ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Trabalho Prático

(Fase 1)

Alunos:	Guilherme Cepeda	47531
	Rafael Coelho	47578
	Tiago Martinho	48256

Docentes:	Nuno Leite
	Walter Vieira

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 matches 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

Lista de Figuras

Lista de Tabelas

1. 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.

2. Formulação do Problema

Inicialmente, começámos por interpretar o enunciado e identificar quais as entidades, associações, atributos e restrições de integridade que constam no problema.

De seguida, foi criado o **Modelo Entidade-Associação** e, só depois, o seu respetivo **Modelo Relacional**.

2.1. Modelo Entidade-Associação

Na figura abaixo, apresentamos a nossa solução do problema no Modelo EA:

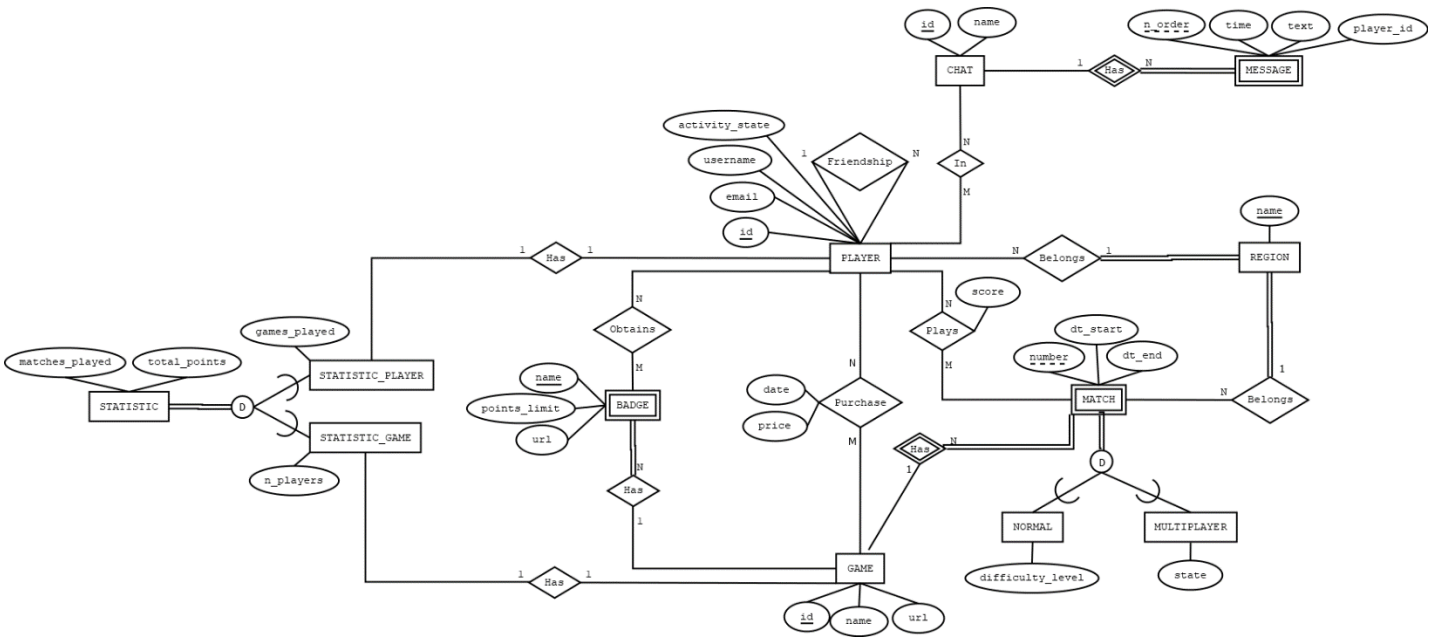

Figura 1 – Diagrama do Modelo Entidade-Associação

No modelo apresentado acima identificamos as seguintes entidades e os seus atributos:

Entidades	Atributos
REGION	name
PLAYER	id, email, username, activity_state
CHAT	id, name
MESSAGE	n_order, time, text, player_id
GAME	id, name, url
MATCH	number, dt_start, dt_end
NORMAL	difficulty_level
MULTIPLAYER	state
BADGE	name, points_limit, url
STATISTIC	matches_played, total_points
STATISTIC_PLAYER	games_played
STATISTIC_GAME	n_players

Tabela 1 – Entidades e os seus atributos

2.2. Modelo Relacional

Iremos proceder à passagem do Modelo EA para o Modelo Relacional seguindo as regras. Para a identificação das chaves sublinhamos apenas para identificar que se trata de uma Primary Key (**PK**) e identificamos explicitamente as Foreign Key (**FK**) e chaves candidatas (**AK**).

- **REGION** (name)

- **PLAYER** (id, email, username, activity_state, region_name)

  AK: { email, username }

  FK: { region_name } ref. REGION.name

- **FRIENDSHIP** (player1_id, player2_id)

  FK: { player1_id } ref. PLAYER.id

  { player2_id } ref. PLAYER.id

- **CHAT** (id, name)

❖ **CHAT_LOOKUP** (chat_id, player_id)

FK: { chat_id } ref. CHAT.id

{ player_id } ref. PLAYER.id

❖ **MESSAGE** (n_order, chat_id, time, text, player_id)

FK: { chat_id } ref. CHAT.id

{ player_id } ref. PLAYER.id

❖ **GAME** (id, name, url)

AK: { name }

❖ **PURCHASE** (player_id, game_id, date, price)

FK: { player_id } ref. PLAYER.id

{ game_id } ref. GAME.id

❖ **MATCH** (number, game_id, dt_start, dt_end)

FK: { game_id } ref. GAME.id

❖ **MATCH_NORMAL** (match_number, game_id, difficulty_level)

FK: { match_number } ref. MATCH.number

{ game_id } ref. MATCH.game_id

❖ **MATCH_MULTIPLAYER** (match_number, game_id, state)

FK: { match_number } ref. MATCH.number

{ game_id } ref. MATCH.game_id

❖ **PLAYER_SCORE** (player_id, match_number, game_id, score)

FK: { player_id } ref. PLAYER.id

{ match_number } ref. MATCH.number

{ game_id } ref MATCH.game_id

❖ **BADGE** (name, game_id, points_limit, url)

FK: { game_id } ref. GAME.id

❖ **PLAYER_BADGE** (player_id, b_name, game_id)

FK: **{** b_name } ref. BADGE.name

{ game_id } ref. BADGE.game_id

{ player_id } ref. PLAYER.id

❖ **STATISTIC_PLAYER** (player_id, matches_played, total_points, games_played)

FK: { player_id } ref. PLAYER.id

❖ **STATISTIC_GAME** (game_id, matches_played, total_points, n_players)

FK: { game_id } ref. GAME.id

2.3. Restrições de Integridade

email→ PLAYER→ varchar→ segue o formato "%@%.%"
activity_state→ PLAYER→ varchar→ valores possíveis "Active", "Inactive" e ""Banned"
name→ REGION→ varchar→ Segue o formato [a-z] ou [A-Z] e 20 dígitos
url→ GAME, BADGE → varchar→ segue o formato "https://%"
time→ MESSAGE→ timestamp→ data com valores até aos segundos
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"

2.4. Regras de Negócio:

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

- Sistema que atualiza a pontuação e estado durante a execução do jogo
- Cada partida está associada a uma região e apenas jogadores dessa região a podem jogar

3. Solução Proposta

Aqui foi criado o código PL/pgSQL que permitiu resolver todos os exercícios do enunciado.

Tivemos um cuidado especial no **controlo de exceções**. Para isso criámos funções auxiliares que verificam uma determinada condição e se essa condição falhar, é lançada uma exceção com um determinado ERRCODE e uma mensagem personalizada. Por exemplo, na verificação se um player existe através do seu id.

Quanto aos **níveis de isolamento**, apenas são mencionados os que têm diferente do nível por omissão, ou seja, quando não é dito nada, está a ser usado o nível read committed.

a. 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)**.

b. Remover o modelo físico

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

c. 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.

d. Mecanismos para criar, desativar e banir o jogador

Foi criado o procedimento **criarJogador** realizada a inserção de dados na tabela PLAYER com o comando insert dados o email, região, username e estado de atividade do jogador. Este verifica se já existe outro jogador com o mesmo email ou nome, se não houver, insere o novo jogador na tabela PLAYER.

Foi criado um procedimento auxiliar **mudarEstadoJogador** que atualiza o estado de atividade do jogador na tabela PLAYER, recebendo como entrada o id do jogador e o novo estado deste. Desta forma, os procedimentos **desativarJogador** e **banirJogador** apenas precisam de chamar o procedimento auxiliar com o novo estado de "Inactive" e "Banned", respetivamente.

e. 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.

f. 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.

g. 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.

h. 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.

A 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.

Devido à primeira consideração mencionada foi definido o nível de isolamento **repeatable read** para garantir que não existe dupla (e concorrente) atribuição de crachás.

i. 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 e o m_time a uma mensagem no momento da sua criação. Isto é necessário porque o n_order 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. É importante referir, que isto também poderia ter sido feito através de uma ***procedure***.

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.

j. 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 à tabela **CHAT_LOOKUP** verificar 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.

Foi definido o nível de isolamento **repeatable read** uma vez que as verificações feitas inicialmente têm de continuar a ser válidas para evitar uma tentativa de dupla (e concorrente) adição do jogador à conversa.

k. 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.

Foi definido o nível de isolamento **repeatable read** devido às verificações efetuadas continuarem a ter de ser válidas no momento de inserir a mensagem na tabela. Por exemplo, o utilizador continua a ter de estar na conversa.

l. 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.

m.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.

n. 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.

4. Avaliação Experimental

Foi criado um script de testes às funcionalidades desde a alínea 2d a 2n para cenários normais e de erro, onde demonstramos o correto funcionamento da nossa lógica e pensamento. Este script, ao ser executado, lista, para cada teste, o seu nome e indicação se ele correu ou não com sucesso, como demonstrado em baixo:

Sucesso: "teste 1: Inserir jogador com dados bem passados: Resultado OK"
Insucesso: "teste 1: Inserir jogador com dados bem passados: Resultado FAIL"

Para correta utilização destes testes assume-se que a seguinte ordem de scripts foi executada:

createTable → insertTable → 2d-2n → tests

Para uso real da base de dados deve-se remover também o modelo físico e recriar as tabelas/procedimentos, visto os testes serem intrusivos:

removeTable → createTable → insertTable → 2d-2n → tests

5. Conclusões

Com este trabalho foi construído um sistema 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 nível de isolamento da base de dados

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

Referências

[1] GeeksForGeeks, PostgreSQL – CREATE PROCEDURE
https://www.geeksforgeeks.org/postgresql-create-procedure/, 2020

[2] JavaTPoint, PostgreSQL Functions
https://www.javatpoint.com/postgresql-functions, 2021

[3] PostgreSQL, Transactions Isolation Levels
https://www.postgresql.org/docs/14/transaction-iso.html, 2023

[4] Walter Vieira, Vistas e Extensões SQL para programação
Microsoft PowerPoint - SisInf_M2_SP_Trig_Func(v2).pptx (isel.pt), 2023