Material de Banco de Dados

Release 2016

Rodrigo Dornel

set 27, 2017

Sumário

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> —Linus Torvalds

CAPÍTULO 1

Informações

Licença

CAPÍTULO 2

Introdução

Este material será usado dentro da disciplina de Banco de Dados do professor Rodrigo Dornel.

Todo material será desenvolvido durante o ano letivo com a colaboração dos alunos.

Todo e qualquer conteúdo incluído dentro deste material será avaliado antes de ser publicado.

Todo e quelquer texto, imagem, vídeo ou ainda qualquer conteúdo externo deverá ser referenciado, citando o autor ou proprietário do conteúdo.

CAPÍTULO 3

Linguagem SQL

Ótimo local para buscar refências e exemplos de comandos em diversos SGBD´s.

http://www.w3schools.com/sql/

CREATE

- Comando utilizado para criar os principais objetos em um banco de dados.

Neste tópico vamos trabalhar com as diversas variações do comando `CREATE` relacionados ao início dos trabalhos com criação das entidades no banco de dados.

O Primeiro comando é o `CREATE DATABASE`, que cria o Banco de dados e suas dependências, como arquivos e metadados dentro do sistema. Vale lembrar que alguns sistemas gerenciadores de bancos de dados podem implementar maneiras diferentes de tratar os bancos de dados ou espaços de trabalho de cada usuário ou sistema. Sugiro a leitura do link abaixo, que explica como o Oracle trabalha, ao contrário do SQL Server que vemos em sala de aula.

http://www.oracle.com/technetwork/pt/articles/database-performance/introducao-conceito-de-tablespaces-495850-ptb.html

No nosso banco de dados de Exemplo temos a criação básica de um banco de dados e a criação de uma tabela chamada `Clientes`. Depois usamos o comando use para posicionar a execução dos comandos no banco de dados `MinhaCaixa`.

```
1  CREATE DATABASE MinhaCaixa;
2
3  use MinhaCaixa;
4
5  CREATE TABLE Clientes (
6    ClienteCodigo int,
7    ClienteNome varchar(20)
8  );
```

Podemos ter variações do comando `CREATE TABLE` de acordo com a necessidade. Abaixo temos diversas implementações do comando `CREATE` e suas CONSTRAINT´s.

CONSTRAINT PRIMARY KEY & IDENTITY

Nesse exemplo adicionamos uma chave primária ao campo `ClienteCodigo` e configuramos a propriedade `IDENTITY` que vai gerar um número com incremento de (um) a cada inserção na tabela `Clientes`. Você pode personalizar o incremento de acordo com sua necessidade, neste exemplo temos (1,1) iniciando em um e incrementando um.

```
CREATE TABLE Clientes (
  ClienteCodigo int IDENTITY (1,1) CONSTRAINT PK_Cliente PRIMARY KEY,
  ...
);
```

CONSTRAINT FOREIGN KEY

Neste exemplo temos a criação da `FOREIGN KEY` dentro do bloco de comando `CREATE`. Se tratando de uma chave estrangeira temos que tomar o cuidado de referenciar tabelas que já existem para evitar erros. Repare que no comando abaixo estamos criando uma tabela nova chamada `Contas` e especificando que o código de cliente deverá estar cadastrado na tabela de Cliente, portanto deve existir antes uma tabela `Cliente` que será referenciada nessa chave estrangeira `FOREIGN KEY`. Repare que sempre damos um nome para a `CONSTRAINT`, isso é uma boa prática, para evitar que o sistema dê nomes automáticos.

```
CREATE TABLE Contas
  (
  AgenciaCodigo int,
  ContaNumero VARCHAR (10) CONSTRAINT PK_CONTA PRIMARY KEY,
  ClienteCodigo int,
  ContaSaldo MONEY,
  ContaAbertura datetime
  CONSTRAINT FK_CLIENTES_CONTAS FOREIGN KEY (ClienteCodigo) REFERENCES␣
↪Clientes(ClienteCodigo)
  );
```

ALTER TABLE ADD CONSTRAINT

Também podemos adiconar CONSTRAINT´s através do comando `ALTER TABLE ... ADD CONSTRAINT`. Geralmente após criar todas as entidades podemos então criar as restrições entre elas.

```
ALTER TABLE Contas ADD CONSTRAINT FK_CLIENTES_CONTAS FOREIGN KEY (ClienteCodigo)
REFERENCES Clientes(ClienteCodigo);
```

CONSTRAINT´s de domínio

```
ALTER TABLE Clientes ADD CONSTRAINT chk_cliente_saldo CHECK ([ClienteNascimento] <␣
↪GETDATE() AND ClienteNome <> 'Sara');
```

Abaixo a mensagem de tentativa de violação da `CONSTRAINT` acima.

```
The INSERT statement conflicted with the CHECK constraint "chk_cliente_saldo". The␣
↪conflict occurred in database "MinhaCaixa", table "dbo.Clientes".
```

Apenas checando uma condição, data de nascimento menor que data atual. No SQL Server para pegarmos a data atual usamos `GETDATE()`:

```
ALTER TABLE Clientes ADD CONSTRAINT TESTE CHECK ([ClienteNascimento] < GETDATE());
```

INSERT

- Comando utilizando para popular as tabelas no banco.

O comando `INSERT` também possui algumas variações que devem ser respeitadas para evitar problemas. O primeiro exemplo abaixo mostra a inserção na tabela `Clientes`. Repare que logo abaixo tem um fragmento da criação da tabela `Clientes` mostando que o campo `ClienteCodigo` é `IDENTITY`, portanto não deve ser informado no momento do `INSERT`.

```
INSERT Clientes (ClienteNome) VALUES ('Nome do Cliente');

CREARTE TABLE Clientes
  (
  ClienteCodigo int IDENTITY CONSTRAINT PK_CLIENTES PRIMARY KEY...
```

Quando vamos fazer o `INSERT` em uma tabela que não possui o campo `IDENTITY` passamos o valor desejado, mesmo que o campo seja `PRIMARY KEY`.

```
INSERT Clientes (ClienteCodigo, ClienteNome) VALUES (1, 'Nome do Cliente');

CREATE TABLE Clientes
  (
  ClienteCodigo int CONSTRAINT PK_CLIENTES PRIMARY KEY...

INSERT Clientes (colunas) VALUES (valores);

INSERT INTO Clientes SELECT * FROM ...
```

SELECT - Nível 1

- Comando utilizado para recuperar as informações armazenadas em um banco de dados.

O comando `SELECT` é composto dos atributos que desejamos, a ou as tabela(s) que possuem esses atributos e as consdições que podem ajudar a filtrar os resultados desejados. Não é uma boa prática usar o `*` ou *star* para trazer os registros de uma tabela. Procure especificar somente os campos necessários. Isso ajuda o motor de execação de consultas a construir bons planos de execução. Se você conhecer a estrutura da tabela e seus índices, procure tirar proveito disso usando campos chaves, ou buscando e filtrando por atributos que fazem parte de chaves e índices no banco de dados.

```
SELECT * FROM Clientes;
```

- O Comando `FROM` indica a origem dos dados que queremos.

Na consulta acima indicamos que queremos todas as informações de clientes. É possível especificar mais de uma tabela no comando `FROM`, porém, se você indicar mais de uma tabela no comando `FROM`, lembre-se de indicar os campos que fazem o relacionamento entre as tabelas mencionadas na cláusula `FROM`.

- O comando `WHERE` indica quais as consições necessárias e que devem ser obedecidadas para aquela consulta.

Procure usar campos restritivos ou indexados para otimizar sua consulta. Na tabela `Clientes` temos o código do cliente como chave, isso mostra que ele é um bom campo para ser usado como filto.

```
SELECT ClienteNome FROM Clientes WHERE ClienteCodigo=1;
```

- Um comando que pode auxiliar na obtenção de metadados da tabela que você deseja consultar é o comando `sp_help`. Esse comando mostrar a estrutura da tabela, seus atributos, relacionamentos e o mais importante, se ela possui índice ou não.

```
sp_help clientes
```

- Repare que a tabela Clientes possui uma chave no `ClienteCodigo`, portanto se você fizer alguma busca ou solicitar o campo `ClienteCodigo` a busca será muito mais rápida. Caso você faça alguma busca por algum campo que não seja chave ou não esteja "indexado" (Veremos índice mais pra frente) a busca vai resultar em uma varredura da tabela, o que não é um bom negócio para o banco de dados.
- Para escrever um comando `SELECT` procuramos mostrar ou buscar apenas os atributos que vamos trabalhar, evitando assim carregadar dados desecessários e que serão descartados na hora da montagem do formulário da aplicação. Também recomendamos o uso do nome da Tabela antes dos campos para evitar erros de ambíguidade que geralmente aparecem quando usamos mais de uma tabela.

```
SELECT Clientes.ClienteNome FROM Clientes;
```

- Você pode usar o comando `AS` para dar apelidos aos campos e tabelas para melhorar a visualiação e compreensão.

```
SELECT Clientes.ClienteNome AS Nome FROM Clientes;

SELECT C.ClienteNome FROM Clientes AS C;
```

- Você pode usar o operador `ORDER BY` para ordenar os registros da tabela.

Procure identificar os campos da ordenação e verificar se eles possuem alguma ordenação na tabela através de algum índice. As opererações de ordenação são muito custozas para o banco de dados. A primeira opção traz os campos ordenados em ordem ascendente `ASC`, não precisando informar o operador. Caso você deseje uma ordenação descendente `DESC` você deverá informar o `DESC`.

```
SELECT Clientes.ClienteNome FROM Clientes
  ORDER BY Clientes.ClienteNome;

SELECT Clientes.ClienteNome FROM Clientes
  ORDER BY Clientes.ClienteNome DESC;
```

- Outro operador que é muito utilizado em parceria com o `ORDER BY` é o `TOP`, que permite limitar o conjunto de linhas retornado. Caso ele não esteja associado com o `ORDER BY` ele trará um determinado conjunto de dados baseado na ordem em que estão armazenados. Caso você use um operador `ORDER BY` ele mostrar os `TOP` maiores ou menores. O Primeiro exemplo mostra as duas maiores contas em relação ao seu saldo. A segunda, as duas menores.

```
SELECT TOP 2 ContaNumero, ContaSaldo FROM Contas
  ORDER BY ContaSaldo DESC;

SELECT TOP 2 ContaNumero, ContaSaldo FROM Contas
  ORDER BY ContaSaldo;
```

- Podemos usar mais de uma tabela no comando `FROM` como falamos anteriormente, porém devemos respeitar seus relacionamentos para evitar situações como o exemplo abaixo. Execute o comando e veja o que acontece.

```
SELECT * FROM Clientes, Contas;
```

- A maneira correta deve levar em consideração que as tabelas que serão usadas tem relação entre si "chaves", caso não tenham, poderá ser necessário passar por um outra tabela antes. Lembre-se das tabelas associativas.

```
SELECT CLientes.ClienteNome, Contas.ContaSaldo
  FROM Clientes, Contas
  WHERE Clientes.ClienteCodigo=Contas.ClienteCodigo;
```

- O comando `LIKE` é usado para encontrar registros usando parte do que sabemos sobre ele. Por exemplo podemos buscar todas as pessoas que tenham nome começado com `R`, usando um coringa `%` (Percentual). Podemos fazer diversas combinação com o `%`.

Documentação do comando LIKE

```
SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua LIKE 'a%' AND ClienteRua NOT␣
↪LIKE 'E%';

SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua LIKE '%a%';

SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua LIKE '%a';

SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua NOT LIKE 'a%';
```

- O Comando `CASE` é utilizado quando queremos fazer validações e até gerar novar colunas durante a execução da consulta. No exemplo abaixo fazemos uma classificação de um cliente com base no seu saldo, gerando assim uma nova coluna `Curva Cliente`.

```
SELECT ContaNumero,
  CASE WHEN ContaSaldo < 200 THEN 'Cliente C' WHEN ContaSaldo < 500 THEN 'Cliente B
↪'
  ELSE 'Cliente A' END AS 'Curva Cliente'
  FROM dbo.Contas;
```

- Podemos incluir em nossas consultas diversos operadores condicionais: = (igual), <> (diferente), >, <, <=, >=, `OR` (ou), `AND` (e) e `BETWEEN`.

```
SELECT Nome_agencia, Numero_conta, saldo
  FROM Conta
  WHERE saldo > 500 AND Nome_agencia = 'Joinville';

SELECT AgenciaCodigo FROM dbo.Agencias
  WHERE AgenciaCodigo BETWEEN 1 AND 3;
```

- O `ALIAS` ou apelido ajuda na exibição de consultas e tabelas. Dessa forma podemos dar nomes amigáveis para campos e tabelas durante a execução de consultas. Use sempre o `AS` antes de cada `ALIAS`, mesmo sabendo que não é obrigatório.

```
SELECT Nome_agencia,C.Numero_conta,saldo AS [Total em Conta],
    Nome_cliente,D.Numero_conta AS 'Conta do Cliente'
  FROM Conta AS C, Depositante AS D
  WHERE C.Numero_conta=D.Numero_conta AND Nome_cliente IN ('Rodrigo','Laura')
  ORDER BY saldo DESC
```

- O comando `DISTINCT` serve para retirar do retorno da consulta registros repitidos.

```
SELECT DISTINCT Cidade_agencia FROM Agencia;
```

- A SUB CONSULTA, `IN` e `NOT IN` são poderosos recursos para autilizar em buscas e filtragem de registros. Podemos criar subconjuntos de registros e usar operadores como `IN` para validar se os registros estão dentro daquele subconjunto.

```
SELECT AgenciaCodigo FROM dbo.Agencias
  WHERE AgenciaCodigo NOT IN ('1','4');

SELECT Contas.ContaNumero, Contas.ContaSaldo, Contas.AgenciaCodigo
  FROM Contas INNER JOIN
    (
    SELECT AgenciaCodigo, MAX(ContaSaldo) AS VALOR
    FROM Contas
    GROUP BY AgenciaCodigo
    ) AS TB2
  ON
  TB2.AgenciaCodigo=Contas.AgenciaCodigo AND TB2.VALOR=Contas.ContaSaldo;
```

- Os operadores `UNION` e `UNION ALL` ajudam a consolidar conjuntos de registros que são retornados por consultas distintas. O operador `ALL` faz a junção dos consultos sem eliminar duplicados. Precisamos obedecer o mesmo número de colunas e tipos de dados entre as consultas.

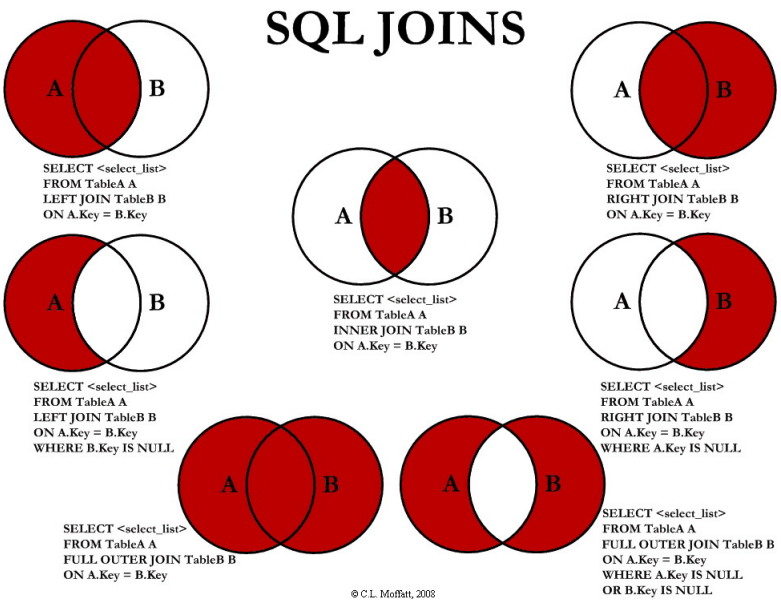
```
SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 1
UNION
SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 2;

SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 1
UNION ALL
SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 1;
```

SELECT - Nível 2

- Existem diversos tipos de `JOINS`. O mais tradicional e restritivo é o `JOIN` ou `INNER JOIN` que requer que o registros usado na comparação exista em ambas as tabelas.

No exemplo abaixo, o `ClienteCodigo` não poderá ser vazio em nenhuma das tabelas envolvidas, caso isso ocorra, aquela linha não será retornada no resultado.



Fonte da imagem: Representação Visual das Joins

```
SELECT * FROM Clientes
  JOIN Contas
    ON Clientes.ClienteCodigo=Contas.ClienteCodigo;

SELECT * FROM CLIENTES
  INNER JOIN Contas
    ON Clientes.ClienteCodigo=Contas.ClienteCodigo;
```

- LEFT JOIN

O comando LEFT indica que todos os registros existentes na tabela da sua esquerda serão retornados e os registros da outra tabela da direita irão ser retornados ou então virão em branco.

```
SELECT ClienteNome, ContaSaldo,
  CASE WHEN CartaoCodigo IS NULL THEN 'LIGAR' ELSE 'NÃO INCOMODAR' END AS
↪'NN'
```

```
  FROM Clientes
  INNER JOIN Contas
  ON (Contas.ClienteCodigo = Clientes.ClienteCodigo)
  LEFT JOIN CartaoCredito
  ON (CartaoCredito.ClienteCodigo = Clientes.ClienteCodigo);
```

- RIGHT

Já o comando RIGHT traz todos os registros da tabela da direita e os registos da tabela da esquerda, mostrando em branco

```
SELECT * FROM CartaoCredito RIGHT JOIN Clientes ON CartaoCredito.
↪ClienteCodigo=Clientes.ClienteCodigo;
```

- FULL

O comando full retorna todos os registros das tabelas relacionadas, mesmo que não existe um correspondente entre elas.

```
SELECT * FROM CartaoCredito FULL OUTER JOIN Clientes ON CartaoCredito.
↪ClienteCodigo=Clientes.ClienteCodigo;
```

- CROSS

Efetua um operação de produto cartesiano, para cada registro de uma tabela ele efetua um relacionamento com os registros das outras tabelas.

```
SELECT * FROM CLIENTES CROSS JOIN Contas;
```

- As FUNÇÕES DE AGREGAÇÃO, `SUM`, `MIN`, `MAX`, `COUNT`, `AVG` permitem um nível mais robusto de informação, criando cojuntos de dados agrupados, médias entre outros, permitindo que possamos resumir e totatlizar comjuntos de resultados. Sempre que usarmos a função de agregação em conjunto com um campo agregador devemos usar a função `GROUP BY` para indicar qual o compo será o responsável pelo agrupamento das informações.

Caso você deseje comparar conjuntos de informações contidos na função de agragação você deve compará-los usando o `HAVING`.

```
SELECT TOP 2 AgenciaNome, SUM(ContaSaldo) AS TOTAL
  FROM Contas,  Agencias
  WHERE Agencias.AgenciaCodigo=Contas.AgenciaCodigo
  GROUP BY AgenciaNome
  HAVING SUM(ContaSaldo) > (SELECT MAX(ContaSaldo) AS VALORMETA FROM Contas AS␣
↪META)
  ORDER BY 2 DESC;

SELECT SUM( Contas.ContaSaldo),
  AgenciaCodigo, ContaNumero
  FROM Contas
  GROUP BY AgenciaCodigo,ContaNumero
  --WHERE COM AVG ???
  --WHERE COM SUBCONSULTA ???
  HAVING SUM( Contas.ContaSaldo) > (SELECT AVG( Contas.ContaSaldo) FROM  Contas); -
↪-667,0833

SELECT MAX(ContaSaldo) FROM  Contas;
SELECT MIN(ContaSaldo) FROM  Contas;
SELECT AVG(ContaSaldo) FROM  Contas;
SELECT COUNT(*), COUNT(CONTAS.ClienteCodigo), COUNT(DISTINCT CONTAS.ClienteCodigo)␣
↪FROM  Contas;
```

- EXISTS

O comando EXISTS é pareceido com o comando IN, quando queremos comparar mais de um campo contra uma subconsulta.

```
SELECT * FROM  Contas C
    WHERE EXISTS
                      (SELECT * FROM  CartaoCredito CC
                              WHERE C.ClienteCodigo=CC.ClienteCodigo
                              AND C.AgenciaCodigo=CC.AgenciaCodigo
                      )
```

- FUNÇÕES DE Data e Hora

```
SET DATEFORMAT YDM

SET LANGUAGE PORTUGUESE

SELECT YEAR(getdate()) -YEAR( Clientes.ClienteNascimento),
  DATEDIFF(YEAR,ClienteNascimento,GETDATE()),
  DATEPART(yy,ClienteNascimento),
  dateadd(yy,1,ClienteNascimento),
  EOMONTH(GETDATE()),
  DATENAME(MONTH,(GETDATE()))
FROM  Clientes;
```

```
SELECT * FROM  Contas
  WHERE YEAR(ContaAbertura) = '2011'
  ORDER BY ContaAbertura;
```

- Variáveis

Muitas vezes necessitamos armazenar determinados valores para uso posterior. Um exemplo é gardar um valor total em uma variável para que ele seja usado em cálculo de percentual por exemplo

```
declare @numero int
set @numero = 1

declare @dia int
set @dia = (select day(getdate()))
```

- SELECT INTO

```
      SELECT Clientes.ClienteNome,
      DATEDIFF(YEAR,Clientes.ClienteNascimento,GETDATE()) AS IDADE
      INTO ClientesIdade -- O comando INTO vem depois do campos listados no SELECT e antes do FROM.
      FROM Clientes

      SELECT * FROM ClientesIdade
```

- CAST, CONVERT e concatenação

Comandos utilizados para converter tipos de dados e concatenar Strings.

```
SELECT Clientes.ClienteNome + Clientes.ClienteCidade FROM Clientes;

SELECT Clientes.ClienteNome + ' ' + Clientes.ClienteCidade FROM Clientes;

SELECT Clientes.ClienteNome + ' de ' + Clientes.ClienteCidade FROM Clientes;

SELECT Clientes.ClienteNome + ' - R$ ' + CAST (Contas.ContaSaldo AS VARCHAR(10) )FROM Clientes INNER JOIN Contas ON Contas.ClienteCodigo = Clientes.ClienteCodigo;
```

```
8
9  SELECT Clientes.ClienteNome + ' - R$ ' + CONVERT  (VARCHAR(10), Contas.
   ↪ContaSaldo )FROM Clientes INNER JOIN Contas ON Contas.ClienteCodigo =␣
   ↪Clientes.ClienteCodigo;
```

UPDATE

- Comando utilizado para alterar registros em um banco de dados. Antes de executar qualquer comando `UPDATE`, procure se informar sobre transações (será abordado mais pra frente).
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
1  UPDATE CartaoCredito SET CartaoLimite = 1000 WHERE ClienteCodigo = 1;
```

DELETE

- Comando utilizado para deletes registros em um banco de dados.
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
1  DELETE FROM CartaoCredito WHERE ClienteCodigo = 1;
```

VIEW

- Comando utilizado para alterar registros em um banco de dados. Antes de executar qualquer comando `UPDATE`, procure se informar sobre transações (será abordado mais pra frente).
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
1  CREATE VIEW ClientesIdade
2  AS
3  SELECT ClienteNome,DATEDIFF(YEAR,ClienteNascimento,GETDATE()) AS Idade         FROM␣
   ↪dbo.Clientes;
```

PROCEDURES

- Uma procedure é um bloco de comandos ou instruções SQL organizados para executar uma ou mais tarefas. Ela pode ser utilizada para ser acionada através de uma chamada simples que executa uma série de outros comandos.

```
1  CREATE PROCEDURE uspRetornaSaldo
2  @Nome nvarchar(50)
3  AS
4  SELECT Clientes.ClienteNome, Contas.ContaSaldo
5  FROM Clientes
6  INNER JOIN Contas ON Clientes.ClienteCodigo=Contas.ClienteCodigo
7  WHERE Clientes.ClienteNome = @Nome;
```

- Execução da procedure

```
exec uspRetornaSaldo 'Ana';
```

IF

Comando utilizado para checar condições.

```
CREATE PROCEDURE uspRetornaSaldo2
@Nome nvarchar(50)
AS
BEGIN

        IF @Nome = 'Ana'
                BEGIN
                SELECT Clientes.ClienteNome, Contas.ContaSaldo
                FROM Clientes
                INNER JOIN Contas ON Clientes.ClienteCodigo=Contas.ClienteCodigo
                WHERE Clientes.ClienteNome = @Nome;
                END
        ELSE
                BEGIN
                SELECT @Nome
                END

END
```

WHILE

Comando utilizado para realizar laços de repetição.

```
DECLARE @contador INT
SET @contador = 1
WHILE @contador <= 5
BEGIN
  SELECT @contador
  SET @contador = @contador + 1
END
```

FUNÇÕES

- Uma função é ...

```
CREATE FUNCTION fnRetornaAno (@data DATETIME)
RETURNS int
AS
  BEGIN
  DECLARE @ano int
  SET @ano = YEAR(@data)

      RETURN @ano

END
```

- Chamada ou execução da função

```
SELECT dbo.fnRetornaAno(GETDATE())

SELECT dbo.fnRetornaAno(Clientes.ClienteNascimento) FROM dbo.Clientes
```

TRANSAÇÕES

Transações

Comando utilizado para alterar registros em um banco de dados. Antes de executar qualquer comando `UPDATE`, procure se informar sobre transações (será abordado mais pra frente). Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
BEGIN TRAN --> Inicia a transação

UPDATE dbo.CartaoCredito SET CartaoLimite = CartaoLimite * 1.1

COMMIT --> Finaliza a transação

--OR

ROLLBACK --> Desfaz a transação
```

Execute primeiro sem o WHERE e verifique que nenhuma linha será alterada. Depois remova o comentário e verá que apenas uma linha foi alterada.

```
BEGIN TRAN

UPDATE dbo.CartaoCredito SET CartaoLimite = CartaoLimite * 1.1
--WHERE ClienteCodigo = '12'

IF (@@ROWCOUNT > 1 OR @@ERROR > 0)

  ROLLBACK

ELSE

  COMMIT
```

Try Catch

```
BEGIN TRY

  SELECT 1/0

END TRY

BEGIN CATCH
  SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

TRIGGERS

- Comando vinculado a uma tabela que executa um ação assim que algum comando de UPDATE, INSERT ou DELETE é executado na tabela onde a trigger está vinculada.

Trigger para INSERT

```
CREATE TRIGGER trgINSERT_CLIENTE
ON Clientes
FOR INSERT
AS
BEGIN
INSERT clientes_audit
SELECT *,[TRG_OPERACAO] = 'INSERT', [TRG_DATA]=GETDATE(), [TRG_FLAG]='NEW' FROM␣
↪Inserted
END;
```

Trigger para DELETE

```
CREATE TRIGGER trgDELETE_CLIENTE
ON dbo.Clientes
FOR DELETE
AS
BEGIN
INSERT dbo.clientes_audit SELECT *,[TRG_OPERACAO] = 'DELETE',[TRG_DATA]=GETDATE(),
↪[TRG_FLAG]='OLD' FROM Deleted
END;
```

Trigger para UPDATE

```
CREATE TRIGGER trgUPDATE_CLIENTE
ON dbo.Clientes
FOR UPDATE
AS
BEGIN
INSERT dbo.clientes_audit SELECT *,[TRG_OPERACAO] = 'UPDATE',[TRG_DATA]=GETDATE(),
↪[TRG_FLAG]='OLD' FROM Deleted
INSERT dbo.clientes_audit SELECT *,[TRG_OPERACAO] = 'UPDATE',[TRG_DATA]=GETDATE(),
↪[TRG_FLAG]='NEW' FROM Inserted
END;
```

INDICES

- Criação de índices e estatísitcas

Os índices garantem um bom desempenho para as consultas que serão realizadas no banco de dados. Comece verificando com a procedure sp_help os metadados das tabelas para verificar se não existe um índice que possa ajudar na sua consulta.

Caso precise criar um índice comece analisando os campos que estão na sua cláusula WHERE. Esses campos são conhecidos como predicados. Ainda dentro da cláusula WHERE procure filtrar primeiramente os campos com maior seletividade, que possam filtar os dados de forma que não sejam trazidos ou pesquisados dados descessários.

Em seguida olhe os campos da cláusula SELECT e adicione eles no índice.

- Atenção Leia o material complementar na biblioteca Virtual
- Exemplo

A consulta abaixo busca nome e data de nascimentos do cliente com base em uma data passada pelo usuário ou sistema. Como primeiro passo vamos olhar a cláusula WHERE e em seguida a cláusula SELECT. Dessa forma temos um índice que deverá conter ClienteNascimento e ClienteNome onde ClienteNascimento é o predicado.

Comando

```
SELECT Clientes.ClienteNome, Clientes.ClienteNascimento
FROM Clientes
WHERE ClienteNascimento >= '1980-01-01'
```

Índice

```
CREATE INDEX IX_NOME ON Clientes
(
        ClienteNascimento,
        ClienteNome
)
```

EXERCÍCIOS

1. Mostre quais os clientes tem idade superior a média.

```
SELECT ClienteNome, YEAR(GETDATE()) - YEAR(ClienteNascimento) AS idade
  FROM Clientes
  WHERE YEAR(GETDATE()) - YEAR(ClienteNascimento) >
    (
      SELECT AVG(YEAR(GETDATE()) -YEAR(ClienteNascimento)) AS IDADE FROM␣
↪Clientes
    );
```

2. Mostre qual agência tem quantidade de clientes acima da média.

```
SELECT AgenciaNome, COUNT(ClienteCodigo) AS QDTE
FROM Contas INNER JOIN Agencias
  ON Agencias.AgenciaCodigo = Contas.AgenciaCodigo
GROUP BY AgenciaNome
HAVING COUNT(ClienteCodigo) >
  (SELECT COUNT(DISTINCT ClienteCodigo)/
  COUNT(DISTINCT AgenciaCodigo) FROM Contas);
```

3. Mostre o nome da agência o saldo total, o mínimo, o máximo e a quantidade de clientes de cada agência.

```
SELECT AgenciaNome, SUM(ContaSaldo) AS TOTAL ,MIN(ContaSaldo) AS MINIMO,␣
↪MAX(ContaSaldo) AS MAXIMO,
COUNT(Contas.ClienteCodigo) AS QTDE_CLIENTES
FROM Contas INNER JOIN dbo.Agencias ON Agencias.AgenciaCodigo = Contas.
↪AgenciaCodigo
GROUP BY dbo.Agencias.AgenciaNome;
--ATENCAO AQUI PARA COUNT(*) E COUNT(DISTINT)
```

4. Mostre o percentual que cada agencia representa no saldo total do banco.

```
SELECT AgenciaNome, SUM(ContaSaldo) / (SELECT SUM(ContaSaldo) FROM dbo.
↪Contas) * 100 AS PERCENTUAL
FROM Contas INNER JOIN dbo.Agencias ON Agencias.AgenciaCodigo = Contas.
↪AgenciaCodigo
GROUP BY dbo.Agencias.AgenciaNome;
```

5. Mostre as duas cidades que tem o maior saldo total

```
SELECT TOP 2 AgenciaCidade, SUM(ContaSaldo) AS SALDO_TOTAL
FROM Contas INNER JOIN Agencias ON Agencias.AgenciaCodigo = Contas.
↪AgenciaCodigo
GROUP BY AgenciaCidade
ORDER BY 2 DESC;
```

6. Mostre qual a agência tem o maior montante de emprestimo

```
SELECT TOP 1 AgenciaCidade, Emprestimos.EmprestimoTotal
FROM dbo.Emprestimos INNER JOIN Agencias ON Agencias.AgenciaCodigo =
↪Emprestimos.AgenciaCodigo
ORDER BY 2 DESC;
```

7. Mostre qual o menor valor devido, o maior e o total devido da tabela devedor

```
SELECT MIN(DevedorSaldo) AS MINIMO, MAX(DevedorSaldo) AS MAXIMO,
↪SUM(DevedorSaldo) AS TOTAL
FROM dbo.Devedores;
```

8. Mostre o nome do cliente, se ele tem cartão de crédito, apenas do cliente que é o maior devedor.

```
SELECT TOP 1 --Experimente remover o TOP 1 para conferir o resultado
ClienteNome
,CASE WHEN dbo.CartaoCredito.ClienteCodigo IS NULL THEN 'NÃO TEM CARTÃO
↪CRÉDITO' ELSE 'TEM CARTÃO CRÉDITO' END AS 'CARTAO'
,DevedorSaldo FROM dbo.Clientes
INNER JOIN dbo.Devedores ON Devedores.ClienteCodigo = Clientes.
↪ClienteCodigo
LEFT JOIN dbo.CartaoCredito ON CartaoCredito.ClienteCodigo = Clientes.
↪ClienteCodigo
ORDER BY 3 DESC;
```

9. Mostre o nome do cliente, a idade, o saldo total em conta, seu total devido, seu total emprestado e se tem cartão de crédito ou não. Os valores nulos devem aparecer como 0.00. A ordenação dever ser sempre pelo maioir devedor.

```
SELECT Clientes.ClienteNome, DATEDIFF(YEAR,Clientes.ClienteNascimento,
↪GETDATE()) AS IDADE,
ISNULL(Devedores.DevedorSaldo,0) AS DevedorSaldo, ISNULL(Emprestimos.
↪EmprestimoTotal,0) AS EmprestimoTotal,
CASE WHEN CartaoCredito.CartaoCodigo IS NULL THEN 'NÃO TEM' ELSE 'TEM'
↪END AS CARTAOCREDITO
FROM Clientes
LEFT JOIN Devedores ON Devedores.ClienteCodigo = Clientes.ClienteCodigo
LEFT JOIN Emprestimos ON Emprestimos.ClienteCodigo = Clientes.
↪ClienteCodigo
LEFT JOIN CartaoCredito ON CartaoCredito.ClienteCodigo = Clientes.
↪ClienteCodigo
ORDER BY 3 DESC;
```

10. Utilizando a questão anterior, incluia o sexo do cliente e mostre também a diferença entre o que o ele emprestou e o que ele está devendo.

```
SELECT Clientes.ClienteNome, DATEDIFF(YEAR,Clientes.ClienteNascimento,
↪GETDATE()) AS IDADE,
ISNULL(Devedores.DevedorSaldo,0) AS DevedorSaldo, ISNULL(Emprestimos.
↪EmprestimoTotal,0) AS EmprestimoTotal,
CASE WHEN .CartaoCredito.CartaoCodigo IS NULL THEN 'NÃO TEM' ELSE 'TEM'
↪END AS CARTAOCREDITO,
CASE WHEN ClienteNome LIKE '%a' THEN 'FEMININO' ELSE 'MASCULINO' END AS
↪SEXO,
ISNULL((Emprestimos.EmprestimoTotal-DevedorSaldo),0) AS DIFERENÇA
FROM Clientes
LEFT JOIN Devedores ON Devedores.ClienteCodigo = Clientes.ClienteCodigo
LEFT JOIN Emprestimos ON Emprestimos.ClienteCodigo = Clientes.
↪ClienteCodigo
LEFT JOIN CartaoCredito ON CartaoCredito.ClienteCodigo = Clientes.
↪ClienteCodigo
```

```
ORDER BY 3 DESC;
```

11. Insira um novo cliente chamado Silvio Santos, crie uma conta para ele com saldo de R$ 500,00 na agência Beira Mar. Cadastre um cartão de crédito com limite de 5000,00.

```
INSERT Clientes (ClienteNome, ClienteRua, ClienteCidade, ClienteNascimento) VALUES  ('Silvio Santos', 'Rua João Colin, 1234', 'Joinville','1980-01-01' );

SELECT @@IDENTITY --RETORNA O CÓDIGO DO CLIENTE GERADO PELO AUTO INCREMENTO --> IDENTITY

INSERT Contas (AgenciaCodigo ,ContaNumero , ClienteCodigo , ContaSaldo , ContaAbertura)
OUTPUT INSERTED.* --RETORNA OS REGISTROS INSERIDOS NA TABELA
VALUES (5,'C-999',14,500,'2016-01-01');

INSERT CartaoCredito ( AgenciaCodigo , ClienteCodigo , CartaoCodigo , CartaoLimite)
VALUES  (5,14,'1234-1234-1234-1234',5000);
```

12. Altere a rua do cliente Ana para Rua da Univille.

```
UPDATE dbo.Clientes SET ClienteRua = 'Rua da Univille' WHERE ClienteNome = 'Ana';
```

13. Apague todos os registros do cliente Vania.

```
DECLARE @ClienteCodigo INT = (SELECT ClienteCodigo FROM dbo.Clientes WHERE ClienteNome = 'Vânia')

DELETE FROM dbo.Emprestimos WHERE ClienteCodigo = @ClienteCodigo
DELETE FROM dbo.Devedores WHERE ClienteCodigo = @ClienteCodigo
DELETE FROM dbo.Depositantes WHERE ClienteCodigo = @ClienteCodigo
DELETE FROM dbo.CartaoCredito WHERE ClienteCodigo = @ClienteCodigo
DELETE FROM dbo.Contas WHERE ClienteCodigo = @ClienteCodigo
DELETE FROM dbo.Clientes WHERE ClienteCodigo = @ClienteCodigo
```

14. Mostre nome e sobrenome e se o cliente for homem, mostre Sr e se for mulher Sra na frente do nome.Use o MinhaCaixa_Beta para resolver essa questão.

```
USE MinhaCaixa_Beta
GO
SELECT
CASE WHEN ClienteSexo = 'M' THEN 'Sr. ' + dbo.Clientes.ClienteNome + ' ' + dbo.Clientes.ClienteSobrenome
ELSE 'Sra. ' + dbo.Clientes.ClienteNome + ' ' + dbo.Clientes.ClienteSobrenome END AS Cliente
FROM dbo.Clientes
```

15. Mostre os bairros que tem mais clientes.

```
USE MinhaCaixa_Beta
GO
SELECT COUNT(dbo.Clientes.ClienteCodigo) AS Quantidade,
dbo.Clientes.ClienteBairro
FROM dbo.Clientes
GROUP BY dbo.Clientes.ClienteBairro
ORDER BY 1 desc
```

16. Mostre a renda de cada cliente convertida em dólar.

```
USE MinhaCaixa_Beta
GO
SELECT ClienteNome +' ' + ClienteSobrenome AS Cliente,
CONVERT(DECIMAL(10,2),Clientes.ClienteRendaAnual / 3.25) AS RENDADOLAR
FROM dbo.Clientes
```

17. Mostre o nome do cliente, o número da conta, o saldo da conta, apenas para os 15 melhores clientes

```
    USE MinhaCaixa_Beta
GO
SELECT TOP 15
ClienteNome +' ' + ClienteSobrenome AS Cliente, Contas.ContaNumero,
SUM(MovimentoValor*MovimentoTipo) AS Saldo
FROM dbo.Clientes INNER JOIN dbo.Contas ON Contas.ClienteCodigo =
↪Clientes.ClienteCodigo
INNER JOIN dbo.Movimentos ON Movimentos.ContaNumero = Contas.ContaNumero
GROUP BY ClienteNome + ' ' + ClienteSobrenome , Contas.ContaNumero
ORDER BY 3 DESC
```

18. Mostre quais são os 5 dias com maior movimento (valor) no banco

```
USE MinhaCaixa_Beta
GO
SELECT TOP 5 DAY(Movimentos.MovimentoData) AS DIA,
SUM(dbo.Movimentos.MovimentoValor*dbo.Movimentos.MovimentoTipo) AS VALOR
FROM dbo.Movimentos
GROUP BY DAY(Movimentos.MovimentoData)
ORDER BY 2 DESC
```

19. Crie uma função que receba o código do estado civil e mostre ele por extenso
20. Crie uma função que receba o código do sexo e mostre ele por extenso
21. Crie um procedure que receba o número da conta e cadastre um cartão de crédito com limite de R$ 500 para o cliente caso ele não tenha (MinhaCaixa).
22. Use o script abaixo para criar uma procedure que receba a matricula, disciplina, ano e calcule o total de pontos e a média do aluno

```
CREATE TABLE Notas
(
Matricula INT,
Materia CHAR (3),
Ano INT,
Nota1 FLOAT,
Nota2 FLOAT,
Nota3 FLOAT,
Nota4 FLOAT,
TotalPontos FLOAT,
MediaFinal FLOAT
);
INSERT Notas (Matricula, Materia, Ano, Nota1, Nota2, Nota3, Nota4) VALUES
↪ (1,'BDA',2016,7,7,7,7);
```

23. Use o script abaixo para criar duas procedures:

Uma procedure para cadastrar os alunos em duas matérias (BDA e PRG). Exemplo: exec procedure @matricula, @materia, @ano

(matricular 6 alunos)

Uma procedure que receba a matricula, disciplina, ano, bimestre, aulas dadas, notas e faltas. Quando a condição dentro da procedure identificar que é o quarto bimestre calcule o total de pontos, total de faltas, percentual de frequencia,a média do aluno e calcule o resultado final, A, E ou R.

Exemplo: exec procedure @matricula, @materia, @ano, 1, 32, 7, 0

```
CREATE TABLE Notas
(
Matricula INT,
Materia CHAR (3),
Ano INT,
    Aulas1 INT,
    Aulas2 INT,
    Aulas3 INT,
    Aulas4 INT,
Nota1 FLOAT,
Nota2 FLOAT,
Nota3 FLOAT,
Nota4 FLOAT,
    Falta1 INT,
    Falta2 INT,
    Falta3 INT,
    Falta4 INT,
TotalPontos FLOAT,
    TotalFaltas INT,
    TotalAulas INT,
MediaFinal FLOAT,
    PercentualFrequencia float,
    Resultado char(1)
);
```

CAPÍTULO 4

Administração de Banco de Dados

Objeito de aprender tarefas básicas do dia a dia de um administrador de banco de dados

Segurança

Logins

Manutenção

Rotinas

CAPÍTULO 5

Extra

Documentação dos SGBD

SGBD	Versão	Link
MariaDB	Inglês (Oficial)	https://mariadb.com/kb/en/mariadb/documentation/
MySQL	Inglês (Oficial)	https://dev.mysql.com/doc/
PostgreSQL	Inglês (Oficial) Português (8.0)	https://www.postgresql.org/docs/manuals/ http://pgdocptbr.sourceforge.net/pg80/index.html
SQL Server	Português (2016) Português (2016 Desenvolvimento)	https://msdn.microsoft.com/pt-br/library/ms130214.aspx https://technet.microsoft.com/pt-br/library/bb500155(v=sql.105).aspx
SQLite	Inglês (Oficial)	https://www.sqlite.org/docs.html

Sites Interessantes

- Database Cast: Podcast sobre banco de dados

Palestras

- A Arquitetura (Peculiar) do Stack Overflow
- PostgreSQL no Debian

Como Contribuir?

- Crie um fork do projeto no GitHub.

- Faça suas alterações no seu fork.
  - Se possível, utilize o plugin EditorConfig no seu editor de texto.
  - Escreva o conteúdo usando reStructuredText.
    * Fique atento a marcação dos títulos.
    * Utilize um bloco de código com syntax highlight para código.

      Exemplo:

```
.. code-block:: sql

  SELECT * FROM tabela;
```

  - Adicione os link nos arquivos `index.rst` caso tenha criado algum arquivo novo.
  - Adicione os arquivos modificados (`git add`) e faça o commit (`git commit`).
- Crie um pull request no GitHub.
- Espere sua contribuição ser aprovada.

Git

Links de material

Livros / Documentação

- Pro Git (v2 inglês) (v1 português)
- Getting Git Right

Tutoriais

- git - guia prático - sem complicação!
- Try Git

Vídeos

- Introdução ao Git
- Git para quem gosta de Git

Ferramentas

- Learn Git Branching

Como Compilar o Material com o Sphinx

Instalar o Python

- Usar preferencialmente a versão 3.
- Ambientes Unix provavelmente já possuem ele instalado.
- Pode ser encontrado em https://www.python.org/downloads/.

- No Windows, durante a instalação marcar para instalar o "pip" também.

Instalar as Dependências

Dentro do diretório do código do material executar:

```
pip install -r requirements.txt
```

Compilar o Material

Para listar as opções de compilação execute:

```
make help
```

Alguns exemplos:

HTML

```
make html
```

PDF

```
make latexpdf
```

> **Nota:** É necessário que o LaTeX esteja instalado no sistema para gerar o PDF.

ePub

```
make epub
```

Nota: Após a execução do comando, o material compilado, junto com alguns outros arquivos, podem ser encontrados dentro do diretório `_build` na raíz do projeto.