **“Finished”** e caso seja fazemos um ciclo onde verificamos se algum dos players que pertence a match que acabou tem pontos suficientes para lhe ser atribuído um crachá, através do procedimento **associarCrachá** da alínea h verificamos se algum crachá irá ser atribuído aos players da match em questão.

# Criar os mecanismos necessários para banir os jogadores que constem na vista “jogadorTotalInfo”

Pretende-se que a remoção de um tuplo de um utilizador sobre a vista “jogadorTotalInfo" se equivale a banir o utilizador.

Para tal, foi criada uma função que retorna um gatilho e posteriormente definido o gatilho que em vez da remoção de um (ou mais) tuplo(s) sobre a vista.

A função começa por verificar que a operação que ativou o *trigger* foi um delete e nesse caso faz uso do procedimento anteriormente definido na alínea d para banir o jogador com o identificador do utilizador do tuplo que se pretendia remover. Não é necessário retornar nada uma vez que banir o utilizador já resulta na remoção do tuplo da tabela.

Caso o gatilho corra numa outra qualquer operação, este avisa o cliente e não faz qualquer processamento.

# Testes das funcionalidades

# Conclusões

Com este trabalho foi construído um SGBD para a gestão de jogos, jogadores e as partidas que estes efetuam para a empresa **“GameOn”**, e este poderá ser adaptado para ser utilizado por outras empresas.

O modelo EA foi desenvolvido de modo a ser percetível de um ponto de vista geral. O modelo relacional foi utilizado para dividir e esclarecer todo o tipo de relações possíveis impondo restrições aos seus atributos apelando ao senso comum em alguns casos e a sua passagem foi realizada de forma sistemática à custa das regras de passagem pré-definidas.

Acreditamos ter utilizado e aplicado de forma correta a matéria falada nas aulas, nomeadamente utilização de *triggers*, criação de funções e *stored procedures* assim como o nivel de isolamento da base de dados

Por fim, acreditamos ter atingido os objetivos de aprendizagem na realização deste trabalho.

# Referências

|  |  |
| --- | --- |
| [1] | Wikipedia, “Big data --- Wikipedia, The Free Encyclopedia,” http://en.wikipedia.org/w/index.php?title=Big\_data&oldid=648786139, 2015. |
| [2] | X. Ding, X. Zhu e G. Wu, “Data mining with big data,” *IEEE Transactions on Knowledge and Data Engineering,* vol. 26, n.º 1, pp. 97-107, 2014. |
| [3] | J. Andrews, S. Buzzi, W. Choi, S. Hanly, A. Lozano, A. Soong e J. Zhang, “What Will 5G Be?,” *IEEE Journal on Selected Areas in Communications,* vol. 32, n.º 6, pp. 1065-1082, 2014. |
| [4] | L. Boytsov, “Indexing Methods for Approximate Dictionary Searching: Comparative Analysis,” *J. Exp. Algorithmics,* vol. 16, n.º may, p. 1.81, 2011. |
| [5] | T. Jurkiewicz e K. Mehlhorn, “On a Model of Virtual Address Translation,” *J. Exp. Algorithmics,* vol. 19, n.º jan, pp. 1-18, 2015. |
| [6] | J. Neumann, The Computer and the Brain, New Haven, CT, USA: Yale University Press, 1958. |
| [7] | B. Kernighan e P. Plauger, The Elements of Programming Style, New York, NY, USA: McGraw-Hill, Inc., 1982. |