



INSTITUTO DE GESTÃO E
TECNOLOGIA DA INFORMAÇÃO

A Linguagem SQL

Bootcamp: Analista de Banco de Dados

Gustavo Aguilar

2021

A Linguagem SQL

Bootcamp: Analista de Banco de Dados

Gustavo Aguilar

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1. Introdução à Linguagem SQL	6
1.1. Banco de dados relacional: uma breve revisão	6
1.2. História da linguagem SQL	16
1.3. Padronização da linguagem SQL, classes de instruções SQL e dialetos	22
1.4. Uma “Sopa de Letras”	25
Capítulo 2. Introdução ao SQL Server	30
2.1. Introdução ao SQL Server	30
2.2. Conceitos básicos do SQL Server	31
2.3. Instalação do SQL Server	31
2.4. Introdução à T-SQL	32
2.5. Instalação e overview das ferramentas Client.....	38
2.5.1. Instalação e Overview do SQL Server Management Studio.....	38
2.5.2. Instalação e Overview do Azure Data Studio	39
2.6. Banco de dados de exemplo AdventureWorks	39
Capítulo 3. Linguagem de Definição de Dados (DDL)	40
3.1. Criação de estruturas de dados	42
3.2 Alteração de estruturas de dados	44
3.3. Remoção de estruturas de dados	46
Capítulo 4. Linguagem de Manipulação de Dados (DML)	48
4.1. Selecionando dados	48
4.2. Operadores aritméticos e de concatenação.....	52
4.3. Ordenando dados	56
4.4. Filtrando dados	58

4.4.1. Filtrando Dados com TOP / DISTINCT	62
4.5. Funções de caracteres, de data e hora.....	64
4.6. Tipos de dados e conversões Explícitas x Implícitas	76
Capítulo 5. Agrupamento de Dados e Funções de Agregação	79
5.1. Funções de agregação	79
5.2. Agrupamentos de dados	83
5.2.1. Filtrando agrupamentos de dados	86
Capítulo 6. Junção de Tabelas (JOIN)	89
6.1. Introdução à Junção de Tabelas.....	89
6.2. INNER JOIN, CROSS JOIN e OUTER JOIN	95
Capítulo 7. Subconsultas	102
7.1. Subconsulta escalar e multivalorada	102
7.2. Subconsultas correlacionadas e com operadores de conjuntos	108
Capítulo 8. Inserção, Atualização e Exclusão de Dados.....	117
8.1. Inserindo Dados.....	117
8.2. Atualizando Dados	120
8.3. Excluindo Dados	122
Capítulo 9. Linguagem de Controle de Transação (TCL).....	126
9.1. Conceitos Básicos de Transação.....	126
9.2. Controle de Transações.....	128
Capítulo 10. Linguagem de Controle de Acesso a Dados (DCL).....	130
10.1. Segurança em Bancos de Dados	130
10.2. Concessão e revogação de privilégios	132
Capítulo 11. Programação com a Linguagem T-SQL.....	134
11.1. Visões (Views) e CTEs	134

11.2. Stored Procedures	139
11.3. Functions	142
11.4. Triggers.....	143
11.5. Linked Server e Query Distribuída	144
Capítulo 12. Mapeamento Objeto Relacional.....	147
12.1. Introdução ao Mapeamento Objeto Relacional	148
12.2. Introdução ao Sequelize	151
12.2.1. Instalação e Configuração do Sequelize para NodeJS	152
12.2.2. Dialetos e Opções da Engine	156
12.3. Criando e usando um modelo de dados básico no Sequelize	161
12.4. Definição do Modelo de Dados no Sequelize	170
12.5. Usando e consultando o modelo de dados no sequelize	177
Capítulo 13. Controle de Versão de Banco de Dados.....	182
13.1. Introdução ao Controle de Versão de Banco de Dados	182
13.2. Introdução ao VersionSQL.....	187
13.3. Utilização do VersionSQL	192
13.4. Introdução ao Flyway	196
13.5. Utilização do Flyway	198
Referências.....	204

Capítulo 1. Introdução à Linguagem SQL

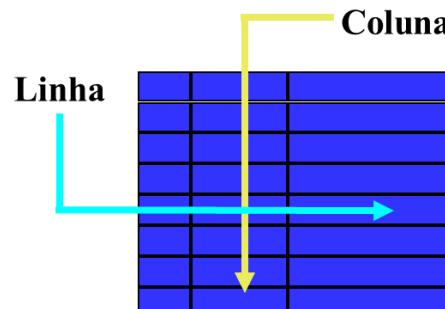
Quando falamos de **Linguagem SQL** (abreviatura para *Structured Query Language*, traduzido para o português como Linguagem Estruturada de Consulta), estamos intrinsicamente falando da **Teoria de Banco de Dados Relacional**, mais especificamente de modelos de dados e bancos de dados relacionais.

Desde o advento dos sistemas gerenciadores de bancos de dados relacionais (SGBDR ou RDBMS, abreviatura para *Relational Database Management System*), a linguagem SQL é, sem dúvida, a linguagem de consulta e manipulação de dados mais utilizada em projetos de desenvolvimento de sistemas e aplicações apoiadas em bancos de dados. Conhecer não somente os comandos e opções existentes nessa linguagem, mas saber o contexto em que ela está inserida e dominar a teoria relacional a partir da qual ela foi concebida, são aspectos fundamentais para o sucesso de um projeto de banco de dados.

1.1. Banco de dados relacional: uma breve revisão

O **Modelo de Dados Relacional** foi criado por Edgar Frank Codd, em junho de 1970, na publicação do seu artigo *A Relational Model of Data for Large Shared Data Banks*. Nesse artigo, Codd propôs um modelo onde a representação dos dados é independente de como eles são organizados e armazenados internamente nos servidores.

O grande sucesso da proposta de Codd está no fato dele ter se baseado na álgebra relacional e teoria de conjuntos, trazendo para a comunidade um conceito simplificado e mais trivial para o raciocínio lógico de desenvolvedores e projetistas de bancos de dados: **dados vistos como um conjunto de tabelas, dispostos em linhas e colunas**.



Com um determinado registro (dados inter-relacionados) sendo visto como uma **tupla** (linha formada por uma lista ordenada de colunas), a representação tabular trouxe grande facilidade para manipular e visualizar dados, em detrimento dos modelos hierárquicos e de rede em voga na época, contribuindo enormemente para o “boom” dos bancos de dados relacionais.

Figura 1 – Exemplo de Armazenamento Tabular.

COD_CLIENTE	NOM_CLIENTE	NUM_TEL_CLIENTE	IND_SEXO_CLIENTE
1	JOÃO	33333333	M
2	MARIA	99999999	F
3	JOSÉ	88888888	M
4	TIAGO	66666666	M
5	PEDRO	44444444	M
6	MATEUS	11111111	M
7	ANA	77777777	F
8	TERESA	55555555	F

Fonte: Gustavo Aguilar (2002).

Junto com a teoria da representação tabular dos dados, Codd apresentou os conceitos e recursos a serem usados na modelagem relacional, descritos a seguir:

- **Tabela:** objeto correspondente à entidade do modelo conceitual, constituindo a estrutura onde os dados serão armazenados no modelo físico do banco de dados.
- **Domínio:** natureza dos valores permitidos para um determinado conjunto de dados. Exemplos:

CÓDIGO CLIENTE → NUMBER

NOME CLIENTE → STRING

DATA NASCIMENTO → DATE

- **Coluna:** correspondente ao conceito de atributo do modelo conceitual, acrescido do domínio do dado que será armazenado na coluna. Comumente chamado de **campo**.

- **Chave Primária (“Primary Key” / “PK”):** coluna (campo) ou conjunto de colunas (chave primária composta), que identificam unicamente a tupla (linha) em uma tabela. Exemplo: número do CPF.
- **Chave Candidata:** que não repetem valor em uma tabela e que são candidatas à chave primária. Exemplos:

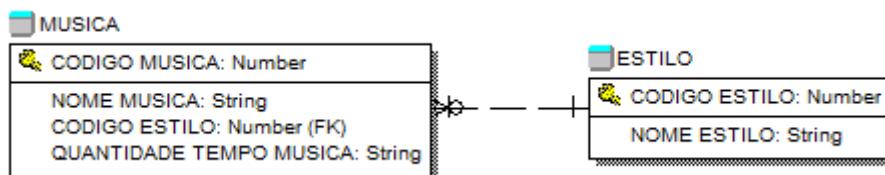
NÚMERO DO TÍTULO DO ELEITOR

NÚMERO DO RG

NÚMERO DO CPF

- **Chave Estrangeira (“Foreign Key” / “FK”):** é o elo de ligação entre duas tabelas, se constituindo na(s) coluna(s) da chave primária da tabela estrangeira, relacionada com a tabela em questão.

Figura 2 – Exemplo de Chave Estrangeira (FK).

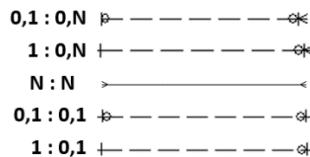


Fonte: Gustavo Aguilar (2002).

- **Relacionamento:** aplica-se o mesmo princípio do modelo conceitual para o mapeamento das relações entre as tabelas, sendo que alguns relacionamentos podem gerar novas tabelas.
- **Cardinalidade:** no modelo lógico relacional, assume uma importância ainda maior, pois além de representar a quantidade de elementos de uma entidade A que podem ser associados a elementos de uma entidade B e vice-versa, indicam também as restrições de nulidade das chaves estrangeiras.

No modelo lógico, uma das notações mais usadas para a cardinalidade de relacionamentos é a ***Information Engineering (IE)***, mostrada a seguir.

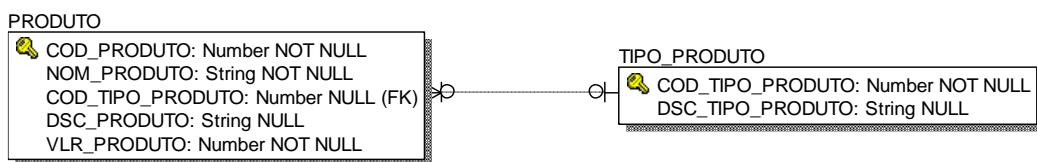
Figura 3 – Notação IE.



Fonte: Gustavo Aguilar (2002).

Essa notação, no tocante à chave estrangeira, indica a nulidade dessa coluna, ou seja, se ela sempre estará associada a um registro da outra tabela ou se a chave estrangeira aceitará um valor nulo. Cardinalidades 0,1 e 0,N implicam em chaves estrangeiras que aceitam valores nulos, como mostrado no exemplo abaixo:

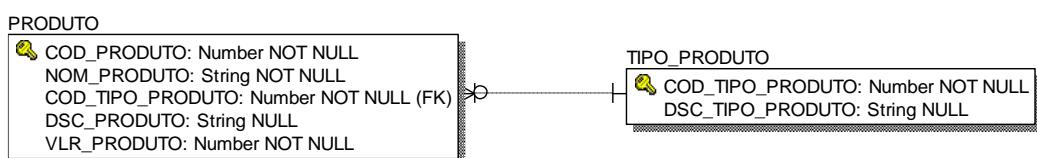
Figura 4 – Exemplo de Relacionamento 0,1:0,N.



Fonte: Gustavo Aguilar (2002).

Já as cardinalidades 1:0,N e 1:0,1 garantem que as chaves estrangeiras não aceitem nulo e estejam sempre associadas com um registro da tabela estrangeira, como mostrado no exemplo a seguir.

Figura 5 – Exemplo de Relacionamento 1:0,N.



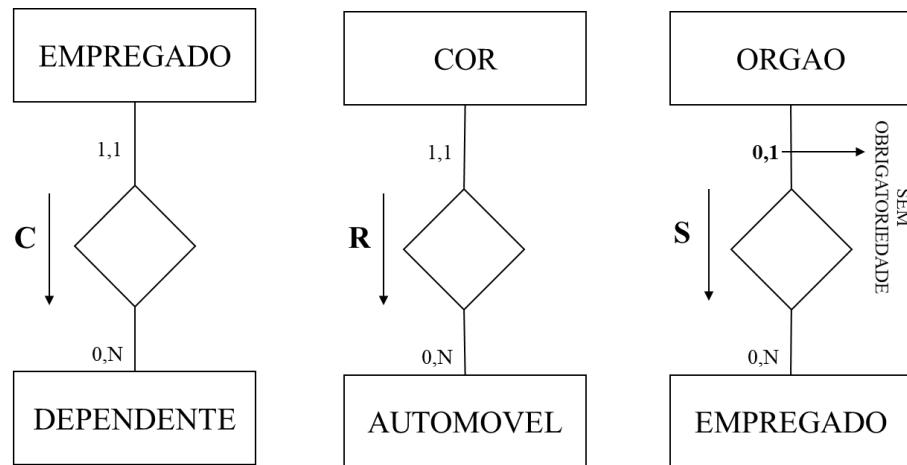
Fonte: Gustavo Aguilar (2002).

- **Restrições de integridade:** regra que garante a consistência dos dados em um relacionamento entre duas tabelas. São de **três tipos:**

- **Cascade (C):** ao deletar um registro, deleta o registro da chave estrangeira também (deleção em cascata).
- **Restrict (R):** não deleta um registro que seja chave estrangeira.
- **Set Null (S):** deleta registro e seta chave estrangeira para nulo.

A representação da restrição de integridade é sempre feita no sentido tabela com a chave primária → tabela com a chave estrangeira, como mostrado nos exemplos abaixo.

Figura 6 – Exemplos Restrições de Integridade.



Fonte: Gustavo Aguilar (2002).

Para definir corretamente as restrições de integridade, pode-se usar o algoritmo a seguir como regra geral:

Posso eliminar chave primária (registro pai)?

Se não puder → RESTRICT

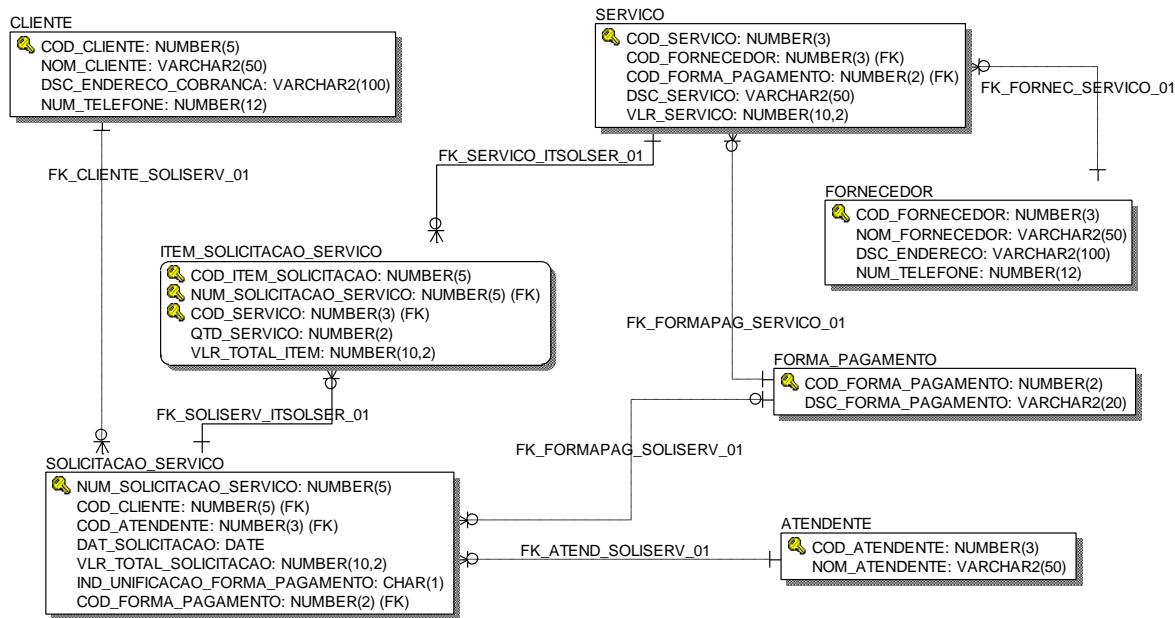
Se puder:

Preciso manter os Filhos (registros na tabela da FK)?

Se precisar → SET NULL

Exemplo de modelo físico de dados para o sistema de uma agência de turismo, a ser implementado no SGBD Oracle:

Figura 7 – Exemplo de Modelo Físico de Dados para Oracle.



Fonte: Gustavo Aguilar (2002).

Além de todo esse trabalho espetacular de lançamento do modelo relacional, no ano de 1981, Codd publicou um conjunto de 12 regras, as famosas **12 Regras de Codd** (disponível em: <https://computing.derby.ac.uk/c/codds-twelve-rules>), com o propósito de definir o que seria necessário para um sistema de gerenciamento de bancos de dados ser considerado relacional. Codd publicou essas regras como parte de uma empreitada pessoal, na tentativa de impedir que sua visão e seu trabalho acerca de bancos de dados relacionais fosse deturpada ou diluída, dados os constantes esforços de fornecedores de bancos de dados em ajustar ou redesenvolver os produtos existentes na época (SGBDs hierárquicos, de rede etc.) com um viés relacional.

Essas regras são numeradas de 1 à 12, mas Codd definiu também uma regra base, na qual todas as doze regras foram baseadas, **a Regra 0:** “*Para qualquer sistema que seja anunciado como ou reivindicado a ser um sistema de gerenciamento de banco de dados relacional, esse sistema deve ser capaz de gerenciar bancos de dados inteiramente por meio de suas capacidades relacionais.*”

- **Regra 1: A Regra da Informação.**

“*Todas as informações em uma base de dados relacional são representadas explicitamente no nível lógico e por apenas uma maneira: valores em tabelas.*”

Com essa regra, Codd quis garantir que até as informações de catálogo, como nomes das tabelas e colunas, e objetos de sistema, estivessem armazenados em tabelas. Além disso, procurou tornar a tarefa do administrador de manter a base de dados em um estado de integridade geral mais simples e mais eficaz.

- **Regra 2: Regra de Garantia de Acesso.**

“*Todo e qualquer dado (valor atômico) em uma base de dados relacional tem a garantia de ser logicamente acessível, recorrendo a uma combinação do nome da tabela, valor da chave primária e nome da coluna.*”

Com essa regra, Codd definitivamente deixa para trás as necessidades dos SGBDs não relacionais da época, onde havia-se a necessidade de saber a forma

como os dados estavam armazenados internamente, ou seja, do usual endereçamento orientado à computador.

- **Regra 3: Tratamento Sistemático de Valores Nulos.**

“Valores nulos (distintos da cadeia de caracteres vazia ou uma cadeia de caracteres em branco e distintos de zero ou qualquer outro número) são suportados no sistema gerenciador de banco de dados totalmente relacional, para representar informações ausentes e informações inaplicáveis de maneira sistemática, independente do tipo de dados.”

Técnicas anteriores exigiam a definição de um valor especial (peculiar a cada coluna ou campo), para representar informações ausentes. Codd quis eliminar essa premissa, pois acreditava que diminuiria a produtividade do usuário.

- **Regra 4: Catálogo Online e Dinâmico Baseado no Modelo Relacional.**

“A descrição da base de dados (dicionário de dados – grifo nosso) é representada no nível lógico da mesma forma que os dados ordinários, de modo que os usuários autorizados possam aplicar a mesma linguagem relacional à sua query que se aplica aos dados regulares.”

- **Regra 5: Regra da Linguagem de Dados Compreensiva.**

“Um sistema relacional pode suportar várias linguagens e vários modos de uso de terminal. No entanto, deve haver pelo menos uma linguagem cujas declarações sejam expressas por alguma sintaxe bem definida, como cadeias de caracteres e abrangente em suporte a todos os itens a seguir:

- Definição de dados e de visões.
- Manipulação de dados (interativo e por programa).
- Restrições de integridade.
- Autorização.

- Controle de transação (begin, commit e rollback)."

- **Regra 6: Regra da Atualização de Visões.**

"Todas as visualizações que são teoricamente atualizáveis, também são atualizáveis pelo sistema."

Com essa regra, Codd alerta para o tratamento das atualizações das visões de dados (views).

- **Regra 7: Inserções, Atualizações e Deleções de Alto Nível.**

"A capacidade de lidar com uma relação base ou uma relação derivada como um único operando, aplica-se não apenas à recuperação de dados, mas também à inserção, atualização e exclusão de dados."

Nesse requerimento, Codd procura proporcionar aos sistemas gerenciadores de bancos de dados um escopo maior, para otimizar as ações (tarefas) em tempo de execução e, ao mesmo tempo, fornecer uma experiência mais rápida para o usuário.

- **Regra 8: Independência Física dos Dados.**

"Os programas de aplicativos e as atividades do terminal permanecem logicamente inalterados sempre que forem feitas alterações nas representações de armazenamento, ou nos métodos de acesso."

- **Regra 9: Independência Lógica dos Dados.**

"Os programas de aplicativos e atividades de terminal permanecem logicamente inalterados quando mudanças que preservem a informação são feitas às essas tabelas."

Essa regra garante que os comandos de manipulação dos dados permaneçam inalterados quando de alterações de estruturas das tabelas, desde que os dados originais das mesmas sejam preservados.

- **Regra 10: Independência de Integridade.**

“Restrições de integridade específicas de uma determinada base de dados relacionais devem ser definidas na linguagem do banco de dados e armazenadas no catálogo, não nos programas de aplicação.”

Com essa regra, Codd procura simplificar o desenvolvimento, deixando a cargo do banco de dados a garantia da integridade dos dados.

- **Regra 11: Independência de Distribuição.**

“Um sistema gerenciador de banco de dados relacional tem independência de distribuição.”

Por independência de distribuição, Codd quis dizer que o SGBD relacional precisa ter uma linguagem que permita aos programas ficarem inalterados quando uma distribuição de dados for introduzida no banco de dados ou quando os dados forem redistribuídos.

- **Regra 12: Regra de Não Subversão.**

“Se um sistema relacional tiver uma linguagem de baixo nível, esse nível não poderá ser usado para subverter ou ignorar as regras de integridade e restrições expressas na linguagem relacional de nível superior.”

Essa regra expressa a preocupação de Codd com os fornecedores de SGBDs que estavam recodificando seus produtos como relacionais, mas que haviam nascido com suporte a uma interface abaixo da interface de restrição relacional, o que torna muito fácil burlar várias regras colocadas por ele, principalmente as de restrição de integridade. Com essa subversão de baixo nível, o banco de dados poderia passar para um status não íntegro, como registros filhos órfãos, mesmo havendo as restrições de integridade definidas nas tabelas do banco de dados.

Olhando para essas regras, e, em especial, para a **regra de número 5 (Linguagem de Dados Compreensiva)**, fica clara a preocupação de Codd para a

“exigência” específica da existência de uma **linguagem de alto nível** que abstraísse os detalhes internos da *engine* do banco de dados e do “bit-a-bit” do armazenamento dos dados. Através dessa linguagem, os usuários de bancos de dados relacionais conseguiram, de forma fácil e padronizada, criar suas estruturas de dados e manipular os dados armazenados nas mesmas. Estava sedimentado, dessa forma, um dos pilares responsáveis para o grande sucesso e uso em larga escala da **Linguagem SQL!**

1.2. História da linguagem SQL

Quando Codd publicou a teoria do Modelo de Dados Relacional, em 1970, ele era um pesquisador no laboratório da IBM, em San José. Entretanto, com o intuito de preservar o faturamento gerado pelos produtos “pré-relacionais”, por exemplo, o SGBD hierárquico IMS, a IBM optou inicialmente por não implementar as ideias relacionais de Codd.

Descontente com essa situação e acreditando no salto evolutivo que o modelo de dados relacional poderia trazer para os sistemas gerenciadores de bancos de dados e para a história da computação, Codd procurou grandes clientes da IBM para mostrar-lhes as novas potencialidades e benefícios da implementação do modelo relacional. A pressão desses clientes sobre a IBM forçou-a a incluir, no seu **projeto Future Systems (FS)**, em andamento no Centro de Pesquisas de Almaden (*Almaden Research Center*, na época *San Jose Research*), em San José, Califórnia, o **subprojeto System R**.

O projeto *Future Systems* foi um projeto de pesquisa e desenvolvimento realizado na IBM no início dos anos 70, com o objetivo de desenvolver uma linha revolucionária de produtos de informática, incluindo novos modelos de software que simplificariam o desenvolvimento de software, explorando hardwares poderosos e modernos. Em 1974, então, dentro do projeto *Future System*, iniciava-se a construção do **System R**, um sistema de banco de dados experimental com o propósito de demonstrar que as vantagens de usabilidade do modelo de dados relacional poderiam

ser implementadas em um sistema com suas funcionalidades completas e com alto desempenho, quesitos altamente requeridos no uso cotidiano de ambientes de produção.

Um dos resultados mais expressivos do projeto *System R* foi a criação da linguagem para consulta e manipulação de dados chamada inicialmente de **SEQUEL** (pronunciada como "síquel"), um acrônimo para **Structured English Query Language** (Linguagem de Consulta Estruturada em inglês). Os progenitores dessa linguagem, **Donald Chamberlin** e **Ray Boyce**, a partir do modelo estabelecido por Codd em seu artigo “*A Relational Model of Data for Large Shared Data Banks*”, desenvolveram e lançaram a linguagem de programação em seu próprio artigo **SEQUEL: A Structured English Query Language**. Eles pegaram as linguagens de Codd com o objetivo de projetar uma linguagem relacional que fosse mais acessível aos usuários, sem um treinamento formal em matemática ou ciência da computação.

Essa versão original da linguagem, que continha um conjunto de comandos que permitiam definir a estrutura de dados, inserir, atualizar, excluir e manipular dados, foi usada inicialmente nos sistemas de bancos de dados relacionais originais da IBM, os “*System R*”.

Entretanto, como **SEQUEL** era uma marca registrada pela empresa de aeronaves Hawker Siddeley, a linguagem **SEQUEL**, através da eliminação das vogais, foi renomeada para **SQL (Structured Query Language)** - Linguagem de Consulta Estruturada). Devido a isso, até hoje ouvimos comumente a sigla, em inglês, ser pronunciada como "síquel" (SEQUEL), ao invés de "és-kiú-él" (SQL), letra a letra. No entanto, em português, a pronúncia mais comum é letra a letra: S ("esse") - Q ("quê") - L ("éle").

Nos anos seguintes, a SQL não se tornou disponível publicamente, como vemos nos dias de hoje. No entanto, em 1979, a *Oracle*, então conhecida como *Relational Software*, criou sua própria versão da linguagem SQL chamada *Oracle V2*, que foi lançada comercialmente. É importante observar que a SQL não foi a primeira linguagem de programação para bancos de dados, mas o crescimento em larga

escala da sua utilização foi em grande parte devido à sua intuição e facilidade de uso. As linguagens de consulta, até então, eram na maioria **linguagens procedurais**, o que requeria que o usuário especificasse o caminho (navegação, posição etc.) para se chegar ao resultado, como mostrado no pseudocódigo abaixo:

Repita para cada linha de dados:

Se a cidade for São Paulo, retorno a linha; caso contrário, não faça nada.

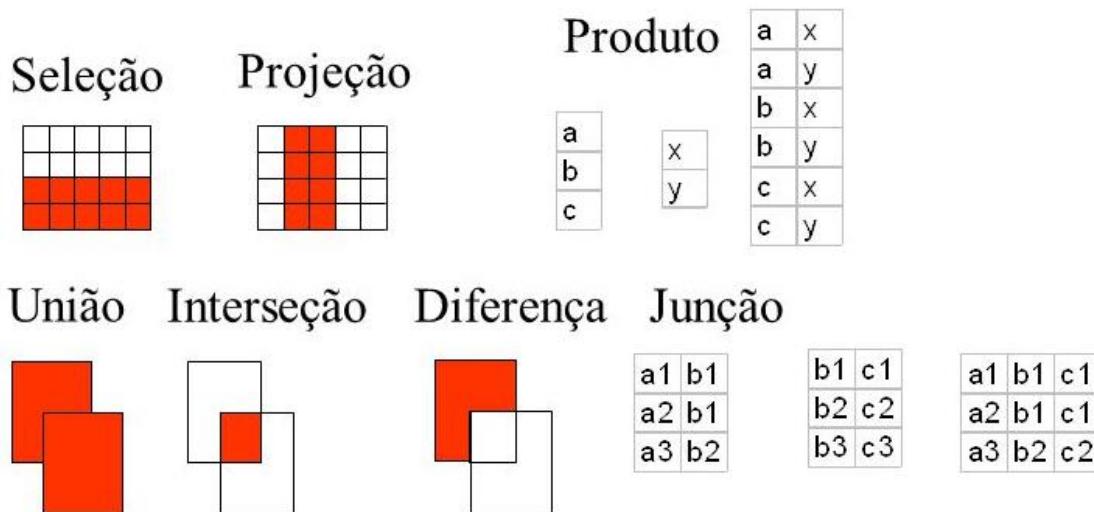
Mova para a próxima linha.

Já a linguagem SQL, sendo uma **linguagem declarativa**, requeria apenas especificar a forma (conteúdo desejado) do resultado e não o caminho para se chegar a ele, como mostrado no pseudocódigo abaixo:

Retorne todos os clientes cuja cidade é São Paulo.

De certa forma, essa peculiaridade reduziu o ciclo de aprendizado dos desenvolvedores e administradores de bancos de dados, fazendo com que a Linguagem SQL caísse no “gosto” rapidamente da comunidade.

Outro fator que contribuiu para a popularização da linguagem SQL, no sentido de prover respaldo e coerência, foi a grande confiabilidade pelo fato de Codd ter se baseado na **Álgebra Relacional**, e consequentemente na **Teoria de Conjuntos**, para sustentar sua fundamentação teórica acerca dos modelos de dados relacionais. Utilizando-se das já conhecidas operações da Álgebra Relacional, como **seleção**, **projeção e junção**, e das operações da teoria de conjuntos, como **união**, **interseção**, **diferença** e **produto cartesiano**, Codd quase levou a comunidade ao “êxtase” ao propor uma forma mais abstrata e natural para o armazenamento dos dados, e uma maneira de manipulá-los mais próxima das relações humanas baseadas em relacionamentos e correlações.

Figura 8 – Operações da Álgebra Relacional.

Fonte: Marta Mattoso.

Usando-se dos operadores de Álgebra Relacional, Codd conseguiu fundamentar a viabilidade e usabilidade do modelo relacional e seu armazenamento de dados no formato tabular, em tuplas. A título informacional, pois não é o objetivo dessa disciplina aprofundar em álgebra relacional, os operadores usados por Codd em sua teoria estão relacionados na figura abaixo, e, a seguir, alguns exemplos de utilização deles.

Figura 9 – Operações/Operadores em Álgebra Relacional.

OPERAÇÃO	SÍMBOLO	SINTAXE
Projeção	π ("pi")	π <lista de campos> (Tabela)
Seleção/ Restrição	σ ("sigma")	σ <condição de seleção> (Tabela)
União	\cup	(Tabela 1) \cup (Tabela 2)
Interseção	\cap	(Tabela 1) \cap (Tabela 2)
Diferença	-	(Tabela 1) - (Tabela 2)
Produto Cartesiano	X	(Tabela 1) X (Tabela 2)
Junção	$ X $	(Tabela 1) $ X $ <condição de junção> (Tabela 2)
Divisão	\div	(Tabela 1) \div (Tabela 2)
Renomeação	ρ ("rho")	ρ Nome(Tabela)
Atribuição	\leftarrow	Variável \leftarrow Tabela

Fonte: Júnior Galvão (2016).

- **Restrição (σ):** retornar somente as tuplas que satisfaçam a uma condição.

Exemplo: retornar os dados dos empregados que estão com salário menor que 2.000.

Figura 10 – Seleção na Álgebra Relacional.

$$\sigma_{\text{salario} < 2000}(\text{Empregado})$$

Empregado

codEmp	Nome	Salario	idade	codDep
200	Pedro	3.000,00	45	001
201	Paulo	2.200,00	43	001
202	Maria	2.500,00	38	001
203	Ana	1.800,00	25	002

Resultado

codEmp	Nome	Salario	idade	codDep
203	Ana	1.800,00	25	002

Fonte: Vania Bogorny.

- **Projeção (π):** retornar um ou mais atributos desejados. Exemplo: retornar o nome e a idade dos empregados.

Figura 11 – Projeção na Álgebra Relacional.

$$\pi_{\text{nome, idade}}(\text{Empregado})$$

Empregado

codEmp	Nome	Salario	idade	codDep
200	Pedro	3.000,00	45	001
201	Paulo	2.200,00	43	001
202	Maria	2.500,00	38	001
203	Ana	1.800,00	25	002

Resultado

Nome	idade
Pedro	45
Paulo	43
Maria	38
Ana	25

Fonte: Vania Bogorny.

- **Interseção (\cap)**: retornar tuplas comuns entre duas entidades (tabelas).

Exemplo: retornar o nome e o CPF dos funcionários que estão internados como pacientes.

$$\Pi_{\text{nome,CPF}} (\text{FUNCIONARIO}) \cap \Pi_{\text{nome,CPF}} (\text{PACIENTE})$$

- **União (\cup)**: retornar a união das tuplas de duas entidades (tabelas). Exemplo: retornar o nome e o CPF dos médicos e pacientes cadastrados no hospital.

$$\Pi_{\text{nome,CPF}} (\text{MEDICO}) \cup \Pi_{\text{nome,CPF}} (\text{PACIENTE})$$

- **Diferença ($-$)**: retornar um conjunto de tuplas, que é o conjunto de tuplas da primeira entidade, menos as tuplas existentes na segunda entidade. Exemplo: retornar o código dos ambulatórios onde nenhum médico dá atendimento.

$$\Pi_{\text{codigoA}} (\text{AMBULATORIO}) - \Pi_{\text{codigoA}} (\text{MEDICO})$$

- **Produto Cartesiano (\times)**: retornar todas as combinações possíveis de tuplas de duas entidades. Exemplo: retornar o nome dos médicos e os estados onde eles podem trabalhar.

$$\Pi_{\text{medico,uf}} (\text{MEDICO} \times \text{ESTADO})$$

No entanto, o sucesso prolongado da SQL não pode ser atribuído apenas às suas qualidades. A ajuda dos padrões, assunto que trataremos no próximo capítulo, não apenas ajudou a SQL a se aproximar de um uso quase que universal na comunidade de banco de dados, mas também acrescentou atributos-chave que floresceram em suas especificações posteriores. Tudo isso começou quando o **American National Standards Institute (ANSI)**, que é o Instituto Americano Nacional de Padrões, se envolveu.

1.3. Padronização da linguagem SQL, classes de instruções SQL e dialetos

O *American National Standards Institute* (ANSI) é uma organização particular estadunidense, sem fins lucrativos, que atua há mais de 100 anos como administradora e coordenadora do **sistema norte-americano de normas e conformidades voluntários**. Fundado em 1918 por cinco sociedades de engenharia e três agências governamentais, o Instituto continua sendo uma organização privada sem fins lucrativos, apoiada por um grupo diversificado de organizações do setor público e privado.



Ao longo de sua história, o ANSI manteve como meta principal a melhoria da competitividade global das empresas americanas e a qualidade de vida nos Estados Unidos, promovendo e facilitando padrões consensuais voluntários e sistemas de avaliação de conformidade, promovendo sua integridade. Um desses padrões, talvez o mais conhecido mundialmente pela comunidade da computação, é o **padrão publicado para a Linguagem SQL**.

Embora a SQL tenha sido originalmente criada pela IBM, após a sua publicação rapidamente surgiram várias linguagens desenvolvidas pelos fornecedores de SGBDs relacionais, cada uma com suas adaptações à Linguagem SQL original. Por exemplo, podemos citar o dialeto PL/SQL da Oracle. Essa expansão levou à necessidade de ser criado e adaptado um padrão para a Linguagem SQL. Essa tarefa foi realizada, pioneiramente, pela *American National Standards Institute*, em 1986, com a publicação do padrão **ANSI X3.135-1986**. Este foi o começo do que as pessoas chamam erroneamente de padrão ANSI para SQL. Na verdade, não existem padrões ANSI, apenas padrões desenvolvidos por comitês aprovados pela ANSI, muitos deles operando de acordo com os Requisitos Essenciais da ANSI.

Após alguns meses, ao lançamento do ANSI X3.135-1986, a **International Organization for Standardization (ISO)** publicou um padrão tecnicamente idêntico, o **ISO 9075-1987**, para o cenário internacional. Na época da publicação inicial desses padrões, especificações mais detalhadas sobre a linguagem SQL se faziam

necessárias, mas o ANSI X3.135-1986 ajudou a definir as bases para alguns recursos importantes para a linguagem de codificação. Esse padrão, por exemplo, deu a capacidade de invocar recursos de SQL de quatro linguagens de programação: COBOL, FORTRAN, Pascal e PL / I.

Esse padrões foram revisados em conjunto, primeiramente em **1989 (ANSI X3.135-1989 e ISO / IEC 9075: 1989)** e novamente em **1992 (ANSI X3.135-1992 e ISO / IEC 9075: 1992)**. A edição de 1989 incluía suporte para duas linguagens de programação adicionais, Ada e C. Essas edições se tornaram coloquialmente conhecidas como **SQL-86, SQL-89 e SQL-92**. Portanto, se você ouvir esses nomes em referência a um formato SQL, lembre-se que ele está se referindo às várias edições desse padrão.

O padrão foi revisado novamente em **1999 (SQL3), 2003, 2008, 2011 e 2016**, que continua sendo a edição atual (**ISO / IEC 9075: 2016**). A norma internacional ISO / IEC 9075 para SQL é desenvolvida pelo **ISO/IEC Joint Technical Committee (JTC) 1 for Information Technology** e desde a edição de 2003 foi subdividida em nove partes, cada uma cobrindo um aspecto do padrão geral, sob o título **Tecnologia da Informação – Linguagens de Banco de Dados – SQL**.

Mesmo com todos esses esforços e publicações para se ter um padrão único da Linguagem SQL, os fornecedores de SGBDs continuaram fazendo suas próprias implementações baseadas nos padrões da Linguagem SQL, mas com pequenas adaptações ou personalizações. Dentre as mais conhecidas, além da já citada PL/SQL da Oracle, temos também a **T-SQL da Microsoft** (usada no **SQL Server**) e a **SQL PL da IBM**.

Devido a isso, é comum encontrar algumas pequenas diferenças na sintaxe de implementação para um mesmo tipo de comando SQL nos diversos sistemas gerenciadores de bancos de dados relacionais (comumente referidos pela comunidade como **dialetos**). Para exemplificar, tomemos como base uma simples query SQL para retornar os N registros de uma tabela com a cláusula TOP:

- Sintaxe de uso no SGBD (dialeto) **SQL Server**:

SELECT TOP [QTDE DE REGISTROS] [COLUNA(S)] FROM [TABELA]

- Sintaxe de uso nos SGBDs (dialeto) **MySQL** e **PostgreSQL**:

SELECT [COLUNA(S)] FROM [TABELA] LIMIT [QTDE DE REGISTROS]

- Sintaxe de uso no SGBD (dialeto) **Oracle**:

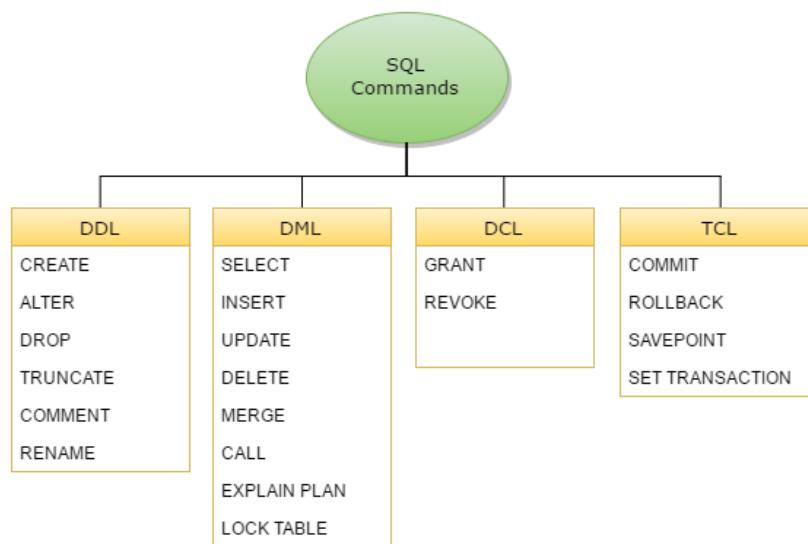
SELECT [COLUNA(S)] FROM [TABELA] WHERE ROWNUM < [QTDE REGISTROS]

- Sintaxe de uso no SGBD (dialeto) **Firebird**:

SELECT FIRST [QTDE DE REGISTROS] [COLUNA(S)] FROM [TABELA]

Dentro do padrão ISO SQL, encontramos **quatro classes (conjuntos) de comandos da Linguagem SQL**, cada uma delas com um propósito bem definido na etapa de construção do modelo de dados físico do banco de dados:

- **Linguagem de Definição de Dados** (DDL – *Data Definition Language*): comandos para a criação do banco de dados e dos objetos no banco de dados, como tabelas, índices, constraints etc., e para alteração e exclusão de objetos.
- **Linguagem de Manipulação de Dados** (DML – *Data Manipulation Language*): comandos para inserir, atualizar, deletar e consultar dados.
- **Linguagem de Controle de Dados** (DCL – *Data Control Language*): comandos para gerenciar permissões de acesso para usuários e objetos, ou seja, permissões para executar comandos DDL, DML, DCL e TCL.
- **Linguagem de Controle de Transação** (TCL – *Transaction Control Language*): comandos para realizar controle das transações de dados no banco de dados.

Figura 12 – Classes de Comandos SQL.

Fonte: [www.learnhowtocode.info.](http://www.learnhowtocode.info/)

1.4. Uma “Sopa de Letras”

Siglas sempre fizeram e farão parte do cotidiano de um profissional de TI, principalmente daqueles que se propõem a trabalhar com banco de dados. Algumas delas já foram apresentadas nos capítulos anteriores, mas se faz necessário apresentar algumas novas e reforçar na diferenciação de algumas delas.

Primeiramente, é preciso estar bem clara a diferenciação entre **a SQL** e **o SQL**. Quando se diz “a SQL”, está se fazendo menção à **Linguagem SQL**. Já no segundo caso, quando se diz “o SQL”, está se fazendo menção ao **SQL Server**, sistema gerenciador de banco de dados relacional desenvolvido pela Microsoft. Adicionalmente, encontramos também o nome de outros SGBDs fazendo menção à Linguagem SQL, de forma a indicar que são relacionais: **MySQL** fornecido pela Oracle (www.mysql.com), **PostgreSQL** desenvolvido pela PostgreSQL Global Development Group e de código aberto (www.postgresql.org), **SQLite** desenvolvido por Richard Hipp, sendo um banco de dados portátil e de código aberto (www.sqlite.org) etc.

Outra sigla, uma das mais faladas atualmente, “NOSQL”, pode se referenciar a duas tecnologias diferentes, mas nenhuma delas relacionadas diretamente à Linguagem SQL. A diferença entre essas duas tecnologias é feita na grafia: **NoSQL** versus **NOSQL**.

O termo **NoSQL** surgiu em 1998, quando **Carlo Strozzi** usou-o para batizar um banco de dados relacional que havia desenvolvido e que não fornecia nenhuma forma da linguagem SQL para consulta. Seu nome era pronunciado como “*nosequel*”, e ele era relacional, rápido, portátil, de código aberto que executava e interagia com o sistema operacional *UNIX*. Para manipulação dos dados, ele disponibilizava “operadores” que executavam uma única função, e o fluxo de dados era fornecido pelo próprio mecanismo de redirecionamento de entrada/saída do *UNIX*. Dessa forma, cada operador processava alguns dados e os transmitia ao próximo operador por meio da função *pipe* do *UNIX*.

Os dados do *NoSQL* eram armazenados em arquivos *ASCII UNIX* e, portanto, podiam ser manipulados por utilitários *UNIX* comuns, como *ls*, *mv*, *cp* e editores como o ‘vi’. Essa forma alternativa para manipulação dos dados, **sem a Linguagem SQL**, foi o motivador do nome dado para esse banco, *NoSQL* (“*non SQL*”), no sentido de realmente não ser um banco de dados SQL. Entretanto, pelo fato do formato dos arquivos ser baseado em uma relação ou tabela, com linhas e colunas, era considerado um SGBD relacional.

Figura 13 – Logo do NoSQL de Carlo Strozzi.



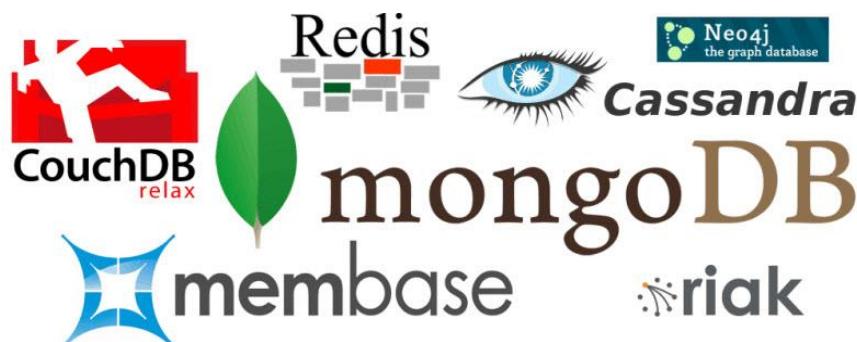
Fonte: Kyle Hart (1998).

Strozzi afirmava categoricamente que seu produto “nada tinha a ver com o recém-nascido **Movimento NOSQL**”, e que “enquanto o primeiro é um pacote de software bem definido, é um banco de dados relacional para todos os efeitos e apenas

intencionalmente não usa SQL como uma linguagem de consulta; o recém-chegado (NOSQL) é principalmente um conceito (e de nenhuma maneira novo) que se afasta do modelo relacional e, portanto, deveria ter sido chamado mais apropriadamente de ‘NoREL’, ou algo nesse sentido, já que não será baseado em SQL e é apenas uma consequência óbvia de não ser relacional, e não o contrário”.

Apesar de toda a ênfase de Strozzi, Johan Oskarsson, então um desenvolvedor na *Last.fm*, reintroduziu o termo *NOSQL*, no início de 2009, quando organizou um evento para discutir “bancos de dados distribuídos, não relacionais e de código aberto”. O termo tentava rotular o surgimento de um número crescente de repositórios de dados distribuídos, não relacionais, incluindo clones de código aberto do *Google Bigtable/MapReduce* e do *DynamoDB* da *Amazon*. Desde então, o termo *NOSQL* tem sido atribuído aos **sistemas gerenciadores de bancos de dados não relacionais** (e não somente mais não SQL), que fornecem um mecanismo de armazenamento de dados que não é baseado no formato tabular (linhas x colunas).

Figura 14 – Logo dos SGBDs NOSQL mais usados.



Fonte: Arte dos Dados (2013).

Nessa abordagem, o termo *NoSQL* passou a ser escrito como **NOSQL** (reparem a grafia maiúscula da letra ‘o’), para fazer menção aos SGBDs não relacionais, **“Não Apenas SQL”** (**‘Not Only SQL’**), e objetivando enfatizar que esses SGBDs podiam suportar outras linguagens de consulta, que não a SQL.

Figura 15 – Logo do Movimento NOSQL.

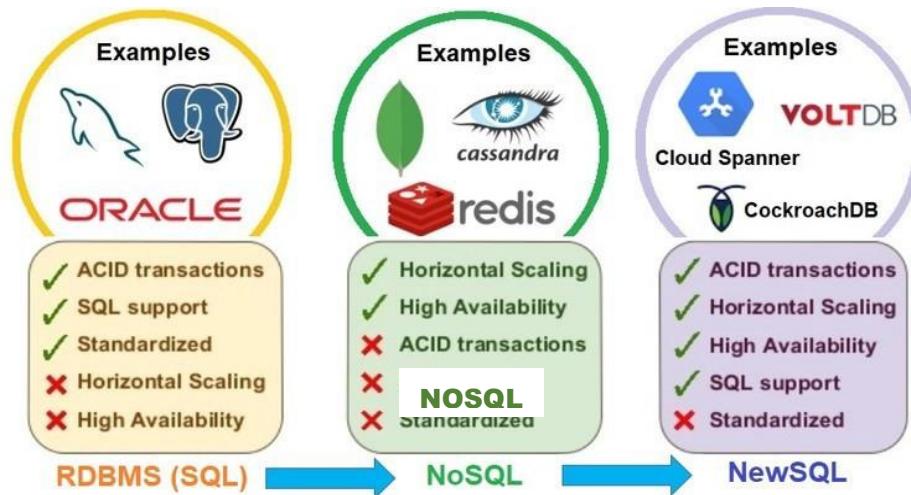


Fonte: Ylanite (2009).

Outra sigla que precisa ser diferenciada é a **UnSQL**, criada em 29 de julho de 2011, quando o fundador da Couchbase e inventor do sistema gerenciador de banco de dados CouchDB (NOSQL), Damien Katz, juntamente com Richard Hipp, inventor do SQLite, publicaram um artigo ('UNQL QUERY LANGUAGE UNVEILED BY COUCHBASE AND SQLITE', disponível em: <https://www.couchbase.com/press-releases/unql-query-language>) para anunciar o trabalho em conjunto acerca da criação de uma linguagem de consulta padrão NOSQL, a **UnQL** (*Unstructured Query Language*). Com pronúncia “Uncle”, o objetivo era que essa linguagem se popularizasse da mesma forma que a linguagem SQL, do movimento relacional, havia se popularizado. Foi criado até um site para o projeto (<http://unql.sqlite.org/index.html/wiki?name=UnQL>), mas até o momento ela não se popularizou e não virou um padrão ISO como a SQL.

Por fim, é preciso mencionar a sigla **NewSQL**, que, ao contrário do que parece, não é uma nova versão da Linguagem SQL. Trata-se de uma nova geração de sistemas gerenciadores de bancos de dados relacionais distribuídos que procura fornecer o mesmo desempenho escalável de um SGBD NOSQL distribuído ao mesmo tempo que tenta garantir todas as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) de transações de dados dos SGBDs relacionais. Dentro dessa geração, encontramos novos players (Amazon Aurora, CockroachDB, CosmosDB, Google Spanner, VoltDB etc.) e SGBDs já conhecidos, mas com a engine reformulada (MySQL Cluster, SQL Server Column Store etc.).

Figura 16 – Evolução dos SGBDs Distribuídos.



Fonte: Gustavo (2018).

Capítulo 2. Introdução ao SQL Server

2.1. Introdução ao SQL Server

Para aplicarmos na prática os conceitos e instruções da Linguagem SQL que serão expostos nessa disciplina, utilizaremos um banco de dados SQL Server. O **Microsoft SQL Server** é um sistema gerenciador de banco de dados relacional, desenvolvido pela Microsoft, cuja primeira versão (1.0) foi lançada em 1989. A versão atual é a 2019 (<https://www.microsoft.com/pt-br/sql-server/sql-server-2019>) e será usada neste curso, no sistema operacional Windows.

Figura 17 – Versões do SQL Server.

Versão	Ano de Lançamento
2019	2019
2017	2017
2016	2016
2014	2014
2012	2012
2008 R2	2010
2008	2008
2005	2005
2000	2000
7.0	1998
6.5	1996
6.0	1995
4.2.1	1994
4.2	1992
1.1	1991
1.0	1989

Fonte: Gustavo (2020).

A título de conhecimento, além da engine de banco de dados relacional, na família de produtos SQL Server existem outros três produtos:

- **Microsoft Analysis Services (SSAS)**: engine analítica (OLAP) que permite a criação de cubos e dimensões.
- **Microsoft Reporting Services (SSRS)**: engine de exibição de dados, que permite a criação de relatórios e dashboards.
- **Microsoft Integration Services (SSIS)**: engine para extração, transformação e carga de dados (ETL).

2.2. Conceitos básicos do SQL Server

Em linhas gerais, a macro arquitetura desses produtos pode ser resumida como *uma instância (binária) existente em um servidor, que contém os recursos (banco de dados / pasta SSRS / catálogo SSIS), onde as estruturas (tabelas / cubos / relatórios / pacotes SSIS) para armazenamento, processamento e exibição dos dados são criadas*, como ilustrado abaixo.

Figura 18 – Macroarquitetura de Produtos da Família SQL.



Fonte: Gustavo (2017).

2.3. Instalação do SQL Server

Para instalar o SQL Server 2019, de forma que você possa reproduzir os exemplos expostos, realizar os exercícios e praticar, baixe o instalador abaixo, assista à respectiva aula deste capítulo e reproduza em seu computador.

- SQL Server 2019 Developer Edition:
 - <https://go.microsoft.com/fwlink/?linkid=866662>.

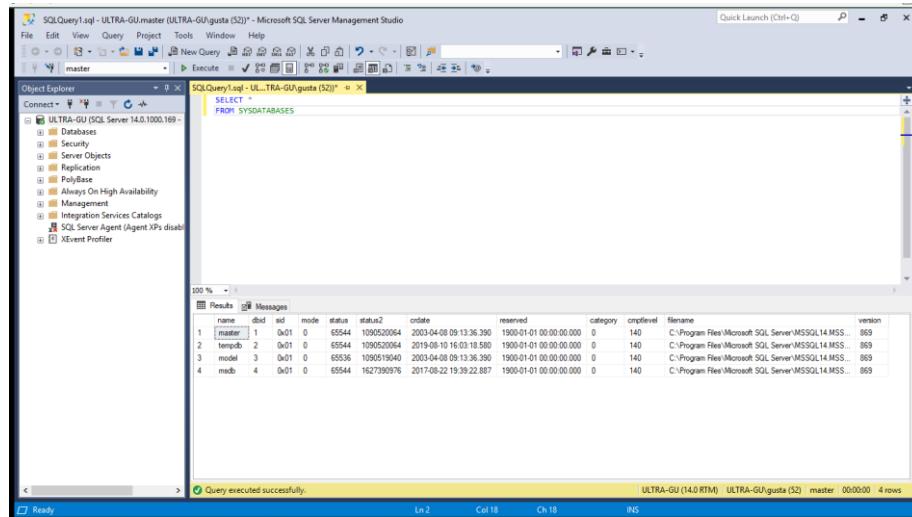
2.4. Introdução à T-SQL

A implementação da Microsoft, no SQL Server, da Linguagem SQL padrão ISO / IEC 9075, é chamada **Transact SQL (T-SQL)**. A T-SQL é uma linguagem declarativa, possui todas as opções e instruções existentes no padrão ISO, como variáveis, loop, controle de fluxo e decisões, funções etc., além de possuir algumas instruções que não estão no padrão ISO e instruções com sintaxe ligeiramente diferente do padrão.

Para executar queries SQL no SQL Server, utilizamos uma ferramenta client. Client é uma interface gráfica (GUI), que permite se conectar no banco de dados e realizar atividades, como criar bancos de dados, tabelas, usuários, bem como manipular dados e realizar tarefas administrativas.

A ferramenta client oficial da Microsoft, para o SQL Server, é o **Microsoft SQL Server Management Studio (SSMS)**, disponível para sistema operacional Windows.

Figura 19 – SQL Server Management Studio (SSMS) 18.5.

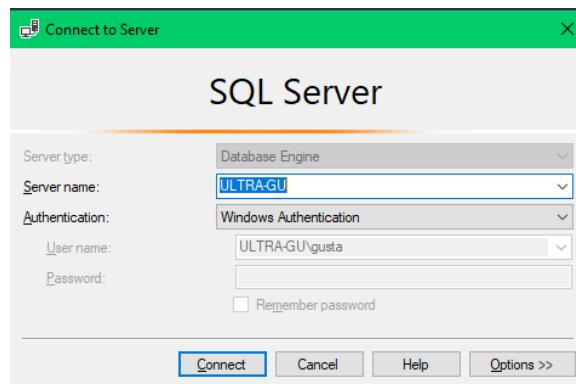


Fonte: Gustavo (2019).

Para executar os comandos T-SQL no SQL Server, primeiramente é necessário conectar-se ao banco de dados. No escopo deste curso e tendo seguido

os procedimentos de instalação exibidos nas aulas, para se conectar ao SQL Server usando o Microsoft SQL Server Management Studio, basta clicar no botão **Connect** mostrado na figura abaixo.

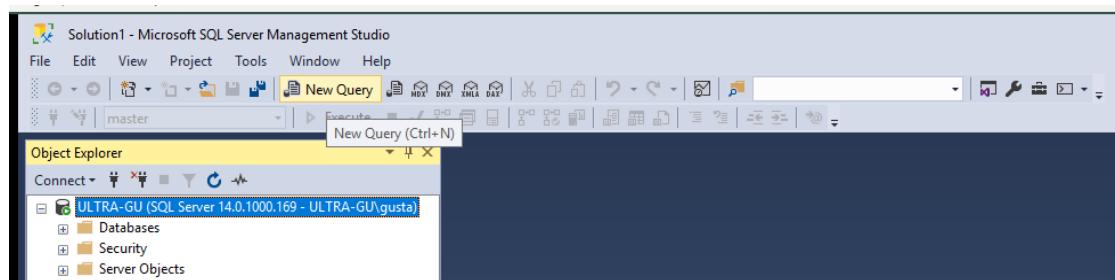
Figura 20 – Tela de Conexão do Management Studio 18.5.



Fonte: Gustavo (2019).

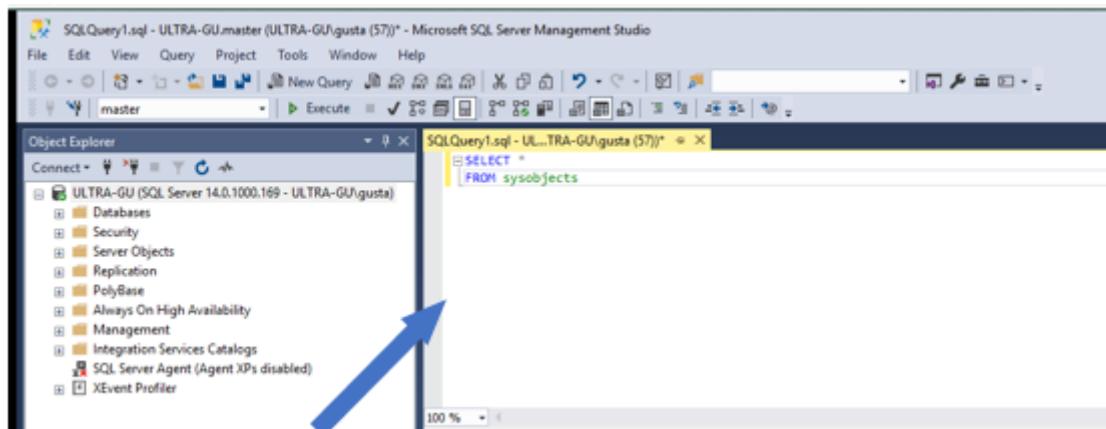
Clicando no botão **New Query** destacado abaixo, abrirá uma nova área onde é possível digitar os comandos que se deseja executar.

Figura 21 – Botão para Abrir Tela de Query no SSMS 18.5.



Fonte: Gustavo (2019).

Figura 22 – Tela para Escrever e Executar Query no SSMS 18.



Fonte: Gustavo (2019).

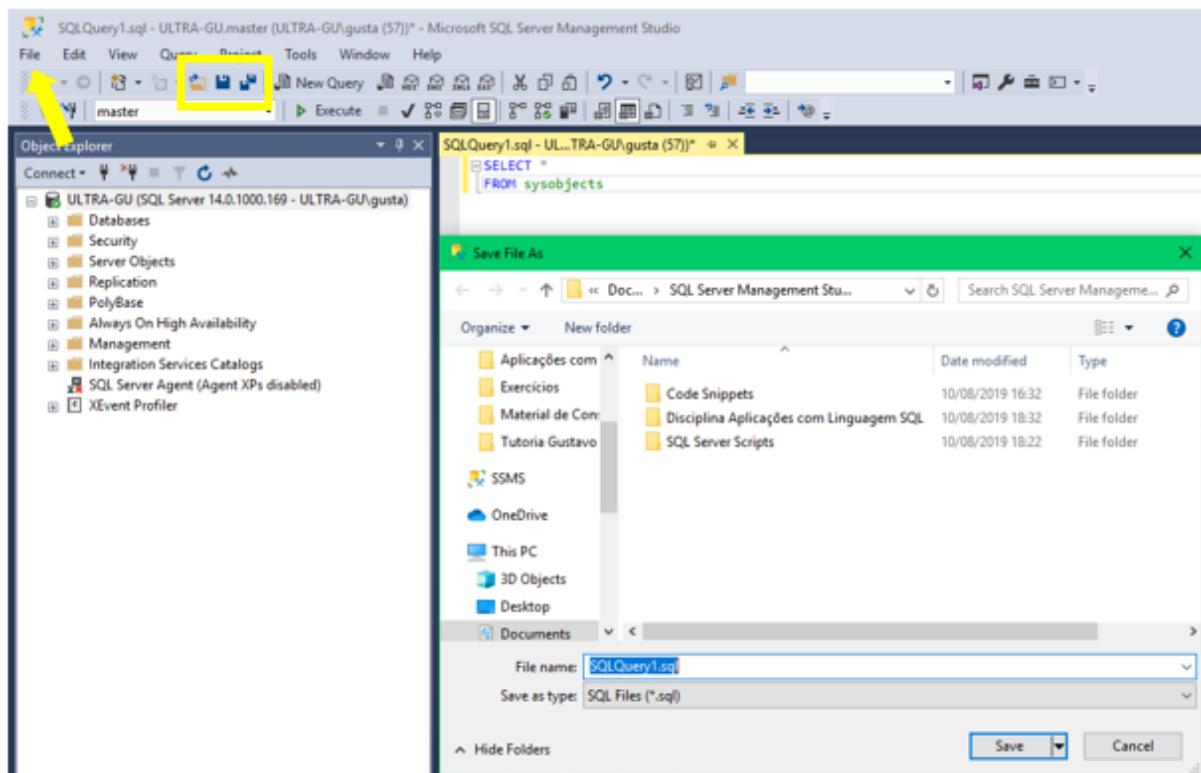
Para executar a query, basta selecionar o bloco de instruções que deseja executar (caso contrário, executará todos os comandos existentes na janela) e clicar no botão **Execute** (ou usar a tecla de atalho **F5**).

Figura 23 – Botão para Executar Query no SSMS 18.5.



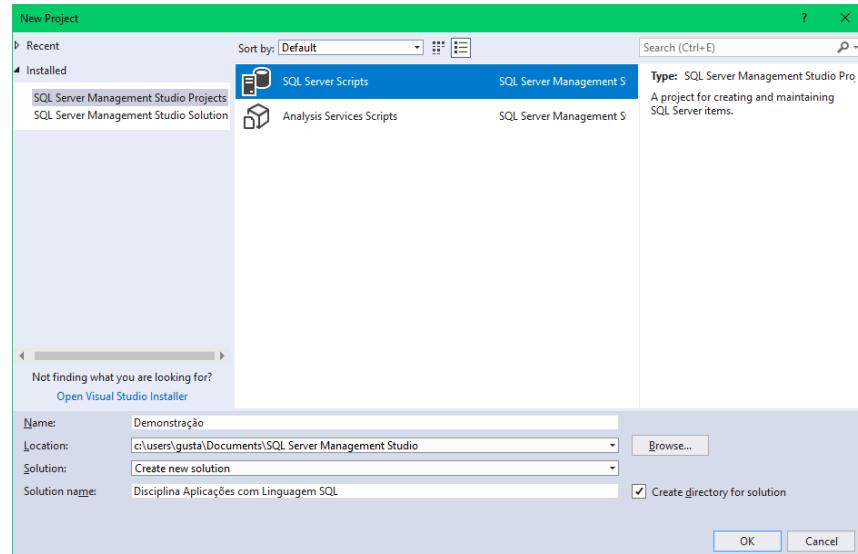
Fonte: Gustavo (2019).

Importante saber também, que os comandos T-SQL podem ser salvos em um arquivo texto, chamado **script**, normalmente com a extensão “**sql**”. No SSMS, isso é feito no menu **File**, ou nos botões de salvar da barra de ferramentas.

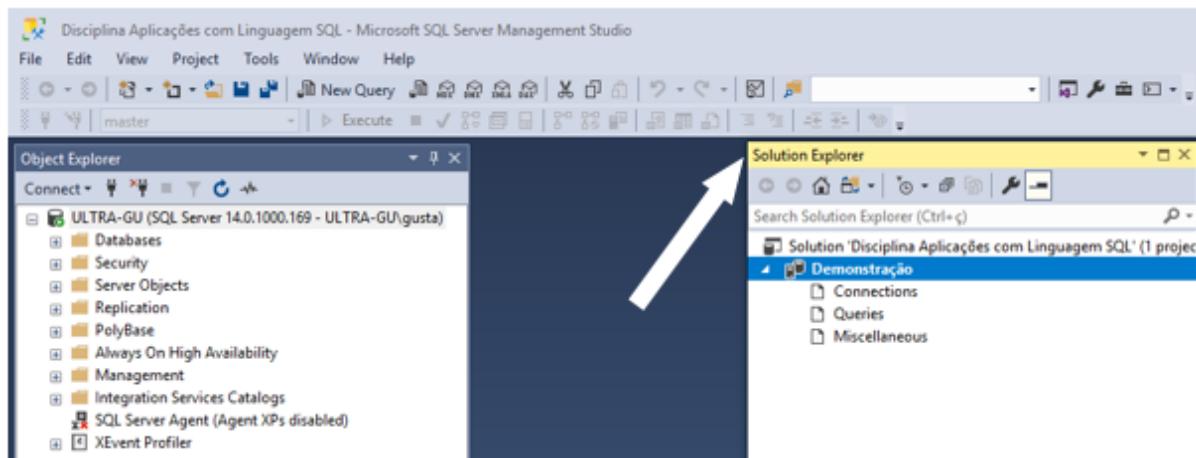
Figura 24 – Salvando um Script no SSMS 18.5.

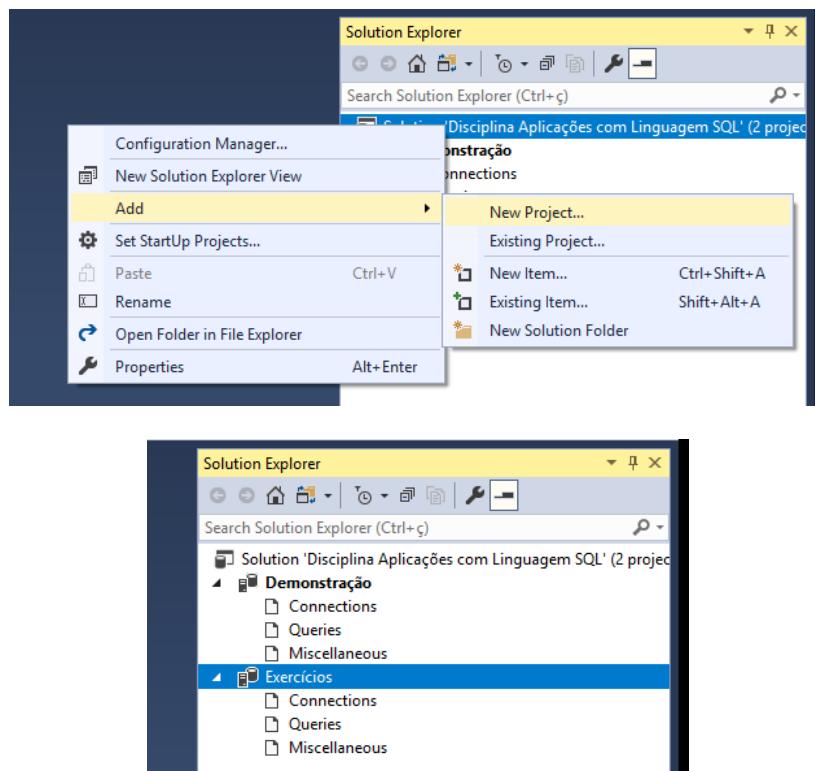
Fonte: Gustavo (2019).

Por fim, faz-se importante mencionar os recursos que o SSMS fornece para trabalhar com um projeto de banco de dados, no tocante à organização dos scripts SQL. O SSMS fornece o recurso **Solução (Solution)**, dentro do qual pode ser criado um ou mais **Projetos de Scripts SQL Server**.

Figura 25 – Criando Nova Solução e Projeto no SSMS 18.5.**Fonte: Gustavo (2019).**

Esse recurso é visualizado na janela **Solution Explorer** do SSMS.

Figura 26 – Janela Solution Explorer no SSMS 18.5.**Fonte: Gustavo (2019).**

Figuras 27 e 28 – Adicionando um Projeto à Solução.**Fonte: Gustavo (2019).**

Dentro de cada projeto, pode-se adicionar um ou mais scripts que já estejam salvos em arquivos .sql, as conexões aos bancos de dados usados por eles, ou até mesmo criar um script ou conexão, já adicionando-os ao projeto.

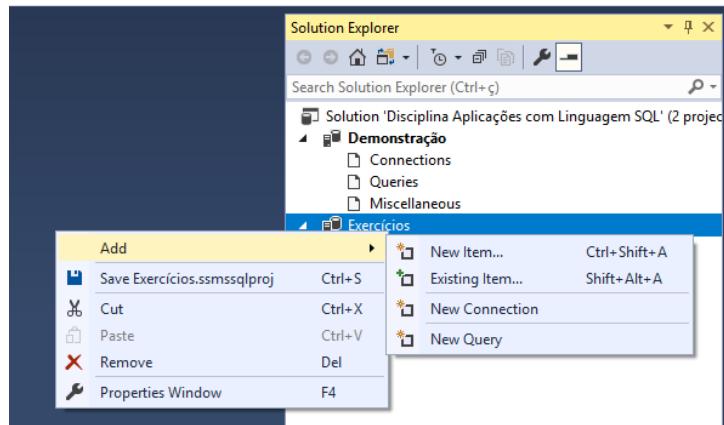
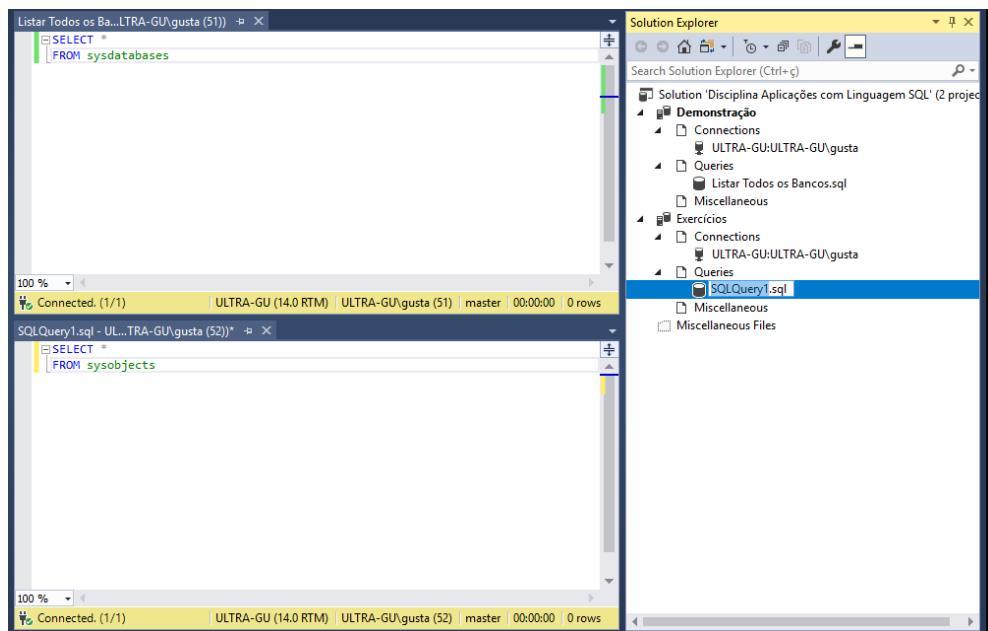
Figura 29 – Adicionando Itens ao Projeto.**Fonte: Gustavo (2019).**

Figura 30 – Exemplo de projetos e seus itens.



Fonte: Gustavo (2019).

2.5. Instalação e overview das ferramentas Client

Para instalar os clients do SQL Server 2019, de forma que você possa reproduzir os exemplos expostos, realizar os exercícios e praticar, baixe os instaladores abaixo, assista à respectiva aula deste capítulo e reproduza em seu computador.

- SQL Server Management Studio 18.5 (SSMS): <https://aka.ms/ssmsfullsetup>.
- Azure Data Studio (ADS): <https://go.microsoft.com/fwlink/?linkid=2135512>.

2.5.1. Instalação e Overview do SQL Server Management Studio

Vide aula 2.5.1 da instalação e overview do Management Studio.

2.5.2. Instalação e Overview do Azure Data Studio

Vide aula 2.5.2 da instalação e overview do Azure Data Studio.

2.6. Banco de dados de exemplo AdventureWorks

Para demonstrar os comandos da Linguagem SQL, usaremos como banco de dados de exemplo o tradicional **Adventure Works**, disponibilizado pela Microsoft em <https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks> e já populado com dados fictícios. Usaremos neste curso o banco de dados **AdventureWorks2019**, cujo link direto para download do backup desse banco é:

- <https://github.com/Microsoft/sql-server-samples/releases/download/adventureworks/AdventureWorks2019.bak>.

O modelo de dados para esse banco de dados pode ser encontrado em:

- <https://improveandrepeat.com/2019/02/use-the-adventureworks-sample-database-for-your-examples>.

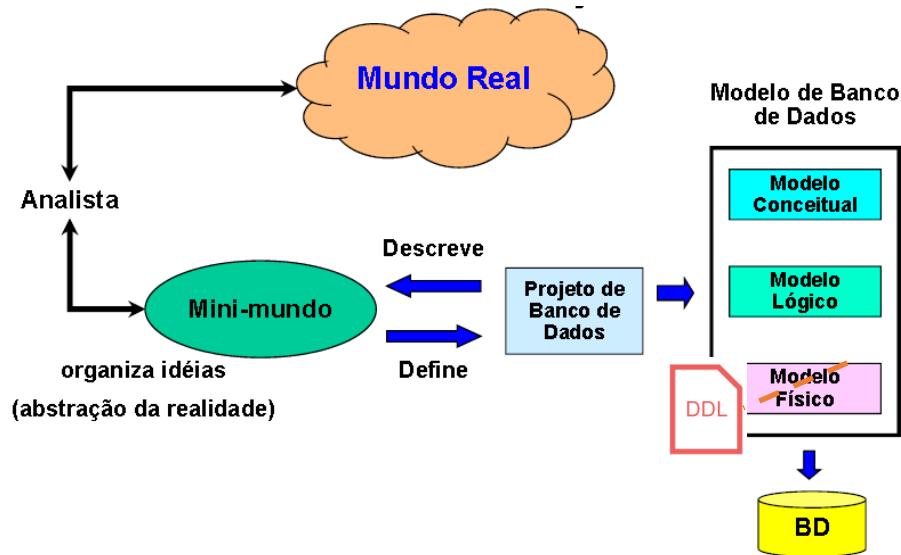
Para fazer o restore (instalar) o banco de dados de exemplo, de forma que você possa reproduzir os exemplos expostos, realizar os exercícios e praticar, assista à respectiva aula deste capítulo.

Capítulo 3. Linguagem de Definição de Dados (DDL)

A Linguagem de Definição de Dados, mais comumente mencionada como **DDL** (abreviação de *Data Definition Language*), é uma classe da Linguagem SQL com comandos que permitem criar, alterar e excluir objetos de banco de dados.

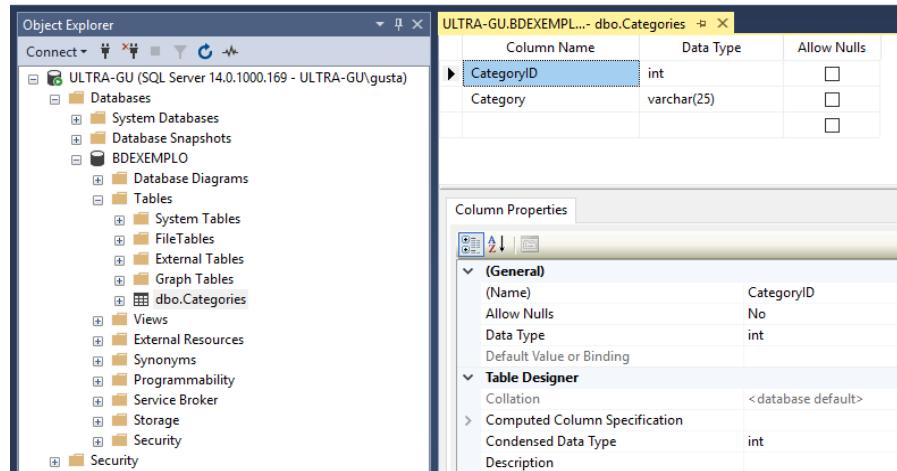
Dentro de um **Projeto de Banco de Dados**, essa classe é utilizada, principalmente, na etapa de **geração do esquema físico no banco de dados**, com base no modelo de dados físico elaborado.

Figura 31 – Modelagem de Dados.



Fonte: <https://sites.google.com/site/qiyqiyhvhj/home/o-fatecano>.

A maioria dos SGBDs fornece interface gráfica para implementação do modelo de dados físico, mas a utilização de scripts SQL é bem útil para questões de versionamento e validações.

Figura 32 – Designer Gráfico de Tabelas no SSMS.

Fonte: Gustavo (2019).

Grande parte das ferramentas CASE de modelagem de dados também fornece recursos para a geração dos scripts DDL a partir de um determinado modelo físico, como no exemplo abaixo de um script DDL, gerado pela ferramenta *ERWin* para um modelo de dados físico no SGBD Oracle.

Figura 33 – Script DDL.

```
CREATE TABLE ATENDENTE (
    COD_ATENDENTE          NUMBER(3) NOT NULL,
    NOM_ATENDENTE           VARCHAR2(50) NOT NULL
);

ALTER TABLE ATENDENTE
    ADD ( CONSTRAINT PK_ATEND PRIMARY KEY (COD_ATENDENTE)
) ;

CREATE SEQUENCE SQ_ATEND_COD_ATENDENTE
NOCACHE
NOCYCLE;

CREATE TABLE CLIENTE (
    COD_CLIENTE              NUMBER(5) NOT NULL,
    NOM_CLIENTE               VARCHAR2(50) NOT NULL,
    DSC_ENDERECO_COBRANCA    VARCHAR2(100) NOT NULL,
    NUM_TELEFONE               NUMBER(12) NOT NULL
);
```

Fonte: Gustavo (2002).

3.1. Criação de estruturas de dados

A criação de estruturas ou objetos, no banco de dados, é feita através da instrução SQL **CREATE**. Através desse comando, consegue-se criar o banco de dados, as tabelas, os índices, visões, chaves estrangeiras, constraints, sinônimos, sequences etc.

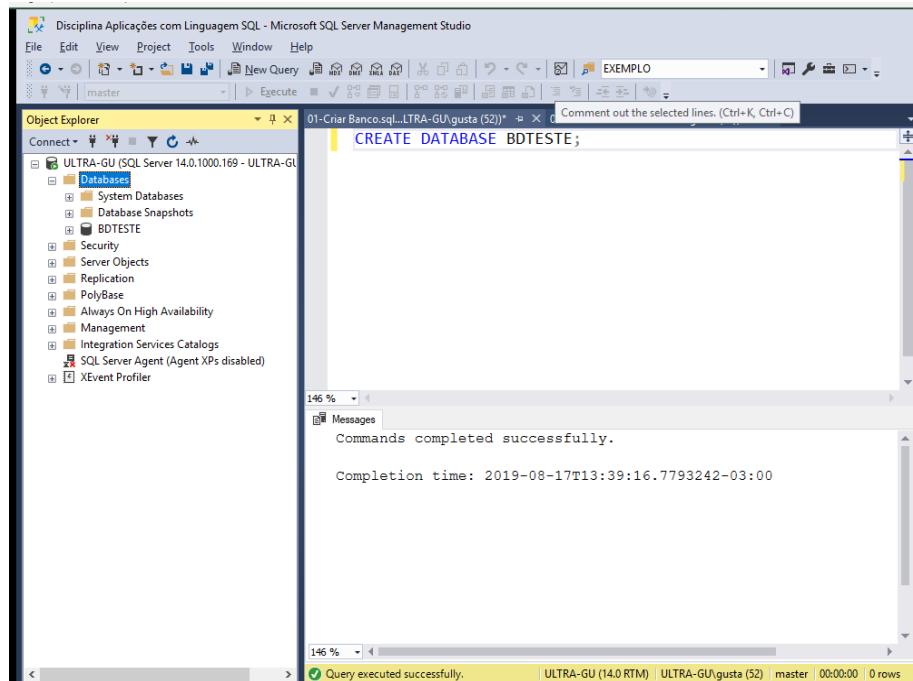
Para **criar um banco de dados**, a sintaxe do comando SQL é:

CREATE DATABASE <Nome_do_Banco_de_Dados>;

Executando o comando SQL dessa forma, sem passar parâmetros adicionais (como tamanho, conjunto de caracteres de idioma etc.), o banco de dados será criado com as opções default da instância em questão.

Para exemplificar, se desejamos criar o banco de dados de nome **BDTESTE**, devemos executar ***CREATE DATABASE BDTESTE;***

Figura 34 – Exemplo de Criação de Banco de Dados.



Fonte: Gustavo (2019).

Importante ressaltar que, apesar da instrução CREATE DATABASE pertencer ao padrão ISO da Linguagem SQL, cada SGBD implementa os parâmetros adicionais necessários e específicos da sua plataforma no comando de criação de um banco de dados, como mostrado no exemplo abaixo entre o comando no SQL Server versus no Oracle:

Figura 35 – Create Database SQL Server x Oracle.

```
CREATE DATABASE [SQLServer] ON PRIMARY
( NAME = N'SQLServer', FILENAME = N'D:\DATA\SQLServer.mdf',
  SIZE=102400KB, MAXSIZE=1024000KB, FILEGROWTH=102400KB)
LOG ON
( NAME = N'SQLServer_log', FILENAME = N'E:\LOG\SQLServer_log.ldf',
  SIZE=51200KB, MAXSIZE=3072000KB, FILEGROWTH=51200KB)
GO
ALTER DATABASE [SQLServer] SET COMPATIBILITY_LEVEL = 140
GO
ALTER DATABASE [SQLServer] SET AUTO_CLOSE OFF
GO
ALTER DATABASE [SQLServer] SET AUTO_SHRINK OFF
GO
ALTER DATABASE [SQLServer] SET AUTO_UPDATE_STATISTICS ON
GO
ALTER DATABASE [SQLServer] SET RECOVERY FULL
GO
```

```
create database Oracle
controlfile reuse
logfile 'D:\ORAWIN95\DATABASE\log1orcl.ora'
        size 200K reuse,
'D:\ORAWIN95\DATABASE\log2orcl.ora'
        size 200K reuse
datafile 'D:\ORAWIN95\DATABASE\sys1orcl.ora'
        size 20M reuse autoextend on
        next 10M maxsize 200M
character set WE8ISO8859P1;
```

Fonte: Gustavo (2019).

Para criar as tabelas, que são as estruturas onde os dados são inseridos, consultados e manipulados, usa-se o comando SQL **CREATE TABLE**. No SQL Server, sua sintaxe básica é:

```
CREATE TABLE <Nome_da_Tabela>
(
    column1 datatype [ NULL | NOT NULL ],
    column2 datatype [ NULL | NOT NULL ], ....
);
```

Um exemplo de código para criar uma tabela de nome **ALUNO**, seria:

CREATE TABLE ALUNO

(

```
COD_ALUNO int NOT NULL,  
NOM_ALUNO varchar(100) NOT NULL  
);
```

Outro tipo de objeto muito comum, criado com o comando CREATE da classe DDL da Linguagem SQL, é o **sinônimo**. Um objeto desse tipo fornece um nome alternativo para outro objeto (tabela, view, função etc.) do banco de dados. No SQL Server, sua sintaxe básica é:

```
CREATE SYNONYM <Nome_do_Sinônimo> FOR <Nome_do_Objeto>;
```

Um exemplo de código para criar um sinônimo, de nome TB_ALUNO_NEW para a tabela de nome **ALUNO**, seria:

```
CREATE SYNONYM TB_ALUNO_NEW FOR ALUNO;
```

Existem várias outras possibilidades de utilização da instrução CREATE, bem como outros comandos SQL da classe DDL, que, ao longo do curso, serão mostrados à medida que formos aprofundando na Linguagem SQL.

3.2 Alteração de estruturas de dados

É normal que, após a criação das estruturas de dados, surjam necessidades de alteração delas. Na maioria das vezes, não é necessário recriá-las para aplicar essa alteração, uma vez que a Linguagem SQL fornece o comando **ALTER** para esse tipo de situação.

Com esse tipo de comando, é possível, por exemplo, desde que não viole as regras de unicidade, adicionar uma chave primária a uma tabela já criada, como demonstrado na sintaxe abaixo:

ALTER TABLE ALUNO**ADD CONSTRAINT PK_ALUNO PRIMARY KEY CLUSTERED****(****COD_ALUNO****);**

Com o comando ALTER, é possível também adicionar uma nova coluna a uma tabela que já esteja criada, como mostrado no exemplo abaixo:

- Adicionar Coluna COD_CURSO_FK na tabela ALUNO

ALTER TABLE ALUNO ADD COD_CURSO_FK int NULL;

Da mesma forma, desde que os dados existentes na tabela não violem as regras de nulidade e de tipos de dados, é possível também alterar algumas definições e propriedades de colunas já existentes em uma tabela, como mostrado nos comandos de exemplo abaixo:

- Alterar Coluna COD_CURSO_FK Para Não Nula

ALTER TABLE ALUNO ALTER COLUMN COD_CURSO_FK int NOT NULL;

- Alterar Tamanho da Coluna NOM_CURSO

ALTER TABLE CURSO**ALTER COLUMN NOM_CURSO varchar(200) NOT NULL;**

Outra possibilidade com o comando ALTER, desde que as regras de integridade referencial dos dados não sejam violadas, é adicionar uma chave estrangeira entre duas tabelas que já estejam criadas, como no exemplo abaixo:

- Adicionar Chave Estrangeira COD_CURSO_FK na Tabela ALUNO

ALTER TABLE ALUNO

ADD CONSTRAINT FK_ALUNO_CURSO FOREIGN KEY

(

COD_CURSO_FK

) REFERENCES CURSO

(

COD_CURSO

)

ON UPDATE NO ACTION

ON DELETE NO ACTION;

Existem várias outras possibilidades de utilização da instrução ALTER, e algumas delas serão utilizadas nos capítulos seguintes à medida que formos aprofundando na Linguagem SQL.

3.3. Remoção de estruturas de dados

Um comando muito útil da linguagem SQL, também pertencente à classe DDL, é o DROP. Com ele, é possível remover as estruturas de dados e demais objetos de um banco de dados, devendo ser usado com muita cautela e atenção para não remover objetos por engano.

Usando o DROP, pode-se remover banco de dados, tabelas, views, funções, índices etc., como mostrado nos exemplos a seguir.

- Remover Sinônimo

DROP SYNONYM TB_ALUNO_NEW;

- Remover Banco de Dados

DROP DATABASE BDTESTE_DROP;

- Remover Tabela (Com FK)

DROP TABLE ALUNO;

Uma observação importante é quanto à remoção de alguns recursos que não são considerados objetos independentes, como colunas de uma tabela. Nesse caso, deve-se usar o comando ALTER, uma vez que o objeto a ser alterado é a tabela onde a coluna se encontra, em conjunto com o comando DROP, como demonstrado abaixo.

- Remover Coluna NOM_ALUNO da tabela ALUNO

ALTER TABLE ALUNO DROP COLUMN NOM_ALUNO;

Capítulo 4. Linguagem de Manipulação de Dados (DML)

Sem sombra de dúvidas, os comandos da linguagem SQL para manipulação de dados, pertencentes à classe DML (*Data Manipulation Language*), são os comandos mais usados em sistemas de bancos de dados relacionais.

Inicialmente, vamos falar do comando **SELECT**, usado para selecionar dados de forma a exibi-los ao usuário. Apesar de alguns atribuírem o comando **SELECT** à classe DQL (*Data Query Language*), no padrão ISO / IEC 9075: 2016, ele pertence à classe DML. A classe DQL pertence a um outro padrão, menos conhecido da Linguagem SQL, chamado *ODMG*, que não é o foco desse curso.

4.1. Selecionando dados

Como introduzido acima, na Linguagem SQL, a instrução utilizada para retornar (consultar) os dados armazenados no banco de dados é o **SELECT**. Essa instrução, baseada na operação de **projeção da Álgebra Relacional**, retorna as colunas e seus respectivos dados no formato tabular. Em sua estrutura mais simples (forma mínima), é composto de duas cláusulas:

- **SELECT <relação de colunas ou * >**: declaração informando que o comando se trata de uma consulta, seguida da relação de colunas que se deseja exibir no resultado. Caso se utilize o ***** (asterisco) no lugar da lista de colunas, serão retornadas todas as colunas que existirem no objeto em questão (especificado na cláusula *FROM*), **no momento da execução do comando**.
- **FROM**: origem dos dados, ou seja, o nome do(s) objeto(s) onde estão os dados que serão selecionados.

Para exemplificar as duas opções existentes para formar a lista com a relação de colunas a serem exibidas, pode-se executar no banco de dados de exemplo *AdventureWorks2017* usado nesse curso, os seguintes comandos:

- Selecionar todas as linhas das colunas **Name** e **ProductNumber** da tabela **Product**

```
SELECT Name, ProductNumber
```

```
FROM Production.Product;
```

- Selecionar todas as linhas de todas as colunas da tabela **Product**

```
SELECT *
```

```
FROM Production.Product;
```

Um recurso muito interessante e útil, que pode ser usado tanto para nomes de colunas quanto de tabelas, é o **alias**. Com ele, é possível criar um nome alternativo (apelido) existente apenas em tempo de execução para cada coluna / tabela em um comando SQL.

No caso de *alias* de colunas, é criado usando-se a cláusula **AS** após o nome da coluna, colocando-se o *alias* desejado em seguida, como mostrado abaixo:

```
SELECT Name AS Nome_do_Produto, ProductNumber AS  
Número_Produto
```

```
FROM Production.Product;
```

O efeito prático da utilização de *alias* para colunas é modificar o cabeçalho (*label*) da coluna que é exibido no resultado da query, como no exemplo abaixo:

Figura 36 – Alias de Coluna.

The screenshot shows a SQL query window with the following code:

```
SELECT Name AS Nome_do_Produto,
       ProductNumber AS Número_do_Produto
    FROM Production.Product
      GO
```

Below the code, the results pane displays the output of the query:

Nome_do_Produto	Número_do_Produto
Adjustable Race	AR-5381
Bearing Ball	BA-8327
BB Ball Bearing	BE-2349
Headset Ball Bearings	BE-2908
Blade	BL-2036

Two blue arrows point from the column headers "Nome_do_Produto" and "Número_do_Produto" in the results table back to their respective alias definitions in the query code above them.

Fonte: Gustavo (2019).

Para *alias* de tabelas, o uso é bem mais amplo, permitindo que o *alias* seja usado em outras cláusulas da instrução SELECT, como nas cláusulas *WHERE*, *GROUP BY*, *ORDER BY* etc., que serão vistas mais à frente. Além disso, a utilização de *alias* é muito útil nas situações em que existem colunas com o mesmo nome em tabelas diferentes participando do mesmo comando SQL, sendo necessário distinguir cada coluna (veremos isso mais à frente quando falarmos de junção de tabelas).

Para se criar um *alias* para tabela, pode-se usar, na cláusula **FROM**, a cláusula **AS** após o nome da tabela seguido do *alias* desejado, ou simplesmente omitir a cláusula **AS** e colocar o *alias* após o nome da tabela, como mostrado nos exemplos abaixo.

- *Alias* de Tabela Criado **com** a Cláusula **AS**

SELECT Name, ProductNumber FROM Production.Product **AS P**;

- *Alias* de Tabela Criado **sem** a Cláusula **AS**

SELECT Name, ProductNumber FROM Production.Product **P**;

Uma vez definido o *alias* para uma tabela, ele pode ser referenciado na relação de colunas da cláusula SELECT, como exemplificado abaixo.

- *Alias* de Tabela nas Colunas

```
SELECT P.Name, P.ProductNumber FROM Production.Product P;
```

Obs.: ao se definir *alias* de tabelas, não é obrigatório usá-los na relação de colunas, desde que não haja ambiguidade nos nomes das colunas.

Alias de tabela pode ser usado em conjunto com *alias* de colunas, como mostrado no exemplo abaixo:

- *Alias* de Tabela + *Alias* de Coluna

```
SELECT P.Name AS Nome_do_Produto, P.ProductNumber AS Número_Produto  
FROM Production.Product P;
```

EXPRESSÃO CASE

Um outro recurso muito útil, que pode ser utilizado na instrução SELECT, é a expressão **CASE**. Esse tipo de expressão compara um valor de entrada a uma lista de possíveis valores correspondentes, funcionando da seguinte maneira:

- Se uma correspondência for encontrada, o primeiro valor correspondente será retornado como resultado da expressão CASE. Dessa forma, múltiplas correspondências não são permitidas.
- Se nenhuma correspondência for encontrada, a expressão CASE retorna o valor encontrado em uma cláusula ELSE, se existir.
- Se nenhuma correspondência for encontrada e nenhuma cláusula ELSE existir, a expressão CASE retornará NULL.

No exemplo a seguir, está sendo usada a expressão CASE na coluna **ProductSubcategoryID**, de forma que retorne o valor '**Mountain Bikes**' se o valor da coluna for igual a 1; '**Road Bikes**' se for igual a 2; '**Touring Bikes**' se for igual a 3; e '**Unknown Category**' caso não seja nenhum dos três valores.

```
SELECT ProductID, Name, ProductSubcategoryID,  
CASE ProductSubcategoryID  
    WHEN 1 THEN 'Mountain Bikes'  
    WHEN 2 THEN 'Road Bikes'  
    WHEN 3 THEN 'Touring Bikes'  
    ELSE 'Unknown Category'  
END AS SubCategoryName  
FROM Production.Product;
```

Figura 37 – Amostra do Resultado do Exemplo de CASE

	ProductID	Name	ProductSubcategoryID	SubCategoryName
29	990	Mountain-500 Black, 42	1	Mountain Bikes
30	991	Mountain-500 Black, 44	1	Mountain Bikes
31	992	Mountain-500 Black, 48	1	Mountain Bikes
32	993	Mountain-500 Black, 52	1	Mountain Bikes
33	973	Road-350-W Yellow, 40	2	Road Bikes
34	974	Road-350-W Yellow, 42	2	Road Bikes
35	975	Road-350-W Yellow, 44	2	Road Bikes
36	976	Road-350-W Yellow, 48	2	Road Bikes

Fonte: Gustavo (2019).

4.2. Operadores aritméticos e de concatenação

Além da principal função de retornar os dados das colunas de uma tabela, a instrução SELECT pode realizar cálculos matemáticos e manipulações dos valores

selecionados em tempo de execução. O resultado desse processo é uma nova coluna a ser retornada pela instrução SELECT, chamada de **expressão calculada**, que inicialmente não tem nome (pode-se definir um *alias* para ela), não existe fisicamente na tabela e deve ser escalar (retornar apenas um valor por linha).

As expressões calculadas são construídas utilizando-se **operadores aritméticos ou de concatenação**, que no SQL Server são:

Operador	Descrição
+	Adição / Concatenação
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)

Para melhor exemplificar a utilização dos operadores aritméticos na formação de uma expressão calculada, vamos ver alguns exemplos:

– Expressão Calculada Multiplicando Duas Colunas

```
SELECT UnitPrice, OrderQty, (UnitPrice * OrderQty) AS TotalValue  
FROM Sales.SalesOrderDetail;
```

UnitPrice	OrderQty	TotalValue
2024,994	1	2024,994
2024,994	3	6074,982
2024,994	1	2024,994
2039,994	1	2039,994
2039,994	1	2039,994
2039,994	2	4079,988
2039,994	1	2039,994
28,8404	3	86,5212
28,8404	1	28,8404
5,70	6	34,20

– Expressão Calculada Usando o Operador de Divisão e Multiplicação

```

SELECT UnitPrice, UnitPriceDiscount,
       (UnitPriceDiscount / UnitPrice) * 100 AS DiscountPercentual
FROM Sales.SalesOrderDetail;
    
```

UnitPrice	UnitPriceDiscount	Discount Percentual
2,495	0,10	4,00
3,975	0,10	2,51
4,3221	0,10	2,31
4,495	0,10	2,22
4,495	0,10	2,22
4,495	0,10	2,22
4,75	0,10	2,10
4,75	0,10	2,10

– Expressão Calculada Usando o Operador de Subtração

```

SELECT UnitPrice, UnitPriceDiscount,
       UnitPrice - UnitPriceDiscount AS UnitPriceWithDiscount
FROM Sales.SalesOrderDetail
    
```

Unit Price	Unit Price Discount	Unit Price With Discount
1,3282	0,02	1,3082
1,374	0,00	1,374
1,374	0,00	1,374
1,374	0,00	1,374
1,374	0,00	1,374
1,374	0,00	1,374

Importante destacar que os **operadores aritméticos** são usados com **colunas do tipo de dados não string** (numéricos, datas, hora, moeda etc.). Para trabalhar com **strings**, o operador a ser usado é o de **concatenação (+)**.

– Expressão Calculada Usando o Operador de Concatenação

```

SELECT FirstName, MiddleName, LastName,
       FirstName + ' ' + MiddleName + ' ' + LastName AS NomeCompleto
    
```

FROM Person.Person;

FirstName	MiddleName	LastName	NomeCompleto
Xavier	C	Adams	Xavier C Adams
Ronald	L.	Adina	Ronald L. Adina
Osarum...	Uwaifiokun	Agbonile	Osarumwense U...
Samuel	N.	Agcaoili	Samuel N. Agcaoili
James	T.	Aguilar	James T. Aguilar
Robert	E.	Ahlering	Robert E. Ahlering
François	P	Ajenstat	François P Ajenstat
Kim	III	Akesson	III

Obs.: ao usar operadores, verifique primeiramente a ordem de avaliação deles no SGBD em questão. No SQL Server, por exemplo, os operadores aritméticos e de concatenação são avaliados na seguinte ordem (sequência):

1º → / (divisão)

4º → + (adição / concatenação)

2º → * (multiplicação)

5º → - (subtração)

3º → % (módulo)

Dessa forma, para que o operador de subtração, por exemplo, seja avaliado primeiramente, deve-se usar parênteses, como no exemplo abaixo. Sem os parênteses, a operação de multiplicação seria feita primeiro, para depois fazer-se a operação de subtração:

```
SELECT UnitPrice, UnitPriceDiscount, OrderQty,
```

```
(UnitPrice - UnitPriceDiscount) * OrderQty AS TotalWithDiscount
```

```
FROM Sales.SalesOrderDetail;
```

4.3. Ordenando dados

Além das cláusulas *SELECT* e *FROM*, uma instrução *SELECT* pode ter também a cláusula **ORDER BY**. Essa cláusula é **opcional** e é utilizada para ordenar os dados retornados pela instrução *SELECT*.

Com relação à sintaxe, ela é a última cláusula em uma instrução *SELECT* e, **por default**, ao utilizá-la, a ordenação é feita de forma ascendente (A à Z, 0 à 9). Entretanto, pode-se usar as opções **ASC** (ascendente) ou **DESC** (descendente, de Z a A, 9 a 0), para personalizar a ordem de exibição dos dados.

– Classificando Ascendentemente pelo Primeiro Nome

`SELECT FirstName, MiddleName, LastName`

`FROM Person.Person`

`ORDER BY FirstName;`

Similar a

`ORDER BY FirstName ASC;`

– Classificando Descendentemente pelo Primeiro Nome

`SELECT FirstName, MiddleName, LastName`

`FROM Person.Person`

`ORDER BY FirstName DESC;`

Figura 38 – ORDER BY ASC x ORDER BY DESC.

FirstName	MiddleName	LastName	FirstName	MiddleName	LastName
A.	Francesca	Leonetti	Zoe	L	Bailey
A.	Scott	Wright	Zoe	L	Bell
A. Scott	NULL	Wright	Zoe	L	Brooks
Aaron	L	Wright	Zoe	NULL	Cook
Aaron	C	Yang	Zoe	L	Cooper
Aaron	M	Young	Zoe	A	Cox

Fonte: Gustavo (2019).

Outra possibilidade da cláusula ORDER BY é usar **mais de uma coluna para ordenar o resultado, cada uma delas com sua ordem desejada** (ascendente ou descendente), como no exemplo abaixo.

- **Classificando Ascendentemente pelo Primeiro e Descendentemente pelo Último**

```
SELECT FirstName, MiddleName, LastName  
FROM Person.Person  
ORDER BY FirstName ASC, LastName DESC;
```

Ordenação descendente pela coluna *LastName*

FirstName	MiddleName	LastName
A.	Scott	Wright
A.	Francesca	Leonetti
A. Scott	NULL	Wright
Aaron	A	Zhang
Aaron	M	Young
Aaron	C	Yang
Aaron	I	Wright
Aaron	L	Washington
Aaron	V	Wang
Aaron	NULL	Simmons
Aaron	J	Sharma
Aaron	NULL	Shan

Uma flexibilidade muito interessante dessa cláusula, é que a(s) coluna(s) usada(s) no ORDER BY não necessariamente precisam estar na lista de colunas da cláusula SELECT, como mostrado no exemplo a seguir.

- **Ordenando por uma Coluna que não Está na Lista de Colunas do SELECT**

```
SELECT FirstName, MiddleName, LastName  
FROM Person.Person  
ORDER BY ModifiedDate ASC;
```

FirstName	MiddleName	LastName
Guy	R	Gilbert
Waleed	NULL	Heloo
Pat	NULL	Coleman
Kevin	F	Brown
Roberto	NULL	Tamburello
Qiang	NULL	Wang
-

Para além disso, uma outra possibilidade que existe na cláusula ORDER BY é usar *alias* de coluna para determinar a ordenação, ao invés de usar o nome das colunas, como mostrado a seguir.

– Usando **Alias de Coluna** no ORDER BY

```
SELECT Name AS Nome_do_Produto, ProductNumber AS Número_do_Produto  
FROM Production.Product  
ORDER BY Nome_do_Produto ASC;
```

4.4. Filtrando dados

Além da operação de projeção da Álgebra Relacional vista até agora (especificação das colunas a serem retornadas), a Linguagem SQL também possui cláusulas para realizar a **operação de seleção (restrição)**. Com essas cláusulas, é possível restringir as tuplas (linhas/dados) retornadas pela operação de projeção (cláusula SELECT).

A primeira delas, e a mais conhecida de todas, é a cláusula **WHERE**. Em termos de sintaxe, ela é usada **após** a cláusula *FROM* e utiliza-se dos operadores lógicos ou de comparação para filtrar os resultados retornados.

No exemplo a seguir, a cláusula **WHERE** está sendo usada para retornar as colunas com o nome e a cor, somente dos produtos que **sejam da cor preta**:

– Listar Produtos da Cor Preta (Black)

```
SELECT Name, Color  
FROM Production.Product  
WHERE Color = 'Black'  
ORDER BY Name;
```

Name	Color
Chainring	Black
Full-Finger Gloves, L	Black
Full-Finger Gloves, M	Black
Full-Finger Gloves, S	Black
Half-Finger Gloves, L	Black
Half-Finger Gloves, M	Black
Half-Finger Gloves, S	Black
HL Crankarm	Black
HL Crankset	Black
HL Mountain Frame - Black, 38	Black
HL Mountain Frame - Black, 40	Black

Em termos de operadores, cada SGBD procura manter o respectivo caractere indicado no padrão ISO, mas nada impede que existam outros operadores ou operadores representados por caracteres diferentes em alguns SGBDs.

No SQL Server, os **operadores de comparação** existentes são:

Operador de Comparação	Significado
=	Igual a
>	Maior que
<	Menor que
>=	Maior que ou igual a
<=	Menor que ou igual a
<>	Diferente de
!=	Diferente de (não é padrão ISO)
!>	Não é maior que (não é padrão ISO)
!<	Não é menor que (não é padrão ISO)

Na cláusula *WHERE*, em conjunto com os operadores de comparação, pode-se utilizar também os **operadores lógicos**, permitindo que mais de uma condição (filtro) possa ser realizada na operação de restrição da instrução SELECT.

No exemplo a seguir, o operador lógico **OR (ou)** está sendo utilizado junto à cláusula *WHERE*, para retornar as informações de nome e cor somente dos produtos que tenham/sejam da cor preta ou da cor prata:

– Listar Produtos da Cor Preta (Black) ou Prata (Silver)

```
SELECT Name, Color
```

```
FROM Production.Product
```

```
WHERE Color = 'Black'
```

OR Color = 'Silver'

```
ORDER BY Name;
```

Name	Color
Chain	Silver
Chainring	Black
Chainring Bolts	Silver
Chainring Nut	Silver
Freewheel	Silver
Front Brakes	Silver
Front Derailleur	Silver
Front Derailleur Cage	Silver
Front Derailleur Linkage	Silver
Full-Finger Gloves, L	Black
Full-Finger Gloves, M	Black

Na prática, os operadores lógicos testam a verdade de alguma condição, **retornando um tipo de dados booleano com o valor TRUE, FALSE ou UNKNOWN**. No SQL Server, os **operadores lógicos** existentes são:

Operador	Significado
ALL	TRUE se todas as comparações forem verdadeiras
AND	TRUE se ambas as expressões booleanas forem verdadeiras
ANY	TRUE se qualquer um de um conjunto de comparações for verdadeiro
BETWEEN	TRUE se o operando/expressão estiver dentro de um intervalo
EXISTS	TRUE se uma subconsulta contiver quaisquer linhas
IN	TRUE se o operando for igual a um de uma lista de valores/expressões
LIKE	TRUE se o operando corresponder a um padrão
NOT	Inverte o valor de qualquer outro operador booleano
OR	TRUE se qualquer expressão booleana for TRUE
SOME	TRUE se algumas das comparações forem verdadeiras

Nas aulas gravadas são mostrados mais exemplos acerca da utilização dos operadores lógicos, mas por hora é preciso acrescentar que, assim como os operadores aritméticos, os operadores lógicos e de comparação também podem ser usados em conjunto. Dessa forma, é preciso se atentar também à ordem de avaliação de cada um deles pelo SGBD em questão.

No SQL Server, os **operadores lógicos e de comparação** são avaliados após os operadores aritméticos e de concatenação. Entre si, os operadores lógicos e de comparação são avaliados na seguinte ordem (sequência):

1º → =, >, <, <=, <, !=, !, !<, !> (operadores de comparação)

2º → NOT

3º → AND

4º → ALL, ANY, BETWEEN, IN, LIKE, OR, SOME

Dessa forma, para que o operador *OR*, por exemplo, seja avaliado antes do operador *AND*, deve-se usar parênteses, como no exemplo abaixo. Sem os parênteses, a avaliação da expressão lógica *AND* seria feita primeiro, para depois se avaliar a expressão com o operador *OR*:

- Produtos com Nome que Iniciam com 'Chain' e que Sejam da Cor Preta ou Prata

SELECT Name,Color

FROM Production.Product

WHERE Name LIKE 'Chain%'

Name	Color
Chain	Silver
Chaining	Black
Chainring Bolts	Silver
Chainring Nut	Silver

AND (Color = 'Black' OR Color = 'Silver')

ORDER BY Name;

- Produtos com Nome que Iniciam com 'Chain' e que Sejam da Cor Preta, ou Todos Produtos da Cor Prata

SELECT Name,Color FROM Production.Product

WHERE Name LIKE 'Chain%'

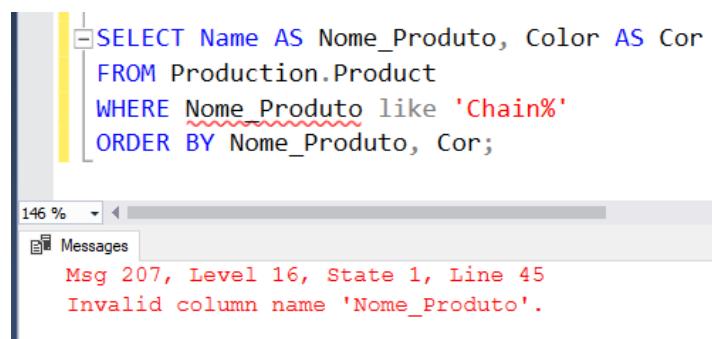
AND Color = 'Black' OR Color = 'Silver'

Name	Color
Chainring	Black
Freewheel	Silver
Front Derailleur Cage	Silver
Front Derailleur Linkage	Silver
Lock Ring	Silver
Rear Derailleur Cage	Silver
UL Mountain Frame - Silver - 42	Silver

ORDER BY Color;

Com relação à utilização de *alias* de coluna, ao contrário do que ocorre na cláusula *ORDER BY*, onde é possível utilizá-lo, na cláusula *WHERE* não. Isso ocorre devido ao fato da cláusula *WHERE* ser processada antes da cláusula *SELECT*, de forma que o *alias* ainda não está definido quando o filtro é aplicado.

Figura 39 – Erro ao usar *alias* na cláusula WHERE.



A screenshot of the SQL Server Management Studio interface. The query window contains the following code:

```
SELECT Name AS Nome_Produto, Color AS Cor
FROM Production.Product
WHERE Nome_Produto like 'Chain%'
ORDER BY Nome_Produto, Cor;
```

The 'Messages' tab shows the following error:

```
Msg 207, Level 16, State 1, Line 45
Invalid column name 'Nome_Produto'.
```

Fonte: Gustavo (2019).

4.4.1. Filtrando Dados com *TOP / DISTINCT*

Existem outras duas cláusulas, a ***TOP*** e a ***DISTINCT***, que são muito usadas também para filtrar dados, mas que, na prática, acabam apenas restringindo (limitando) a quantidade de linhas ou ocorrências duplicadas do resultado de uma instrução SELECT, respectivamente.

A cláusula ***TOP (N)*** irá atuar no resultado de uma instrução SELECT, **retornando as *N* primeiras linhas encontradas**. No exemplo abaixo, deseja-se retornar os cinco primeiros produtos (na ordem alfabética):

```
SELECT TOP (5) Name AS Nome_Produto
FROM Production.Product
ORDER BY Nome_Produto;
```

Nome_Produto
Adjustable Race
All-Purpose Bike Stand
AWC Logo Cap
BB Ball Bearing
Bearing Ball

Importante observar que, como são retornadas as N primeiras linhas, a cláusula *ORDER BY* impacta diretamente no resultado retornado. Por exemplo, se na query acima a ordenação fosse decrescente (*ORDER BY Nome_Produto DESC*), o resultado seria como mostrado ao lado.

Nome_Produto
Women's Tights, S
Women's Tights, M
Women's Tights, L
Women's Mountain Shorts, S
Women's Mountain Shorts, M

Já a cláusula ***DISTINCT*** irá atuar na eliminação dos resultados repetidos retornados (tuplas / linhas repetidas) pela instrução *SELECT* onde ela for usada. No exemplo abaixo, deseja-se saber as cores de produtos existentes (sem necessidade de retornar cores repetidas).

```
SELECT DISTINCT Color AS Cores_de_Produtos
FROM Production.Product
WHERE Color IS NOT NULL
ORDER BY Cores_de_Produtos;
```

Cores_de_Produtos
Black
Blue
Grey
Multi
Red
Silver
Silver/Black
White
Yellow

Importante observar que a cláusula ***DISTINCT*** atua em todas as colunas da cláusula *SELECT*. Caso exista mais de uma coluna no *SELECT*, basta apenas uma cláusula *DISTINCT*, uma vez que o que será eliminado será a ocorrência duplicada da combinação das colunas, ou seja, as tuplas duplicadas. No exemplo abaixo, incluindo a coluna nome do produto, o efeito da cláusula *DISTINCT* será retornar as ocorrências não repetidas da combinação das colunas *Color* e *Name*.

```
SELECT DISTINCT Color AS Cores_de_Produtos, Name AS Nome_Produto
FROM Production.Product
WHERE Color IS NOT NULL
ORDER BY Color, Name;
```

Cores_de_Produtos	Nome_Produto
Black	Chainring
Black	Full-Finger Gloves, L
Black	Full-Finger Gloves, M
Black	Full-Finger Gloves, S
Black	Half-Finger Gloves, L
Black	Half-Finger Gloves, M
Black	Half-Finger Gloves, S
Black	HL Crankarm
Black	HL Crankset
Black	HL Mountain Frame - Black, 38

4.5. Funções de caracteres, de data e hora

Funções de caracteres são recursos de programação da Linguagem SQL utilizados para modificar ou realizar um tratamento específico em uma **cadeia de caracteres (string)**. Nos SGBDs, essas funções já vêm prontas para serem utilizadas junto às queries SQL, sendo conhecidas como ***built-in string functions***.

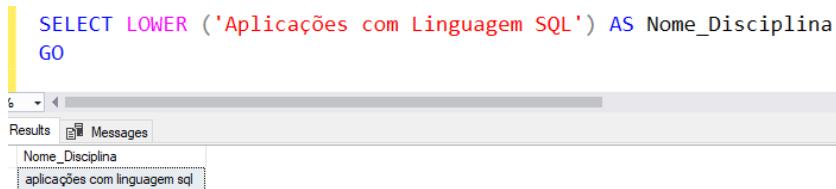
Essas funções podem receber como parâmetro de entrada uma string, uma coluna ou uma expressão, e retornam os dados alterados/tratados em um formato do tipo texto. Quando as funções de caracteres recebem parâmetros que não são do tipo string, os dados de entrada são implicitamente convertidos em um tipo de dados texto, de forma que a ação desejada possa ser executada.

Obs.: mais informações sobre tipos de dados do SQL Server podem ser vistas na documentação do produto, disponível em: <https://docs.microsoft.com/pt-br/sql/t-sql/data-types/data-types-transact-sql?view=sql-server-2017>.

Cada uma das funções de caracteres possui sua sintaxe específica e, no SQL Server, as mais utilizadas são:

- **LOWER (string):** converte a string passada como parâmetro em caracteres minúsculos.

Figura 40 – Função LOWER com cadeia de caracteres.



The screenshot shows a SQL query being run in SSMS. The query is:

```
SELECT LOWER ('Aplicações com Linguagem SQL') AS Nome_Disciplina
GO
```

The results pane shows the output of the query:

Nome_Disciplina
aplicações com linguagem sql

Fonte: Gustavo (2019).

- **UPPER (string):** converte a string passada como parâmetro em caracteres maiúsculos.

Figura 41 – Função UPPER em Colunas.

```

SELECT Name AS Nome_Original, UPPER (Name) AS Nome_Maiúsculo
FROM Production.Product
ORDER BY Name
GO

```

Nome_Original	Nome_Maiúsculo
Adjustable Race	ADJUSTABLE RACE
All-Purpose Bike Stand	ALL-PURPOSE BIKE STAND
AWC Logo Cap	AWC LOGO CAP
BB Ball Bearing	BB BALL BEARING

Fonte: Gustavo (2019).

- **SUBSTRING (string, posição, quantidade)**: retorna parte da string a partir da posição especificada até a quantidade de caracteres informada.

Figura 42 – Função SUBSTRING.

```

--SUBSTRING
SELECT SUBSTRING ('Aplicações com Linguagem SQL',16,13) AS Parte_da_String
GO

```

Parte_da_String
Linguagem SQL

Fonte: Gustavo (2019).

- **LEFT (string, quantidade)**: retorna a parte mais à **esquerda** de uma string, contendo a quantidade de caracteres informada.

Figura 43 – Função LEFT.

```

SELECT Name AS Nome_Original, LEFT (Name,5) AS Cinco_Primeiros_Caracteres
FROM Production.Product
ORDER BY Name
GO

```

Nome_Original	Cinco_Primeiros_Caracteres
Adjustable Race	Adju
All-Purpose Bike Stand	All-P
AWC Logo Cap	AWC L

Fonte: Gustavo (2019).

- **RIGHT (string, quantidade)**: retorna a parte mais à **direita** de uma string, contendo a quantidade de caracteres informada.

Figura 44 – Função RIGHT.

```

SELECT RIGHT ('Aplicações com Linguagem SQL',3) AS Parte_Direita
GO
SELECT Name AS Nome_Original, RIGHT (Name,5) AS Cinco_Ultimos_Caracteres
FROM Production.Product
ORDER BY Name
GO
    
```

Results Messages

Parte_Direita
SQL

Nome_Original	Cinco_Ultimos_Caracteres
Adjustable Race	Race
All-Purpose Bike Stand	Stand
AWC Logo Cap	o Cap
BB Ball Bearing	aring
Bearing Ball	Ball

Fonte: Gustavo (2019).

- **LEN (string)**: retorna a quantidade de caracteres encontrados na string, excluindo espaços à direita.

Figura 45 – Função LEN.

```

SELECT LEN ('Aplicações com Linguagem SQL') AS Qtde_Caracteres
GO
    
```

Results Messages

Qtde_Caracteres
28

Fonte: Gustavo (2019).

- **DATALENGTH (string)**: retorna a quantidade de caracteres encontrados na string, incluindo espaços à direita.

Figura 46 – Função DATALENGTH.

```

SELECT DATALENGTH ('Aplicações com Linguagem SQL') AS Qtde_Com_DATALENGTH,
       LEN ('Aplicações com Linguagem SQL') AS Qtde_Com_LEN
GO
    
```

Qtde_Com_DATALENGTH	Qtde_Com_LEN
32	28

Fonte: Gustavo (2019).

- **CHARINDEX (string_a_encontrar, string_onde_procurar, [início]):** procura uma string a ser encontrada em uma outra string onde a busca deve ser feita, retornando a posição inicial na string onde foi feita a busca (se encontrada). Opcionalmente, pode-se especificar a posição de início a partir da qual a busca na string deve ser feita.

Figura 47 – Função CHARINDEX.

```

SELECT CHARINDEX ('SQL', 'Aplicações com Linguagem SQL') AS Inicio_String_SQL,
       CHARINDEX ('T-SQL', 'Aplicações com Linguagem SQL') AS String_Inexiste
GO
--Especificando uma Posição de Início para a Busca
SELECT CHARINDEX ('SQL', 'Aplicações com Linguagem SQL',27) AS A_Partir_da_27ª
GO
    
```

Inicio_String_SQL	String_Inexiste
26	0

A_Partir_da_27ª
0

Fonte: Gustavo (2019).

- **REPLACE (string, string_a_encontrar, string_substituta):** substitui, na string alvo da busca, todas as ocorrências de uma string a ser encontrada, pelo valor de uma string substituta informada.

Figura 48 – Função REPLACE.

```
SELECT REPLACE ('Aplicações com Linguagem ABC', 'ABC', 'SQL') AS String_Trocada
GO

Results Messages
String_Trocada
Aplicações com Linguagem SQL
```

Fonte: Gustavo (2019).

- **REPLICATE (string, quantidade):** repete uma string por uma quantidade especificada de vezes.

Figura 49 – Função REPLICATE.

```
SELECT REPLICATE ('SQL ', 5) AS String_Replicada
GO

Results Messages
String_Replicada
SQL SQL SQL SQL SQL
```

Fonte: Gustavo (2019).

- **REVERSE (string):** retorna a ordem inversa de uma string.

Figura 50 – Função REVERSE.

```
SELECT REVERSE ('Linguagem SQL') AS String_Invertida
GO

char(15), null)
Results Messages
String_Invertida
LQS megaugnIL
```

Fonte: Gustavo (2019).

- **LTRIM (string):** remove todos os espaços à **esquerda** da string.

Figura 51 – Função LTRIM.

The screenshot shows a SQL query window with the following code:

```
SELECT LTRIM (' Aplicações com Linguagem SQL') AS Espaços_a_Esquerda_Removidos
GO
```

The results pane displays the output:

Results	Messages
Espaços_a_Esquerda_Removidos	Aplicações com Linguagem SQL

Fonte: Gustavo (2019).

- **RTRIM (string):** remove todos os espaços à **direita** da string.

Figura 52 – Função RTRIM.

The screenshot shows a SQL query window with the following code:

```
SELECT RTRIM ('Aplicações com Linguagem SQL   ') AS Espaços_a_Direita_Removidos
GO
```

The results pane displays the output:

Results	Messages
Espaços_a_Direita_Removidos	Aplicações com Linguagem SQL

Fonte: Gustavo (2019).

- **TRIM (string):** remove todos espaços do **início ou final** de uma string.

Figura 53 – Função TRIM.

The screenshot shows a SQL query window with the following code:

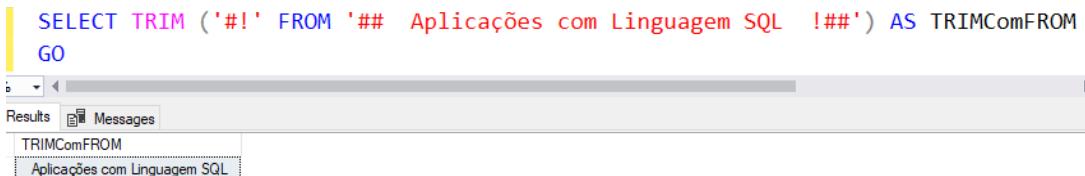
```
SELECT TRIM ('    Linguagem SQL    ') AS Removidos
GO
```

The results pane displays the output:

Results	Messages
Removidos	Linguagem SQL

Fonte: Gustavo (2019).

- **TRIM ('caracteres' FROM string):** remove os espaços e os caracteres especificados do início (antes do espaço) ou do final (depois do espaço) da string em questão.

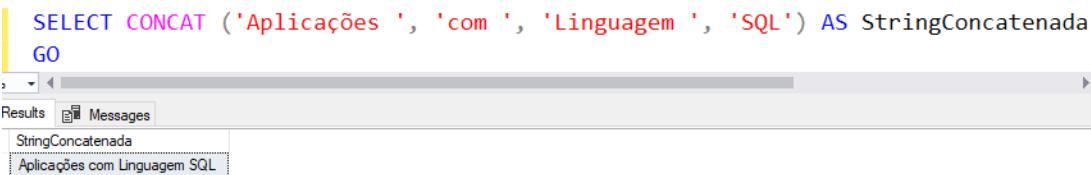
Figura 54 – Função TRIM com FROM.


```
SELECT TRIM ('#!' FROM '## Aplicações com Linguagem SQL ##') AS TRIMComFROM
GO
```

The screenshot shows an SQL query window with the results tab selected. The output is a single row with the value 'Aplicações com Linguagem SQL'.

Fonte: Gustavo (2019).

- **CONCAT (string, string_1, string_N)**: concatena uma ou mais strings.

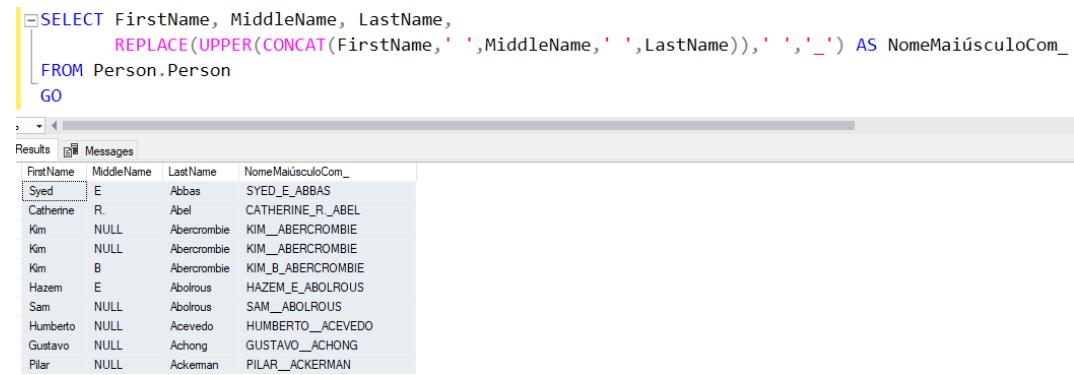
Figura 55 – Função CONCAT.


```
SELECT CONCAT ('Aplicações ', 'com ', 'Linguagem ', 'SQL') AS StringConcatenada
GO
```

The screenshot shows an SQL query window with the results tab selected. The output is a single row with the value 'Aplicações com Linguagem SQL'.

Fonte: Gustavo (2019).

Para além disso, as funções de caracteres podem ser usadas em conjunto, com a saída (retorno) de uma determinada função, sendo o parâmetro de entrada de outra função. No exemplo a seguir, a função *CONCAT* concatena três colunas e as passa como parâmetro de entrada para a função *UPPER*, que converte a string inteira para letras maiúsculas, passando-a como parâmetro de entrada para a função *REPLACE*, que, por sua vez, substitui os espaços por underline “_”.

Figura 56 – Funções de Caracteres Usadas em Conjunto.


```
SELECT FirstName, MiddleName, LastName,
       REPLACE(UPPER(CONCAT(FirstName, ' ', MiddleName, ' ', LastName)), ' ', '_') AS NomeMaiúsculoCom_
  FROM Person.Person
GO
```

The screenshot shows an SQL query window with the results tab selected. The output is a table with columns FirstName, MiddleName, LastName, and NomeMaiúsculoCom_. The data is as follows:

FirstName	MiddleName	LastName	NomeMaiúsculoCom_
Syed	E	Abbas	SYED_E_ABBA
Catherine	R.	Abel	CATHERINE_R_ABEL
Kim	NULL	Abercrombie	KIM_ABERCROMBIE
Kim	NULL	Abercrombie	KIM_ABERCROMBIE
Kim	B	Abercrombie	KIM_B_ABERCROMBIE
Hazem	E	Abolrous	HAZEM_E_ABOLROU
Sam	NULL	Abolrous	SAM_ABOLROU
Humberto	NULL	Acevedo	HUMBERTO_ACEVEDO
Gustavo	NULL	Achong	GUSTAVO_ACHONG
Pilar	NULL	Ackerman	PILAR_ACKERMAN

Fonte: Gustavo (2019).

Figura 57 – Tabela Resumo de Funções de Caracteres.

Função	Ação
LOWER	Converte a string em caracteres minúsculos.
UPPER	Converte a string em caracteres maiúsculos.
SUBSTRING	Retorna uma parte da string.
LEFT	Retorna a parte esquerda de uma string com a quantidade especificada de caracteres.
RIGHT	Retorna a parte direita de uma string com a quantidade especificada de caracteres.
LEN	Retorna a quantidade de caracteres encontrados na string, excluindo espaços à direita.
DATALENGTH	Retorna a quantidade de caracteres encontrados na string, incluindo espaços à direita.
CHARINDEX	Procura uma string dentro de uma segunda string, retornando à posição inicial da primeira expressão, se encontrada.
REPLACE	Substitui todas as ocorrências de um valor por outro valor.
REPLICATE	Repete uma string por um número especificado de vezes.
REVERSE	Retorna a ordem inversa de uma string.
CONCAT	Retorna uma string, resultante da concatenação (junção) de duas ou mais strings, de ponta a ponta.

TRIM	Remove o caractere de espaço ou outros caracteres especificados do início ou final de uma string.
LTRIM	Trunca todos os espaços à esquerda.
RTRIM	Trunca todos os espaços à direita.

Fonte: Gustavo (2019).

Funções de data e hora:

Uma outra categoria de ***built-in functions***, específica para se trabalhar com os tipos de dados de data/hora, é a de **funções de data e hora**. Com essas funções, é possível extrair partes dos campos de data/hora, alterá-los para um determinado formato, bem como realizar operações de cálculo.

Obs.: o SQL Server (e a grande maioria dos SGBDs) não oferece meios para que sejam inseridas datas e horas como **valores literais**. Dessa forma, é necessário usar sequências de caracteres (**strings literais**) delimitadas com **aspas simples**, seguindo-se o formato* de data e hora especificado nas configurações regionais do sistema operacional. Com isso, o SQL Server consegue converter, fidedignamente, as strings literais em data e hora.

* *Formato Brasileiro: 'DD-MM-AAAA hh:mm:ss[.nnnnnnnn]'*

Formato Americano: 'YYYY-MM-DD hh:mm:ss[.nnnnnnnn]'

A / Y → ano

h → hora

nnn → milisegundos

M → mês

m → minuto

nnnnn → microsegundos

D → dia

s → segundos

nnnnnnn → nanosegundos

As funções mais básicas na Linguagem SQL para se trabalhar com data e hora são as funções que retornam parte da informação de um dado do tipo data e/ou

hora. A seguir, são listadas as principais funções básicas de data e hora disponíveis no SQL Server, com um exemplo envolvendo todas elas ao final.

- **DAY (data)**: retorna um inteiro representando o **dia da data**.
- **MONTH (data)**: retorna um inteiro representando o **mês da data**.
- **YEAR (data)**: retorna um inteiro representando o **ano da data**.
 - **Separando Dia, Mês e Ano de um Campo com Data e Hora (datetime)**

```
SELECT DISTINCT DueDate AS Data_e_Hora_Vencimento,
               DAY(DueDate) AS Dia_Vencimento,
               MONTH(DueDate) AS Mês_Vencimento,
               YEAR(DueDate) AS Ano_Vencimento
FROM Purchasing.PurchaseOrderDetail
ORDER BY DueDate DESC;
```

Figura 58 – Resultado exemplo de funções de data e hora.

Data_e_Hora_Vencimento	Dia_Vencimento	Mês_Vencimento	Ano_Vencimento
2014-10-22 00:00:00.000	22	10	2014
2014-09-22 00:00:00.000	22	9	2014
2014-08-17 00:00:00.000	17	8	2014
2014-08-16 00:00:00.000	16	8	2014
2014-08-15 00:00:00.000	15	8	2014
2014-08-14 00:00:00.000	14	8	2014
2014-08-13 00:00:00.000	13	8	2014

Fonte: Gustavo (2019).

Alternativamente a essas funções, pode-se usar a função **DATENAME**, que além de retornar o dia, mês e ano de uma data-hora, pode retornar outras

partes/informações, como o dia da semana, o nome por extenso do mês, a hora, minutos, segundos etc.

Sua sintaxe é **DATENAME (datepart, date)**, onde *datepart* são as opções abaixo, e *date* é um campo ou uma cadeia de caracteres no formato do tipo data-hora.

Figura 59 – Opções *datepart* para funções de data e hora.

datepart	Informação extraída	Exemplo do retorno
year, yyyy, yy	Ano	2019
month, mm, m	Mês	October
dayofyear, dy, y	Dia do ano	303
day, dd, d	Dia do mês	30
week, wk, ww	Número da semana	44
weekday, dw	Dia da semana	Tuesday
hour, hh	Hora	12
minute, n	Minutos	15
second, ss, s	Segundos	32
millisecond, ms	Milisegundos	123
microsecond, mcs	Microsegundos	123456
nanosecond, ns	Nanosegundos	123456700

Fonte: Gustavo (2019).

– Listando Dia da Semana, Hora e Minutos

```
SELECT DATENAME (weekday,GETDATE()) AS Dia_da_Semana,
```

```
DATENAME (hour,GETDATE()) AS Hora,
```

```
DATENAME (n,GETDATE()) AS Minutos;
```

	Results	Messages	
	Dia_da_Semana	Hora	Minutos
1	terça-feira	23	40

Duas funções também muito interessantes, que possibilitam calcular a diferença entre duas datas, ou fazer uma adição (de dias, horas, minutos etc.), são as funções **DATEDIFF** e **DATEADD**, respectivamente. Ambas utilizam as opções de *datepart*, listadas na Figura 59, para indicar a unidade de medida da diferença a ser retornada (no caso de *DATEDIFF*), ou a unidade da medida do valor que está sendo somado à data (no caso de *DATEADD*). A seguir, são mostradas as sintaxes e exemplos para cada uma delas.

- **DATEDIFF** (*datepart* , Data_Inicial , Data_Final):

- **Calcular diferença em dias entre duas datas**

```
SELECT DATEDIFF(day, '2018-12-31 23:59:59', '2019-01-01 00:00:00');
```

→ 1

```
SELECT DATEDIFF(day, '2018-12-31', '2019-01-07'); → 7
```

```
SELECT DATEDIFF(day, '2018', '2019'); → 365
```

- **Calcular diferença em horas entre duas datas**

```
SELECT DATEDIFF(hour, '2018-12-31 23:59:59', '2019-01-01 00:00:00');
```

→ 1

```
SELECT DATEDIFF(hour, '2018-12-31', '2019-01-01'); → 24
```

```
SELECT DATEDIFF(hour, '2018', '2019'); → 8.760
```

- **Calcular diferença em minutos entre duas datas**

```
SELECT DATEDIFF(minute, '2018-12-31 23:59:59', '2019-01-01 00:00:00'); → 1
```

```
SELECT DATEDIFF(minute, '2018-12-31', '2019-01-01'); → 1.440
```

```
SELECT DATEDIFF(minute, '2018', '2019'); → 525.600
```

- **DATEADD** (datepart , Quantidade , Data):

- Adicionar 1 hora a uma data

```
SELECT DATEADD (hour, 1, '2019-01-01 00:30:00');
```

Resultado ➔ ‘2019-01-01 01:30:00.000’

- Adicionar 2 horas a uma data

```
SELECT DATEADD (hour, 2, '2019-01-01 22:30:00');
```

Resultado ➔ ‘2019-01-02 00:30:00.000’

- Adicionar 6 meses a uma data

```
SELECT DATEADD (month, 6, '2019-01-15');
```

Resultado ➔ ‘2019-07-15 00:00:00.000’

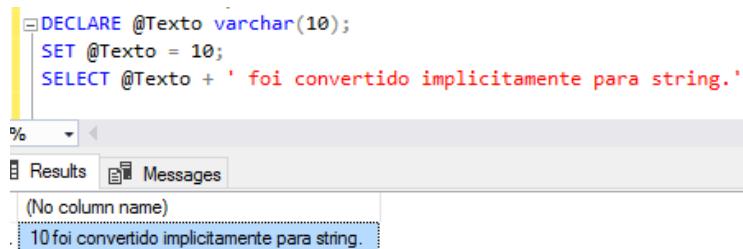
4.6. Tipos de dados e conversões Explícitas x Implícitas

Além da conversão implícita de strings literais, no formato de data/hora para o tipo de dados *datetime*, o SQL Server realiza outras conversões implícitas de forma totalmente transparente para o usuário. As mais comuns acontecem quando são feitas comparações entre dois tipos de dados diferentes, como quando um *smallint* é comparado a um *int*, e o *smallint* é implicitamente convertido para *int* antes que a comparação prossiga.

Outra conversão implícita muito comum é feita na atribuição de valores a uma variável ou a uma coluna. Nesse caso, o tipo de dados resultante é o definido na declaração da variável ou o da coluna. Por exemplo, no código abaixo, a variável **Texto** é definida com o tipo *varchar*, é atribuído um valor do tipo *int* a ela e, em seguida, feita uma concatenação dessa variável com uma string. Como

consequência, o valor inteiro **10** é convertido implicitamente em um **varchar**, retornando, na instrução **SELECT**, uma string de caracteres.

Figura 60 – Exemplo de Conversão Implícita.



The screenshot shows a SQL query window with the following code:

```
--DECLARE @Texto varchar(10);
--SET @Texto = 10;
--SELECT @Texto + ' foi convertido implicitamente para string.'
```

The results pane shows a single row with the value "10 foi convertido implicitamente para string." highlighted.

Fonte: Gustavo (2019).

Obs.: para operadores de comparação ou outras expressões, o tipo de dados resultante depende das regras de precedência do tipo de dados de cada SGBD. As do SQL Server podem ser encontradas em:

- <https://docs.microsoft.com/pt-br/sql/t-sql/data-types/data-type-precedence-transact-sql?view=sql-server-2017>.

Além das conversões implícitas, é possível fazer também **conversões explícitas** usando-se funções padrões da Linguagem SQL. Na T-SQL, são disponibilizadas as funções **CAST()** e **CONVERT()**, sendo a primeira do padrão ISO.

Figura 61 – Exemplo de conversão explícita de dados.

```
--Convertendo Explicitamente Tipo de Dados MONEY para VARCHAR
SELECT CAST ( $1090.50 AS VARCHAR(10) ) AS VALOR
```



The screenshot shows a SQL query window with the following code:

```
--Convertendo Explicitamente Tipo de Dados MONEY para VARCHAR
SELECT CAST ( $1090.50 AS VARCHAR(10) ) AS VALOR
```

The results pane shows a single row with the value "1090.50" highlighted.

Fonte: Gustavo (2019).

A grande maioria dos SGBDs fornece um mapa como o mostrado abaixo, indicando as conversões implícitas e explícitas permitidas entre os tipos de dados existentes no produto, bem como as conversões não possíveis.

Figura 62 – Mapa de conversões de tipos de dados.

O mapa de conversões de tipos de dados é uma matriz bidimensional que mostra a permissibilidade de conversões entre diferentes tipos de dados. As linhas representam o tipo de origem ('From') e as colunas representam o tipo de destino ('To'). Os ícones na grade indicam a natureza da conversão:

- Explicit conversion** (Conversão explícita): Representada por um ícone azul.
- Implicit conversion** (Conversão implícita): Representada por um ícone laranja.
- Conversion not allowed** (Conversão não permitida): Representada por um ícone com uma cruz vermelha.
- Requires explicit CAST to prevent the loss of precision or scale that might occur in an implicit conversion.** (Requer CAST explícito para evitar a perda de precisão ou escala que pode ocorrer em uma conversão implícita): Representada por um ícone com um diamante cinza.
- Otherwise, the conversion must be explicit.** (Caso contrário, a conversão deve ser explícita): Representada por um ícone com um círculo branco.

Alguns tipos de dados são listados na diagonal principal (que não é visível na grade) e não aparecem na grade:

- binary, varbinary, char, nchar, nvarchar, date, smalldatetime, datetimeoffset, decimal, numeric, float, real, tinyint, smallint, int, money, smallmoney, bit, timestamp, uniqueidentifier, image, text, ntext, sql_variant, xml, CLR UDT, hierarchyid.

Fonte: Microsoft (2017).

Capítulo 5. Agrupamento de Dados e Funções de Agregação

Em muitas situações, pode haver a necessidade de agrupar dados em conjuntos ou executar um cálculo (contagem, soma, média etc.) em um conjunto de linhas, ao invés de fazê-lo linha a linha. Para isso, a Linguagem SQL disponibiliza cláusulas para agrupamento de dados, como a **GROUP BY** e as **funções de agregação**.

5.1. Funções de agregação

Nesse tipo de função, os valores de várias linhas são agrupados para formar um único valor resumido (agregado), sendo muito útil para consultas analíticas e cálculos matemáticos.

Antes de se começar a trabalhar com funções de agregação, é preciso estar bem claro que:

- Funções agregadas retornam um valor único (escalar) e podem ser usadas nas instruções SELECT.
- Com exceção da função COUNT(*), as funções agregadas ignoram valores nulos (NULL).
- Funções agregadas não geram um *alias* de coluna, sendo permitido defini-lo explicitamente.
- Funções agregadas operam em todas as linhas retornadas pela instrução SELECT (respeitando, é claro, os filtros definidos na cláusula WHERE).
- Se não existir uma cláusula GROUP BY (que será vista mais adiante), não pode existir colunas na cláusula SELECT que não estejam em uma função agregada.

As funções de agregação da Linguagem SQL estão presentes na grande maioria dos SGBDs. No SQL Server, algumas das mais usadas são:

- **MIN (expressão)**: retorna o **menor** número, data/hora mais **antiga** ou a **primeira** ocorrência de uma string.
- **MAX (expressão)**: retorna o **maior** número, data/hora mais **recente** ou **última** ocorrência de uma string.

--Usando as funções MIN e MAX

```
SELECT MIN(UnitPrice) AS Preço_Mínimo, MAX(UnitPrice) AS  
Preço_Máximo  
FROM Sales.SalesOrderDetail;
```

Preço_Mínimo	Preço_Máximo
1,3282	3578,27

- **COUNT (*)** ou **COUNT (expressão)**: com (*), conta todas as linhas, incluindo aquelas com valores nulos (*NULL*). Quando uma coluna é especificada como <expressão>, retorna a contagem de linhas não nulas para essa coluna, como demonstrado no exemplo abaixo.

Figura 63 – COUNT (*) x COUNT (expressão).

The screenshot shows a SQL query in the query editor and its results in the results grid. The query is:

```
SELECT COUNT(*) AS Total_de_Produtos,  
       COUNT(Color) AS Total_de_Produtos_Com_Cor  
  FROM Production.Product;
```

The results grid has two columns: 'Total_de_Produtos' and 'Total_de_Produtos_Com_Cor'. The data row shows values 504 and 256 respectively.

Total_de_Produtos	Total_de_Produtos_Com_Cor
504	256

Fonte: Gustavo (2019).

- **SUM (expressão)**: **soma** todos os valores numéricos não nulos de uma coluna.

- **AVG (expressão):** calcula a **média** de todos os valores não nulos retornados, ou seja, é o resultado de SUM / COUNT.

Figura 64 – Funções SUM e AVG.

A screenshot of SQL Server Management Studio (SSMS) showing a query window and a results window. The query window contains the following SQL code:

```
SELECT SUM(LineTotal) AS Valor_Total_de_Vendas,
       AVG(LineTotal) AS Valor_Médio_de_Vendas
  FROM Sales.SalesOrderDetail;
```

The results window shows the output of the query:

Valor_Total_de_Vendas	Valor_Médio_de_Vendas
109846381.399888	905.449206

Fonte: Gustavo (2019).

Uma opção muito interessante, presente na maioria dos SGBDs, é a de se utilizar funções de agregação em conjunto com as outras funções vistas anteriormente (funções de data e hora, de caracteres etc.), como demonstrado no exemplo abaixo.

Figura 65 – Funções de agregação em conjunto com outras.

A screenshot of SQL Server Management Studio (SSMS) showing a query window and a results window. The query window contains the following SQL code:

```
SELECT MIN(YEAR(orderdate)) AS Ano_da_Compra_Mais_Antiga,
       MAX(YEAR(orderdate)) AS Ano_da_Compra_Mais_Recente
  FROM Sales.SalesOrderHeader;
```

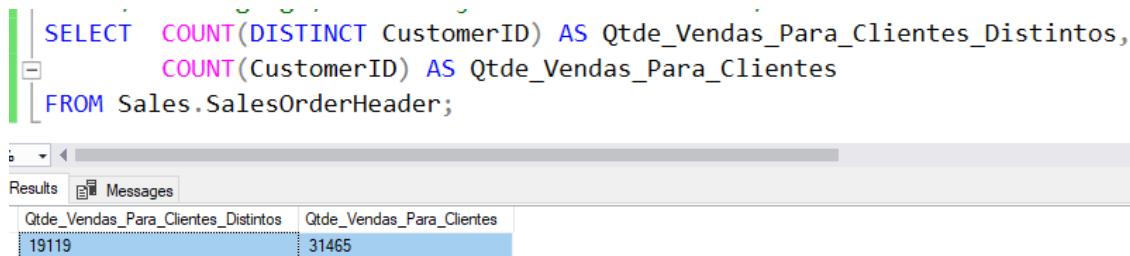
The results window shows the output of the query:

Ano_da_Compra_Mais_Antiga	Ano_da_Compra_Mais_Recente
2011	2014

Fonte: Gustavo (2019).

Outro recurso muito útil de ser utilizado em conjunto com as funções agregadas é a cláusula *DISTINCT*. Quando usada com uma função de agregação, o DISTINCT remove valores duplicados de uma coluna antes de calcular o valor final. Com isso, é possível agregar ocorrências únicas dos valores de uma coluna, como no exemplo abaixo, onde conta-se as compras feitas por clientes diferentes.

Figura 66 – DISTINCT com funções de agregação.



```
SELECT COUNT(DISTINCT CustomerID) AS Qtde_Vendas_Para_Clientes_Distintos,
       COUNT(CustomerID) AS Qtde_Vendas_Para_Clientes
  FROM Sales.SalesOrderHeader;
```

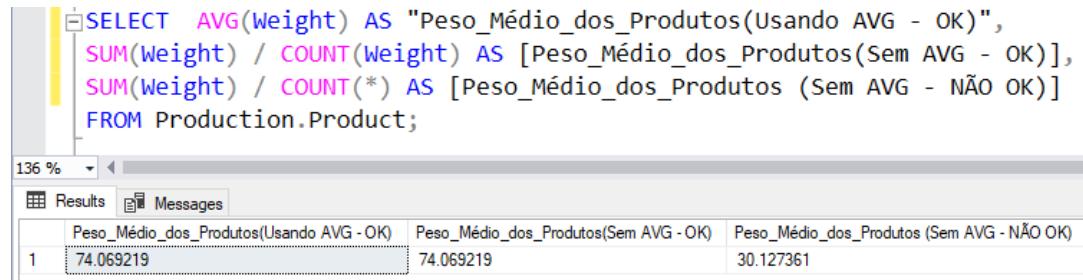
Qtde_Vendas_Para_Clientes_Distintos	Qtde_Vendas_Para_Clientes
19119	31465

Fonte: Gustavo (2019).

Para além disso, é muito importante ter em mente o comportamento das funções de agregação com relação a valores nulos, de forma a não serem feitos cálculos errados ou obter-se resultados inesperados. Como informado anteriormente, **com exceção da função COUNT(*), todas as funções de agregação ignoram valores nulos.** Isso significa que, ao usar a função *SUM*, por exemplo, ela somará apenas valores não nulos, uma vez que valores nulos não são avaliados como zero.

Estando isso exposto, pode-se notar, no exemplo abaixo, o cálculo da média dos pesos dos produtos sendo calculado de forma errada ao usar COUNT(*), uma vez que a coluna que contém essa informação (*weight*) contém valores nulos.

Figura 67 – Valores nulos em funções de agregação.



```
SELECT AVG(Weight) AS "Peso_Médio_dos_Produtos(Usando AVG - OK)",
       SUM(Weight) / COUNT(Weight) AS [Peso_Médio_dos_Produtos(Sem AVG - OK)],
       SUM(Weight) / COUNT(*) AS [Peso_Médio_dos_Produtos (Sem AVG - NÃO OK)]
  FROM Production.Product;
```

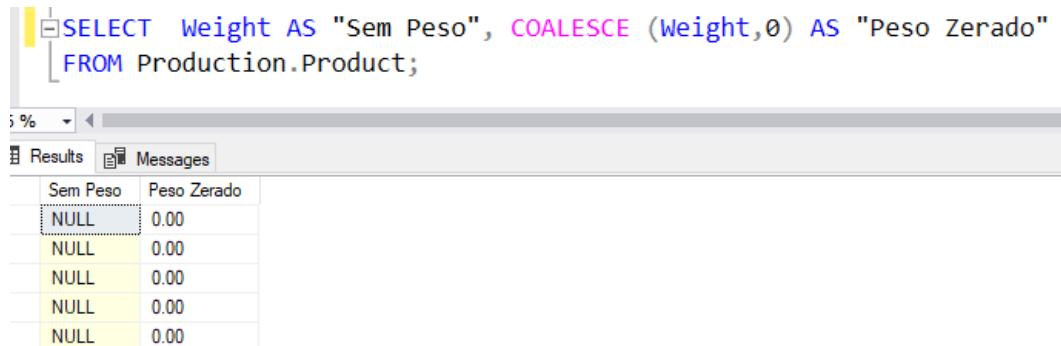
Peso_Médio_dos_Produtos(Usando AVG - OK)	Peso_Médio_dos_Produtos(Sem AVG - OK)	Peso_Médio_dos_Produtos (Sem AVG - NÃO OK)
74.069219	74.069219	30.127361

Fonte: Gustavo (2019).

Para se tratar de valores nulos em tempo de execução, a Linguagem SQL fornece funções para substitui-los por valores válidos. No SQL Server, temos a função **COALESCE** para esse propósito, devendo-se **passar como parâmetros para ela a coluna e o valor com o qual os valores nulos devem ser substituídos.** Dessa

forma, podemos fazer a média do exemplo acima de forma correta, tratando os nulos antes, como mostrado no exemplo abaixo.

Figura 68 – Tratando Valores Nulos com COALESCE.



```
SELECT Weight AS "Sem Peso", COALESCE (Weight,0) AS "Peso Zerado"
FROM Production.Product;
```

Sem Peso	Peso Zerado
NULL	0.00

Fonte: Gustavo (2019).

5.2. Agrupamentos de dados

Em vários momentos em um projeto de banco de dados, principalmente em projetos de caráter analítico, surgem necessidades para agrupar os dados retornados por uma query em grupos ou subconjuntos. Na maioria delas, fazer a summarização de dados somente após esse agrupamento se faz essencial para as regras de análise dos dados.

Como exemplos de agrupamento com summarização de dados, podemos citar uma query que retorne o total de vendas agrupado por dia, por produto ou por vendedor, bem como uma query que retorne os limites mínimos e máximos de preço em cada categoria de produtos. Na Linguagem SQL, a cláusula responsável por prover essa possibilidade é a **GROUP BY**. Em termos de sintaxe, ela é inserida após a cláusula *FROM* ou após a cláusula *WHERE* (quando há um filtro na query).

SELECT < lista de colunas / expressões >

FROM < tabela >

WHERE < condição de filtro – opcional >

GROUP BY < lista de colunas / expressões >**ORDER BY < lista de colunas / expressões – opcional >**

Dessa forma, a cláusula **GROUP BY** criará grupos e agrupará as linhas em cada grupo, conforme determinado pelas combinações exclusivas dos elementos (uma coluna/múltiplas colunas/expressões) especificados na cláusula. A título de exemplo, na query abaixo está sendo contabilizado o total de vendas agrupadas pelo ID do funcionário (ID *NULL* significa compras feitas pela Web, sem vendedor).

Figura 69 – Exemplo de GROUP BY Simples.

The screenshot shows a SQL query in the query editor and its results in the results grid. The query is:

```
SELECT SalesPersonID AS ID_do_Vendedor, COUNT(SalesOrderID) AS Total_de_Vendas
FROM Sales.SalesOrderHeader
GROUP BY SalesPersonID
ORDER BY 2 DESC
```

The results grid displays the following data:

ID_do_Vendedor	Total_de_Vendas
NULL	27659
277	473
275	450
279	429
276	418

Fonte: Gustavo (2019).

Um ponto de atenção muito importante ao se trabalhar com agrupamentos na Linguagem SQL refere-se às situações onde é exibido o erro abaixo*.

Figura 70 – Erro de GROUP BY.

The screenshot shows a SQL query in the query editor and its results in the results grid. The query is:

```
SELECT SalesPersonID AS ID_do_Vendedor, OrderDate AS Ano_da_Venda,
       COUNT(SalesOrderID) AS Total_de_Vendas
FROM Sales.SalesOrderHeader
GROUP BY SalesPersonID
ORDER BY OrderDate DESC
```

The results grid shows an error message:

Msg 8120, Level 16, State 1, Line 11
Column 'Sales.SalesOrderHeader.OrderDate' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

Fonte: Gustavo (2019).

Esse erro acontece devido à ordem lógica de processamento das operações (cláusulas) nos SGBDs. No SQL Server, como mostrado no quadro abaixo, a cláusula **GROUP BY** é processada antes do **SELECT**. Isso significa que, se uma coluna ou expressão não estiver presente no **GROUP BY** ou não estiver em uma função de

agregação, ela não fará parte do conjunto de dados (colunas) passado para a cláusula **SELECT**, no qual ela agirá selecionando as colunas especificadas e que estejam disponíveis, ou seja, que foram passadas pelas cláusulas anteriores.

<i>Ordem Lógica das Operações</i>	<i>Cláusula</i>
5	SELECT
1	FROM
2	WHERE
3	GROUP BY
4	HAVING
6	ORDER BY

*Código correto: *SELECT SalesPersonID AS ID_do_Vendedor,
OrderDate AS Ano_da_Venda,
COUNT(SalesOrderID) AS Total_de_Vendas
FROM Sales.SalesOrderHeader
GROUP BY SalesPersonID, OrderDate
ORDER BY OrderDate DESC;*

Em decorrência dessa mesma ordem lógica de processamento, não é possível utilizar um *alias* definido na cláusula **SELECT**, na cláusula **GROUP BY**, uma vez que no momento do processamento da cláusula **GROUP BY**, o *alias* ainda não foi processado (será definido somente quando a cláusula **SELECT** for processada).

Figura 71 – Erro ao tentar usar alias no GROUP BY.

The screenshot shows a SQL query in the query editor:

```
SELECT OrderDate AS Data_da_Venda, SUM(TotalDue) AS Valor_Total_de_Vendas
FROM Sales.SalesOrderHeader
GROUP BY Data_da_Venda
ORDER BY 1 ASC;
```

An error message is displayed in a tooltip: "Alias não permitido! Usar coluna OrderDate!"

In the messages pane at the bottom, the error is shown:

```
Msg 207, Level 16, State 1, Line 44
Invalid column name 'Data_da_Venda'.
```

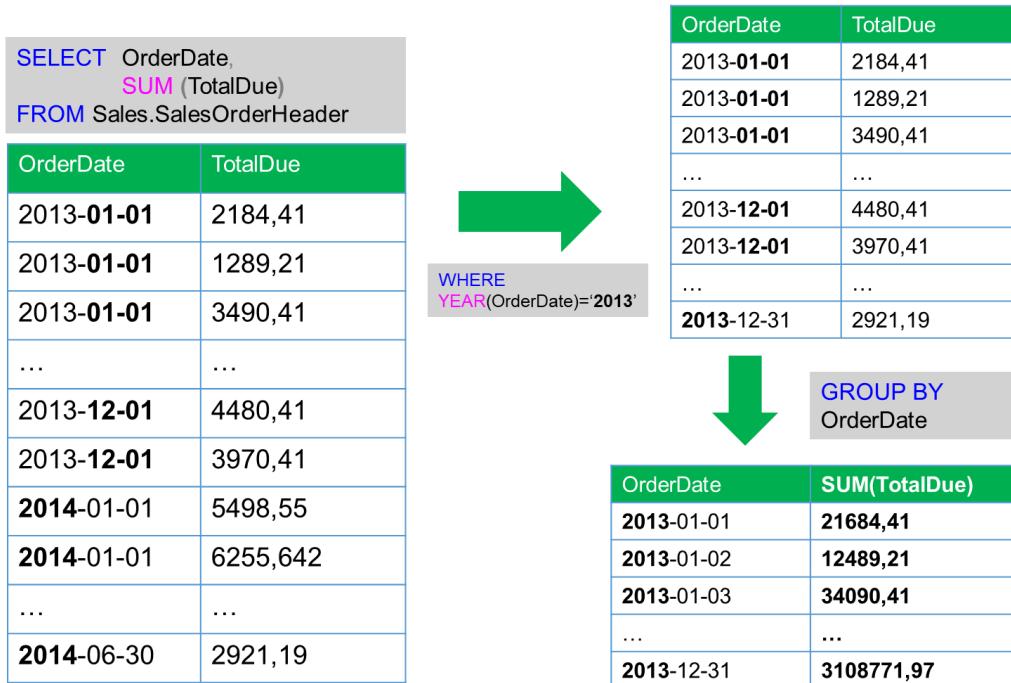
Fonte: Gustavo (2019).

5.2.1. Filtrando agrupamentos de dados

Outra característica, também devida à ordem lógica de processamento, diz respeito à realização do filtro da cláusula *WHERE* antes do conjunto de dados ser passado para a etapa de agrupamento, ou seja, da execução da cláusula *GROUP BY*. Dessa forma, ao contrário do que pode parecer, a cláusula *WHERE* não tem a capacidade de filtrar os grupos propriamente ditos, fazendo apenas o papel de filtro dos dados (tuplas) que serão agrupados na etapa posterior de execução da cláusula *GROUP BY* (o que pode ou não reduzir a quantidade de grupos exibidos).

No exemplo a seguir, podemos constatar que, na etapa de processamento da cláusula *WHERE*, todas as vendas diárias que não são do ano de 2013 (filtro *YEAR(OrderDate)='2013'*) não são passadas para a etapa de agrupamento. Dessa forma, na etapa de processamento da cláusula *GROUP BY*, os valores de todas as vendas do ano de 2013 são agrupados (sumarizados) por dia (*GROUP BY OrderDate*).

Figura 72 – Ordem de execução do WHERE com GROUP BY.



Fonte: Gustavo (2019).

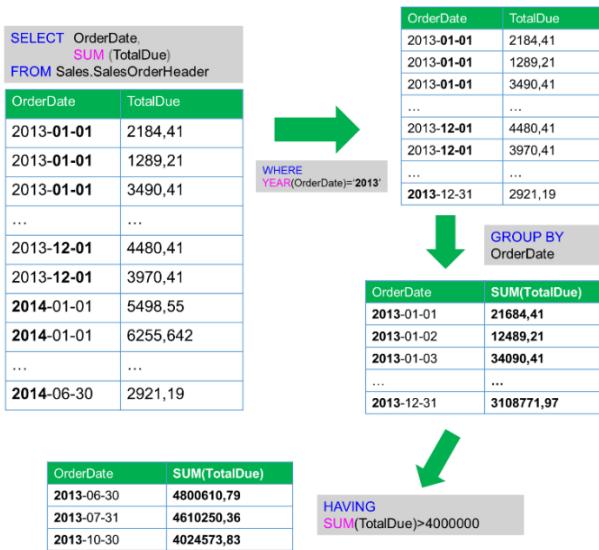
Devido a isso, a Linguagem SQL disponibiliza uma cláusula específica para possibilitar o filtro de agrupamentos de dados: a cláusula **HAVING**. Com ela sendo processada após a cláusula *GROUP BY*, e consequentemente após a cláusula *WHERE*, é possível filtrar os grupos usando inclusive o resultado das funções de agregação. Dessa forma, é muito importante ter claro em mente que a cláusula *HAVING* só opera (atua) nos grupos formados pela cláusula *GROUP BY*.

Sua sintaxe é muito semelhante à da cláusula *WHERE*, sendo possível utilizar também todos os operadores de comparação vistos anteriormente:

```
SELECT < lista de colunas / expressões >
FROM < tabela >
WHERE < condição de filtro – opcional >
GROUP BY < lista de colunas / expressões >
HAVING < condição de filtro de grupos – opcional >
ORDER BY < lista de colunas / expressões – opcional >
```

Assim sendo, usando a mesma query do exemplo da Figura 71 e incluindo a cláusula *HAVING* para filtrar e exibir somente os grupos de dias com vendas superiores a 4 milhões, no ano de 2013, teríamos a seguinte ordem de execução:

Figura 73 – Ordem de execução do HAVING com GROUP BY.



Fonte: Gustavo (2019).

Importante esclarecer que o uso das funções de agregação, na cláusula *HAVING*, não é obrigatório, bem como é possível usar as cláusulas *WHERE* e *HAVING* em conjunto. Entretanto, as colunas a serem usadas como filtro, na cláusula *HAVING*, precisam estar definidas na cláusula *GROUP BY* ou estarem sendo usadas em uma função de agregação. Outro ponto importante, e que também está relacionado com a ordem de execução das cláusulas, é em relação à impossibilidade do uso de alias na cláusula *HAVING*, visto que ela é processada antes da cláusula *SELECT* onde o alias é definido.

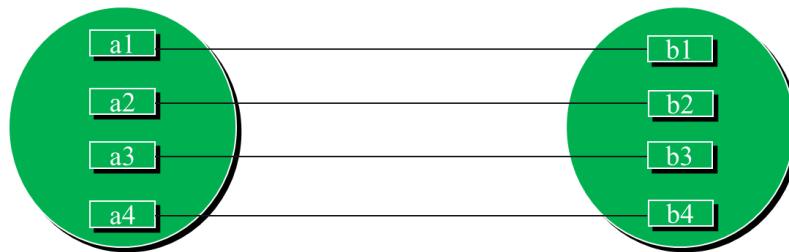
Capítulo 6. Junção de Tabelas (JOIN)

6.1. Introdução à Junção de Tabelas

A junção de tabelas está intrinsecamente inserida na teoria de banco de dados relacional, onde uma tabela pode estar relacionada a uma ou mais tabelas. Dessa forma, a Linguagem SQL vem prover, em termos de cláusulas e instruções, os recursos necessários para fazer esse relacionamento entre duas ou mais tabelas, ou seja, o famoso **JOIN** (junção).

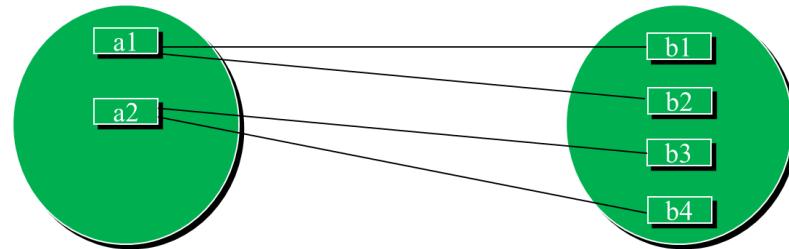
Antes de adentrarmos as cláusulas e mecanismos para se fazer a junção entre tabelas, é preciso lembrar que esse recurso (*join* entre tabelas) está diretamente relacionado com a **Teoria de Conjuntos**, onde elementos de um conjunto estão relacionados com elementos de outro conjunto, como mostrado nos exemplos abaixo.

Figura 74 – Relacionamento 1x1.



Fonte: Gustavo (2019).

Figura 75 – Relacionamento 1xN.



Fonte: Gustavo (2019).

Até agora, as queries que foram vistas não estavam considerando o relacionamento (*join*) entre as tabelas, ou seja, eram queries em apenas uma tabela (um conjunto de dados), como mostrado no exemplo abaixo.

Figura 76 – Queries sem Join.

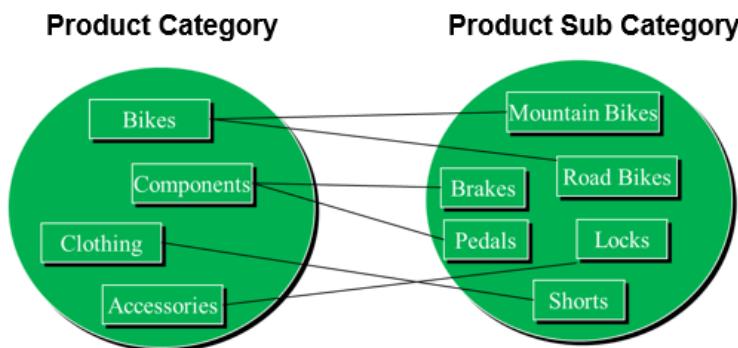


Fonte: Concurso INEP/ENADE (2017).

Na teoria de banco de dados relacional, para se realizar um relacionamento entre as tabelas, são usados os recursos de chave primária/secundária e chave estrangeira (*foreign key*). Dessa forma, a coluna de uma tabela é comparada com a coluna de outra tabela, construindo-se o elo desejado (tipos de joins que serão vistos mais à frente) entre elas.

A título de exemplo, suponha que tenhamos duas tabelas (conjuntos), como os mostrados abaixo, de *Categoria de Produtos* e *Sub Categoria de Produtos*:

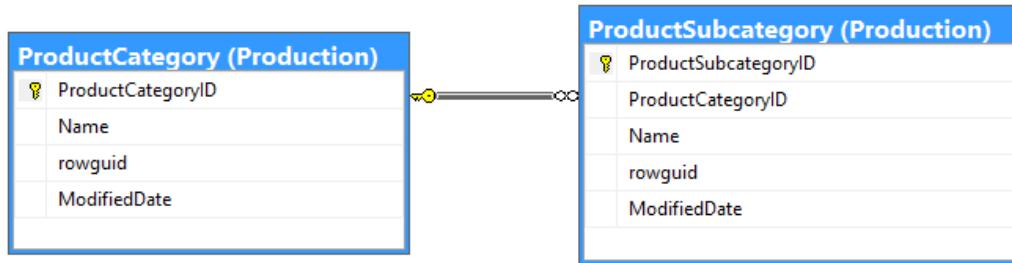
Figura 77 – Relacionamento Entre Dois Conjuntos.



Fonte: Gustavo (2019).

Olhando para o modelo relacional, teríamos algo como a figura mostrada abaixo, onde a chave primária da tabela de categoria de produtos é chave estrangeira na tabela de subcategorias de produtos (uma categoria de produtos tem uma ou mais subcategorias de produtos).

Figura 78 – Modelo Relacional.



Fonte: Gustavo (2019).

Com isso, para recuperarmos a lista de categorias de produtos e suas respectivas subcategorias, é necessário fazer um relacionamento entre essas duas tabelas. Um dos meios mais antigos e simples para se fazer essa operação é usando o operador de comparação “=” (padrão ANSI SQL-89), como mostrado no exemplo abaixo.

Figura 79 – Exemplo de JOIN com Operador de Comparação.

```
SELECT Production.ProductCategory.Name AS Nome_da_Categoria,
       Production.ProductSubCategory.Name AS Nome_da_SubCategoria
  FROM Production.ProductCategory, Production.ProductSubCategory
 WHERE Production.ProductCategory.ProductCategoryID = Production.ProductSubCategory.ProductCategoryID
 ORDER BY Nome_da_Categoria ASC, Nome_da_SubCategoria ASC;
```

Nome_da_Categoria	Nome_da_SubCategoria
Accessories	Bike Racks
Accessories	Bike Stands
Accessories	Bottles and Cages
Accessories	Cleaners
Accessories	Fenders
Accessories	Helmets
Accessories	Hydration Packs
Accessories	Lights
Accessories	Locks
Accessories	Panniers
Accessories	Pumps
Accessories	Tires and Tubes
Bikes	Mountain Bikes
Bikes	Road Bikes
Bikes	Touring Bikes
Clothing	Bib-Shorts
Clothing	Caps
Clothing	Gloves

Fonte: Gustavo (2019).

Perfeitamente factível também realizar o join entre mais de duas tabelas, como mostrado no exemplo abaixo, onde deseja-se listar os produtos e suas respectivas categorias e subcategorias. Perceba que para fazer o join entre mais de duas tabelas com o operador de comparação “=”, faz-se necessário usar em conjunto o operador lógico “AND”.

Figura 80 – Exemplo de JOIN entre três tabelas.

The screenshot shows a SQL query in the Query Editor and its results in the Results grid. The query joins three tables: Production.Product, Production.ProductCategory, and Production.ProductSubCategory. It selects the names of products, their subcategory, and category. The WHERE clause ensures that the product subcategory ID matches the product subcategory ID, and the product category ID matches the product category ID. The ORDER BY clause sorts the results by product name. The results grid displays 15 rows of data, mapping items like 'All-Purpose Bike Stand' to 'Bike Stands' under 'Accessories' in the 'Components' category.

Nome_da_Produto	Nome_da_SubCategoria	Nome_da_Categoria
All-Purpose Bike Stand	Bike Stands	Accessories
AWC Logo Cap	Caps	Clothing
Bike Wash - Dissolver	Cleaners	Accessories
Cable Lock	Locks	Accessories
Chain	Chains	Components
Classic Vest, L	Vests	Clothing
Classic Vest, M	Vests	Clothing
Classic Vest, S	Vests	Clothing
Fender Set - Mountain	Fenders	Accessories
Front Brakes	Brakes	Components
Front Derailleur	Derailleurs	Components
Full-Finger Gloves, L	Gloves	Clothing
Full-Finger Gloves, M	Gloves	Clothing
Full-Finger Gloves, S	Gloves	Clothing

Fonte: Gustavo (2019).

Para melhorar a visibilidade do código, deixando a query mais limpa, pode-se usar o recurso de *alias* tanto para as tabelas quanto para as colunas. Para o exemplo mostrado na Figura 80, o código ficaria da seguinte forma:

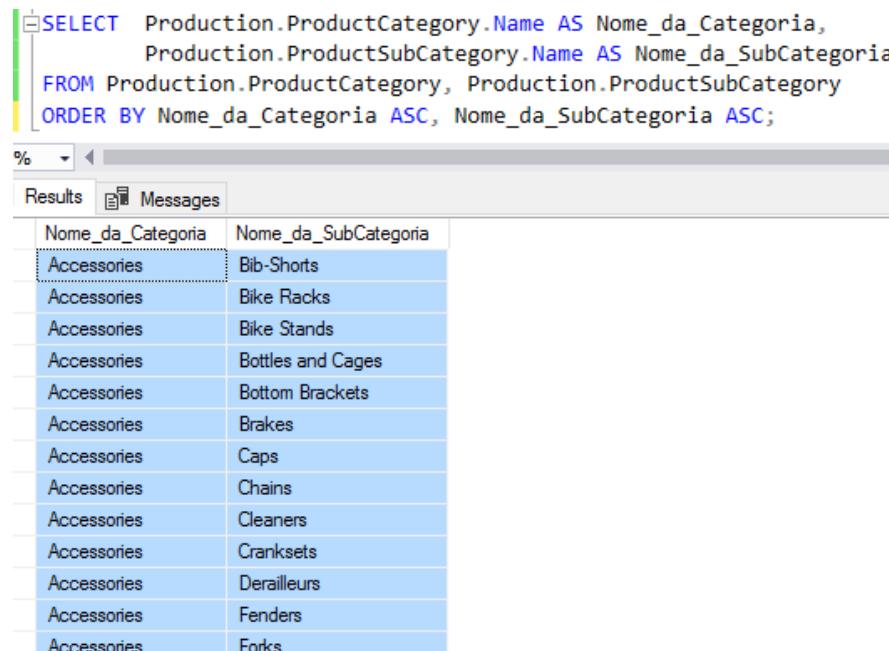
```
SELECT      P.Name AS Nome_da_Produto,  
            S.Name AS Nome_da_SubCategoria,  
            C.Name AS Nome_da_Categoria  
FROM        Production.Product P, Production.ProductCategory C,  
            Production.ProductSubCategory S  
WHERE       P.ProductSubCategoryID = S.ProductSubcategoryID  
AND         C.ProductCategoryID = S.ProductCategoryID  
ORDER BY    Nome_da_Produto ASC;
```

Ao se trabalhar com queries multitabelas, é muito importante conhecer o conceito de **produto cartesiano** e evitá-lo, uma vez que é muito oneroso para o servidor, em termos de consumo de recurso (CPU, memória RAM e I/O de disco).

Produto cartesiano é um conceito da Teoria de Conjuntos, consistindo no produto entre dois conjuntos. Dessa forma, se temos um conjunto com cinco itens e outro conjunto com seis itens, o produto cartesiano entre eles conterá 30 itens (5×6).

Em banco de dados relacional, um produto cartesiano ocorre quando tabelas são relacionadas em uma query sem se considerar qualquer relação lógica entre elas, ou seja, sem a cláusula/expressão do join. Em outras palavras, isso corresponde a não existência da cláusula *WHERE*, comparando-se os campos entre as tabelas. No exemplo da Figura 79, para gerar um produto cartesiano bastaria retirar a cláusula *WHERE*, como mostrado na figura abaixo. Sem informações sobre o relacionamento entre as tabelas, o processador de consultas do SQL Server exibirá todas as combinações possíveis de linhas, ou seja, todas as linhas da tabela de categoria serão relacionadas com todas as linhas da tabela de subcategoria.

Figura 81 – Exemplo de Produto Cartesiano (ANSI SQL-89).



The screenshot shows a SQL query window with the following code:

```
SELECT Production.ProductCategory.Name AS Nome_da_Categoria,
       Production.ProductSubCategory.Name AS Nome_da_SubCategoria
  FROM Production.ProductCategory, Production.ProductSubCategory
 ORDER BY Nome_da_Categoria ASC, Nome_da_SubCategoria ASC;
```

The results grid displays the Cartesian product of the two tables. The columns are 'Nome_da_Categoria' and 'Nome_da_SubCategoria'. The data shows that every category in the 'ProductCategory' table is paired with every subcategory in the 'ProductSubCategory' table.

Nome_da_Categoria	Nome_da_SubCategoria
Accessories	Bib-Shorts
Accessories	Bike Racks
Accessories	Bike Stands
Accessories	Bottles and Cages
Accessories	Bottom Brackets
Accessories	Brakes
Accessories	Caps
Accessories	Chains
Accessories	Cleaners
Accessories	Cranksets
Accessories	Derailleurs
Accessories	Fenders
Accessories	Forks

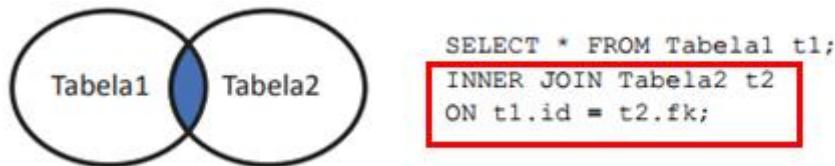
Fonte: Gustavo (2019).

6.2. INNER JOIN, CROSS JOIN e OUTER JOIN

No capítulo anterior, vimos a sintaxe para fazer join entre as tabelas usando a sintaxe do padrão ANSI SQL-89 com o operador “=”. Esse tipo de join é chamado, na Linguagem SQL, de **INNER JOIN**, e significa dizer que, para as linhas serem retornadas, os valores das colunas das tabelas relacionadas devem ser iguais. Na Teoria de Conjuntos, isso representa a operação de **interseção** entre dois conjuntos. Na revisão ANSI SQL-92, foram introduzidas, na Linguagem SQL, as cláusulas específicas para cada tipo de join, de forma a deixar as operações na cláusula *WHERE* apenas para os filtros desejados, e também de forma a evitar produtos cartesianos (mais fáceis de acontecer usando-se somente o operador “=”).

Dessa forma, para realizar um INNER JOIN entre duas tabelas, ou seja, uma operação de interseção, a sintaxe que pode ser usada, a partir do padrão ANSI SQL-92, é a mostrada na figura abaixo.

Figura 82 – INNER JOIN (ANSI SQL-92).



Fonte: Concurso INEP/ENADE (2017).

Usando essa nova sintaxe no exemplo mostrado no capítulo anterior na figura 79 (que usava o operador “=”), a query ficaria como a mostrada abaixo:

```
SELECT Production.ProductCategory.Name AS Nome_da_Categoria,
       Production.ProductSubCategory.Name AS Nome_da_SubCategoria
FROM Production.ProductCategory
INNER JOIN Production.ProductSubCategory
ON Production.ProductCategory.ProductCategoryID = Production.ProductSubCategory.ProductCategoryID
ORDER BY Nome_da_Categoria ASC, Nome_da_SubCategoria ASC;
```

Como resultado do processamento dessa query, todas as linhas de ambas as tabelas, que não satisfizerem à cláusula do join

(*ProductCategory.ProductCategoryID = ProductSubCategory.ProductCategoryID*), não serão retornadas.

Obs.: assim como na sintaxe ANSI SQL-89, usando-se o operador de comparação “=”, a ordem em que as tabelas são escritas em uma query com INNER JOIN, bem como a ordem que as comparações são feitas, não impactam no resultado da consulta. Da mesma forma que na sintaxe do padrão ANSI SQL-89 é perfeitamente factível realizar join entre mais de duas tabelas, usando-se a sintaxe da Linguagem SQL a partir do padrão ANSI SQL-92, como mostrado a seguir, onde o exemplo da Figura 80 foi adaptado para usar a cláusula INNER JOIN.

```
SELECT Production.Product.Name AS Nome_do_Produto,
       Production.ProductSubCategory.Name AS Nome_da_SubCategoria,
       Production.ProductCategory.Name AS Nome_da_Categoria
  FROM Production.Product
 INNER JOIN Production.ProductSubCategory
    ON Production.Product.ProductSubCategoryID = Production.ProductSubCategory.ProductSubcategoryID
 INNER JOIN Production.ProductCategory
    ON Production.ProductCategory.ProductCategoryID = Production.ProductSubCategory.ProductCategoryID
 ORDER BY Nome_do_Produto ASC;
```

Outro tipo de join que pode ser feito tanto usando a sintaxe ANSI SQL-89 quanto a sintaxe ANSI SQL-92, é o **SELF INNER JOIN**. Esse tipo de join nada mais é que o join de uma tabela com ela mesma, que ocorre muito frequentemente em tabelas que representam hierarquias ou quando deseja-se fazer correlações entre as linhas da mesma tabela. Um ponto importante é que esse tipo de join só é possível com a utilização de *alias* para a tabela, uma vez que o nome das tabelas relacionadas será o mesmo. No exemplo abaixo, com a sintaxe SQL-92, deseja-se listar os empregados e seus respectivos gerentes.

Figura 83 – SELF JOIN (ANSI SQL-92).

```
SELECT
    e.first_name + ' ' + e.last_name employee,
    m.first_name + ' ' + m.last_name manager
FROM
    sales.staffs e
INNER JOIN sales.staffs m ON m.staff_id = e.manager_id
ORDER BY
    manager;
```

employee	manager
Mireya Copeland	Fabiola Jackson
Jannette David	Fabiola Jackson
Kali Vargas	Fabiola Jackson
Marcelene Boyer	Jannette David
Venita Daniel	Jannette David
Genna Serrano	Mireya Copeland
Virgie Wiggins	Mireya Copeland
Layla Terrell	Venita Daniel
Bernardine Houston	Venita Daniel

Fonte: [SQLServerTutorial.NET \(2019\)](https://www.SQLServerTutorial.NET).

Para além disso, a Linguagem SQL também disponibiliza uma cláusula para que seja possível fazer um produto cartesiano utilizando a nova sintaxe. Essa cláusula é a **CROSS JOIN** e não exige a comparação (cláusula ON) entre as colunas das tabelas relacionadas. No código abaixo, o exemplo mostrado na Figura 80 foi reescrito usando a sintaxe ANSI SQL-92 para fazer o produto cartesiano.

```
SELECT      Production.ProductCategory.Name AS Nome_da_Categoria,
            Production.ProductSubCategory.Name AS Nome_da_SubCategoria
FROM Production.ProductCategory
CROSS JOIN Production.ProductSubCategory
ORDER BY Nome_da_Categoria ASC, Nome_da_SubCategoria ASC;
```

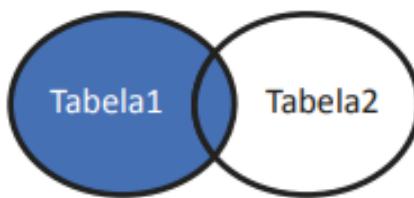
Já no **OUTER JOIN**, é possível retornar as linhas que atendam ao critério do join em ambas as tabelas (assim como acontece no INNER JOIN), **acrescentado de uma** das opções abaixo:

- Linhas da tabela à esquerda, para as quais não foram encontradas linhas correspondentes na tabela à direita → **LEFT OUTER JOIN**.
- Linhas da tabela à direita, para as quais não foram encontradas linhas correspondentes na tabela à esquerda → **RIGHT OUTER JOIN**.

- Linhas da tabela à esquerda, para as quais não foram encontradas linhas correspondentes na tabela à direita, **mais** as linhas da tabela à direita, para as quais não foram encontradas linhas correspondentes na tabela à esquerda → **FULL OUTER JOIN**.

Em resumo, com um OUTER JOIN consegue-se exibir todas as linhas de uma tabela juntamente com as linhas correspondentes entre as tabelas na cláusula de join. Para as linhas que não tiverem correspondência na outra tabela, a coluna retornada é representada com o valor *NULL*.

Figura 84 – LEFT OUTER JOIN (ANSI SQL-92).



Fonte: Concurso INEP/ENADE (2017).

No SQL Server, para se fazer uma operação de LEFT OUTER JOIN, pode-se suprimir a cláusula OUTER, como mostrado na sintaxe abaixo.

```
SELECT *
FROM TABELA1 T1
LEFT JOIN TABELA2 T2
ON T1.id = T2.id_fk;
```

Para exemplificar o LEFT OUTER JOIN, foi escrita a query abaixo para relacionar todos os produtos e seus valores nas vendas realizadas (incluindo os produtos que não tiveram vendas ainda). Note que para os produtos que não tiveram vendas ainda, a coluna *Valor_na_Venda* é retornada com o valor *NULL*.

```
SELECT P.Name AS Nome_Produto, S.UnitPrice AS Valor_na_Venda
```

FROM Production.Product P

LEFT JOIN Sales.SalesOrderDetail S

ON P.ProductID = S.ProductID

ORDER BY Nome_Produto ASC;

Nome_Produto	Valor_na_Venda
Adjustable Race	NULL
All-Purpose Bike Stand	159,00

É importante ressaltar que, ao contrário do que acontece no INNER JOIN, para o LEFT JOIN e para o RIGHT JOIN a ordem em que as tabelas são colocadas impacta diretamente no resultado retornado pela query. No exemplo acima, invertendo-se a ordem no join, o resultado seria outro: todas as vendas e seus respectivos produtos; ou seja, não retornaria produtos que não tiveram vendas.

SELECT P.Name AS Nome_Produto, S.UnitPrice AS Valor_na_Venda

FROM Sales.SalesOrderDetail S

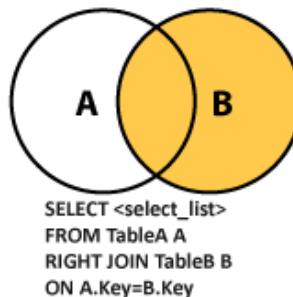
LEFT JOIN Production.Product P

ON P.ProductID = S.ProductID

ORDER BY Nome_Produto ASC;

Com base nessa ordem, podemos concluir que o RIGHT JOIN tem o efeito contrário do LEFT JOIN, como mostrado na operação de conjunto abaixo.

Figura 85 – RIGHT OUTER JOIN (ANSI SQL-92).



Fonte: Gustavo (2019).

Para exemplificar o RIGHT OUTER JOIN, vamos considerar a ordem das tabelas no exemplo acima, com a tabela *Product* à direita, para trazer todos os

produtos e seus valores de vendas, incluindo os produtos que ainda não tiveram vendas. Perceba que é possível obter o mesmo valor do exemplo da página anterior alterando a ordem das tabelas e o sentido do join.

Figura 86 – RIGTH OUTER JOIN (ANSI SQL-92).

```

SELECT P.Name AS Nome_Produto, S.UnitPrice AS Valor_na_Venda
FROM Sales.SalesOrderDetail S
RIGHT JOIN Production.Product P
ON P.ProductID = S.ProductID
ORDER BY Nome_Produto ASC, Valor_na_Venda ASC;

```

Nome_Produto	Valor_na_Venda
Adjustable Race	NULL
All-Purpose Bike Stand	159.00

Fonte: Gustavo (2019).

Importante mencionar que, assim como no INNER JOIN, é possível realizar SELF JOIN usando LEFT ou RIGHT JOIN. Dessa forma, no exemplo da Figura 82, se quiséssemos retornar também os funcionários que não têm gerente, ou seja, os gestores (coluna *manager* = *NULL*), a query ficaria como mostrado na figura a seguir.

Figura 87 – SELF LEFT JOIN (ANSI SQL-92).

```

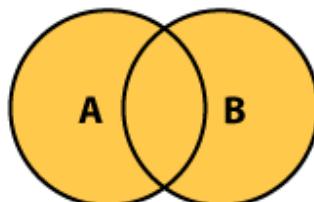
SELECT
    e.first_name + ' ' + e.last_name employee,
    m.first_name + ' ' + m.last_name manager
FROM
    sales.staffs e
LEFT JOIN sales.staffs m ON m.staff_id = e.manager id
ORDER BY
    manager;

```

employee	manager
Fabiola Jackson	NULL
Mireya Copeland	Fabiola Jackson
Jannette David	Fabiola Jackson
Kali Vargas	Fabiola Jackson
Marcelene Boyer	Jannette David
Venita Daniel	Jannette David
Genna Semano	Mireya Copeland
Virgie Wiggins	Mireya Copeland
Layla Terrell	Venita Daniel
Bernardine Houston	Venita Daniel

Fonte: SQLServerTutorial.NET (2019).

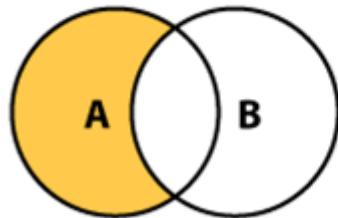
O último tipo de JOIN, o **FULL JOIN**, permite recuperar todas as linhas das tabelas envolvidas mesmo que não haja correspondência na cláusula da operação do join, como mostrado na operação de conjunto representada na figura abaixo:

Figura 88 – FULL OUTER JOIN (ANSI SQL-92).

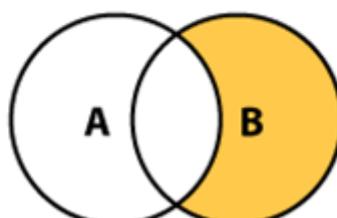
```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key=B.Key
```

Fonte: Gustavo (2019).

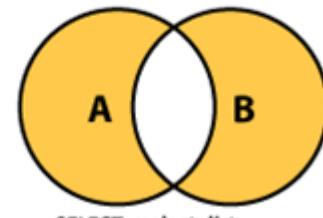
Para além disso, pode-se utilizar a cláusula *WHERE* em conjunto com a cláusula *JOIN*, de forma a executar as seguintes operações de conjuntos:

Figura 89 – Outras operações de conjunto.

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key=B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key=B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key=B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Fonte: Gustavo (2019).

Capítulo 7. Subconsultas

Uma subconsulta é uma instrução SELECT aninhada em outra consulta, ou seja, em outra instrução SELECT. A consulta aninhada, que é a subconsulta, é chamada de **consulta interna** (*inner query*). A consulta que contém a subconsulta é chamada de **consulta externa** (*outer query*).

O objetivo de uma subconsulta é retornar resultados para a consulta externa, de forma a permitir a construção de queries mais complexas, sendo um recurso amplamente usado em projetos de bancos de dados.

A forma dos resultados retornados pela subconsulta determinará se ela é uma **subconsulta escalar** ou uma **subconsulta multivalorada**. Uma subconsulta escalar, da mesma forma que uma função escalar, **retorna um único valor**. Já uma subconsulta multivalorada **retorna valores múltiplos** como se fosse uma tabela com uma coluna apenas. Além disso, a Linguagem SQL permite escrever **subconsultas** que estão **correlacionadas** com a consulta externa, referenciando uma ou mais colunas da consulta externa.

Nos capítulos a seguir, veremos cada tipo de subconsulta existente, sua aplicabilidade e exemplos.

7.1. Subconsulta escalar e multivalorada

Uma subconsulta **escalar** é uma instrução SELECT aninhada internamente a uma consulta externa, escrita de forma a **retornar um único valor**. Esse tipo de subconsulta pode ser usado em qualquer lugar de uma instrução SQL externa, em que uma expressão de valor único é permitida, como em uma instrução SELECT, WHERE, HAVING ou até mesmo em uma cláusula FROM.

Para escrever uma subconsulta escalar, devem ser observadas as seguintes regras que podem variar de acordo com o SGBD em questão:

- A subconsulta deve ser colocada entre **parênteses**.
- Podem existir **vários níveis de subconsultas**. Por exemplo, se tivermos uma consulta de 3 níveis, temos 1 consulta interna dentro de uma subconsulta externa, que, por sua vez, está dentro de uma consulta externa. No SQL Server, há suporte para até 32 níveis de consultas.
- Se a subconsulta retornar um **conjunto vazio**, o resultado da subconsulta será convertido e retornado como um **NULL**.

Uma dica muito útil para se construir subconsultas é escrever primeiramente a consulta mais interna, de forma a ir aumentando a complexidade da query gradativamente. Suponha que no nosso banco de exemplo deseja-se retornar os itens inclusos na última venda realizada. Primeiramente, para retornar a última venda realizada, a query ficaria algo como mostrado abaixo. Perceba que se trata de uma query que retornará apenas um valor (última ordem de venda), se tratando de uma query escalar:

```
SELECT MAX(SalesOrderID) AS Última_Venda
FROM Sales.SalesOrderHeader;
```

De posse dessa subconsulta, já podemos pensar em inseri-la na consulta externa, ficando algo como mostrado abaixo:

```
SELECT SalesOrderID, SalesOrderDetailID, OrderQty, ProductID, LineTotal
FROM Sales.SalesOrderDetail
WHERE SalesOrderID = (
    SELECT MAX(SalesOrderID) AS Última_Venda
    FROM Sales.SalesOrderHeader
)
```

SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	LineTotal
75123	121315	1	878	21.980000
75123	121316	1	879	159.000000
75123	121317	1	712	8.990000

Importante destacar que, precedendo a subconsulta escalar, devem ser usados operadores de comparação que exigem a comparação com um valor único. Esses operadores são: `=`, `!=`, `<`, `<=`, `>` e `>=`. Caso a consulta mais interna retorne mais de um valor, será exibido o erro como mostrado na figura abaixo, informando que não é permitido esse tipo de comparação.

Figura 90 – Erro em Subconsulta Escalar.

```
SELECT SalesOrderID, SalesOrderDetailID, OrderQty, ProductID, LineTotal
  FROM Sales.SalesOrderDetail
 WHERE SalesOrderID =      (   SELECT TOP(5) SalesOrderID AS Últimas_Cinco_Vendas
                                FROM Sales.SalesOrderHeader
                                ORDER BY SalesOrderID DESC
                           )
GO

% ▾
Results Messages
Msg 512, Level 16, State 1, Line 13
Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or
when the subquery is used as an expression.
```

Fonte: Gustavo (2019).

Além do uso na cláusula WHERE, é permitido também usar subconsulta escalar na cláusula HAVING, como mostrado no exemplo abaixo, onde deseja-se listar as vendas diárias do ano de 2013 com valores inferiores à média de vendas do ano anterior:

```
SELECT OrderDate AS Data_da_Venda,
       SUM(TotalDue) AS Valor_Total_Diário_de_Vendas
    FROM Sales.SalesOrderHeader
   WHERE YEAR(OrderDate)='2013'
 GROUP BY OrderDate
```

```
HAVING SUM(TotalDue) < (  
    SELECT AVG(TotalDue)  
    FROM Sales.SalesOrderHeader  
    WHERE YEAR(OrderDate)='2012'  
)  
ORDER BY 1 ASC;
```

Percebiam também que no exemplo anterior, podemos encontrar um tipo de subconsulta muito útil, chamado **subconsulta recursiva**. Com esse tipo de subconsulta, é permitido, em uma mesma query, executar um **SELECT** em uma **tabela e uma subconsulta nessa mesma tabela**.

Para as situações em que deseja-se que a subconsulta retorne mais de um valor, a Linguagem SQL fornece o tipo de **consulta multivvalorada**. Esse tipo de subconsulta pode ser usado, então, onde as subconsultas escalares não forem aplicáveis. Da mesma forma que na subconsulta escalar, a subquery (subconsulta) deve ser colocada entre parênteses e podem existir vários níveis de subconsultas, que são as chamadas **subconsultas aninhadas**.

Os operadores que podem ser usados com uma subconsulta multivvalorada são o **IN** e o **EXISTS**, e a mesma dica para a construção de queries com subconsulta escalar aplica-se também à construção de queries com subconsulta multivvalorada: escrever primeiramente a consulta mais interna, de forma a ir aumentando a complexidade da query gradativamente.

O operador IN verifica se a coluna ou expressão informada na cláusula WHERE possui valores que correspondam aos valores retornados pela subconsulta multivvalorada. Para exemplificarmos esse operador, suponha que se deseja retornar a identificação e o número da conta dos clientes da Austrália e França. A query ficaria algo como:

```
SELECT CustomerID, AccountNumber
```

```
FROM Sales.Customer  
WHERE TerritoryID IN (  
    SELECT TerritoryID  
    FROM Sales.SalesTerritory  
    WHERE Name ='Australia' OR Name = 'France'  
);
```

Percebam que nas situações em que houver uma relação entre as tabelas envolvidas em uma subconsulta multivalorada, como no exemplo anterior, a query pode ser reescrita de forma a utilizar join, sendo mais performática, como mostrado no exemplo a seguir:

```
SELECT C.CustomerID, C.AccountNumber  
FROM Sales.Customer C  
JOIN Sales.SalesTerritory T  
ON C.TerritoryID = T.TerritoryID  
WHERE T.Name ='Australia' OR T.Name = 'France'
```

Um operador que pode ser usado junto ao operador IN, mas que por questões de performance deve ser evitado, é o operador **NOT**. Com ele, pode-se inverter o resultado esperado da comparação feita com o operador IN, que, na prática, pode-se traduzir para “**não está contido**”. Assim sendo, com **NOT IN**, a verificação feita é a de que a coluna ou expressão informada na cláusula WHERE possua valores que **não** correspondam a nenhum dos valores retornados pela subconsulta multivalorada, o que exige a leitura da tabela inteira. Para exemplificar, suponha que agora se deseja retornar a identificação e o número da conta de todos os clientes, **exceto** os da Austrália e da França. A query ficaria algo como:

```
SELECT CustomerID, AccountNumber  
FROM Sales.Customer  
WHERE TerritoryID NOT IN (
```

```
SELECT TerritoryID  
FROM Sales.SalesTerritory  
WHERE Name ='Australia' OR Name = 'France'  
);
```

Assim como nas subconsultas escalares, além do uso na cláusula WHERE, é permitido também usar subconsultas multivaloradas na cláusula **HAVING**, seja com o operador **IN**, seja com os operadores **NOT IN**.

A título de exemplo, na query abaixo deseja-se listar o valor total das vendas dos dias no ano de 2013, que no ano anterior foram os 5 dias com maior valor de vendas.

```
SELECT OrderDate AS Data_da_Venda,  
       SUM(TotalDue) AS Valor_Total_Diário_de_Vendas  
  FROM Sales.SalesOrderHeader  
 WHERE YEAR(OrderDate)= '2013'  
 GROUP BY OrderDate  
 HAVING CAST (DAY(OrderDate) AS VARCHAR(2)) + '-' +  
           CAST(MONTH(OrderDate) AS VARCHAR(2))  
        IN (  
             SELECT TOP (5) CAST (DAY(OrderDate) AS VARCHAR(2)) + '-' +  
                   CAST(MONTH(OrderDate) AS VARCHAR(2))  
            FROM Sales.SalesOrderHeader  
           WHERE YEAR(OrderDate)= '2012'  
           GROUP BY OrderDate  
           ORDER BY SUM(TotalDue)  
        )
```

ORDER BY 1 ASC;

7.2. Subconsultas correlacionadas e com operadores de conjuntos

Nos exemplos de subconsultas mostrados anteriormente, sejam elas escalar ou multivalorada, podemos perceber que as subconsultas não possuem nenhuma dependência com a consulta externa e podem ser executadas isoladamente. Esse tipo de subconsulta é denominada **subconsulta independente**.

Já as **subconsultas correlacionadas**, que podem ser escritas como escalares ou multivaloradas, **não podem ser executadas separadamente** da consulta externa. Isso ocorre devido ao fato da **consulta externa passar um valor para a consulta interna** (subconsulta), para ser usado como um parâmetro na execução dela.

Outra peculiaridade é que, diferentemente das subconsultas independentes processadas uma vez, as subconsultas correlacionadas executam várias vezes, de forma que a consulta externa é executada primeiro e, para cada linha retornada, a consulta interna é processada.

Para clarificar, o exemplo a seguir usa uma subconsulta correlacionada para retornar as vendas mais recentes feitas por cada empregado. A subconsulta aceita um valor de entrada da consulta externa, usa a entrada em sua cláusula WHERE, e retorna um resultado escalar para a consulta externa.

```
SELECT Q1.SalesOrderID, Q1.SalesPersonID, Q1.OrderDate  
FROM Sales.SalesOrderHeader AS Q1  
WHERE Q1.OrderDate = (  
    SELECT MAX(Q2.OrderDate)  
    FROM Sales.SalesOrderHeader AS Q2  
    WHERE Q2.SalesPersonID = Q1.SalesPersonID
```

)

ORDER BY Q1.SalesPersonID, Q1.OrderDate;

Utilizando-se do recurso de subconsulta correlacionada, é possível escrever queries bem mais performáticas do que as escritas usando o operador IN. Para isso, usamos o operador **EXISTS** em conjunto com a subconsulta correlacionada. Esse operador, ao contrário do operador IN, interrompe a execução da subconsulta, assim que uma correspondência for encontrada. Com isso, estamos falando que o operador EXISTS, ao invés de recuperar um valor escalar ou uma lista com vários valores em uma subconsulta, ele simplesmente verifica se existe qualquer linha, no resultado, que satisfaça a condição de comparação com a consulta mais externa.

No exemplo a seguir, deseja-se retornar o ID e o bônus das pessoas que fizeram pelo menos uma venda no ano de 2011.

SELECT P.BusinessEntityID, P.Bonus

FROM Sales.SalesPerson P

WHERE EXISTS (

*SELECT **

FROM Sales.SalesOrderHeader V

WHERE V.SalesPersonID = P.BusinessEntityID

AND YEAR(OrderDate) = '2011'

)

ORDER BY 1 ASC;

Conceitualmente falando, o operador EXISTS é equivalente a recuperar os resultados contando as linhas retornadas e comparando se essa contagem é maior que zero.

SELECT P.BusinessEntityID, P.Bonus

FROM Sales.SalesPerson P

WHERE (

SELECT COUNT(*)

FROM Sales.SalesOrderHeader V

WHERE V.SalesPersonID = P.BusinessEntityID

AND YEAR(OrderDate) = '2011'

) > 0

ORDER BY 1 ASC;

Da mesma forma que o operador IN, o operador EXISTS pode ser usado com o operador **NOT**. Dessa forma, usando-se **NOT EXISTS**, consegue-se retornar os resultados para os quais não foram encontradas correspondências na subconsulta correlacionada.

Aproveitando o penúltimo exemplo, se desejássemos retornar as pessoas que não fizeram vendas no ano de 2011, bastaria adicionar o operador NOT, como mostrado a seguir.

SELECT P.BusinessEntityID, P.Bonus

FROM Sales.SalesPerson P

WHERE NOT EXISTS (

SELECT *

FROM Sales.SalesOrderHeader V

WHERE V.SalesPersonID = P.BusinessEntityID

AND YEAR(OrderDate) = '2011'

)

ORDER BY 1 ASC;

Da mesma forma, conceitualmente falando, os operadores NOT EXISTS são equivalentes a recuperar os resultados contando as linhas retornadas e comparando essa contagem com zero.

```
SELECT P.BusinessEntityID, P.Bonus  
FROM Sales.SalesPerson P  
WHERE (  
    SELECT COUNT(*)  
    FROM Sales.SalesOrderHeader V  
    WHERE V.SalesPersonID = P.BusinessEntityID  
    AND YEAR(OrderDate) = '2011'  
) = 0  
ORDER BY 1 ASC;
```

Além dos tipos de subconsultas vistos anteriormente, onde temos uma consulta externa e a consulta interna (subconsulta), a Linguagem SQL possui **operadores de conjuntos** para realizar operações entre duas ou mais **subconsultas independentes entre si**. Nesse tipo de subconsulta, ao contrário das subconsultas vistas anteriormente, onde a consulta externa não é independente (apenas a consulta interna), **todas as subconsultas são independentes e podem ser executadas separadamente**.

Para realizar as operações de conjunto **UNIÃO**, **INTERSEÇÃO** e **DIFERENÇA**, a linguagem T-SQL disponibiliza os operadores de conjunto **UNION**, **INTERSECT** e **EXCEPT**, respectivamente. Esses operadores, apesar de cada um se prestar a um propósito, possuem algumas características em comum:

- **Cada conjunto de entrada** é o resultado de uma consulta (subconsulta), que pode incluir qualquer instrução SELECT, mas **não pode possuir uma cláusula ORDER BY**. Caso seja necessário ordenar o resultado, o ORDER BY deve ser colocado na consulta geral (query completa com todos as subconsultas e os respectivos operadores de conjuntos).
- Os conjuntos de entrada devem ter o **mesmo número de colunas** e as colunas devem ter **tipos de dados compatíveis**. Se não forem inicialmente

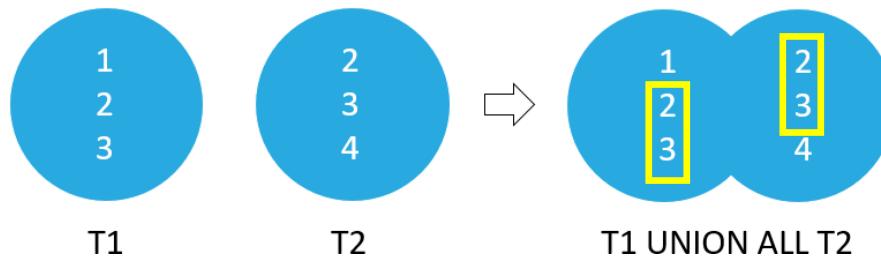
compatíveis, devem ser compatíveis através da conversão implícita (caso os tipos de dados a suportem de acordo com as regras de precedência de tipos de dados vistas na página 67) ou da conversão explícita usando CAST ou CONVERT.

- Um valor nulo (NULL) em um conjunto é tratado como igual a outro valor nulo em outro conjunto.

Isso posto, vamos inicialmente ver a operação de união entre duas ou mais subconsultas que pode ter duas vertentes:

- Usando o operador **UNION ALL**: não elimina as linhas duplicadas.

Figura 91 – Operação com UNION ALL.



Fonte: sqlitetutorial.net (2019).

A sintaxe para usar essa vertente é:

SELECT1

UNION ALL

SELECT2

UNION ALL

SELECT[n]

[ORDER BY]

No exemplo a seguir, deseja-se retornar todas as linhas, de forma que seja possível identificar aqueles médicos que também são pacientes do hospital e vice-versa:

SELECT nome, cpf, telefone

FROM Tab_Medicos

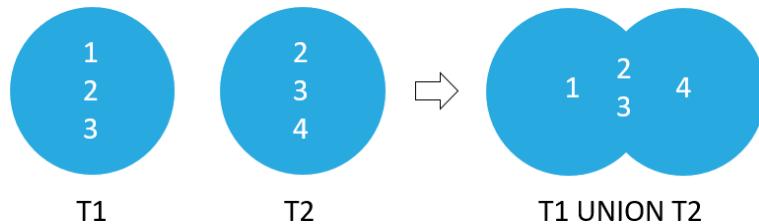
UNION ALL

SELECT nome, cpf, telefone

FROM Tab_Pacientes;

- Usando o operador **UNION**: exclui as linhas duplicadas.

Figura 92 – Operação com UNION.



Fonte: sqlitetutorial.net (2019).

A sintaxe para usar essa vertente é:

SELECT1

UNION

SELECT2

UNION

SELECT[n]

[ORDER BY]

Usando o mesmo exemplo anterior, mas supondo que agora quiséssemos apenas retornar a lista com nomes distintos das pessoas que estão cadastradas na base de dados do hospital, independentemente se é médico ou paciente, a query precisaria de usar o operador UNION para eliminar as linhas duplicadas.

SELECT nome, cpf, telefone

```
FROM Tab_Medicos
```

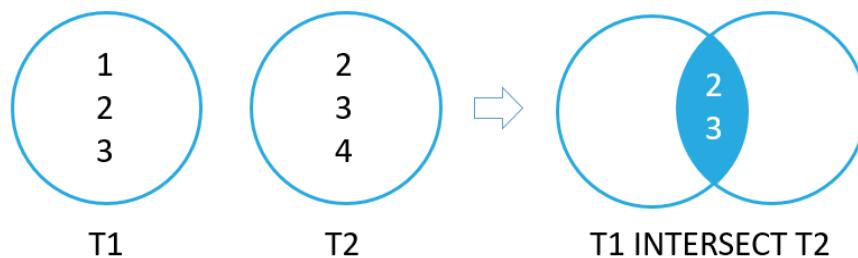
```
UNION
```

```
SELECT nome, cpf, telefone
```

```
FROM Tab_Pacientes;
```

Para realizar a operação de **interseção**, onde deseja-se encontrar os valores comuns a dois ou mais conjuntos, usa-se o operador **INTERSECT** da Linguagem T-SQL. Esse operador retornará apenas as linhas distintas que existirem em ambos os conjuntos, ou seja, linhas repetidas são eliminadas por padrão.

Figura 93 – Operação com INTERSECT.



Fonte: sqlitetutorial.net (2019).

A sintaxe é:

```
SELECT1
```

```
INTERSECT
```

```
SELECT2
```

```
INTERSECT
```

```
SELECT[n]
```

```
[ORDER BY]
```

Usando o mesmo exemplo anterior, mas supondo que agora quiséssemos retornar a lista com nomes distintos das pessoas que são médicos e pacientes, a query precisaria de usar o operador INTERSECT.

```
SELECT nome, cpf, telefone
```

```
FROM Tab_Medicos
```

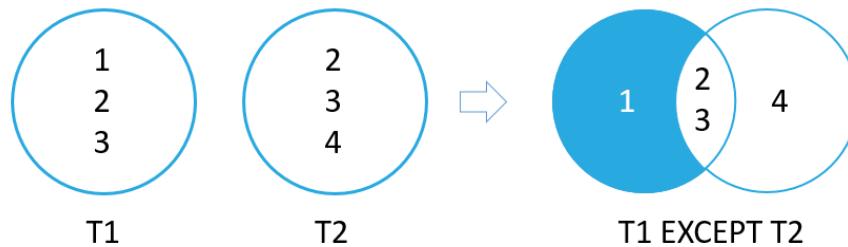
INTERSECT

```
SELECT nome, cpf, telefone
```

```
FROM Tab_Pacientes;
```

Por fim, para realizar a operação de **exceção (diferença)**, onde deseja-se retornar as linhas distintas da subconsulta de entrada à esquerda (ou acima na query) que não são retornadas pela subconsulta de entrada à direita (ou abaixo na query), usa-se o operador **EXCEPT** da Linguagem T-SQL.

Figura 94 – Operação com EXCEPT.



Fonte: [sqlitetutorial.net](https://www.sqlitetutorial.net) (2019).

A sintaxe é: **SELECT1**

EXCEPT

SELECT2

EXCEPT

SELECT[n]

[ORDER BY]

Usando o mesmo exemplo anterior, mas supondo que agora quiséssemos retornar a lista com nomes distintos das pessoas que são médicos e não são pacientes, a query precisaria de usar o operador EXCEPT.

```
SELECT nome, cpf, telefone
```

FROM Tab_Medicos

EXCEPT

SELECT nome, cpf, telefone

FROM Tab_Pacientes;

Importante observar que como se trata de uma operação de subtração, a **ordem dos conjuntos na operação irá afetar diretamente o resultado**. Dessa forma, no exemplo anterior, se quiséssemos retornar a lista com nomes distintos das pessoas que são pacientes e não são médicos, a ordem das subconsultas precisaria ser invertida:

SELECT nome, cpf, telefone

FROM Tab_Pacientes

EXCEPT

SELECT nome, cpf, telefone

FROM Tab_Medicos;

Capítulo 8. Inserção, Atualização e Exclusão de Dados

Ainda dentro da classe DML da Linguagem SQL, encontram-se os comandos para realizar a inserção, a atualização e a exclusão de dados, que serão vistos com mais detalhes nos capítulos a seguir.

8.1. Inserindo Dados

Para **inserir** uma ou mais linhas em uma tabela ou visão (será vista no capítulo 8), ação essa também referida como **persistir dados**, é usado o comando **INSERT**. Apesar de existirem outras formas de inserir dados em uma tabela, como usando os comandos de carga massiva de dados (operações de *bulkload*), o comando INSERT é o mais usado.

Sua sintaxe mais simples é mostrada abaixo, sendo que a lista de colunas é opcional, mas, ao informá-las, o código fica mais autoexplicativo e seguro, diminuindo o risco de inserção de valores de uma coluna em outra, e sendo possível também definir a ordem de inserção dos valores das colunas na tabela. Ao não informar a lista de colunas, é preciso inserir os valores para as colunas na ordem em que elas estão criadas na tabela.

INSERT [INTO] <Tabela ou visão> [lista_de_colunas]

VALUES (Valor | Expressão | NULL | DEFAULT)

Supondo que exista uma tabela de nome TAB1, com as colunas COL1 e COL2, ambas do tipo de dados numérico, o comando para inserir **uma linha** nessa tabela ficaria algo como:

INSERT INTO TAB1 (COL1, COL2)

VALUES (100, 500);

Omitindo a lista de colunas, ficaria como mostrado abaixo, indicando que o valor 100 seria inserido na primeira coluna da tabela TAB1 e o valor 500 na segunda coluna.

```
INSERT INTO TAB1
```

```
VALUES (100, 500);
```

Supondo agora que COL2 fosse do tipo de dados string, o valor ou expressão a ser inserida precisa ser colocada entre aspas simples (' '), como demonstrado abaixo:

```
INSERT INTO TAB1 (COL1, COL2)
```

```
VALUES (100, 'Valor não numérico');
```

Para as situações em que a coluna aceita valores nulos e deseja-se **omitir o valor a ser inserido**, deve-se especificar **NULL** na lista de valores a serem inseridos:

```
INSERT INTO TAB1 (COL1, COL2)
```

```
VALUES (100, NULL);
```

Nos casos em que a coluna foi criada com um **valor default** e deseja-se inserir esse valor, deve-se especificar **DEFAULT** na lista de valores a serem inseridos, como mostrado no exemplo abaixo:

Figura 95 – INSERT com DEFAULT.

```
CREATE TABLE TAB1 (COL1 int, COL2 int DEFAULT 99, COL3 int);
INSERT INTO TAB1
VALUES (10,DEFAULT,20),
(11,DEFAULT,21);
SELECT * FROM TAB1;
```

	COL1	COL2	COL3
1	10	99	20
2	11	99	21

Fonte: Gustavo (2019).

Usando apenas um comando INSERT, é possível inserir **mais de uma linha de uma só vez**, basta separar as várias cláusulas VALUES com vírgula. No nosso exemplo anterior, se desejássemos inserir 3 linhas, o comando ficaria como:

```
INSERT INTO TAB1 (COL1, COL2)  
VALUES    (100, 'Valor não numérico') ,  
          (200, 'Valor não numérico 2') ,  
          (300, 'Valor não numérico 3');
```

Uma possibilidade muito interessante do comando INSERT é **utilizar uma consulta SELECT** em outra tabela ou conjunto de tabelas para **retornar os valores a serem inseridos** na tabela em questão. No nosso exemplo, supondo que tivéssemos também a tabela TAB2 com as colunas COL3 (numérica), COL4 (numérica) e COL5 (string), e desejássemos inserir na TAB1 todos os valores da COL4 e COL5 da tabela TAB2, o comando ficaria como mostrado abaixo.

```
INSERT INTO TAB1 (COL1, COL2)  
SELECT COL4, COL5 FROM TAB2;
```

Na mesma linha de se usar uma consulta SELECT para inserir dados em uma tabela, pode-se também **executar uma procedure** (será vista com mais detalhes no capítulo 8), como mostrado no exemplo abaixo.

```
INSERT INTO TAB1 (COL1, COL2)  
EXEC PROC1;
```

Outra possibilidade de inserir dados em uma tabela é usando a cláusula **SELECT INTO**. Essa cláusula é similar à INSERT INTO SELECT, com a exceção de que a tabela destino dos dados não pode estar criada, visto que a cláusula **SELECT INTO criará a tabela em tempo de execução**.

Essa nova tabela é criada com a estrutura física definida pela lista de colunas na cláusula SELECT. Cada coluna na nova tabela terá o mesmo nome, tipo de dados e anulabilidade da coluna correspondente (ou expressão) da lista de colunas na cláusula SELECT. Entretanto, **índices e constraints não são criados na nova tabela**. A sintaxe para essa opção é mostrada abaixo.

SELECT coluna1, coluna2, ...

INTO Nova_Tabela FROM Tabela_Origem;

Figura 96 – SELECT INTO.

The screenshot shows a SQL Server Management Studio window with three queries and their results. The top query is:

```
SELECT * FROM TAB1;
```

The middle query is:

```
SELECT *
INTO TAB2 FROM TAB1;
```

The bottom query is:

```
SELECT * FROM TAB2;
```

Below the queries, there are two result grids labeled 'Results' and 'Messages'. The 'Results' grid contains two rows of data:

	COL1	COL2	COL3
1	10	99	20
2	11	99	21

The 'Messages' grid is empty.

Fonte: Gustavo (2019).

8.2. Atualizando Dados

Para alterar dados existentes em uma tabela ou visão, a Linguagem SQL fornece a instrução **UPDATE**. Essa instrução opera em um conjunto de linhas definido por uma condição em uma cláusula WHERE ou join, que possui a mesma estrutura que uma cláusula WHERE em uma instrução SELECT e pode-se valer dos mesmos

recursos. Para atribuir o novo valor à coluna, é usada a cláusula **SET**, que permite também a atualização dos valores de uma ou mais colunas, separadas por vírgulas.

Dessa forma, a sintaxe da instrução UPDATE, em linhas gerais, é:

UPDATE <Nome_Tabela>

SET <Coluna1> = { valor | expressão | DEFAULT | NULL },

<Coluna2> = { valor | expressão | DEFAULT | NULL },

<ColunaN> {...n}

WHERE <Condição>;

Considerando os valores da tabela TAB2 do exemplo mostrado na figura 96, se quiséssemos atualizar os valores da coluna COL3 para 500, de todas as linhas onde o valor da coluna COL1 fossem iguais a 10, o comando SQL ficaria algo como:

UPDATE TAB2

SET COL3 = 500

WHERE COL1 = 10;

Importante destacar que se não for especificada a cláusula WHERE, a atualização será feita para todas as linhas da tabela. Dessa forma, omitindo a cláusula WHERE no exemplo anterior, não mais atualizariam os apenas as linhas onde o valor da coluna COL1 fossem iguais a 10, e sim todas as linhas da tabela.

UPDATE TAB2

SET COL3 = 500;

Como informado inicialmente, é perfeitamente possível atualizar mais de uma coluna de uma vez, bastando separá-las por vírgula. Dessa forma, se, no exemplo, quiséssemos atualizar os valores da coluna COL3 para 500 e da coluna COL1 para 600, de todas as linhas onde o valor da coluna COL1 fosse igual a 10, o comando SQL ficaria algo como:

UPDATE TAB2

SET COL3 = 500, COL1 = 600

WHERE COL1 = 10;

8.3. Excluindo dados

Para excluir dados existentes em uma tabela ou visão, a Linguagem SQL fornece a instrução **DELETE**. Essa instrução, assim como a cláusula UPDATE, opera em um conjunto de linhas definido por uma condição em uma cláusula WHERE ou join, que possui a mesma estrutura que uma cláusula WHERE em uma instrução SELECT e pode-se valer dos mesmos recursos. Sua sintaxe na forma mais simples é:

DELETE FROM <Nome_Tabela>

WHERE <Condição>;

Considerando os valores da tabela TAB2 do exemplo mostrado na figura 96, se quiséssemos excluir as linhas onde o valor da coluna COL1 fossem iguais a 10, o comando SQL ficaria algo como:

DELETE FROM TAB2

WHERE COL1 = 10;

Importante destacar que **se não for especificada a cláusula WHERE, todas as linhas da tabela serão excluídas**. Dessa forma, omitindo a cláusula WHERE no exemplo anterior, não mais excluiríamos apenas as linhas onde o valor da coluna COL1 fossem iguais a 10, e sim todas as linhas da tabela.

DELETE FROM TAB2;

Para as situações em que deseja-se excluir todos os dados de uma tabela, por questões de performance, é preferível usar o comando **TRUNCATE** da classe DDL da Linguagem SQL. Esse comando remove todas as linhas de uma tabela sem registrar as exclusões de linhas individualmente, **consumindo menos recursos** do servidor de banco de dados e executando **mais rápido** que o DELETE sem a cláusula WHERE. Sua sintaxe é *TRUNCATE TABLE <Nome_da_Tabela>*.

Assim como no comando UPDATE, é possível utilizar uma subconsulta ou JOIN para retornar/obter os valores para as condições de exclusão, como mostrado nos exemplos abaixo retirados da documentação da Microsoft:

--Usando subconsulta

```
DELETE FROM Sales.SalesPersonQuotaHistory  
WHERE BusinessEntityID IN (  
    SELECT BusinessEntityID  
    FROM Sales.SalesPerson  
    WHERE SalesYTD > 2500000  
)
```

--Usando join

```
DELETE FROM Sales.SalesPersonQuotaHistory  
FROM Sales.SalesPersonQuotaHistory AS spqh  
INNER JOIN Sales.SalesPerson AS sp  
ON spqh.BusinessEntityID = sp.BusinessEntityID  
WHERE sp.SalesYTD > 2500000;
```

Para além disso, e contemplando as necessidades de certas operações no banco de dados onde é preciso **atualizar, inserir ou excluir linhas** de uma tabela, existe o comando **MERGE**. Com essa instrução é possível inserir, atualizar e até

excluir linhas de uma tabela, com base em um join com um conjunto de dados de origem, tudo **em uma única instrução**.

O comando MERGE atua nos dados de uma tabela destino, com base em uma ou mais condições de decisão mostradas abaixo, acerca de qual operação (INSERT/UPDATE/DELETE) será executada:

- Quando os dados de origem correspondem aos dados no destino, ocorre a atualização dos dados.
- Quando os dados de origem não têm correspondência no destino, ocorre a inserção desses dados.
- Quando os dados de destino não têm correspondência no conjunto de dados de origem, ocorre a exclusão dos dados na tabela de destino.

A sintaxe do comando MERGE na sua forma mais simples é:

MERGE

[INTO] <Tabela_Destino>

USING <Conjunto de Dados de Origem>

ON <Condição de Comparação para o Merge >

[WHEN MATCHED

THEN <Operação Quando Há Correspondência>]

[WHEN NOT MATCHED BY TARGET

THEN < Operação Quando Não Há Correspondência no Destino>]

[WHEN NOT MATCHED BY SOURCE

THEN < Operação Quando Não Há Correspondência na Origem>]

Para exemplificar, na query abaixo deseja-se atualizar os campos Data_Venda e Valor quando a venda já existir na tabela, e inserir a linha (colunas

`Id_Venda, Origem.Data_Venda, Origem.Produto, Origem.Qtde e Origem.Valor)`
quando a venda ainda não existir:

`MERGE TAB_Venda_Destino AS Destino`

`USING TAB_Venda_Origem AS Origem ON (Origem.Id_Venda = Destino.Id_Venda)`

`WHEN MATCHED THEN -- Registro existe nas 2 tabelas`

`UPDATE SET`

`Destino.Data_Venda = Origem.Data_Venda,`

`Destino.Valor = Origem.Valor`

`WHEN NOT MATCHED THEN -- Registro não existe no destino e deve ser inserido`

`INSERT`

`VALUES (Origem.Id_Venda, Origem.Data_Venda, Origem.Produto, Origem.Qtde,
Origem.Valor);`

Capítulo 9. Linguagem de Controle de Transação (TCL)

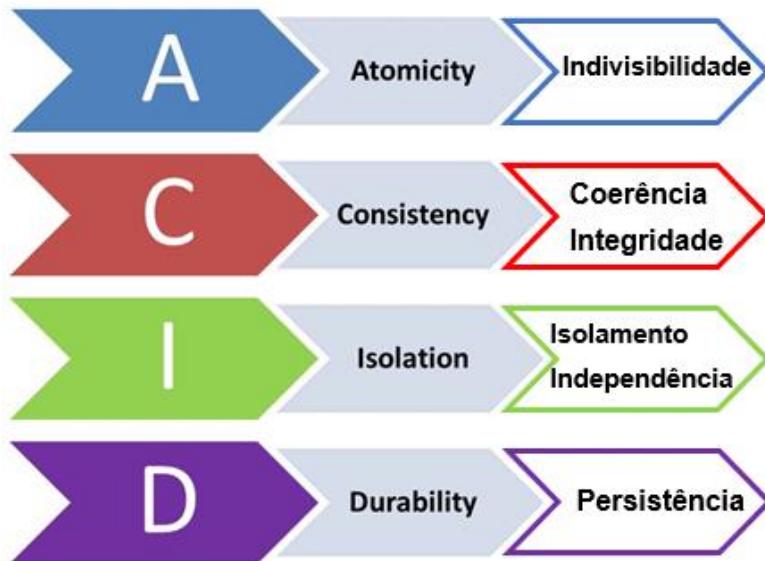
9.1. Conceitos Básicos de Transação

As características que mais contribuíram para o respaldo, robustez e integridade dos sistemas gerenciadores de bancos de dados relacionais foram as suas **propriedades ACID**. Essas propriedades referem-se, em grande parte, às **transações** que são executadas no SGBD, que nada mais são do que uma **sequência de instruções SQL**.

Essas propriedades funcionam como regras para as transações executadas em um SGBD relacional, tendo o seguinte significado:

- **ATOMICIDADE:** requer que as transações sejam “**tudo ou nada**” (“*all or nothing*”), ou seja, se uma parte da transação falhar, a transação inteira deve falhar e o estado do banco de dados permanece inalterado. Em outras palavras, a **transação será executada totalmente ou não será executada**, garantindo, assim, que as transações sejam **atômicas**.
- **CONSISTÊNCIA:** garante que qualquer transação **levará o banco de dados de um estado válido a outro**. Em caso de sucesso, a transação cria um novo estado válido dos dados ou, em caso de falha, retorna todos os dados ao estado imediatamente anterior ao início da transação.
- **ISOLAMENTO:** garante que execuções concorrentes (em paralelo) de transações resultem em um estado do banco de dados que seria obtido caso as transações fossem executadas serialmente. Isso ocasiona que **uma transação em andamento, mas ainda não validada, permanece isolada de qualquer outra operação**, garantindo que a transação não sofra interferência de nenhuma outra transação concorrente.
- **DURABILIDADE:** garante que, uma vez que uma transação tenha sido confirmada, ela permanecerá assim, mesmo no caso de um reinício do sistema, crash, falta de energia ou erros posteriores.

Figura 97 – Propriedades ACID.



Fonte: Gustavo (2019).

Dois termos muito falados quando o assunto é transação são o **COMMIT** e o **ROLLBACK**. Mencionamos commit quando queremos dizer que a transação foi validada, confirmada, persistida. Já o rollback é mencionado nas situações em que a transação não foi confirmada/persistida e precisou ser desfeita.

Em termos de criação, as transações podem ocorrer de duas formas:

- **EXPLICITAMENTE:** iniciadas através de comandos da Linguagem de Controle de Transação (*Transaction Control Language – TCL*), uma classe da Linguagem SQL para controlar transações.
- **IMPLICITAMENTE:** também chamadas de transações com autocommit. As instruções enviadas separadamente para execução são automaticamente inseridas em uma transação pelo SGBD. Essas transações são confirmadas (“commitadas”) automaticamente quando a instrução é bem-sucedida, ou revertidas automaticamente quando a instrução encontra um erro em tempo de execução. É como se o SGBD abrisse a transação explicitamente e acompanhasse o status dela para saber se a confirma no banco de dados ou se a desfaz, tudo isso de forma transparente para o usuário.

Muitos SGBDs, como o SQL Server, permitem os dois modos, mas alguns, como o Oracle, só permitem o modo explícito, sendo fundamental para quem construirá aplicações com a linguagem SQL, conhecer o **commit mode** do SGBD com o qual irá trabalhar.

9.2. Controle de Transações

As transações explícitas são controladas (criadas/confirmadas/desfeitas) usando os comandos da classe TCL, sendo que cada SGBD pode implementar variações ou novos comandos para o propósito de trabalho com transação.

Para iniciar uma transação, é usado o comando **BEGIN TRANSACTION**. Depois de iniciada uma transação, ela deve ser finalizada adequadamente, usando **COMMIT** em caso de sucesso e se desejar persistir a transação, ou **ROLLBACK** em caso de falha ou desistência de persistência da transação.

- **Iniciando e confirmando uma transação:**

BEGIN TRANSACTION;

DELETE FROM HumanResources.JobCandidate

WHERE JobCandidateID = 13;

COMMIT;

- **Iniciando e desfazendo uma transação:**

BEGIN TRANSACTION;

INSERT INTO ValueTable VALUES(1);

INSERT INTO ValueTable VALUES(2);

ROLLBACK;

Importante ressaltar que, dada a propriedade ACID isolamento, objetos envolvidos em uma transação de uma sessão de usuário não estão disponíveis para outras sessões enquanto a transação não for finalizada. Dessa forma, ao trabalhar com transações explícitas, é muito importante ficar atento para não deixar transações

abertas no banco de dados sem nada em execução, simples e puramente por falta do devido tratamento (commit/rollback) a uma transação aberta, pois isso pode causar uma fila enorme de espera devido aos “bloqueios” nos objetos.

Uma possibilidade ao se trabalhar com transações é utilizá-las de forma aninhada em N níveis, como o exemplo mostrado abaixo:

BEGIN TRANSACTION;

```
UPDATE table1 ...;  
BEGIN TRANSACTION;  
    UPDATE table2 ...;  
    SELECT * from table1;  
    COMMIT;  
    UPDATE table3 ...;
```

COMMIT;

Para além disso, um recurso disponível na maioria dos SGBDs é o de nomear transações e usar esses nomes para “commitá-las” a qualquer momento:

BEGIN TRANSACTION T1;

```
    UPDATE table1 ...;  
BEGIN TRANSACTION T2;  
        UPDATE table2 ...;  
        SELECT * from table1;  
COMMIT TRANSACTION T2;  
        UPDATE table3 ...;  
COMMIT TRANSACTION T1;
```

Capítulo 10. Linguagem de Controle de Acesso a Dados (DCL)

10.1. Segurança em Bancos de Dados

Uma preocupação constante, principalmente na era da informação que nos encontramos, é em relação à proteção dos dados armazenados. Na camada de banco de dados, essa proteção deve ser feita externamente ao SGBD (proteção do servidor, firewall, segmentação de rede, mecanismos para evitar invasões e ataques de negação de serviço etc.), bem como internamente.

Dentro do SGBD, a proteção aos objetos e dados, em termos de permissão de acesso ou permissão para executar uma ação (comando), é configurada através da **Linguagem de Controle de Acesso a Dados** (*Data Control Language - DCL*), uma classe da Linguagem SQL também. Dentro dessa linguagem, os comandos mais conhecidos e utilizados são:

- **GRANT**: comando DCL usado para conceder permissões em objetos ou permissões para executar algum comando.
- **REVOKE**: comando DCL usado para remover permissões em objetos ou permissões para executar algum comando.

Em relação aos objetos que podem ser protegidos, os mecanismos e a granularidade de proteção, cada SGBD disponibiliza sua gama de recursos e comandos para habilitar, configurar e gerenciar seu framework de segurança. A título de exemplo, o SQL Server fornece os componentes abaixo para controlar quais usuários podem fazer log-on no SQL Server, quais dados eles podem acessar e quais operações eles podem executar:

- **PERMISSION (Privilégio)**: permissão para executar alguma ação, ou seja, executar algum comando SQL.
- **ROLE (Papel)**: conjunto de privilégios que pode ser concedido a usuários ou a outras roles. São criadas com o comando CREATE ROLE da classe DDL da Linguagem SQL.

- **PRINCIPALS (Entidades):** usuários, grupos ou processos com acesso concedido a uma instância do SQL Server, no nível do servidor ou do banco de dados. As entidades no nível do servidor incluem logins e roles de servidor (*server roles*). As entidades no nível do banco de dados incluem usuários e roles de banco de dados (*database roles*). Também são criados usando-se o comando CREATE da classe DDL.

- Exemplo de comando para criar uma *PRINCIPAL* do tipo login:

```
CREATE LOGIN UsrIGTI
```

```
WITH PASSWORD = 'Pa55w.rd',
```

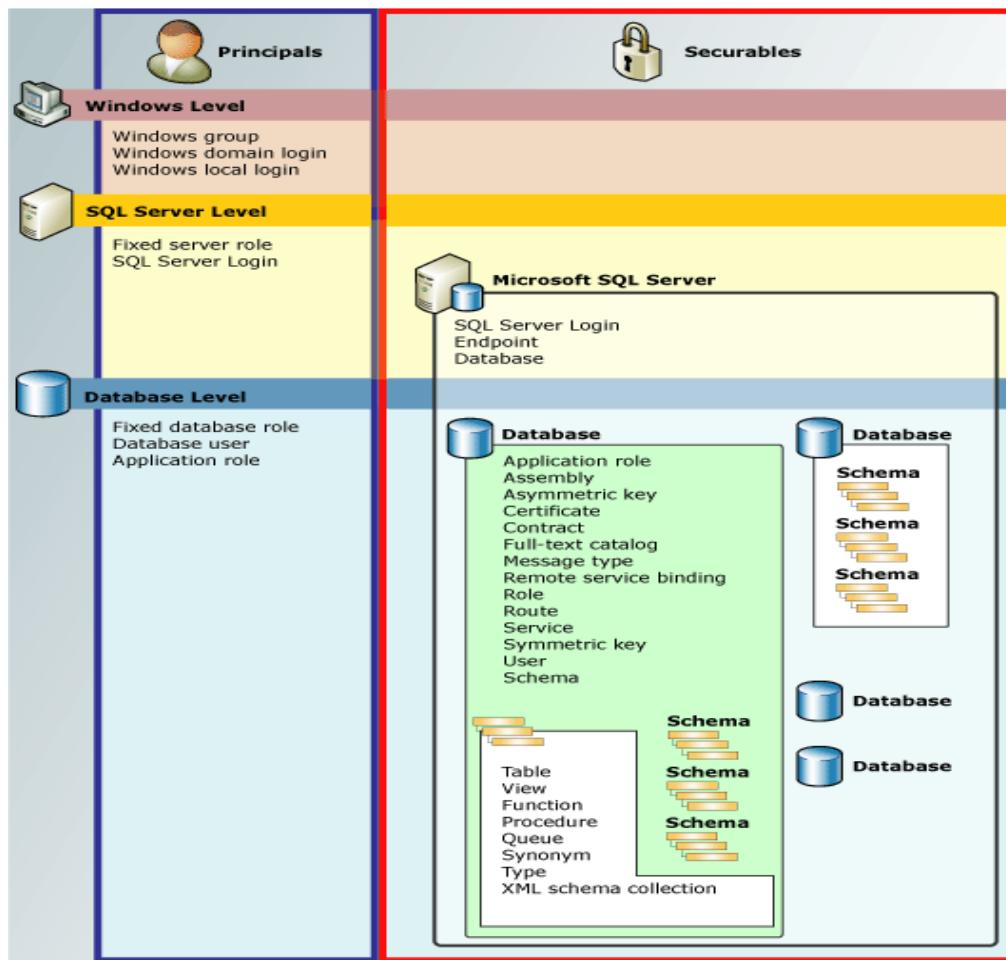
```
DEFAULT_DATABASE = [BDIGTI];
```

- **SECURABLES:** objetos, no nível do servidor ou do banco de dados, nos quais são concedidas as permissões para os *principals*.

Figura 98 – Principals, Permissions e Securables.



Fonte: Microsoft (2008).

Figura 99 – Principals e Securables.

Fonte: Microsoft (2008).

10.2. Concessão e revogação de privilégios

As permissões em bancos de dados relacionais são gerenciadas usando-se as instruções **GRANT** e **REVOKE** da classe DCL da Linguagem SQL. As permissões que se aplicam a tabelas ou a views, para acessar ou manipular dados, são **SELECT**, **INSERT**, **UPDATE** e **DELETE**.

No exemplo abaixo, está sendo concedida a permissão de **SELECT** na tabela *Address* do schema (owner) *Person*, para o usuário *UsrAdventureSystem*.

GRANT SELECT ON [Person].[Address] TO UsrAdventureSystem;

Já no exemplo abaixo, estão sendo concedidas as permissões de *SELECT*, *INSERT*, *DELETE* e *UPDATE* na tabela *Product* do schema (owner) *Production*, para o usuário *UsrAdventureSystem*.

GRANT SELECT, INSERT, DELETE, UPDATE ON [Production].[Product] TO UsrAdventureSystem;

Uma opção bem útil na instrução GRANT é permitir que o usuário que recebeu uma permissão possa concedê-la para outro usuário. Essa possibilidade é feita com a cláusula **WITH GRANT OPTION**. No exemplo abaixo, está sendo concedida a permissão de *DELETE* na tabela *DatabaseLog* para o usuário *UsrAdventureSystem*, permitindo que ele possa conceder essa permissão para outro usuário.

GRANT DELETE ON DatabaseLog

TO UsrAdventureSystem WITH GRANT OPTION;

Da mesma forma que as permissões para permitir a execução de comandos DML, estão disponíveis as permissões para permitir a execução de comandos DDL. No exemplo abaixo, está sendo concedida a permissão de criar uma tabela para o usuário *UsrAdventureSystem*.

GRANT CREATE TABLE TO UsrAdventureSystem;

Por outro lado, para se remover uma permissão, usa-se a cláusula **REVOKE**, cuja sintaxe pode ser observada no exemplo abaixo, onde está sendo removida a permissão de *DELETE* na tabela *DatabaseLog*, do usuário *UsrAdventureSystem*.

REVOKE DELETE ON DatabaseLog FROM UsrAdventureSystem;

Muitas outras permissões e opções de segurança podem ser configuradas no SGBD, mas para efeitos do padrão ISO da Linguagem SQL, as instruções GRANT e REVOKE são suficientes para entender a funcionalidade da classe DCL.

Capítulo 11. Programação com a Linguagem T-SQL

11.1. Visões (Views) e CTEs

Uma view, em linhas gerais, é uma **query armazenada**, criada como um objeto no banco de dados. Ao contrário das tabelas, uma view não armazena dados e são vistas como tabelas virtuais dinâmicas para exibir dados específicos de uma ou mais tabelas/views. Em outras palavras, a **view é definida por uma query no momento da sua criação**, que especifica exatamente os dados que essa view irá retornar.

Views são criadas com o comando **CREATE VIEW** da classe DDL da Linguagem SQL, podendo ser alteradas pelo comando **ALTER VIEW** e excluídas através do comando **DROP VIEW**.

As vantagens de se usar views são:

- Simplificam os relacionamentos complexos entre tabelas, mostrando apenas os dados relevantes e ajudando os usuários a se concentrarem no(s) subconjunto(s) de dados relevante(s) para eles, ou com os quais eles têm permissão para trabalhar. Dessa forma, os usuários não precisam ver as consultas complexas que estão por trás da criação da view, trabalhando de forma transparente como se fosse uma única tabela.
- Fornecem segurança, permitindo que os usuários vejam apenas o que eles estão autorizados a ver. Views podem ser utilizadas para limitar o acesso a determinados conjuntos de dados. Incluindo na query de criação da view, apenas os dados que os usuários estão autorizados a ver; os dados privados são mantidos em sigilo. As visualizações são amplamente usadas também como um mecanismo de segurança, fornecendo aos usuários acesso aos dados através da view, mas não concedendo permissões diretamente às tabelas base da view.

- Fornece uma camada de abstração, além de compatibilidade com versões anteriores do schema físico, caso as tabelas base mudem.
- **Evita problemas de performance causado por filtros indevidos ou ausentes nas queries que consultam diretamente as tabelas base da view;**
- No caso de **views indexadas, oferecem ganhos significativos de performance.**

A título de exemplo, no código abaixo está sendo criada uma view de nome VW_Clientes_Australia para retornar o ID e o Número da Conta dos clientes que são da Austrália:

```
CREATE VIEW VW_Clientes_Australia AS
    SELECT CustomerID, AccountNumber
    FROM Sales.Customer
    WHERE TerritoryID IN (
        SELECT TerritoryID
        FROM Sales.SalesTerritory
        WHERE Name ='Australia'
    );
```

Após criada a view, pode-se utilizá-la como se fosse uma tabela, inclusive participando de joins, agrupamentos e novos filtros.

```
SELECT *
    FROM VW_Clientes_Australia
    WHERE CustomerID BETWEEN 1 AND 100;
```

Uma observação importante é que a **query de criação da view não pode contar a cláusula ORDER BY**. Dessa forma, quando desejar ordenar os dados da

view, deve-se usar a cláusula ORDER BY na query que utilizará a view para retornar os dados, como mostrado abaixo:

```
SELECT *  
FROM VW_Clientes_Australia_All_Info  
ORDER BY CustomerID DESC;
```

Um outro recurso muito interessante do SQL Server, e que muitas vezes pode ser utilizado na falta de uma view, são as **Common Table Expressions (CTE)**. As CTEs fornecem um mecanismo para definir uma subconsulta que pode ser usada em outro lugar de uma query. Uma CTE é definida no início de uma query, pode ser referenciada várias vezes dentro da mesma. Dessa forma, fornecem um meio para dividir os problemas de consulta em unidades menores e mais modulares.

As CTEs possuem algumas características peculiares, como:

- Têm escopo limitado para a execução da consulta externa. Quando a consulta externa termina, a vida útil do CTE também termina;
- Exigem um nome para a expressão da tabela, além de nomes exclusivos para cada uma das colunas referenciadas na cláusula SELECT do CTE;
- Podem usar aliases para colunas;
- Várias CTEs podem ser definidas na mesma cláusula WITH;
- Suportam recursão, na qual a expressão é definida com uma referência a si mesma;
- As seguintes cláusulas não podem ser usadas na CTE:
 - ORDER BY (exceto quando uma cláusula TOP for especificada);
 - INTO e a cláusula OPTION com hints.

SINTAXE DE CRIAÇÃO:

WITH <CTE_name>

AS (<CTE_definition>)

EXEMPLO:

“Quantidade total de clientes e de pedidos, e a média, por bairro”

```
;WITH CTE1 AS (SELECT Id_Bairro, Nm_Cliente, COUNT(Id_Pedido) AS Qt_Pedidos
FROM Clientes LEFT JOIN Pedidos
ON Clientes.Id_Cliente = Pedidos.Id_Pedido
GROUP BY Id_Bairro, Nm_Cliente
),
CTE2 AS ( SELECT Nm_Bairro, COUNT(Nm_Cliente) AS Qt_Clientes, SUM(Qt_Pedidos) AS Qt_Pedidos
FROM Bairro LEFT JOIN CTE1
ON Bairro.Id_Bairro = CTE1.Id_Bairro
GROUP BY Nm_Bairro
)
SELECT Nm_Bairro, Qt_Clientes, Qt_Pedidos, CAST(Qt_Pedidos AS NUMERIC(15, 2)) / Qt_Clientes AS Media
FROM CTE2;
```

CTE Recursiva:

;WITH Estrut_Hierarquia AS

(-- Nível 1 (Âncora)

SELECT Id_Empregado, Nm_Empregado, Id_Superior, 1 AS Nivel

FROM Funcionario

WHERE Id_Superior IS NULL

UNION ALL

-- Níveis 2-N

```
SELECT A.Id_Empregado, A.Nm_Empregado, A.Id_Superior, B.Nivel + 1 AS Nivel
```

```
FROM Funcionarios A
```

```
JOIN Estrut_Hierarquia B ON A.Id_Superior = B.Id_Empregado
```

```
)
```

Figura 100 – SELECT * FROM Estrut_Hierarquia.

	Results	Messages		
	Id_Empregado	Nm_Empregado	Id_Superior	Nivel
1	1	Edvaldo Neves	NULL	1
2	14	Flávio Castro	NULL	1
3	2	Fábricio Amante	1	2
4	3	Caio Lima	1	2
5	9	Vítor Lima	1	2
6	15	Raul Farias	14	2
7	16	Logan Castro	1	2
8	4	Tiago Castro	2	3
9	5	Reginaldo Oliveira	3	3
10	6	Fábio Merazzi	3	3
11	7	Dirceu Resende	2	3
12	8	Luiz Vitor Neves	2	3
13	10	Edimar Lellis	9	3
14	11	Lucas Fardim	9	3
15	12	Aquila Loureiro	9	3
16	13	Rodrigo Almeida	9	3
17	17	Fábio Amante	3	3
18	18	Ariel Neves	3	3
19	19	Leandro Galon	7	4
20	20	Lucas Keller	7	4
21	21	Richardson Folha	19	5
22	22	Rafaela Giugliet	20	5

Fonte: Gustavo (2019).

11.2. Stored Procedures

Um **procedimento armazenado (stored procedure)** é um recurso da Linguagem SQL, criado (compilado) como um objeto no banco de dados, que encapsula um conjunto de instruções SQL. Diferentemente das views, que apenas encapsulam instruções SELECT, as procedures podem incluir uma gama mais ampla de instruções SQL, como UPDATE, INSERT e DELETE, além de instruções SELECT. Além dessas instruções, podem ser usadas instruções de lógica de processamento, controle de fluxos, testes condicionais, variáveis e instruções para tratamento de erros na execução da procedure. As procedures podem possuir parâmetros de entrada e saída, além de um valor de retorno.

Para criar um procedimento armazenado, é usado o comando **CREATE PROCEDURE** da classe DDL da Linguagem SQL. De forma análoga aos outros tipos de objetos, para alterar uma procedure usa-se o comando **ALTER PROCEDURE** e, para excluir, o comando **DROP PROCEDURE**. Para executar uma procedure, é usado o comando **EXECUTE** (ou simplesmente **EXEC**) da classe DML.

Para exemplificarmos, imagine que ao invés de criarmos a view para listar o ID e o Número da Conta dos clientes que são da Austrália, desejássemos criar uma view. O código dessa view seria algo como o mostrado abaixo, podendo inclusive já **conter a cláusula ORDER BY** (não permitida no código da view):

```
CREATE PROCEDURE SP_Clientes_Australia AS
BEGIN
    SELECT CustomerID, AccountNumber
    FROM Sales.Customer
    WHERE TerritoryID IN (
        SELECT TerritoryID
        FROM Sales.SalesTerritory
        WHERE Name = 'Australia'
    )
END
```

)

ORDER BY CustomerID DESC

END;

Em termos de benefícios, os do uso de procedimentos armazenados vão bem mais além do que os benefícios provenientes do uso de views. Dentre esses benefícios, podemos citar:

- **Mais segurança para os objetos do banco de dados:** os usuários podem possuir permissão para executar uma determinada procedure, sem ter permissão para acessar diretamente os objetos que a procedure acessa.
- **Programação modular de fácil manutenção:** a lógica é criada uma vez, no código da procedure, e depois **reutilizada** várias vezes, em diversos módulos da aplicação ou até mesmo em aplicações diferentes, bastando fazer uma chamada à procedure. A manutenção é mais fácil, pois se houver necessidade de alteração de código, **só é preciso alterar a procedure** em questão, e não o código da aplicação (ou de todas aplicações que estiverem usando o código sem ser via procedure).
- **Redução do tráfico de rede:** se a comunicação entre a aplicação e o servidor de banco de dados for feita através de uma rede de dados, o **envio de uma linha de código (EXEC Nome_da_Procedure)** é bem menor (menos bytes) e bem mais rápida de ser enviada para o banco de dados executar, do que várias linhas de código com instruções SQL.
- **Performance:** quando uma instrução SQL é executada, a maioria dos SGBDs cria um plano de execução para ela, relacionando os índices indicados para o acesso aos dados. Esse plano de execução representa a maneira mais eficiente para se executar a instrução naquele momento. Esse processo, chamado de **compilação**, pode aumentar significativamente o tempo de processamento para a execução de instruções SQL. Para as procedures, toda primeira vez que elas são executadas, o mecanismo de banco de dados

compila o código e cria um plano de execução da mesma maneira que o faria para uma instrução SQL isolada, mas depois armazena e **reutiliza esse plano da procedure nas execuções seguintes**. Como as execuções seguintes usam esse **plano de execução pré-compilado** ao invés de gerar um novo plano, o tempo necessário para executar a procedure é reduzido, quando comparado com o mesmo código executado sem ser via procedure.

Uma possibilidade muito interessante para se trabalhar com procedures, é nas operações de inserção, atualização e deleção de dados, onde pode-se, além de evitar acesso direto às tabelas, garantir várias consistências e verificações através do código da procedure. Para viabilizar isso, a linguagem SQL fornece o recurso de passagem de parâmetros para as procedures.

No exemplo abaixo, podemos ver o código da procedure recebendo os parâmetros, que serão os valores das colunas para inserir um novo departamento na tabela *Department*:

```
CREATE PROCEDURE SP_Insere_Departamento
    --Parâmetros de entrada e seus tipos de dados
    --@DepartmentID smallint => não é necessário (coluna IDENTITY)
    @NAme nvarchar(50),
    @GroupName nvarchar(50)
    --@ModifiedDate datetime => possui um default = getdate()

    AS
    BEGIN
        --Exibindo no output os valores que serão inseridos
        PRINT @NAme + '' + @GroupName;

        --Inserindo os valores dos parâmetros de entrada na tabela
        INSERT INTO HumanResources.Department
```

```
VALUES (@Name, @GroupName, DEFAULT);  
END;
```

--Execução da procedure passando os valores a serem inseridos:

```
EXEC SP_Insere_Departamento @Name='EAD', @GroupName='Education'
```

11.3. Functions

Assim como uma procedure, uma function é armazenada como um objeto persistente (compilado) no banco de dados, mas possuindo algumas diferenças fundamentais entre elas, como as mostradas abaixo:

Função (User Defined Function)	Stored Procedure
A função deve retornar um valor.	Procedure pode ou não retornar valores.
Permitirá apenas instruções SELECT, não permitirá usar outras instruções DML (INSERT / UPDATE / DELETE).	Pode ter instruções de seleção e instruções DML, como inserir, atualizar e excluir.
Transações não são permitidas dentro de funções.	Transações são permitidas.
Stored Procedures não podem ser chamadas de função.	Procedures podem ser chamadas de funções.
As funções podem ser chamadas a partir de uma instrução SELECT.	Procedures não podem ser chamadas nas instruções Select / Where / Having e assim por diante.

Um dos tipos de funções, conhecida como **TVF (Table-Value Function)**, possui várias propriedades em comum com as views, mas aceita parâmetros de entrada, podendo consultá-los na instrução SELECT do seu código. Em linhas gerais,

a TVF encapsula uma única instrução SELECT, retornando uma tabela virtual para a query, como mostrado a seguir.

SINTAXE DE CRIAÇÃO:

CREATE FUNCTION <name> (@<parameter_name> AS <data_type>, ...)

RETURNS TABLE AS

RETURN (<SELECT_expression>);

O exemplo a seguir cria uma function, que utiliza um parâmetro de entrada para controlar quantas linhas são retornadas pelo operador TOP:

CREATE FUNCTION TopNProdutos (@qtde AS INT)

RETURNS TABLE AS

RETURN (SELECT TOP (@qtde) productid, name, ListPrice

FROM Production.Products

ORDER BY ListPriceDESC);

CHAMADA À TVF: *SELECT * FROM TopNProdutos(3)*

11.4. Triggers

Já as **TRIGGERS (gatilhos)** são objetos programáticos também armazenados de forma compilada no banco de dados, que são disparados automaticamente, mediante uma ação no banco de dados, instância ou objeto.

Um dos tipos de triggers mais conhecido e usado é o **DML TRIGGER**. Uma trigger DML é executada quando um INSERT, UPDATE ou DELETE modifica os dados em uma tabela ou view (incluindo qualquer INSERT, UPDATE ou DELETE que faz parte de uma instrução MERGE), podendo consultar dados em outras tabelas. As

DDL TRIGGERS são semelhantes às triggers DML, com a diferença que são disparadas quando ocorrem eventos DDL (CREATE, ALTER, DROP etc.). Há também as **LOGON TRIGGERS**, que são uma forma especial de gatilho que dispara quando uma nova sessão é estabelecida.

EXEMPLO DE DML TRIGGER:

```
CREATE TRIGGER TRG_Aviso ON Aluno
```

```
AFTER INSERT AS
```

```
EXEC msdb.dbo.sp_send_dbmail
```

```
@profile_name = 'Coordenador',
```

```
@recipients = 'gustavo.aguilar@igti.edu.br',
```

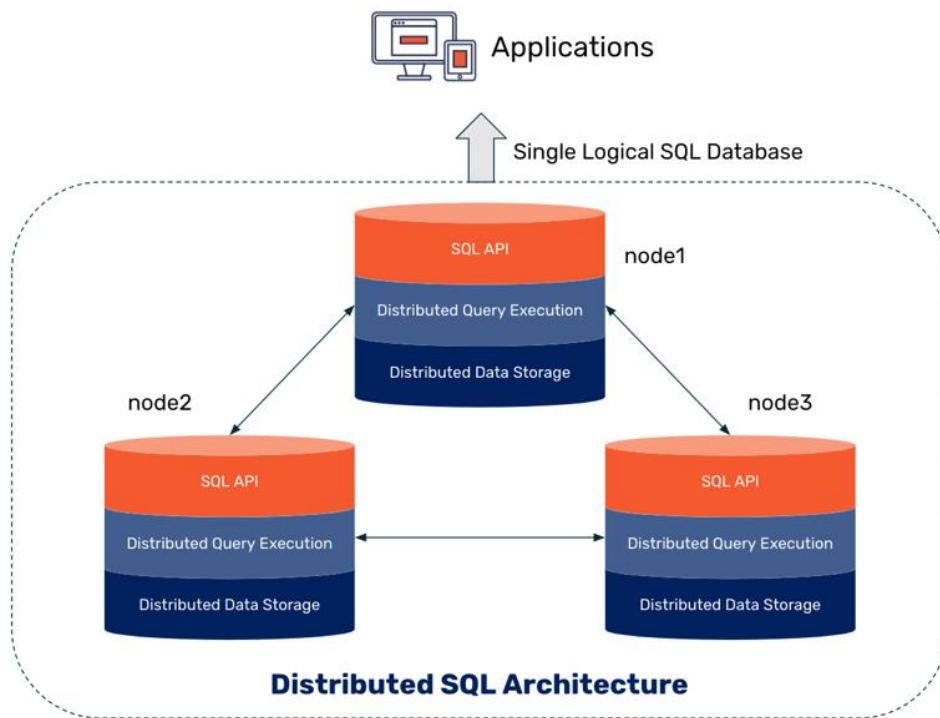
```
@body = 'Novo aluno matriculado no curso',
```

```
@subject = 'Aviso de Novo Aluno';
```

```
GO
```

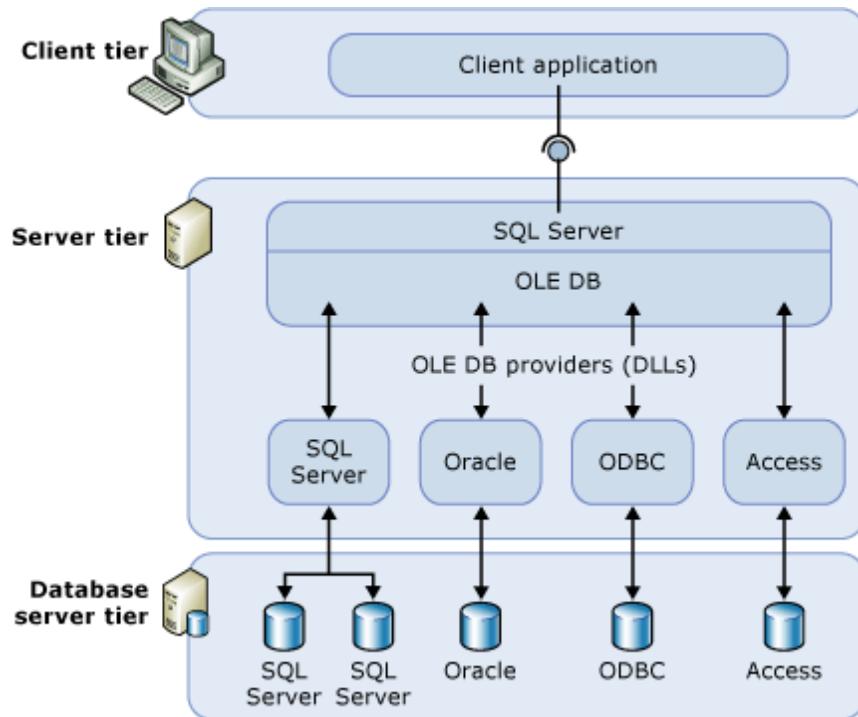
11.5. Linked Server e Query Distribuída

Em linhas gerais, uma query distribuída é uma consulta que executa em mais de uma instância de banco de dados. Uma característica essencial de uma query distribuída é a transparência da localização física dos dados, ou seja, o usuário tem uma visão centralizada do acesso a esses dados.

Figura 101 – Query Distribuída.

Fonte: Sid Choudhury (2019).

O **Linked Server** (no Oracle, chamado de *dblink*) permite que uma instância SQL Server leia dados de fontes de dados remotas e executem comandos nos servidores de banco de dados remotos. Normalmente, os linked server são configurados para permitir que o usuário execute uma instrução T-SQL que inclua tabelas em outra instância do SQL Server ou em outro produto de banco de dados, como Oracle. Além de acessar outros bancos de dados, o linked server pode ser configurado para acessar dados que estejam no Microsoft Access, no Excel e até mesmo na nuvem, como o Azure CosmosDB.

Figura 102 – Linked Server.

Fonte: Microsoft (2019).

Nas aulas gravadas e nas demonstrações, veremos como criar um linked server e como realizar uma query distribuída.

Capítulo 12. Mapeamento Objeto Relacional

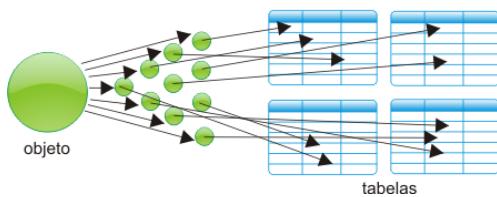
O surgimento dos conceitos de **programação orientada a objetos (POO)** data de alguns anos antes da publicação de Codd acerca do modelo relacional, com a criação da linguagem **Simula 67**, por Ole Johan Dahl e Kristen Nygaard, em 1967. Entretanto, o termo programação orientada a objetos foi criado oficialmente em 1971, por Alan Kay, autor da linguagem de programação *Smalltalk*.

Com a viralização dos sistemas e linguagens de bancos de dados relacionais, uma vertente para atender a demanda da programação orientada a objetos lançava, em paralelo na década de 80, os **sistemas gerenciadores de banco de dados orientado a objetos (SGBDOO)**.

Entretanto, mesmo com a grande quantidade de artigos publicados pelo projeto *ORION*, a massiva campanha comercial em cima do *GemStone* e com todos os pontos positivos de um **banco de dados orientado a objetos (BDOO)**, por exemplo, a persistência transparente (mais performática), grande facilidade de abstração da realidade e menos codificação, os SGBDOOs não chegaram perto da popularidade dos SGBDs relacionais, vistos na quase totalidade, como mais confiáveis e eficientes. Outro ponto determinante se refere às regras e rotinas para tratamento da persistência dos dados, que, em um banco de dados relacional, pode ficar embutida.

É nesse contexto que surgem as técnicas e ferramentas de mapeamento objeto relacional, de forma a criar um elo entre a programação orientada a objetos e os SGBDs relacionais, na busca em aproveitar o melhor desses dois mundos.

Figura 103 – Mapeamento Objeto Relacional.

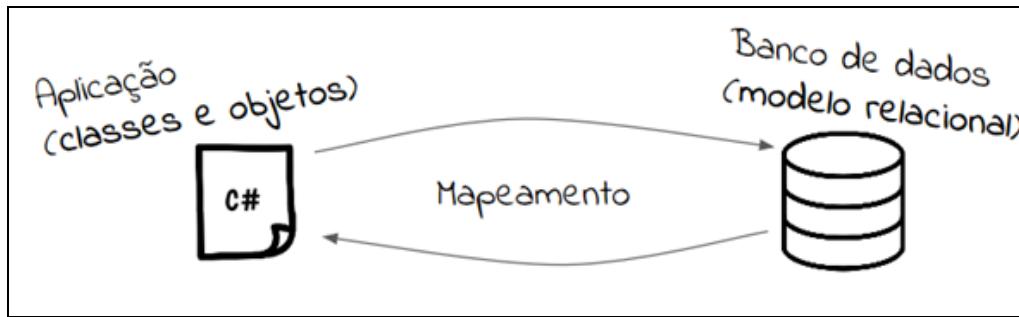


Fonte: David Machado (2010).

12.1. Introdução ao Mapeamento Objeto Relacional

O **mapeamento objeto relacional** (ORM, do inglês *Object Relational Mapping*) é uma técnica usada para conversão de dados entre linguagens de programação orientadas a objetos e bancos de dados relacionais.

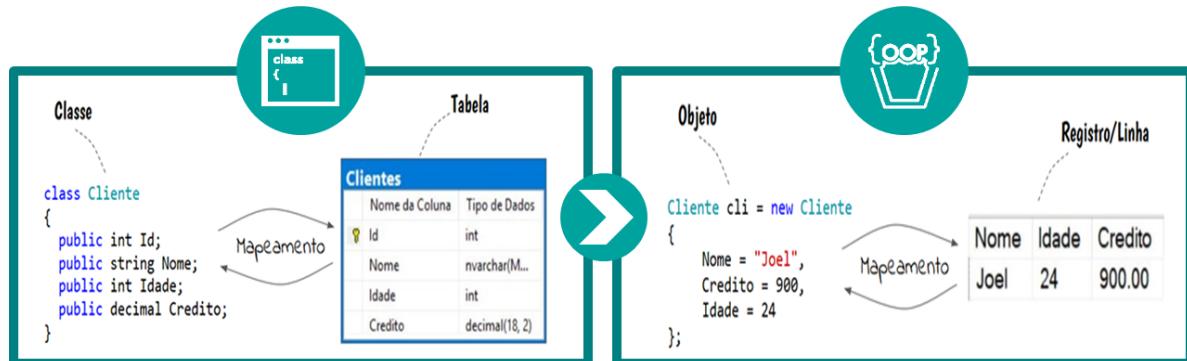
Figura 104 – Mapeamento Objeto Relacional.



Fonte: Joel Rodrigues (2017).

Na prática, trata-se de uma associação das classes de uma aplicação orientada a objetos com as tabelas em um banco de dados relacional. Seguindo nesse caminho, os objetos da aplicação também são mapeados em linhas nas tabelas do banco de dados. Com o mapeamento objeto relacional feito, trabalha-se no nível de código da aplicação com **classes e objetos** e, no nível do banco de dados, com **tabelas e linhas**.

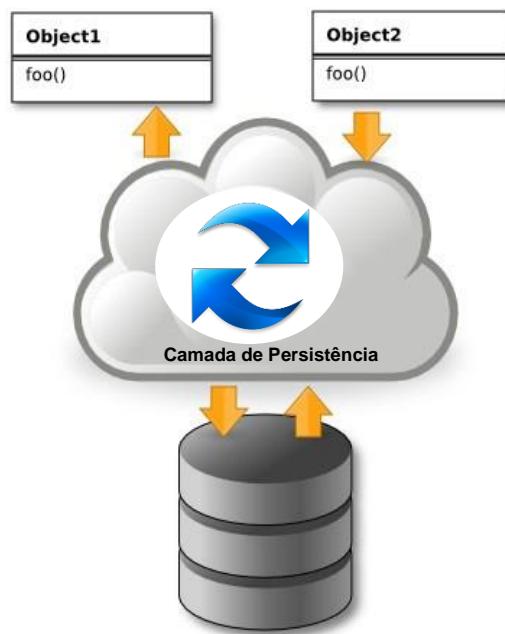
Figura 105 – Mapeamento de Classes e Objetos.



Fonte: Joel Rodrigues (2017).

Esse mapeamento é feito em uma camada chamada **Camada de Persistência de Objetos**. Trata-se de uma biblioteca ou trecho de código que realiza, de forma transparente, todo o **processo de persistência de dados** em um meio não volátil, ou seja, **no banco de dados relacional**. Com isso, o programador não precisa se ater aos comandos SQL do SGBD em questão, pois usa a interface de programação dessa camada de persistência para fazer toda a **manipulação dos dados**.

Figura 106 – Camada de Persistência.



Fonte: Isaque Pinheiro (2019).

Para se fazer esse mapeamento, ou seja, a conversão do modelo orientado a objetos (normalmente um diagrama UML) para o modelo de dados relacional, deve-se utilizar o conjunto de **Regras de Mapeamento Objeto Relacional**. Essas regras, não fazendo parte do escopo deste bootcamp, podem ser consultadas no **Guia para Mapeamento Objeto Relacional**, disponível em <http://www.documentador.pr.gov.br/documentador//acessoPublico.do?action=downloadArquivoUuid&uuid=@gtf-escriba@624abe66-4662-47fa-a33e-1f90de2436df>.

Concluído o mapeamento objeto relacional, a nível de modelo de dados, deve-se definir como será feita a persistência dos dados. Esse processo pode ser feito de forma manual pelo programador, o que se traduz em um esforço muito grande e uma quantidade excessiva de código, ou usando-se uma **ferramenta ORM**. Com essa segunda opção, o mapeamento objeto relacional é configurado na ferramenta, de forma simplificada e rápida, e a persistência é feita implicitamente pelo framework, eliminando o ônus de construção de código para isso. Além disso, por se tratar de ferramentas de mercado planejadas e testadas com bastante critério, adicionam mais segurança e confiança em todo o processo de persistência de dados.

Figura 107 –Ferramenta ORM.



Fonte: Visual Paradigm Gallery.

Apesar de não existir uma vastidão de ferramentas ORM, como acontece com as ferramentas case para modelagem de dados, há, no mercado, uma pluralidade suficiente, com boas opções, para atender as linguagens de programação mais utilizadas. A seguir, são listadas as mais conhecidas e conceituadas com as linguagens suportadas e respectivos sites.



- **Hibernate**
 - ✓ Gratuita.
 - ✓ Para linguagem **Java**.
- **NHibernate**
 - ✓ <http://hibernate.org/orm/>.
 - ✓ Gratuita.
 - ✓ Para **.NET**.
 - ✓ <https://nhibernate.info/>.

- **Django**



- ✓ Grátis;
- ✓ Para linguagem **Python**;
- ✓ www.djangoproject.com/.

- **Ruby on Rails**



- ✓ Grátis;
- ✓ Para linguagem **Ruby**;
- ✓ <https://rubyonrails.org/>.

- **Laravel**



- ✓ Grátis;
- ✓ Para linguagem **PHP**;
- ✓ <https://laravel.com/>.

- **Entity Framework**



- ✓ Grátis / Paga (extensões);
- ✓ Para **.NET**;
- ✓ <https://docs.microsoft.com/>.

- **LLBLGen Pro**



- ✓ Paga;
- ✓ Para **.NET**;
- ✓ <http://www.llblgen.com/>.

- **Sequelize**



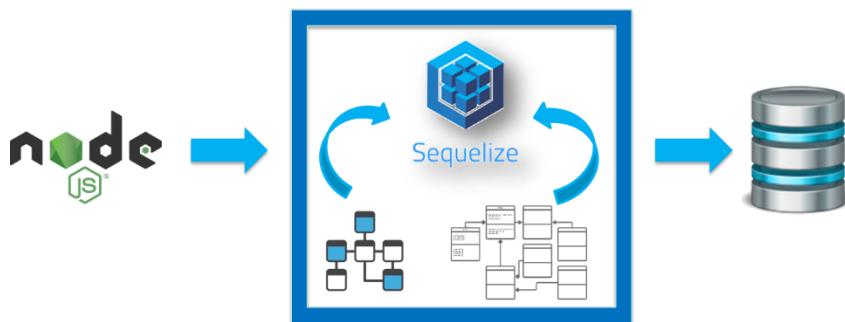
- ✓ Grátis;
- ✓ Para **Node.js**;
- ✓ <http://docs.sequelizejs.com/>.

12.2. Introdução ao Sequelize

O **Sequelize** é uma ferramenta *open source* de mapeamento objeto relacional para *Node.js* versão 4 (ou superior). Ela é multidialecto (comandos SQL dos sistemas gerenciadores de banco de dados), tendo suporte a *MySQL* e *MariaDB*, *PostgreSQL*, *SQLite* e *SQL Server*.

Na prática, ela fornece ao usuário os recursos para mapear os objetos *JavaScript* em tabelas, colunas e linhas nesses SGBDs citados, possibilitando a manipulação de dados relacionais através dos métodos JS.

Figura 108 – ORM Sequelize.



Fonte: Gustavo Aguilar (2019).

Apesar de ser gratuita, oferece suporte a transações, relacionamentos, leitura em cenários de replicação e outros recursos avançados. Além disso, possui uma boa comunidade no *GitHub* (<https://github.com/sequelize>) e uma ótima documentação (<http://docs.sequelizejs.com/>).

Ao longo deste capítulo, será mostrado como instalar e utilizar o Sequelize. Para isso, serão criados pequenos exemplos em *Node.js*, onde usaremos o modelo de dados físico para um sistema de controle de equipamentos de informática, elaborado no capítulo 1, a ser implementado em *MySQL*.

12.2.1. Instalação e Configuração do Sequelize para NodeJS

O Sequelize pode ser instalado em servidor com sistema operacional **Linux** ou **Windows**, sendo ele virtual, em docker ou físico. Antes de instalar o Sequelize, é preciso instalar o sistema gerenciador de banco de dados, onde os dados serão persistidos, ou seja, o dialeto a ser utilizado pelo ORM.

Para efeitos de estudo, pode-se utilizar o **VirtualBox** da Oracle, gratuito e disponível para download em <https://www.virtualbox.org>, que já possui imagens prontas com sistema operacional e que podem ser baixadas no endereço <https://www.oracle.com/technetwork/community/developer-vm/index.html>. Outra opção é usar algum provedor de cloud, como o **Azure** da Microsoft

(<https://azure.microsoft.com>), que fornece uma conta gratuita para testes, e que foi a escolha para as demonstrações desse curso, onde será utilizada uma máquina virtual com sistema operacional **Linux Ubuntu 18** (um passo a passo de como criar uma máquina virtual com Linux pode ser conferido em <https://docs.microsoft.com/learn/modules/create-linux-virtual-machine-in-azure>).

Quanto ao dialeto, será usado o **SGBD MySQL 5.7**, cujo passo a passo de instalação, no Linux Ubuntu, está disponível no link <https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en>. Nessa instalação do MySQL, para apoio aos exemplos, foram criados o **banco de dados BDEquipamentos** e o **usuário usrsequelize (com a senha sequelize)**.

Com o sistema operacional e o SGBD instalados, é possível iniciar a instalação do sequelize, que, de maneira geral, é bem simples:

- i. Logar via terminal no servidor onde o Sequelize será instalado;
- ii. Garantir que os repositórios e os pacotes de sistema estejam atualizados;

sudo apt update

sudo apt upgrade

- iii. Instalar o **npm** (o Sequelize também pode ser instalado via yarn):

sudo apt install npm

OBS.: para verificar a versão do npm instalada: ***npm -v***.

- iv. Criar um diretório para o projeto que será criado:

mkdir equipamentos

- v. Entrar no diretório criado e inicializar o projeto:

cd equipamentos

npm init -y

OBS.: no diretório em questão, será criado um arquivo JSON de nome ***package.json***, contendo as informações básicas do projeto, bem como de licença.

```
{  
  "name": "equipamentos",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": { "test": "echo \\\"Error: no test specified\\\" && exit 1" },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

vi. Instalar o NodeJS:

sudo apt-get install nodejs

OBS.: o executável do NodeJS no Ubuntu será o ***nodejs*** ao invés de ***node***.

vii. Instalar o ***Sequelize***, assim como o CLI (interface de linha de comando) do NodeJs (***sequelize-cli***), o ***Express*** (framework NodeJS) e o ***Body Parser*** (middleware JSON para as requisições):

npm install --save sequelize express body-parser sequelize-cli

Ao final da instalação, será mostrada uma tela como a abaixo, onde pode-se verificar a versão do Sequelize que foi instalada:

Figura 109 – Instalação do Sequelize.

```
gustavoaga@u18seq4: ~/equipamentos
  á á á utils-merge@1.0.1
  á á á á vary@1.1.2
  á á á á sequelize@4.43.0
  á á á á debuglog@5.1.0
  á á á ms@2.1.1
  á á á wks@0.4.6
  á á á @types/node@11.11.0
  á á á sequelize-cli@5.4.0
  á á á cli-color@1.4.0
  á á á ansi-regex@2.1.1
```

Fonte: Gustavo Aguilar (2019).

- viii. Após instalar o Sequelize, deve-se instalar o dialeto do MySQL:

npm install --save mysql2

- ix. Após instalar todos os pré-requisitos, pode-se conferir as dependências mapeadas para o projeto em questão, no arquivo package.json:

Figura 110 – Dependências do Sequelize.

```
gustavoagl@ul8seq4:~/equipamentos$ cat package.json
{
  "name": "equipamentos",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.18.3",
    "express": "^4.16.4",
    "mysql": "^2.16.0",
    "mysql2": "^1.6.5",
    "sequelize": "^4.43.0",
    "sequelize-cli": "^5.4.0"
  }
}
```

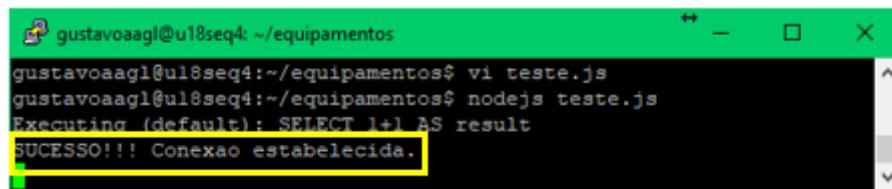
Fonte: Gustavo Aguilar (2019).

- x. Com o Sequelize instalado, é possível fazer um teste de conexão com o banco de dados. A conexão é feita através do método ***authenticate()*** que precisa receber, como input, as variáveis com o nome do banco, usuário, senha, o dialeto a ser usado e a porta.

Para fazer o teste, crie um arquivo (**teste.js**, por exemplo) com o conteúdo a seguir, dentro do diretório do projeto:

```
var Sequelize = require('sequelize')
, sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
{ host: 'localhost',
  dialect: "mysql",
  port: 3306,
  operatorsAliases: false
});
sequelize.authenticate().then
(
  function(err) { console.log('SUCESSO!!! Conexao estabelecida.');}
  function(err) { console.log('ERRO!!! Nao foi possivel conectar.', err);}
);
xi. Para executar o teste de conexão → nodejs teste.js
```

Figura 111 – Teste de Conexão no Sequelize.



```
gustavoagl@u18seq4:~/equipamentos$ vi teste.js
gustavoagl@u18seq4:~/equipamentos$ nodejs teste.js
Executing (default): SELECT 1+1 AS result
SUCESSO!!! Conexao estabelecida.
```

Fonte: Gustavo Aguilar (2019).

Mais detalhes acerca da instalação podem ser encontrados em <http://docs.sequelizejs.com/manual/installation/getting-started.html#installation>.

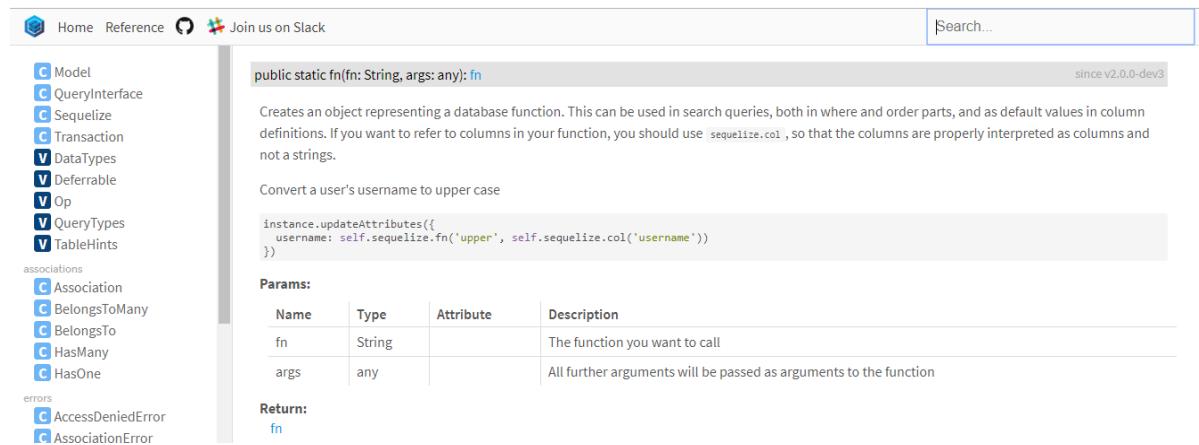
12.2.2. Dialectos e Opções da Engine

A API do Sequelize foi toda codificada a partir da classe principal **sequelize**. Para usá-la, basta importá-la no código Java, da seguinte forma:

```
const Sequelize = require ('sequelize');
```

Essa classe principal possui inúmeros **métodos** que permitem ao programador definir modelos, mapear objetos, persistir dados e manipulá-los. A documentação completa desses métodos, com exemplos de utilização, fica disponível em <http://docs.sequelizejs.com/class/lib/Sequelize.html>.

Figura 112 – Exemplo da Documentação da API do Sequelize.



The screenshot shows a detailed API documentation page for the `fn` method of the Sequelize class. The left sidebar contains navigation links for Model, QueryInterface, Sequelize, Transaction, DataTypes, Deferrable, Op, QueryTypes, TableHints, associations (Association, BelongsToMany, BelongsTo, HasMany, HasOne), and errors (AccessDeniedError, AssociationError). The main content area has a search bar at the top right. The `fn` method is described as creating an object representing a database function. It includes code examples for updating attributes and a table for parameters:

Name	Type	Attribute	Description
fn	String		The function you want to call
args	any		All further arguments will be passed as arguments to the function

Below the table, the `Return:` section indicates that the `fn` method returns an object.

Fonte: Sequelize (2019).

Além dessa classe principal, a biblioteca de conexão para o dialeto (SGBD) que será usado também deve ser instalada no projeto. Entretanto, não é necessário importá-la, pois o Sequelize se incube disso automaticamente.

Como já mencionado no capítulo 2.3, o Sequelize é um ORM **multidialeto**, com suporte aos seguintes sistemas gerenciadores de bancos de dados:

- MariaDB.
- MySQL.
- PostgreSQL.
- Microsoft SQL Server.
- SQLite.

A partir da versão 1.6.0, as bibliotecas do Sequelize se tornaram independentes de dialetos, o que significa que é necessária a instalação, à parte, das bibliotecas dos respectivos conectores. Com o SGBD instalado, isso pode ser feito via NPM, bastando escolher qual dialeto será usado no projeto:

npm install --save pg pg-hstore ➔ para PostgreSQL

npm install --save mysql2 ➔ para MySQL

npm install --save sqlite3 ➔ para SQLite

npm install --save mariasql ➔ para MariaDB

npm install --save tedious ➔ para SQL Server

As principais opções da classe **sequelize** são as requisitadas pelo método de autenticação (**authenticate**) para abrir a conexão com o banco, além dos parâmetros com o **nome do banco, nome do usuário e senha**:

- *dialect* ➔ sistema gerenciador de banco de dados que será usado.
- *host* ➔ hostname ou ip onde está o banco de dados.
- *port* ➔ porta na qual o banco de dados está respondendo.
- *operatorsAliases* ➔ desabilita/habilita o uso de alias para operadores (setado para *false*, evita erros de parse).

Além dessas opções, o Sequelize fornece várias outras **opções úteis para o projeto**, como as descritas a seguir:

- Desabilitar o despejo de informações na console (o default é *console.log*):

➔ *logging: false*

- Especificar opções do dialeto, como o conjunto de caracteres a ser utilizado. Apenas para MySQL, PostgreSQL e SQL. Default é vazio:

dialectOptions:

```
{ socketPath: '/Applications/MAMP/tmp/mysql/mysql.sock',
  supportBigNumbers: true,
  bigNumberStrings: true,
  charset: 'utf8',
```

```
    dialectOptions: { collate: 'utf8_general_ci' }
```

```
}
```

- Desativar a inserção de NULL para valores não definidos (default é *false*):

```
→ omitNull: true;
```

- Para SQLite, opção para armazenar dados em disco (default é *memory*):

```
→ storage: 'path/to/database.sqlite';
```

- Configuração do pool de conexões do banco de dados (default é 1):

```
pool: { max: 25,
         idle: 30000,
         acquire: 60000,
     }
```

OBS.: se existirem, por exemplo, 4 worker processes e a necessidade é de se ter um pool de conexão de 1000, deve-se configurar o pool de cada processo com tamanho máximo (parâmetro **max**) para 25.

- Nível de isolamento de cada transação (default é o default do SGBD):

```
isolationLevel: Transaction.ISOLATION_LEVELS.REPEATABLE_READ
```

Uma outra opção disponível, essencial para quando se está trabalhando com bancos de dados distribuídos **homogêneos**, é a opção de especificar os parâmetros referentes à **replicação de dados**:

Para as **configurações gerais aplicáveis a todas as réplicas**, não é necessário fornecê-las para cada instância. Por exemplo, no código mostrado a seguir, o nome do banco de dados e a porta são propagados para todas as réplicas. O mesmo acontecerá com o usuário e a senha se ela ficar de fora da configuração das réplicas, que tem as seguintes opções: host, porta, nome do banco de dados, nome do usuário e senha.

```
var sequelize = new Sequelize('database', null, null,
{ dialect: 'mysql',
  port: 3306
```

```
replication:  
{  
  read: [ { host: '8.8.8.8', username: 'read-username', password: 'some-password' },  
          { host: '9.9.9.9', username: 'another-username', password: null }  
        ],  
  write: { host: '1.1.1.1', username: 'write-username', password: 'any-password' }  
},  
pool: { max: 20,  
        idle: 30000  
      }  
})
```

O Sequelize usa pool para gerenciar conexões com as réplicas, mantendo internamento **dois pools**, criados usando-se a configuração default do pool. Essa configuração pode ser modificada, bastando especificar o pool desejado após a seção de configuração das réplicas, como mostrado no exemplo acima.

Para cada operação de escrita ou instrução específica para usar a réplica primária (*useMaster: true*), a query utilizará o pool de gravação. Para SELECT, o pool de leitura será usado e as leituras distribuídas nas réplicas secundárias usando-se um algoritmo básico de *round robin*, proporcionando um **balanceamento das cargas de leitura**.

Uma flexibilidade fantástica, também oferecida pelo Sequelize, é possibilidade de execução de query SQL nativa do SGBD, de dentro do código Java. Chamadas de **raw SQL queries**, elas são executadas através do método **query()**.

```
sequelize.query("SELECT * FROM TipoEquipamento")  
.then(RegistrosTabela => {console.log(RegistrosTabela) })
```

Com essa flexibilidade, é possível também mapear facilmente uma query para um modelo predefinido, bastando inserir no comando a opção **Model**, com o nome do modelo a ser associado, como mostrado no exemplo a seguir. Após a execução do comando, cada registro da **tabela Equipamento** se encontrará mapeado para o **modelo Equipamentos**.

```
sequelize.query('SELECT * FROM Equipamento', { model: Equipamentos })
  .then(equipamentos => { console.log(equipamentos) })
```

12.3. Criando e usando um modelo de dados básico no Sequelize

O Sequelize suporta dois modos de gerenciamento do esquema físico (objetos) no banco de dados:

- **Com migrações (*migrations*):** funcionamento similar ao controle de alterações em código-fonte, presente no *GitHub*, *VisualStudio* e no *SourceSafe*, por exemplo. Com as migrações, é possível transpor o banco de dados existente para outro estado e vice-versa, uma vez que essas transições de estado são salvas em arquivos de migração, que descrevem como chegar ao novo estado e como reverter as alterações para retornar ao estado antigo.
- **Automatizada:** Sequelize cria as tabelas, atomicamente, sem opção de rollback.

Embora a primeira possibilidade gaste mais tempo para a configuração, é o modo mais recomendado a ser utilizado em ambientes com muitos servidores e/ou muitos desenvolvedores, pois fornece mais segurança e estabilidade no processo de releases. Por outro lado, o modo automatizado é uma boa opção para ambientes *stand alone* e/ou com equipe pequena, bem como para cenários de geração rápida de um protótipo.

Para usar o modo automatizado, de forma que o Sequelize crie os objetos no banco de dados, é preciso especificar os tipos de dados que se deseja persistir, bem como o nome de cada campo (coluna) e do objeto (tabela) em questão. Isso é feito utilizando o método ***define()***. O Sequelize tem suporte a vários tipos de dados, que podem ser conferidos na documentação, disponível em <http://docs.sequelizejs.com/manual/tutorial/models-definition.html#data-types>.

No exemplo abaixo, foi feita a **definição do modelo** para a tabela ***Tipo_Equipamento*** do modelo de dados físico gerado no capítulo 1:

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
```

```
{ cod_tipo_equipamento: Sequelize.INTEGER,
```

Nome do objeto

```
    dsc_tipo_equipamento: Sequelize.STRING
```

```
});
```

Nome da coluna

Tipo de dados

Além das colunas definidas, o Sequelize cria 3 colunas de controle interno:

- **Id int(11) → chave primária** da tabela (OID), **autoincremental**;
- **createdAt (datetime)** → preenchida quando uma instância do objeto é persistida na tabela;
- **updatedAt (datetime)** → preenchida quando uma coluna é atualizada.

Com o modelo definido, precisa-se criar a representação desse modelo (esquema) no banco de dados para que os dados possam ser persistidos. Isso é feito com o **método de sincronização de esquema** do Sequelize, **sync()**:

Se a tabela existir → dropa e a recria

```
sequelize.sync({ force: true })
```

```
.then(function(err) { console.log('Tabela criado com sucesso!'); },
```

```
function (err) { console.log('Erro ao criar a tabela.', err); });
```

Dessa forma, o trecho de código para a criação do nosso modelo de exemplo, seria:

```
var Sequelize = require('sequelize')
, sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
{ host: 'localhost',
dialect: "mysql",
port: 3306,
operatorsAliases: false
```

```

    });
    sequelize.authenticate()
      .then ( function(err) { console.log('SUCESSO!!! Conexao estabelecida.');?>
        function (err) { console.log('ERRO!!! Nao foi possivel conectar.', err); }
      );
}

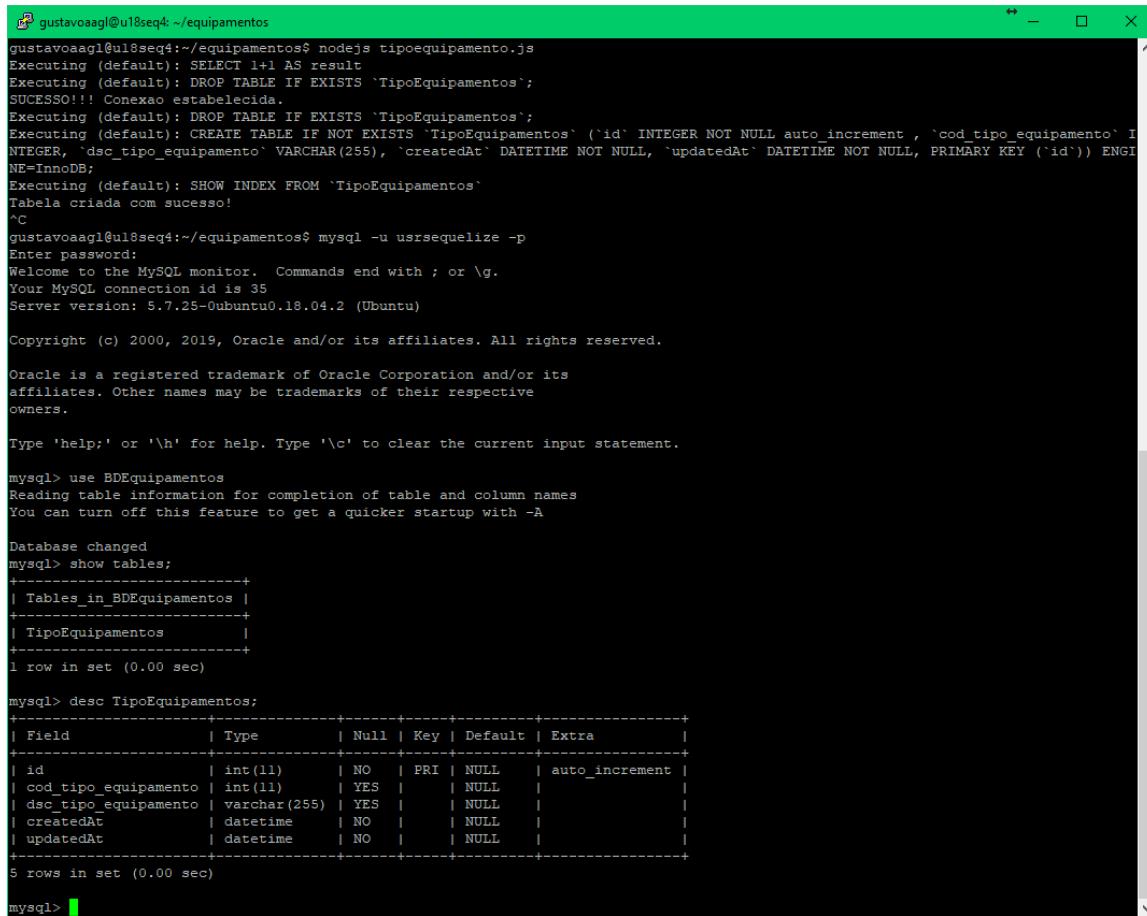
var TipoEquipamento = sequelize.define('TipoEquipamento',
{
  cod_tipo_equipamento: Sequelize.INTEGER,
  dsc_tipo_equipamento: Sequelize.STRING
});

sequelize.sync({ force: true })
  .then( function(err) { console.log('Tabela criada com sucesso!');}
    function (err){ console.log('Erro ao criar a tabela.', err); }
  );

```

Ao salvar esse código em um arquivo com a extensão **.js** (por exemplo, **tipoequipamento.js**) e executá-lo com o comando **nodejs tipoequipamento.js**, o objeto *TipoEquipamento* será gerado no banco de dados como uma tabela. Por default, o Sequelize gera o nome da tabela colocando um “s” ao final do nome do objeto. Dessa forma, será criada automaticamente a tabela *TipoEquipamentos* (por default, o Sequelize coloca o nome no plural), no banco de dados **BDEquipamentos**, existente no MySQL local, como mostrado a seguir.

Figura 113 – Exemplo de Sincronização de Esquema.



```

gustavoagl@ul8seq4:~/equipamentos$ nodejs tipoequipamento.js
Executing (default): SELECT 1+1 AS result
Executing (default): DROP TABLE IF EXISTS `TipoEquipamentos`;
SUCESSO!!! Conexao estabelecida.
Executing (default): DROP TABLE IF EXISTS `TipoEquipamentos`;
Executing (default): CREATE TABLE IF NOT EXISTS `TipoEquipamentos` (`id` INTEGER NOT NULL auto_increment , `cod_tipo_equipamento` INTEGER, `dsc_tipo_equipamento` VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `TipoEquipamentos`
Tabela criada com sucesso!
```
gustavoagl@ul8seq4:~/equipamentos$ mysql -u usrsequelize -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 35
Server version: 5.7.25-0ubuntu0.18.04.2 (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use BDEquipamentos
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----+
| Tables_in_BDEquipamentos |
+-----+
| TipoEquipamentos |
+-----+
1 row in set (0.00 sec)

mysql> desc TipoEquipamentos;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
id	int(11)	NO	PRI	NULL	auto_increment
cod_tipo_equipamento	int(11)	YES		NULL	
dsc_tipo_equipamento	varchar(255)	YES		NULL	
createdAt	datetime	NO		NULL	
updatedAt	datetime	NO		NULL	
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

**Fonte: Gustavo Aguilar (2019).**

Adicionalmente, você pode especificar o **nome da tabela** a ser gerada, ou configurar para que o Sequelize não coloque o nome da tabela no plural (opção **freezeTableName: true**):

```

var TipoEquipamento = sequelize.define('TipoEquipamento',
{
 cod_tipo_equipamento: Sequelize.INTEGER,
 dsc_tipo_equipamento: Sequelize.STRING
},
{ tableName: 'TipoEquipamento' })
```

Pode-se, também, optar por não gerar as colunas default de timestamp (createdAt e updatedAt):

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
{
 cod_tipo_equipamento: Sequelize.INTEGER,
 dsc_tipo_equipamento: Sequelize.STRING
},
{ timestamps: false})
```

Outra opção possível, é customizar o nome das colunas de timestamp:

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
{
 cod_tipo_equipamento: Sequelize.INTEGER,
 dsc_tipo_equipamento: Sequelize.STRING
},
{
 createdAt: 'dat_criacao',
 updatedAt: 'dat_ult_atualizacao'
})
```

Um recurso muito útil também, chamado de modelo **paranoid**, faz com que o registro não seja deletado da tabela quando o método de **destroy** (delete) é invocado. Isso faz com que o registro seja deletado apenas **logicamente**, utilizando o campo **deletedAt** (que também pode ser renomeado) para indicar a data que ele foi excluído. O ponto de atenção nesse modelo é que se não for implementado um expurgo físico da tabela, ela crescerá infinitamente.

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
{
 cod_tipo_equipamento: Sequelize.INTEGER,
 dsc_tipo_equipamento: Sequelize.STRING
},
{
 tableName: 'TipoEquipamento',
 paranoid: true,
 createdAt: 'dat_criacao',
 updatedAt: 'dat_ult_atualizacao',
 deletedAt: 'dat_exclusao'
})
```

**Figura 114 – Tabela Criada com Modelo Paranoid.**

| mysql> desc TipoEquipamento; |              |      |     |         |                |  |
|------------------------------|--------------|------|-----|---------|----------------|--|
| Field                        | Type         | Null | Key | Default | Extra          |  |
| id                           | int(11)      | NO   | PRI | NULL    | auto_increment |  |
| cod_tipo_equipamento         | int(11)      | YES  |     | NULL    |                |  |
| dsc_tipo_equipamento         | varchar(255) | YES  |     | NULL    |                |  |
| dat_criacao                  | datetime     | NO   |     | NULL    |                |  |
| dat_ult_atualizacao          | datetime     | NO   |     | NULL    |                |  |
| dat_exclusao                 | datetime     | YES  |     | NULL    |                |  |

**Fonte: Gustavo Aguilar (2019).**

Com o esquema físico criado, é possível começar a utilizá-lo para persistir os dados, ou seja, inserir registros nas tabelas. No Sequelize, essa etapa é chamada de **criação e persistência de instâncias**, podendo ser feita de duas maneiras:

- **Em dois passos** ➔ cria o objeto e depois o salva (métodos **build** e **save**).
  - ✓ Mais flexível, com opções antes de persistir o objeto.

```
var tpequipamento = TipoEquipamento.build(
 {
 cod_tipo_equipamento: 01,
 dsc_tipo_equipamento: 'Notebook'
 });
tpequipamento.save().then(function(err) { console.log('Registro inserido com sucesso!'); },
 function (err){ console.log('Erro ao inserir o registro', err);});
```

- **Com um passo** ➔ cria o objeto e já persiste (método **create**).
  - ✓ Mais performática.

```
var tpequipamento = TipoEquipamento.create(
 {
 cod_tipo_equipamento: 01,
 dsc_tipo_equipamento: 'Notebook'
 })
.then(function(err) { console.log('Registro inserido com sucesso!'); },
 function (err){ console.log('Erro ao inserir o registro', err);})
```

);

Para efeitos demonstrativos, utilizando o modo para persistir os dados com apenas um passo (create), o código de exemplo ficaria como:

```
var Sequelize = require('sequelize')
, sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
{
 host: 'localhost',
 dialect: "mysql",
 port: 3306,
 operatorsAliases: false
});

sequelize.authenticate()
.then(function(err) { console.log('SUCESSO!!! Conexao estabelecida.');?>
 function (err) { console.log('ERRO!!! Nao foi possivel conectar.', err); });

var TipoEquipamento = sequelize.define('TipoEquipamento',
{
 cod_tipo_equipamento: Sequelize.INTEGER,
 dsc_tipo_equipamento: Sequelize.STRING
},
{
 tableName: 'TipoEquipamento',
 paranoid: true,
 createdAt: 'dat_criacao',
 updatedAt: 'dat_ult_atualizacao',
 deletedAt: 'dat_exclusao'
});

var tpequipamento = TipoEquipamento.create(
{
 cod_tipo_equipamento: 01,
 dsc_tipo_equipamento: 'Notebook'
}
).then(function(err) { console.log('Registro inserido com sucesso!'); },
function (err){ console.log('Erro ao inserir o registro', err);});
```

Ao executar o código com sucesso, podemos constatar que o registro (**01**, **'Notebook'**) foi inserido na tabela TipoEquipamento.

**Figura 115 – Objeto Persistido com o Método Create.**

```

gustavosagl@ultraseq1:/equipamentos$ nodejs tipoequipamento_insert_1p.js
Executing (default): INSERT INTO `tipoequipamento` (id , cod_tipo_equipamento , dsc_tipo_equipamento , dat_criacao , dat_ult_atualizacao , dat_exclusao) VALUES (DEFAULT,1,'Notebook','2019-03-09 19:37:53','2019-03-09 19:37:53');
[OKNESSO!] Conexão estabelecida.
[OKNESSO!] Registro inserido com sucesso!

gustavosagl@ultraseq1:/equipamentos$ mysql -u usrsequelize -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 105
Server version: 5.7.25-0ubuntu0.18.04.2 (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use BDEquipamentos
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from TipoEquipamento;
+----+-----+-----+-----+-----+
| id | cod_tipo_equipamento | dsc_tipo_equipamento | dat_criacao | dat_ult_atualizacao | dat_exclusao |
+----+-----+-----+-----+-----+
| 1 | 1 | Notebook | 2019-03-09 19:37:53 | 2019-03-09 19:37:53 | NULL |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

**Fonte:** Gustavo Aguilar (2019).

Ao executar o código novamente, perceba que o objeto é persistido com sucesso mais uma vez, já que a coluna ID (OID) é autoincrementada:

**Figura 116 – Autoincremento do Atributo OID.**

```

mysql> select * from TipoEquipamento;
+----+-----+-----+-----+-----+
| id | cod_tipo_equipamento | dsc_tipo_equipamento | dat_criacao | dat_ult_atualizacao | dat_exclusao |
+----+-----+-----+-----+-----+
| 1 | 1 | Notebook | 2019-03-09 19:37:53 | 2019-03-09 19:37:53 | NULL |
| 2 | 1 | Notebook | 2019-03-09 19:43:49 | 2019-03-09 19:43:49 | NULL |
+----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

**Fonte:** Gustavo Aguilar (2019).

Para **ler** os objetos persistidos, ou seja, os registros no banco de dados, utiliza-se o método **findAll**, que é o equivalente ao **SELECT \*** na linguagem SQL.

No modelo de exemplo que foi criado, o código para recuperar todos os registros poderia ser:

```

var Sequelize = require('sequelize')
, sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',

```

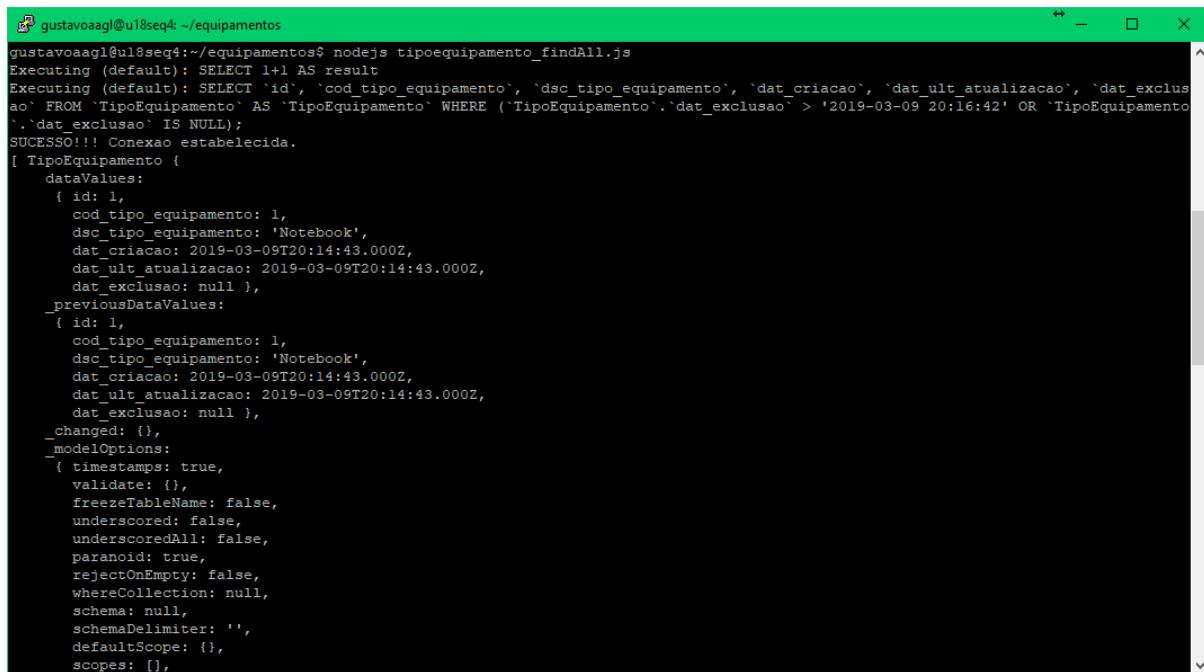
```
{ host: 'localhost',
 dialect: "mysql",
 port: 3306,
 operatorsAliases: false });

sequelize.authenticate()
.then(function(err) { console.log('SUCESSO!!! Conexao estabelecida.');//,
function (err) { console.log('ERRO!!! Nao foi possivel conectar.', err); });

var TipoEquipamento = sequelize.define('TipoEquipamento',
{ cod_tipo_equipamento: Sequelize.INTEGER,
dsc_tipo_equipamento: Sequelize.STRING
},
{ tableName: 'TipoEquipamento',
paranoid: true,
createdAt: 'dat_criacao',
updatedAt: 'dat_ult_atualizacao',
deletedAt: 'dat_exclusao'
});

var tpequipamento = TipoEquipamento.findAll().
then (tpequipamento => {console.log(tpequipamento) })
```

**Figura 117 – Método findAll().**



```
gustavoagl@ul8seq4:~/equipamentos
gustavoagl@ul8seq4:~/equipamentos$ nodejs tipoequipamento_findAll.js
Executing (default): SELECT `id`, `cod_tipo_equipamento`, `dsc_tipo_equipamento`, `dat_criacao`, `dat_ult_atualizacao`, `dat_exclusao` FROM `TipoEquipamento` AS `TipoEquipamento` WHERE (`TipoEquipamento`.`dat_exclusao` IS NULL);
SUCESSO!!! Conexao estabelecida.
[TipoEquipamento {
 dataValues:
 { id: 1,
 cod_tipo_equipamento: 1,
 dsc_tipo_equipamento: 'Notebook',
 dat_criacao: 2019-03-09T20:14:43.000Z,
 dat_ult_atualizacao: 2019-03-09T20:14:43.000Z,
 dat_exclusao: null },
 _previousDataValues:
 { id: 1,
 cod_tipo_equipamento: 1,
 dsc_tipo_equipamento: 'Notebook',
 dat_criacao: 2019-03-09T20:14:43.000Z,
 dat_ult_atualizacao: 2019-03-09T20:14:43.000Z,
 dat_exclusao: null },
 _changed: {},
 _modelOptions:
 { timestamps: true,
 validate: {},
 freezeTableName: false,
 underscored: false,
 underscoredAll: false,
 paranoid: true,
 rejectOnEmpty: false,
 whereCollection: null,
 schema: null,
 schemaDelimiter: '',
 defaultScope: {},
 scopes: [] } }]
```

Fonte: Gustavo Aguilar (2019).

## 12.4. Definição do Modelo de Dados no Sequelize

---

A definição do modelo de dados, ou seja, o mapeamento entre os objetos do modelo orientado a objetos e as tabelas em um banco de dados relacional, é feita através do método **define()**, como visto rapidamente em capítulos anteriores.

```
const Equipmento = sequelize.define('equipmento',
{
 cod_equipamento: Sequelize.INTEGER,
 dsc_equipamento: Sequelize.TEXT
})
```

Além das opções básicas de definição do nome do campo e do tipo de dados, o Sequelize fornece várias outras, das quais as mais interessantes estão listadas a seguir:

- Definição de campo que não aceita ausência de valores (**NOT NULL**):

```
const Equipmento = sequelize.define('equipmento',
{
 cod_equipamento: Sequelize.INTEGER,
 dsc_equipamento: {
 type:Sequelize.TEXT,
 allowNull: false
 }
})
```

- Atribuição de **valor default**, seja fixo ou dinâmico (p. ex. current time):

```
const Equipmento = sequelize.define('equipmento',
{
 cod_equipamento: Sequelize.INTEGER,
 dsc_equipamento: {
 type:Sequelize.TEXT,
 allowNull: false
 },
 ind_ativo: {
 type: Sequelize.BOOLEAN,
 allowNull: false,
 defaultValue: true
 },
 ...
})
```

```
 dat_compra: { type:Sequelize.DATE,
 defaultValue: Sequelize.NOW
 }
})
```

- Criação de **constraint única** (índice único):

```
nom_equipamento: { type:Sequelize.STRING(500),
allowNull: false,
unique: true }
```

- Criação de **chave primária**:

```
cod_equipamento: { type:Sequelize.INTEGER,
allowNull: false,
primaryKey: true }
```

**OBS.:** setando-se a chave primária manualmente com essa opção, o Sequelize não criará o campo **id** na tabela.

- Definição de campo **autoincremental**:

```
cod_equipamento: { type:Sequelize.INTEGER,
allowNull: false,
primaryKey: true,
autoIncrement: true }
```

- Criação de **chave estrangeira**:

```
const Equipmento = sequelize.define('equipamento',
{ cod_equipamento: Sequelize.INTEGER,
dsc_equipamento: {
 type:Sequelize.TEXT,
allowNull: false
},
cod_tipo_equipamento:
{
 type: Sequelize.INTEGER,
references:
```

```
{ //Nome do modelo "pai"
 model: TipoEquipamento,
 //Campo chave da tabela pai
 Key: 'id'
}
```

Um recurso adicional, muito útil na definição e manutenção do modelo de dados, que visa fornecer qualidade e consistência dos dados, é a opção de **Validação de Modelo**. Com as validações, é possibilitado que sejam especificadas validações de formato, conteúdo ou herança para cada atributo do modelo.

As validações são executadas automaticamente quando invocados os métodos *create*, *update* ou *save*. Além disso, também é possível chamar o método **validate ()** de forma a validar manualmente uma instância de um objeto.

O Sequelize fornece várias validações default, que estão implementadas no arquivo *validator.js*. A seguir, são listadas algumas validações interessantes, e a lista completa pode ser encontrada na documentação, no link <http://docs.sequelizejs.com/manual/tutorial/models-definition.html#validations>.

```
validate: {
 is: ["^[a-z]+$",'i'], // Aceita somente letras
 not: ["[a-z]",'i'], // Não aceita letras
 isAlpha: true, // Aceita somente letras
 isAlphanumeric: true, // Aceita somente letras e números
 isNumeric: true, // Aceita somente números
 isInt: true, // Verifica se é um número inteiro válido
 isFloat: true, // Verifica se é um float válido
 isLowercase: true, // Verifica se está tudo em minúsculo
 isUppercase: true, // Verifica se está tudo em maiúsculo
 isEmail: true, // Valida se é formato de e-mail (gu@igti.com.br)
 isUrl: true, // Valida se é formato de URL http://igti.com.br)
 isIP: true, // Valida se é formato IPv4 ou IPv6
 isIPv4: true, // Valida se é formato IPv4
```

```

isIPv6: true, // Valida se é formato IPv6
notEmpty: true, // Não aceita strings vazias strings
equals: 'valor', // Só aceita o valor especificado
contains: 'igti', // Só aceita se tiver a substring especificada
notContains: 'bar', // Não aceita se tiver a substring especificada
notIn: [['abc', 'xyz']], // Valida se o valor não é um dos especificados
isIn: [['igti', 'ead']], // Valida se o valor é um dos especificados
len: [2,10], // Só aceita valores na faixa de tamanho informada
isDate: true, // Só aceita valores de data
max: 23, // Só aceita valor menor ou igual ao especificado
min: 23, // Só aceita valor maior ou igual ao especificado

```

Exemplo: validar se a quantidade de itens da ordem de compra é no mínimo 1 dúzia e no máximo 10 dúzias.

```

OrdemCompra = Sequelize.define('ordemcompra',
{
 cod_ordem_compra: { type: Sequelize.INTEGER },
 qtd_itens:
 {
 type: Sequelize.INTEGER,
 validate:
 {
 notNull: true,
 min: 12,
 max: 120
 }
 }
}

```

Além dessas validações padrões, é possível definir uma validação customizada, como mostrado no exemplo a seguir:

```

var OrdemCompra = sequelize.define('OrdemCompra',
{
 cod_ordem_compra: { type: Sequelize.INTEGER },
 qtd_itens:

```

```
{ type: Sequelize.INTEGER,
 validate: [
 {
 notNull: true,
 min: 12,
 max: 120,
 ValorPar (value)
 {
 if (parseInt(value) % 2 != 0)
 {
 throw new Error('Apenas quantidades pares sao Aceitas!')
 }
 }
 } });
}
```

Com o modelo de dados definido, é possível **exportar** essas **definições**, usando o método **exports()**, de forma que elas possam ser **importadas**, em outro arquivo, com o método **import()**. Com esse método, o objeto retornado é exatamente o mesmo que definido na função do arquivo importado.

A partir da versão 1.5.0 do Sequelize, a importação é armazenada em cache, não havendo mais problemas de performance causados pela invocação frequente da importação de um arquivo.

Antes de fazer a importação das definições, é preciso criar a definição a ser importada, usando o método **export**. No nosso exemplo, no arquivo **tipoequipamento.js**, ficaria:

```
var Sequelize = require('sequelize')
, sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
{
 host: 'localhost',
 dialect: "mysql",
 port: 3306,
 operatorsAliases: false
});

sequelize.authenticate()
.then(function(err) { console.log('SUCESSO!!! Conexao estabelecida.');?>, function (err)
{
 console.log('ERRO!!! Nao foi possivel conectar.', err); });

var TipoEquipamento = sequelize.define('TipoEquipamento',
{
```

```
cod_tipo_equipamento: Sequelize.INTEGER,
dsc_tipo_equipamento: Sequelize.STRING
},
{
 tableName: 'TipoEquipamento',
 paranoid: true,
 createdAt: 'dat_criacao',
 updatedAt: 'dat_ult_atualizacao',
 deletedAt: 'dat_exclusao'
});

module.exports = (sequelize, DataTypes) =>
{
 const TipoEquipamento = sequelize.define('TipoEquipamento',
 {
 cod_tipo_equipamento: Sequelize.INTEGER,
 dsc_tipo_equipamento: Sequelize.STRING },
 {
 tableName: 'TipoEquipamento',
 paranoid: true,
 createdAt: 'dat_criacao',
 updatedAt: 'dat_ult_atualizacao',
 deletedAt: 'dat_exclusao'
 }
)
 return TipoEquipamento;
};

sequelize.sync({ force: true }).then(function(err) {console.log('Tabela criada com sucesso!'); },
 function (err) {console.log('Erro ao criar a tabela.', err); });


```

Já no arquivo de definição do modelo de equipamento (*equipamento.js*), utiliza-se o método *import* para referenciar o modelo *TipoEquipamento* já mapeado:

```
var Sequelize = require('sequelize'),
 sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
 {
 host: 'localhost',
 dialect: "mysql",
 port: 3306,
 operatorsAliases: false
 });


```

```
sequelize.authenticate().then(function(err) {console.log('SUCESSO!!! Conexao estabelecida.');},
function(err) {console.log('ERRO!!! Nao foi possivel conectar.', err);});
```

```
const TipoEquipamento =
sequelize.import ("./home/gustavoagl/equipamentos/tipoequipamento_export.js")
```

```
var Equipamento = sequelize.define('Equipamento',
{
 cod_equipamento: { type:Sequelize.INTEGER,
 allowNull: false,
 autoIncrement: true,
 primaryKey: true
 },
 nom_equipamento: { type:Sequelize.STRING(500),
 allowNull: false,
 unique: true
 },
 dsc_equipamento:{ type:Sequelize.TEXT,
 allowNull: false
 },
 ind_ativo: { type: Sequelize.BOOLEAN,
 allowNull: false,
 defaultValue: true
 },
 dat_compra: { type:Sequelize.DATE,
 defaultValue: Sequelize.NOW
 },
 cod_tipo_equipamento: { type: Sequelize.INTEGER,
 references:
 { model: TipoEquipamento,
 Key: 'id'
 }
 },
 },
{ tableName: 'Equipamento',
paranoid: true,
createdAt: 'dat_criacao',
updatedAt: 'dat_ult_atualizacao',
deletedAt: 'dat_exclusao'
```

Caminho e nome do arquivo com o modelo a ser importado

```

sequelize.sync({ force: true })

.then(function(err) { console.log('Tabela criada com sucesso!'); },
 function (err) { console.log('Erro ao criar a tabela.', err); });

```

Se tratando de um ORM completo, não poderia faltar uma forma para a **criação de índices** na definição do modelo. No Sequelize, isso é feito com a opção **indexes**, como mostrado nos exemplos ao lado, extraídos da própria documentação, em <http://docs.sequelizejs.com/manual/tutorialmodels-definition.html#indexes>.

```

sequelize.define('user', {}, {
 indexes: [
 // Create a unique index on email
 {
 unique: true,
 fields: ['email']
 },
 // Creates a gin index on data with the jsonb_path_ops operator
 {
 fields: ['data'],
 using: 'gin',
 operator: 'jsonb_path_ops'
 },
 // By default index name will be [table]_[fields]
 // Creates a multi column partial index
 {
 name: 'public_by_author',
 fields: ['author', 'status'],
 where: {
 status: 'public'
 }
 },
 // A BTREE index with a ordered field
 {
 name: 'title_index',
 method: 'BTREE',
 fields: ['author', {attribute: 'title', collate: 'en_US', order: 'DESC', length: 5}]
 }
]
})

```

## 12.5. Usando e consultando o modelo de dados no sequelize

O sequelize disponibiliza vários métodos em sua API, correspondentes aos comandos da linguagem SQL do dialeto em questão. Apesar de não existir uma correspondência para 100% dos comandos, não há grandes prejuízos para o mapeamento objeto relacional.

Os métodos mais usados são os com propósito de leitura de dados, como o *findAll()* visto em capítulos anteriores e os métodos abaixo que possibilitam **filtrar a consulta**, sendo os correspondentes a SELECT com WHERE:

- Consultar por registros com ID(s) específico(s):

*TipoEquipamento.findById (123).then (tipoequipamento => {...})*

- Consultar filtrando por atributos:

*Equipamento.findOne({ where: {nom\_equipamento: 'Notebook'} })  
.then(equipamento => {...})*

- Consultar filtrando por atributos e os atributos retornados:

```
Equipamento.findOne({where: {nom_equipamento: 'Notebook'},
 attributes: ['cod_equipamento', 'nom_equipamento', 'ind_ativo']})
.then(equipamento => { ... })
```

Com as opções dos métodos de consulta, é possível também **ordenar** e/ou **agrupar** os registros retornados:

//Similar a SELECT \* FROM Equipamento ORDER BY nom\_equipamento DESC:

```
Equipamento.findAll({order: 'nom_equipamento DESC'})
```

//Similar a SELECT \* FROM Equipamento GROUP BY dat\_compra:

```
Equipamento.findAll({group: 'dat_compra'})
```

Além dessas opções, o Sequelize fornece vários tipos de **operadores** que possibilitam a execução de queries mais complexas com operações combinatórias.

No exemplo abaixo, será executado no banco o comando SELECT \* FROM TipoEquipamento WHERE cod\_tipo\_equipamento IN (12, 13):

```
TipoEquipamento.findAll({ where: { cod_tipo_equipamento:
 {
 [Op.or]: [12, 13]
 }
}});
```

Existem operadores para a grande maioria dos respectivos operadores presentes no sistema gerenciador de banco de dados, como pode-se constatar na listagem abaixo:

### Figura 118 – Operadores.

```
[Op.and]: {a: 5} // AND (a = 5)
[Op.or]: [{a: 5}, {a: 6}] // (a = 5 OR a = 6)
[Op.gt]: 6, // > 6
[Op.gte]: 6, // >= 6
[Op.lt]: 10, // < 10
[Op.lte]: 10, // <= 10
[Op.ne]: 20, // != 20
[Op.eq]: 3, // = 3
[Op.not]: true, // IS NOT TRUE
[Op.between]: [6, 10], // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15
[Op.in]: [1, 2], // IN [1, 2]
[Op.notIn]: [1, 2], // NOT IN [1, 2]
[Op.like]: '%hat', // LIKE '%hat'
[Op.notLike]: '%hat' // NOT LIKE '%hat'
[Op.iLike]: '%hat' // ILIKE '%hat' (case insensitive) (PG only)
[Op.notILike]: '%hat' // NOT ILIKE '%hat' (PG only)
[Op.regexp]: '^h|a|t' // REGEXP/^ 'h|a|t' (MySQL/PG only)
[Op.notRegexp]: '^h|a|t' // NOT REGEXP/!^ 'h|a|t' (MySQL/PG only)
[Op.iRegexp]: '^h|a|t' // ~* 'h|a|t' (PG only)
[Op.notIRegexp]: '^h|a|t' // !~* 'h|a|t' (PG only)
[Op.like]: { [Op.any]: ['cat', 'hat']} // LIKE ANY ARRAY['cat', 'hat'] - also works for iLike and notLike
[Op.overlap]: [1, 2] // && [1, 2] (PG array overlap operator)
[Op.contains]: [1, 2] // @> [1, 2] (PG array contains operator)
[Op.contained]: [1, 2] // <@ [1, 2] (PG array contained by operator)
[Op.any]: [2,3] // ANY ARRAY[2, 3]::INTEGER (PG only)
```

**Fonte: Sequelize (2019).**

Adicionalmente, o Sequelize possui métodos correspondentes às **funções de agregação** dos bancos de dados, para contar ou somar registros, bem como retornar valores máximos e mínimos. Esses métodos podem ser usados de forma isolada (sem filtro), ou em conjunto com operadores:

- Contar as ocorrências de um elemento:

//Equivalente a SELECT COUNT(\*) FROM Equipamento:

```
Equipmento.count().then(x => {
 console.log("Existe um Total de " + x + " equipamentos!"))
})
```

//Equivalente a SELECT COUNT (\*) FROM Equipmento  
WHERE cod\_equipamento > 10:

```
Equipamento.count({ where: {'cod_equipamento': {[Op.gt]: 10}} }).then(x => {
 console.log("Existe um Total de " + x + " equipamentos com código maior que 10.")}
```

- Somar Valores de Atributos:

//Equivalente a SELECT SUM (vlr\_equipamento) FROM Equipamento:

```
Equipamento.sum('vlr_equipamento').then(sum => {})
```

- Retornar o Valor Máximo de um Atributo:

//Equivalente a SELECT MAX (vlr\_equipamento) FROM Equipamento  
WHERE ind\_ativo IN (1,2):

```
Equipamento.max('vlr_equipamento',{where:{ind_ativo: {[Op.or]:[1,2]}}}).then(max=>{})
```

- Retornar o Valor Mínimo de um Atributo:

//Equivalente a SELECT MIN (vlr\_equipamento) FROM Equipamento:

```
Equipamento.min('vlr_equipamento').then(min=>{})
```

Complementando as operações de CRUD (criar, recuperar, **atualizar e deletar**), o Sequelize fornece os métodos **destroy()** e **update()** para remover e atualizar registros da tabela, respectivamente:

//Equivalente a DELETE FROM Equipamento WHERE ind\_ativo = 0:

```
Equipamento.destroy({ where: {ind_ativo: 0 } });
```

//Equivalente a UPDATE Equipamento SET updatedAt = null WHERE deletedAt NOT NULL:

```
Equipamento.update({ updatedAt: null}, {where: {deletedAt: {[Op.ne]: null }}});
```

## Capítulo 13. Controle de Versão de Banco de Dados

---

A topologia de sistemas de três camadas (3-tier), com os servidores de banco de dados, aplicação e camada de apresentação independentes entre si, trouxe inúmeras vantagens para o desenvolvimento de software, bem como para a administração e suporte do ambiente. Entretanto, com essa camada, a persistência de praticamente todos os dados no servidor de banco de dados tornou-se quase uma regra, trazendo também uma necessidade inerente e imprescindível: maior controle sobre a matriz de versionamento e compatibilidade dos softwares das camadas de aplicação e apresentação, com o schema da camada de banco.

Até então, durante a etapa de desenvolvimento, enquanto as modificações e novas implantações em componentes de código geralmente eram controladas através de algum software de versionamento, as alterações no schema físico do banco de dados quase nunca eram. Nesse contexto, surgem as técnicas e ferramentas de controle de versão de banco de dados, de forma a permitir que haja um “de-para” mais robusto entre as versões de códigos e as estruturas de armazenamento necessárias do banco de dados.

### 13.1. Introdução ao Controle de Versão de Banco de Dados

---

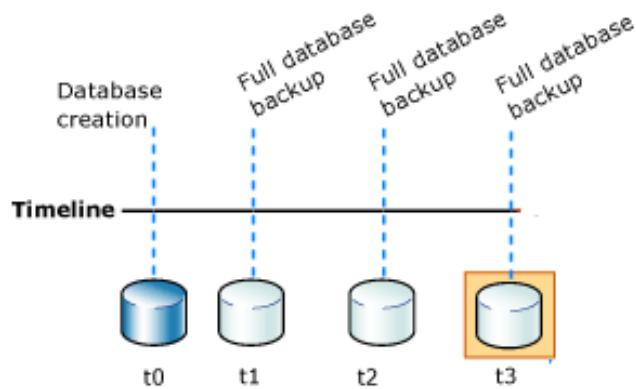
Até o surgimento das ferramentas específicas para versionamento de banco de dados, podemos afirmar que, na quase totalidade dos casos, o controle de versão, quando feito, era com base em uma das duas maneiras abaixo:

- **Versionamento full do banco.**
- **Versionamento de cada alteração no banco de dados em um arquivo separado.**

Na primeira opção, apesar de simples, tem-se desvantagens muito impactantes, como o tamanho do banco (desperdício de espaço de armazenamento com as versões completas e não somente da alteração feita), dificuldade de replicar

para outros ambientes somente a alteração feita, e a imprescindível intervenção humana, sujeita a erros operacionais (o que poderia invalidar o controle de versão).

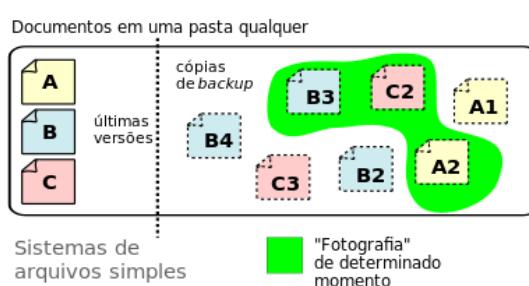
**Figura 119 – Versionamento Full.**



**Fonte: Gustavo Aguilar (2019).**

Na segunda opção, mais granular e otimizada em questões de espaço de armazenamento e replicação das alterações, também pode-se encontrar desvantagens muito relevantes, por exemplo, a necessidade de intervenção humana com perfil metódico, uma vez que cada alteração, por menor que seja, deve ser versionada. Além disso, podemos encontrar também certa complexidade para se gerir e coordenar muitas alterações (elevado número de arquivos/scripts para uma determinada versão).

**Figura 120 – Versionamento de Arquivos.**

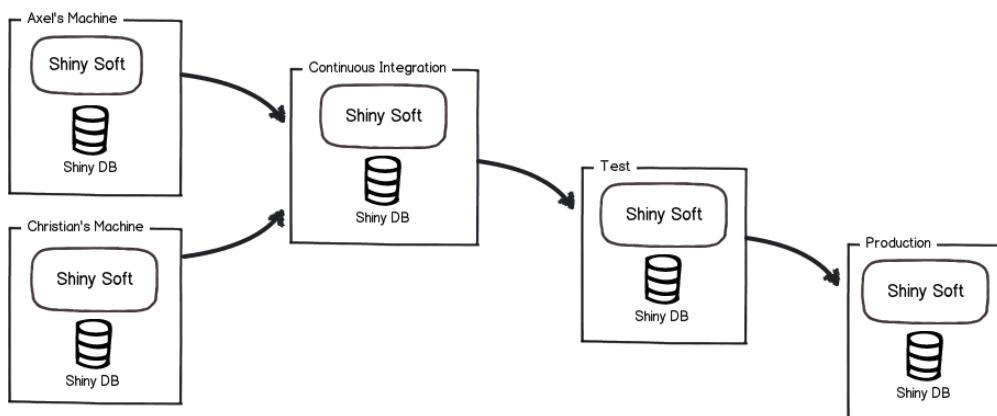


**Fonte: Wikipédia (2019).**

Pensando em eliminar esses problemas e inconveniências, foi introduzida a técnica de migração (**migrations**) de alterações em bancos de dados, atribuída por uma parte da comunidade ao *framework Ruby on Rails*, e segundo outros, ao famoso *Visual Studio* da Microsoft e seu conhecido e poderoso mecanismo de controle de versão de deploy.

Independentemente do precursor, o conceito de migrations, em linhas gerais, pode-se resumir a um pacote de alterações feitas no schema do banco de dados em um determinado momento, com opção para aplicar a versão nova contida na migration (subir versão), desfazê-la (voltar versão) ou até mesmo promovê-la para outro(s) ambiente(s).

**Figura 121 – Migrations.**



**Fonte:** Flyway (2019).

Pode-se perceber que se trata de um conceito que, de certa forma, já era aplicado no versionamento de cada alteração no banco de dados em um arquivo separado, citada no início, mas que na maioria das vezes não continha o script para desfazer a alteração (voltar ao status anterior), além de ser feita de maneira manual. Nesse ínterim, as ferramentas de controle de versão vieram preencher essa lacuna e permitir que as migrações pudessem ser feitas de uma maneira mais automatizada, controlada e garantida, bem como fornecendo uma linha do tempo para o schema do banco de dados, permitindo ao usuário avançar ou retroceder na mesma.

Dentre as diversas ferramentas de controle de versão existentes atualmente, as mais conhecidas e bem-conceituadas são:

- **Visual Studio:**

**Bancos de dados suportados:** SQL Server, Azure SQL Database.

**Repositórios suportados:** Team Foundation Server e Github.

**Tipo de licença:** comercial (possui edição gratuita com limitações).

**Site:** [https://msdn.microsoft.com/en-us/library/hh272690\(v=vs.103\).aspx](https://msdn.microsoft.com/en-us/library/hh272690(v=vs.103).aspx).

- **Redgate Source Control:**

**Bancos de dados suportados:** Oracle e SQL Server.

**Repositórios suportados:** Team Foundation Server, Subversion, git, Mercurial, Perforce, SourceGear Vault, Working Folder.

**Tipo de licença:** comercial.

**Sites:** <http://www.red-gate.com/products/sql-development/sql-source-control>

[www.red-gate.com/products/oracle-development/source-control-for-oracle](http://www.red-gate.com/products/oracle-development/source-control-for-oracle).

- **Datical:**

**Bancos suportados:** SQL Server, Oracle, PostgreSQL, EnterpriseDB, DB2.

**Repositórios suportados:** proprietário.

**Tipo de licença:** comercial.

**Site:** <https://www.datical.com>.

- **DBmaestro Source Controle:**

**Bancos de dados suportados:** SQL Server, Oracle.

**Repositórios suportados:** proprietário.

**Tipo de licença:** comercial.

**Site:** <https://www.dbmaestro.com/database-source-control>.

▪ **Liquibase:**

**Bancos de dados suportados:** MySQL, PostgreSQL, Oracle, SQL Server, SAP ASE, SAP SQL Anywhere, DB2, Apache Derby, HSQL, H2, Informix, Firebird, SQLite.

**Repositórios suportados:** proprietário.

**Tipo de licença:** gratuita.

**Site:** <http://www.liquibase.org>.

▪ **Ruby on Rails Migrations:**

**Bancos suportados:** MySQL, PostgreSQL, SQLite, SQL Server, Oracle.

**Repositórios suportados:** proprietário.

**Tipo de licença:** gratuita.

**Site:** <http://guides.rubyonrails.org/v3.2/migrations.html>.

▪ **Version SQL**

**Bancos de dados suportados:** SQL Server.

**Repositórios suportados:** Github e Subversion.

**Tipo de licença:** comercial (possui edição gratuita com limitações).

**Site:** <https://www.versionsql.com>.

▪ **Flyway**

**Bancos de dados suportados:** Oracle, SQL Server, Azure SQL Database, CockroachDB, DB2, MySQL, MariaDB, Google Cloud SQL, PostgreSQL, Amazon Redshift, Vertica, H2, HyperSQL, Derby, SQLite, SAP HANA, solidDB, SAP ASE, Sybase ASE, Phoenix, EnterpriseDB, Greenplum.

**Repositórios suportados:** proprietário.

**Tipo de licença:** gratuita/paga.

**Site:** <https://flywaydb.org>.

## 13.2. Introdução ao VersionSQL

---

O **VersionSQL** é um software de controle de versão de banco de dados **SQL Server**, desenvolvido para a plataforma Windows pela **MV Webcraft**. Em termos de repositório para armazenamento das versões dos bancos de dados, ele pode utilizar qualquer servidor **Git** ou **Subversion** hospedado em uma rede interna ou na nuvem (**GitHub, Atlassian Bitbucket, Visual Studio Team Services** etc.) por meio de uma conexão HTTPS segura.

Após ser instalado, o VersionSQL adiciona, no painel *Object Explorer* do *SQL Server Management Studio* (SSMS), que é o client oficial do SQL Server, **atalhos** específicos para o fluxo de controle de versões, com o propósito de permitir, de forma facilitada, o check-in de um banco de dados inteiro, pasta (objetos do mesmo tipo no SQL Server Management Studio) ou objetos individuais (tabelas, views, procedures etc.).

Em seu funcionamento, o código T-SQL com as alterações do banco de dados é gravado em um arquivo, com a extensão .sql, organizado em pastas, e enviado para o servidor de controle de versão.

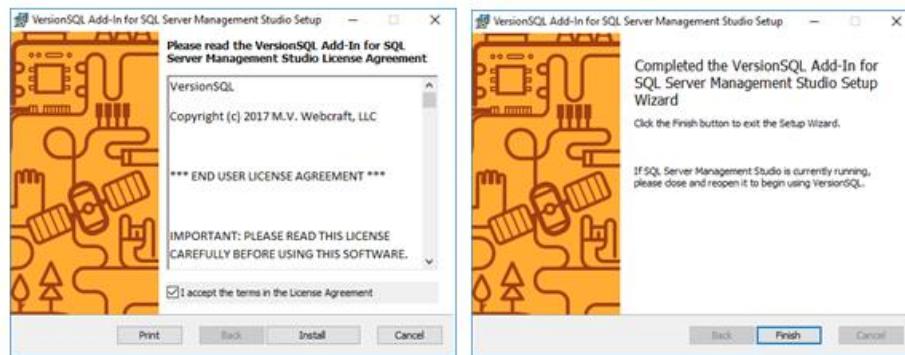
O **VersionSQL Express** é **100% gratuito** para uso com instâncias SQL Server Express e nenhum registro é necessário para utilizá-lo. Já o VersionSQL

Professional, possui suporte para outras edições do SQL, mas é necessária a aquisição da licença, que é por usuário.

Para efeitos de aprendizado nessa disciplina, usaremos a edição VersionSQL Express, com um servidor SQL Server 2017 Express. O instalador pode ser baixado direto do site do fabricante, disponível no link <https://www.versionsql.com/express>.

. A instalação é bem simples, composta basicamente de duas telas, como mostrado a seguir, mas requer que o SQL Server já esteja instalado.

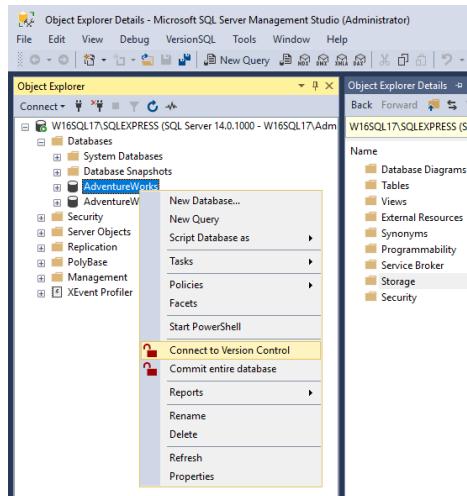
**Figura 122 – Telas do Install do VersionSQL Express.**



**Fonte: Gustavo (2019).**

Após a instalação, é preciso adicionar cada banco de dados ao VersionSQL, clicando com o botão direito sobre eles no Management Studio, e escolhendo a opção **Connect to Version Control**, como mostrado abaixo:

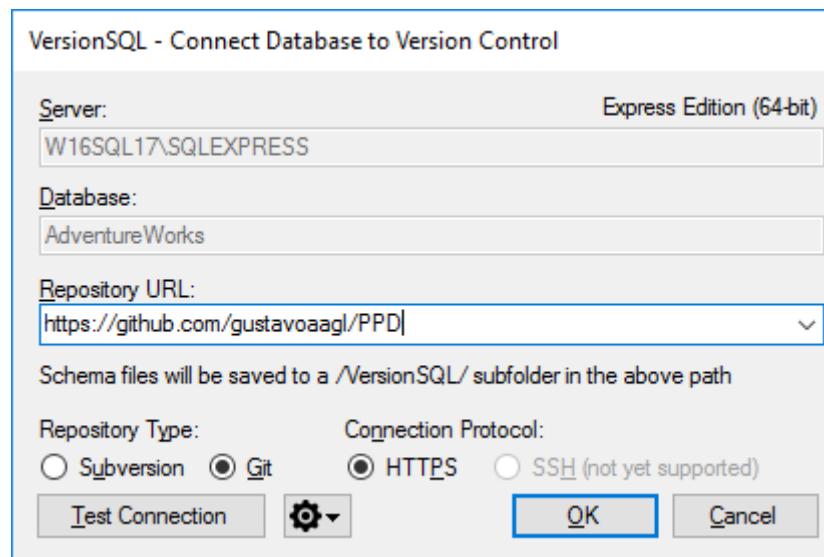
**Figura 123 – Adicionando um banco ao VersionSQL.**



**Fonte: Gustavo (2019).**

Feito isso, na tela seguinte, deve-se selecionar o **tipo de repositório** para armazenamento das versões do banco de dados, bem como o local dele.

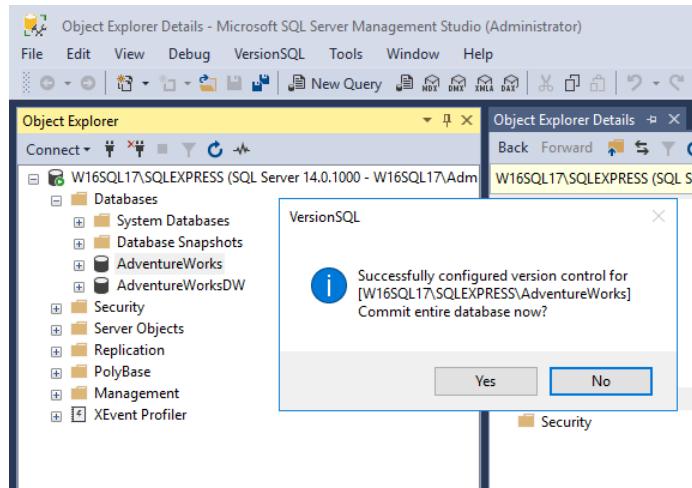
**Figura 124 – Configurar Repositório do VersionSQL.**



**Fonte: Gustavo (2019).**

Após a configuração, será solicitada a primeira sincronização completa (**commit entire database**), de forma a permitir a criação da primeira versão do banco de dados.

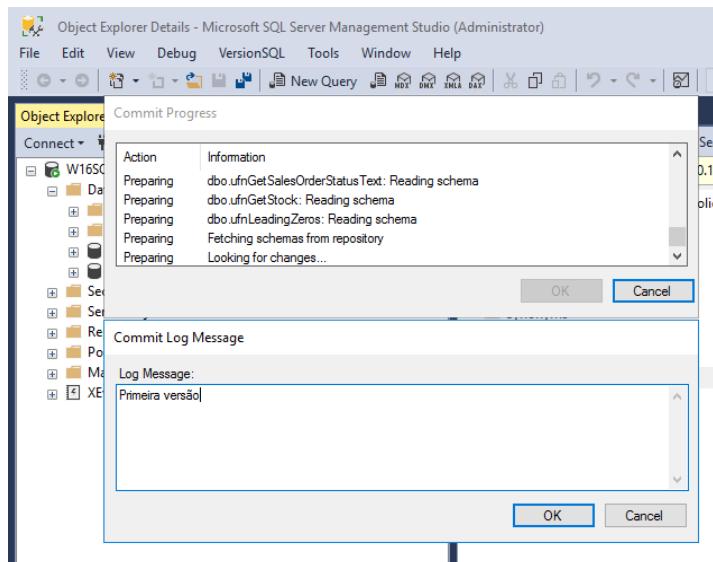
**Figura 125 – Criação da Primeira Versão do Banco.**



**Fonte: Gustavo (2019).**

Pode-se inserir **comentários no log** de cada check-in (criação de nova versão), como mostrado na tela abaixo:

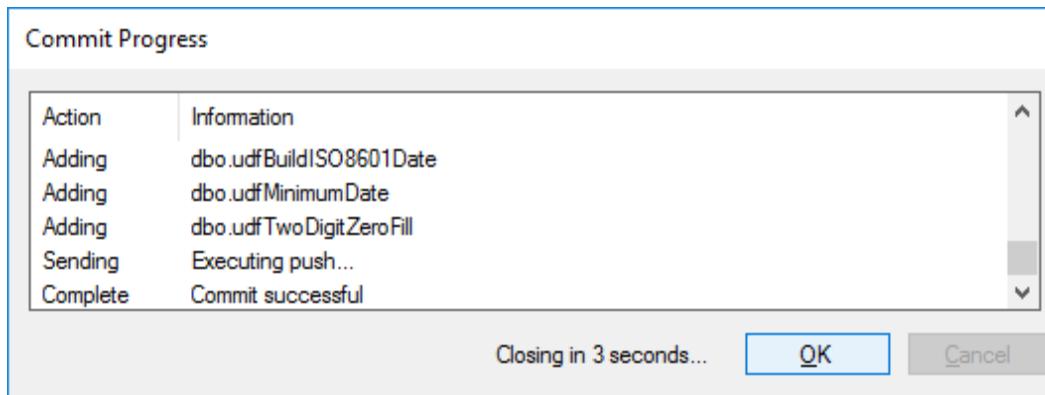
**Figura 126 – Comentários da Versão.**



**Fonte: Gustavo (2019).**

Após o VersionSQL ler toda a estrutura do banco, é mostrado um log do processamento, e é dada como encerrada essa primeira fase do controle de versão, como mostrado abaixo:

**Figura 127 – Log do Progresso do Commit.**



**Fonte: Gustavo (2019).**

Nesse ponto, a estrutura para armazenamento das versões do banco, pastas e os scripts já podem ser encontradas no repositório configurado. Ela é separada por banco, schema e tipo de objeto. No nosso caso, o GitHub ficaria algo semelhante como mostrado a seguir:

**Figura 128 – Estrutura no GitHub.**

The screenshot shows a GitHub repository page for 'gustavoagl / PPD'. The repository details are as follows:

- Branch: master
- Issues: 0
- Pull requests: 0
- Projects: 0
- Wiki: 0
- Insights: 0
- Settings: 0
- Watch: 0
- Star: 0
- Fork: 0

Below the repository details, there is a navigation bar with buttons for 'Create new file', 'Upload files', 'Find file', and 'History'. The main content area shows the file structure:

```
PPD / VersionSQL / W16SQL17 / SQLEXPRESS / AdventureWorks / HumanResources / Table /
```

A table lists the files:

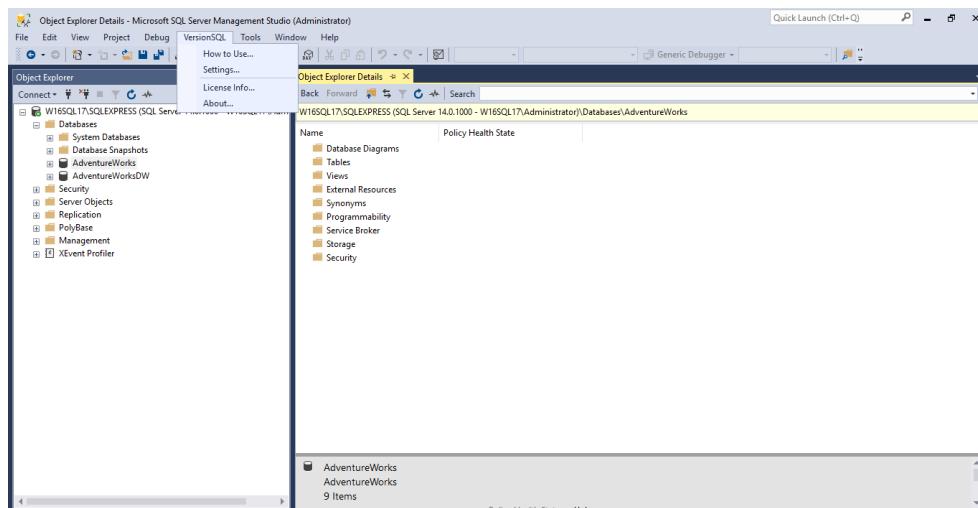
| File                          | Version         | Time           |
|-------------------------------|-----------------|----------------|
| Department.sql                | Primeira versão | 17 minutes ago |
| Employee.sql                  | Primeira versão | 17 minutes ago |
| EmployeeDepartmentHistory.sql | Primeira versão | 17 minutes ago |

**Fonte: Gustavo (2019).**

### 13.3. Utilização do VersionSQL

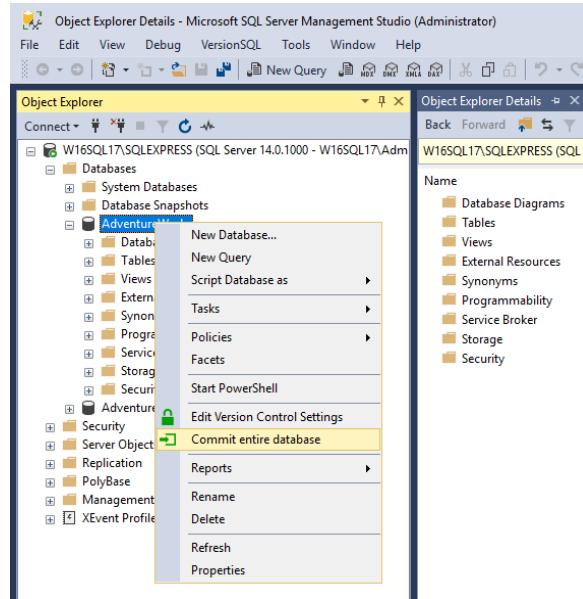
Toda a interação com o controle de versão do VersionSQL é feita utilizando a interface gráfica do SQL Server Management Studio, seja usando o **botão direito**, o **menu VersionSQL** ou o **menu File, Open, File**.

**Figura 129 – Interface de Interação com o VersionSQL.**

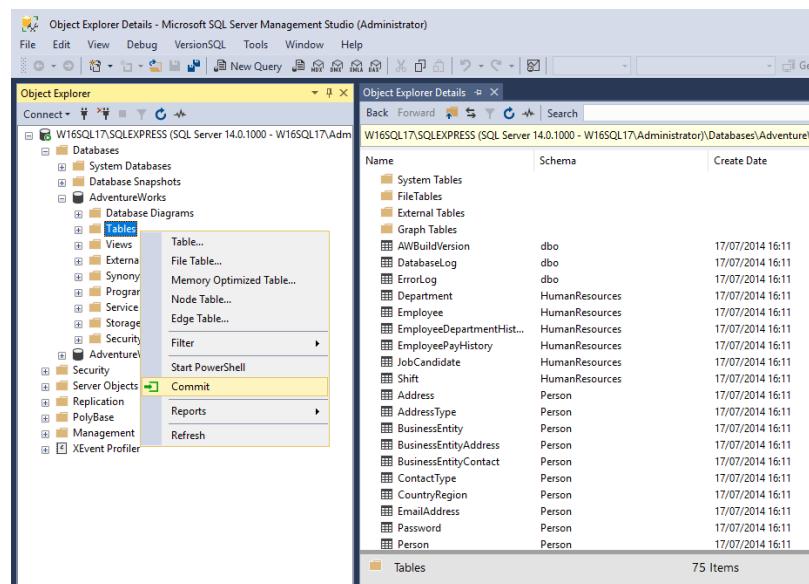


**Fonte: Gustavo (2019).**

Para gerar uma versão do banco inteiro, deve-se clicar com o botão direito sobre o banco desejado e escolher a opção **Commit entire database**.

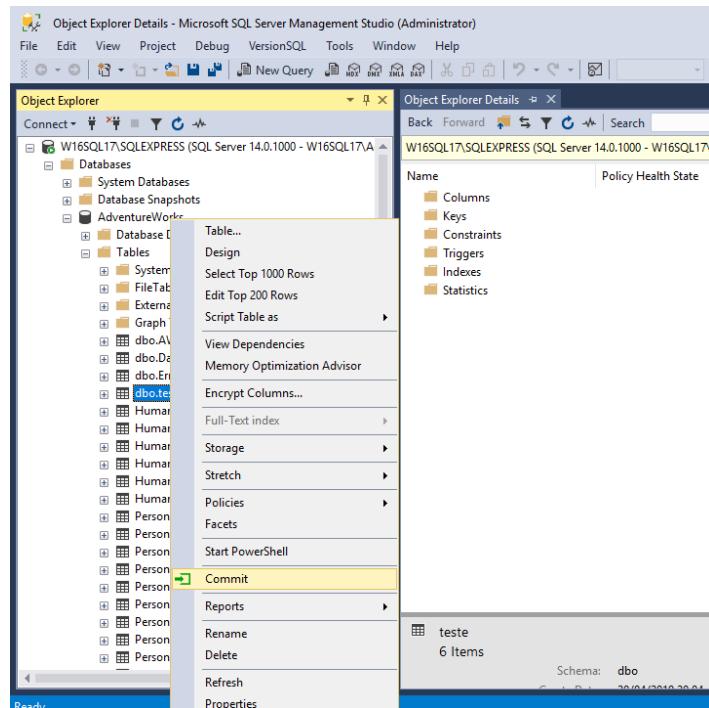
**Figura 130 – Gerando Versão Completa do Banco.****Fonte: Gustavo (2019).**

Para gerar uma versão de uma pasta inteira (tabelas, views, procedures etc.) do SQL Server Management Studio, deve-se clicar com o botão direito sobre a pasta desejada e escolher a opção **Commit**.

**Figura 131 – Gerando Versão de Pasta Inteira do SSMS.****Fonte: Gustavo (2019).**

Já para gerar uma versão de um objeto apenas (tabela/view/procedure etc.), deve-se clicar com o botão direito sobre objeto desejado e escolher a opção **Commit**.

**Figura 132 – Gerando versão de um objeto específico.**

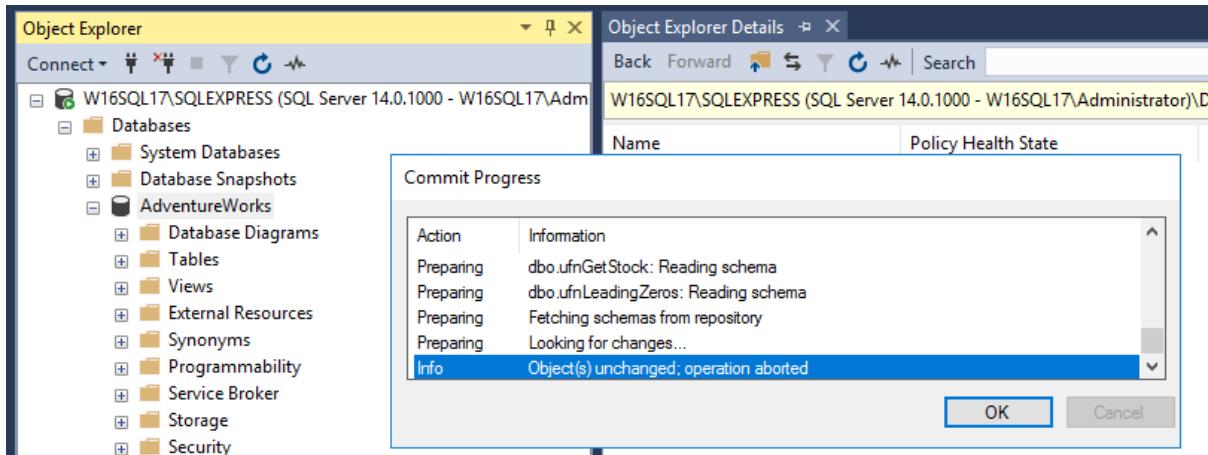


**Fonte: Gustavo (2019).**

Todas essas opções do VersionSQL trabalham com o conceito de migration, visto anteriormente. Com isso, economiza-se em espaço no repositório e tem-se uma visão clara das alterações ao longo da linha do tempo de desenvolvimento ou manutenção do projeto do banco de dados.

Com base nas migrations, o VersionSQL tem controle e inteligência suficiente para verificar que nenhuma alteração foi feita após o último check-in e abortar a operação, como mostrado no exemplo abaixo:

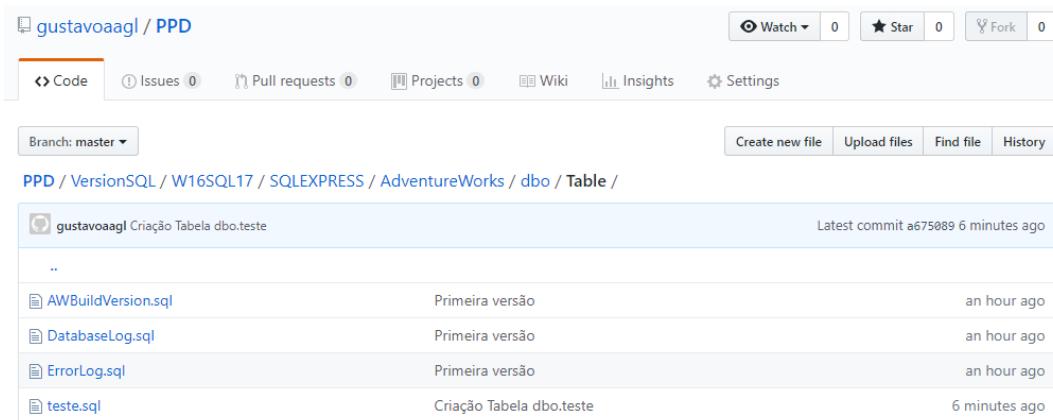
**Figura 133 – Abort de Check-In sem alterações.**



**Fonte: Gustavo (2019).**

Com a utilização de migrations, também consegue fornecer uma visão geral dos objetos do banco de dados, cada um na sua versão atual. Dessa forma, no repositório, sempre que é feito um check-in, ele pode ser facilmente identificado, sem, no entanto, perder a visibilidade das versões atuais dos outros objetos.

**Figura 134 – Estrutura das Migrations após novas versões.**

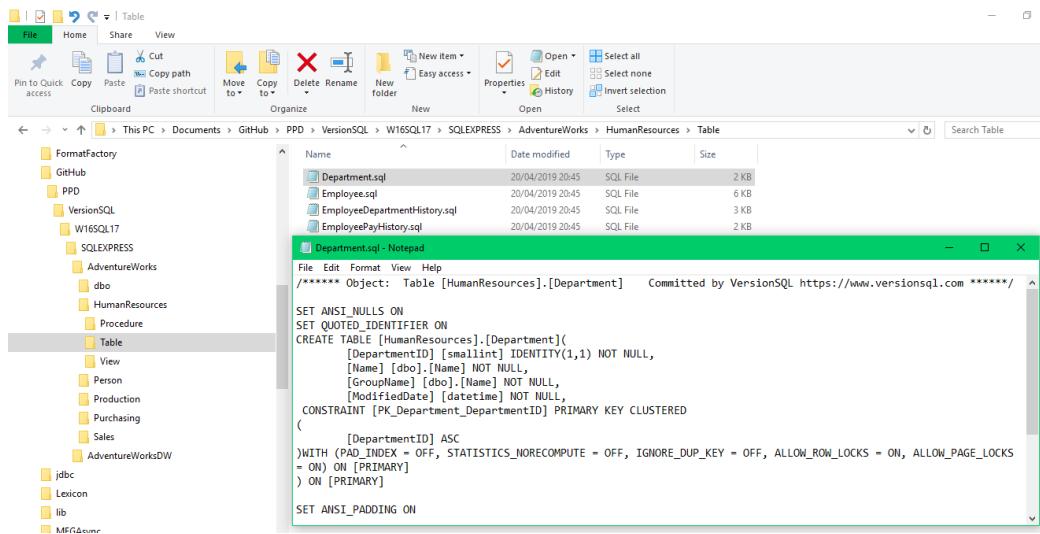


| File               | Description              | Created       |
|--------------------|--------------------------|---------------|
| AWBuildVersion.sql | Primeira versão          | an hour ago   |
| DatabaseLog.sql    | Primeira versão          | an hour ago   |
| ErrorLog.sql       | Primeira versão          | an hour ago   |
| teste.sql          | Criação Tabela dbo.teste | 6 minutes ago |

**Fonte: Gustavo (2019).**

Usando uma ferramenta de exploração do repositório, como o GitHub Desktop, é possível navegar na estrutura do repositório, abrindo e copiando os scripts armazenados.

**Figura 135 – Explorando o Re却itório do VersionSQL.**



**Fonte: Gustavo (2019).**

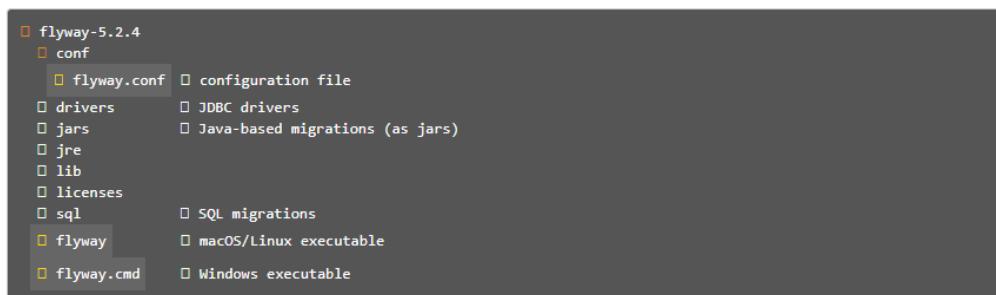
### 13.4. Introdução ao Flyway

O Flyway é um software de controle de versão desenvolvido para as plataformas Windows, Linux, macOS e Docker, com suporte a diversos sistemas gerenciadores de bancos de dados, como Oracle, SQL Server, CockRoachDB, DB2, MySQL, MariaDB, PostgreSQL, Amazon Redshift, SQLite, SAP HANA e Sybase ASE. Em termos de repositório para armazenamento das versões dos bancos de dados, ele utiliza um repositório interno, armazenado em flat file.

Quanto à usabilidade, o Flyway não possui atalhos nas interfaces gráficas como o VersionSQL, mas, por outro lado, permite o controle de versão e migração de alterações em dados (scripts DML), e não somente DDL. Diferentemente do SQLVersion que usa migrations para exportar para o repositório as alterações já feitas no banco de dados, no Flyway as migrations devem ser criadas previamente, em formatos de scripts .sql. Com base nesses scripts é que as alterações são feitas no schema físico do banco de dados, pelo próprio Flyway. Em sua edição gratuita, possui apenas uma ferramenta de linha de comando sem vários recursos presentes nas edições pagas (Pro e Enterprise).

O Flyway pode ser baixado no endereço <https://flywaydb.org/download>. Ele não possui um instalador e basta descompactar o arquivo no local desejado.

**Figura 136 – Estrutura de diretórios.**

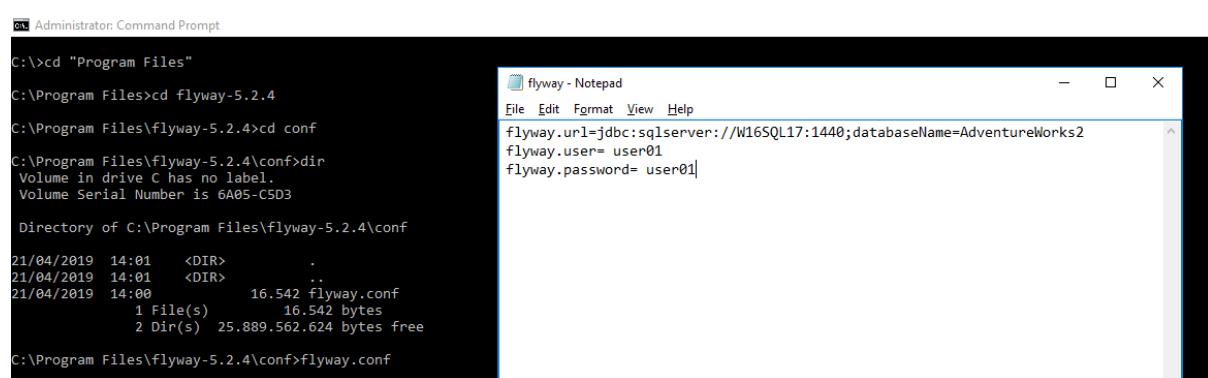


**Fonte:** Flyway (2019).

Antes de começar a utilizá-lo, é preciso configurar a URL do local do banco de dados, bem como o usuário e senha para acesso. Isso é feito no arquivo /conf/flyway.conf. Para o exemplo desse curso, iremos utilizar um banco de dados SQL Server, de forma que a string de conexão fica da maneira mostrada a seguir:

```
flyway.url=jdbc:sqlserver://W16SQL17:1440;databaseName=AdventureWork
s2
flyway.user= user01
flyway.password= user01
```

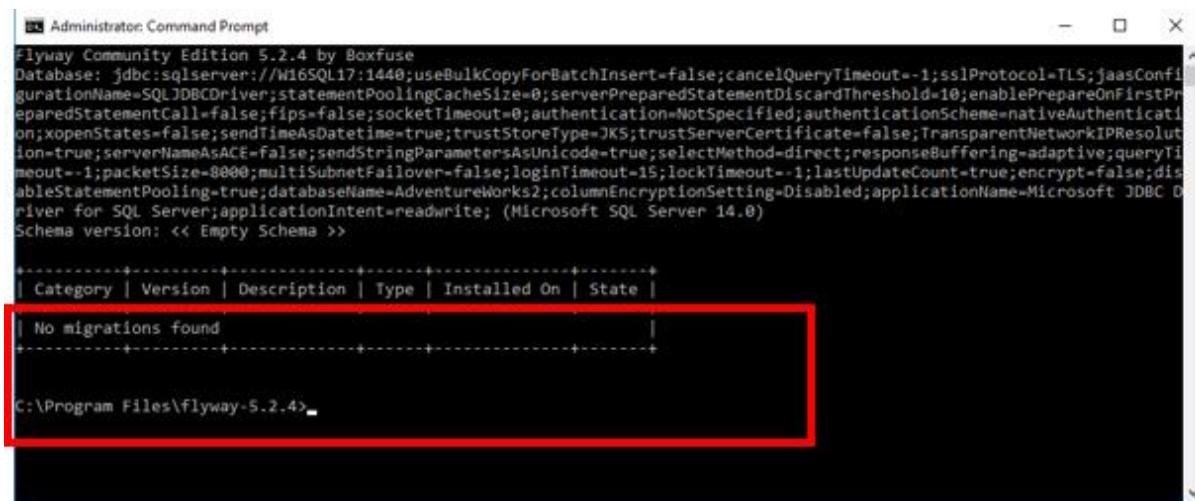
**Figura 137 – Exemplo de arquivo de configuração.**



**Fonte:** Gustavo (2019).

Feito isso, para testar a conexão, basta usar o comando ***flyway info***. Como ainda não foi configurada nenhuma migration, o resultado será como o mostrado abaixo, caso a conexão com o banco seja feita com sucesso. Nesse ponto, o Flyway já está pronto para ser usado, o que será visto no próximo tópico.

**Figura 138 – Teste de Conexão com o Flyway.**



```
Administrator: Command Prompt
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:sqlserver://W16SQL17:1440;useBulkCopyForBatchInsert=false;cancelQueryTimeout=-1;sslProtocol=TLS;jaasConfigurationName=SQLJDBCDriver;statementPoolingCacheSize=0;serverPreparedStatementDiscardThreshold=10;enablePrepareOnFirstPreparedStatementCall=false;fips=false;socketTimeout=0;authentication=NotSpecified;authenticationScheme=nativeAuthentication;xopenStates=false;sendTimeAsDatetime=true;trustStoreType=JKS;trustServerCertificate=false;TransparentNetworkIPResolution=true;serverNameAsACE=false;sendStringParametersAsUnicode=true;selectMethod=direct;responseBuffering=adaptive;queryTimeout=-1;packetSize=8000;multiSubnetFailover=false;loginTimeout=15;lockTimeout=-1;lastUpdateCount=true;encrypt=false;disallowStatementPooling=true;databaseName=AdventureWorks2;columnEncryptionSetting=Disabled;applicationName=Microsoft JDBC Driver for SQL Server;applicationIntent=readwrite; (Microsoft SQL Server 14.0)
Schema version: << Empty Schema >>

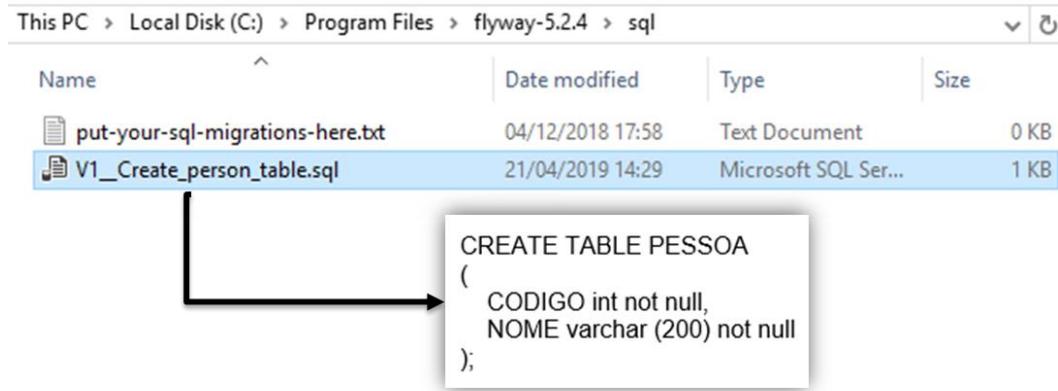
+-----+-----+-----+-----+
| Category | Version | Description | Type | Installed On | State |
+-----+-----+-----+-----+
| No migrations found |
+-----+-----+-----+-----+-----+-----+
C:\Program Files\flyway-5.2.4>
```

Fonte: Gustavo (2019).

### 13.5. Utilização do Flyway

Como dito inicialmente, o Flyway utiliza-se do conceito de migrations para aplicar as alterações no banco de dados e realizar o seu controle de versões. As migrations, por sua vez, são criadas no formato de **arquivos com a extensão .sql**, no **diretório /sql** do caminho de instalação do Flyway.

**Figura 139 – Diretório de Armazenamento das Migrations.**



**Fonte:** Gustavo (2019).

Existem três tipos de migrations, cada um com um propósito específico:

- **Versioned Migrations:** migrations mais utilizadas, que servem para promover uma nova implantação ou alteração no banco de dados.
- **Undo Migrations:** migrations para desfazer o que foi feito por uma versioned migration.
- **Repeatable Migrations:** migrations não versionadas.

Para identificar o tipo de cada migration e controlar as migrações que já foram efetivadas, o Flyway utiliza um padrão de nomenclatura nos arquivos .sql, como mostrado a seguir:

**Prefixo:**

- V para migrations versionada.
- U para migrations de undo.
- R para repeatable migrations.

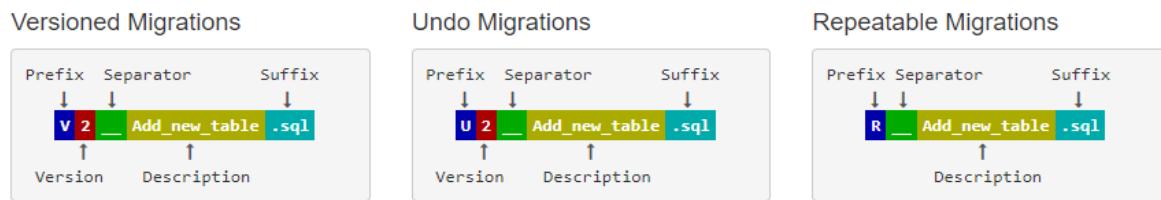
**Versão:** número sequencial para controlar a versão;

**Separador:** dois underscores “\_\_”;

**Descrição:** underscore ou espaço separando as palavras;

**Sufixo:** .sql.

**Figura 140 – Padrão de nomenclatura das migrations.**



**Fonte: Flyway (2019).**

A título de exemplo, para criar a primeira migration, vamos salvar o arquivo **V1\_Cria\_Tabela\_Pessoa.sql** no diretório /sql, com o seguinte script:

```
CREATE TABLE PESSOA
(
 CODIGO int not null,
 NOME varchar (200) not null
);
```

Feito isso, basta executar o comando **flyway migrate** para que as migrations sejam aplicadas no banco de dados, como mostrado a seguir:

**Figura 141 – Implantando Migrations.**

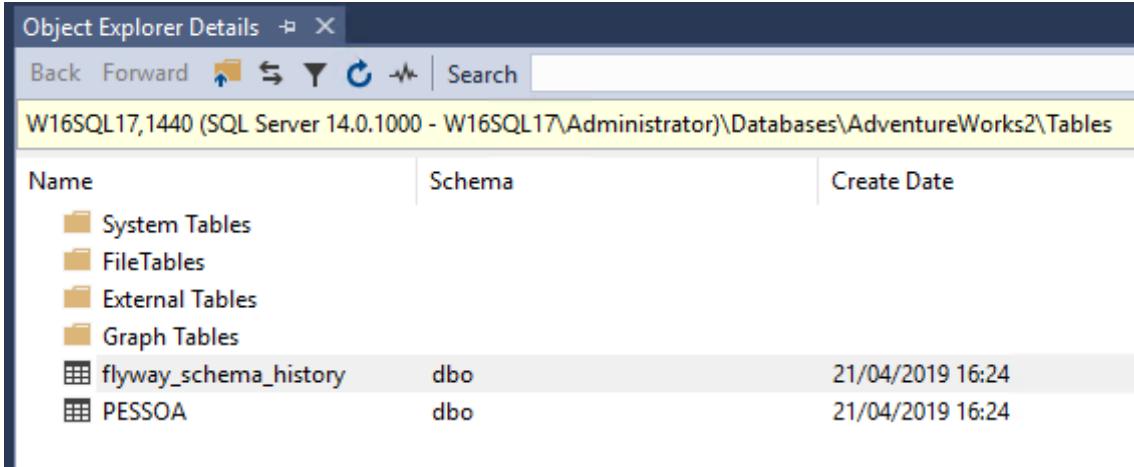
```
C:\Program Files\flyway-5.2.4>flyway migrate
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:sqlserver://W16SQL17:1440;useBulkCopyForBatchInsert=false;cancelQueryTimeout=-1;sslProtocol=TLS;jaasConfigurationName=SQLJDBCDriver;statementPoolingCacheSize=0;serverPreparedStatementDiscardThreshold=10;enablePrepareOnFirstPreparedStatementCall=false;fips=false;socketTimeout=0;authentication=NotSpecified;authenticationScheme=nativeAuthentication;xopenStates=false;sendTimeAsDatetime=true;trustStoreType=JKS;trustServerCertificate=false;TransparentNetworkIPResolution=true;serverNameAsACE=false;sendStringParametersAsUnicode=true;selectMethod=direct;responseBuffering=adaptive;queryTimeout=1;packetSize=8000;multiSubnetFailover=false;loginTimeout=15;lockTimeout=-1;lastUpdateCount=true;encrypt=false;disposableStatementPooling=true;databaseName=AdventureWorks2;columnEncryptionSetting=Disabled;applicationName=Microsoft JDBC Driver for SQL Server;applicationIntent=readwrite; (Microsoft SQL Server 14.0)
Successfully validated 1 migration (execution time 00:00.071s)
Creating Schema History table: [AdventureWorks2].[dbo].[flyway_schema_history]
Current version of schema [dbo]: << Empty Schema >>
Migrating schema [dbo] to version 1 - Cria Tabela Pessoa
Successfully applied 1 migration to schema [dbo] (execution time 00:00.131s)

C:\Program Files\flyway-5.2.4>
```

**Fonte: Gustavo (2019).**

Após a execução com sucesso desse comando, além da implantação do objeto definido no script da migration, por se tratar da **primeira migration**, o Flyway criará, também, uma **tabela para controle do versionamento**, chamada ***flyway\_schema\_history***.

**Figura 142 – Implantação de uma Primeira Migration.**

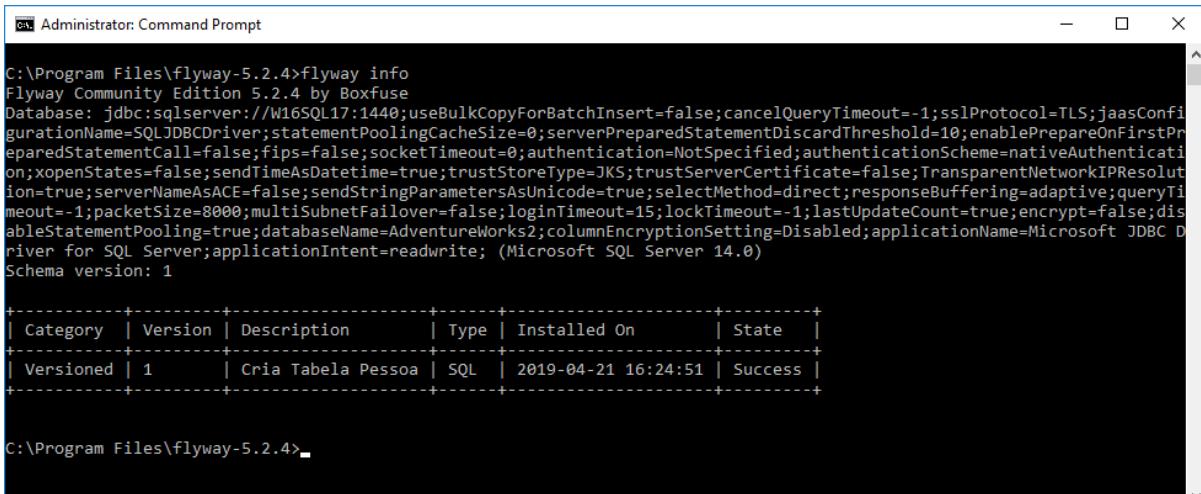


| Name                         | Schema | Create Date      |
|------------------------------|--------|------------------|
| System Tables                |        |                  |
| FileTables                   |        |                  |
| External Tables              |        |                  |
| Graph Tables                 |        |                  |
| <b>flyway_schema_history</b> | dbo    | 21/04/2019 16:24 |
| <b>PESSOA</b>                | dbo    | 21/04/2019 16:24 |

**Fonte: Gustavo (2019).**

Por fim, para verificar informação das migrations já implantadas e versionadas, deve-se utilizar o comando ***flyway info***.

**Figura 143 – Verificando Informações das Migrations.**



```
C:\Program Files\flyway-5.2.4>flyway info
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:sqlserver://W16SQL17:1440;useBulkCopyForBatchInsert=false;cancelQueryTimeout=-1;sslProtocol=TLS;jaasConfigurationName=SQLJDBCDriver;statementPoolingCacheSize=0;serverPreparedStatementDiscardThreshold=10;enablePrepareOnFirstPreparedStatementCall=false;fips=false;socketTimeout=0;authentication=NotSpecified;authenticationScheme=nativeAuthentication;xopenStates=false;sendTimeAsDatetime=true;trustStoreType=JKS;trustServerCertificate=false;TransparentNetworkIPResolution=true;serverNameAsACE=false;sendStringParametersAsUnicode=true;selectMethod=direct;responseBuffering=adaptive;queryTimeout=-1;packetSize=8000;multiSubnetFailover=false;loginTimeout=15;lockTimeout=-1;lastUpdateCount=true;encrypt=false;disallowStatementPooling=true;databaseName=AdventureWorks2;columnEncryptionSetting=Disabled;applicationName=Microsoft JDBC Driver for SQL Server;applicationIntent=readwrite; (Microsoft SQL Server 14.0)
Schema version: 1

+-----+-----+-----+-----+-----+
| Category | Version | Description | Type | Installed On | State |
+-----+-----+-----+-----+-----+
| Versioned | 1 | Cria Tabela Pessoa | SQL | 2019-04-21 16:24:51 | Success |
+-----+-----+-----+-----+-----+

C:\Program Files\flyway-5.2.4>
```

**Fonte: Gustavo (2019).**

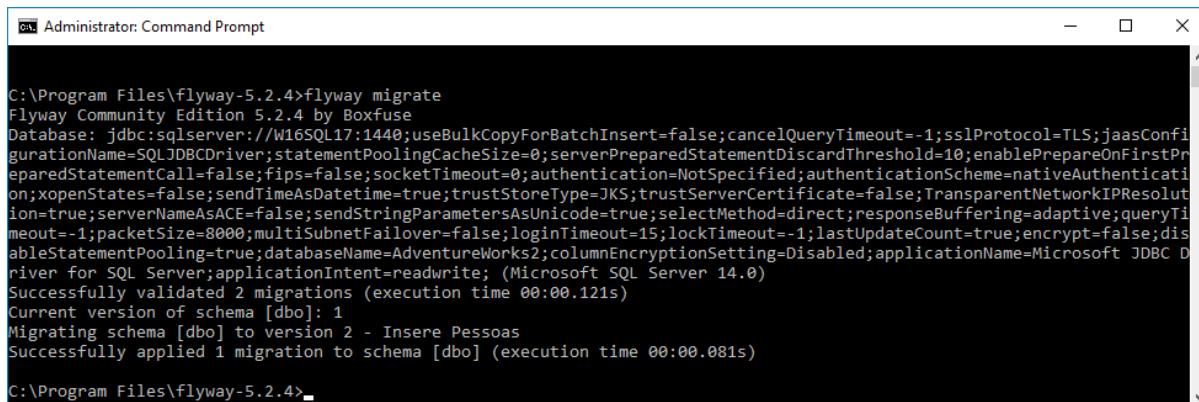
Como dito inicialmente, o Flyway permite que também sejam versionados scripts DML, de manipulação de dados. Para demonstrar isso, vamos criar uma segunda migration, de nome **V2\_\_Insere\_Pessoas.sql**:

```
INSERT INTO PESSOA (CODIGO, NOME) values (1, 'João');
```

```
INSERT INTO PESSOA (CODIGO, NOME) values (2, 'Maria');
```

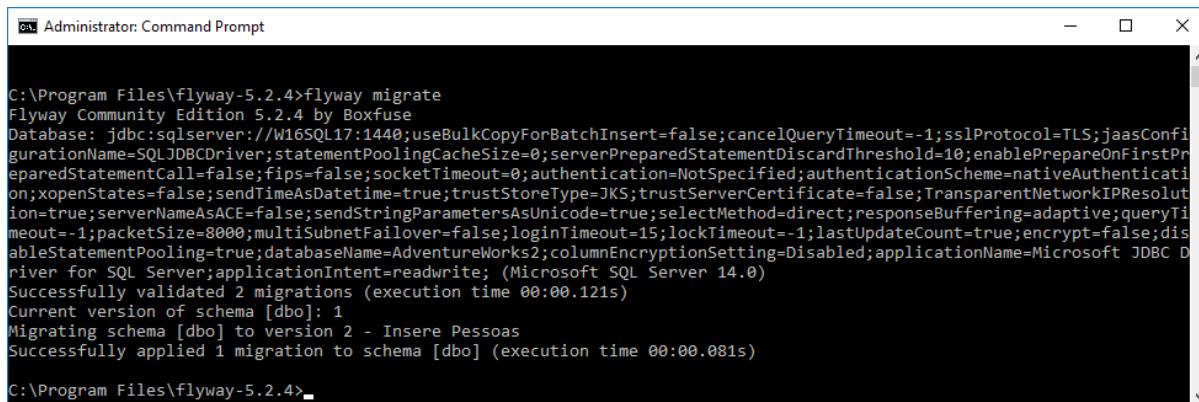
Ao executar novamente o comando **flyway migrate**, esse script será aplicado no banco de dados e incluído no controle de versões.

**Figura 144 – Implantando uma Migration de DML.**



```
C:\Program Files\flyway-5.2.4>flyway migrate
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:sqlserver://W16SQL17:1440;useBulkCopyForBatchInsert=false;cancelQueryTimeout=-1;sslProtocol=TLS;jaasConfigurationName=SQLJDBCDriver;statementPoolingCacheSize=0;serverPreparedStatementDiscardThreshold=10;enablePrepareOnFirstPreparedStatementCall=false;fips=false;socketTimeout=0;authentication=NotSpecified;authenticationScheme=nativeAuthentication;xopenStates=false;sendTimeAsDatetime=true;trustStoreType=JKS;trustServerCertificate=false;TransparentNetworkIPResolution=true;serverNameAsACE=false;sendStringParametersAsUnicode=true;selectMethod=direct;responseBuffering=adaptive;queryTimeout=-1;packetSize=8000;multiSubnetFailover=false;loginTimeout=15;lockTimeout=-1;lastUpdateCount=true;encrypt=false;disallowStatementPooling=true;databaseName=AdventureWorks2;columnEncryptionSetting=Disabled;applicationName=Microsoft JDBC Driver for SQL Server;applicationIntent=readwrite; (Microsoft SQL Server 14.0)
Successfully validated 2 migrations (execution time 00:00.121s)
Current version of schema [dbo]: 1
Migrating schema [dbo] to version 2 - Insere_Pessoas
Successfully applied 1 migration to schema [dbo] (execution time 00:00.081s)

C:\Program Files\flyway-5.2.4>
```



```
C:\Program Files\flyway-5.2.4>flyway migrate
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:sqlserver://W16SQL17:1440;useBulkCopyForBatchInsert=false;cancelQueryTimeout=-1;sslProtocol=TLS;jaasConfigurationName=SQLJDBCDriver;statementPoolingCacheSize=0;serverPreparedStatementDiscardThreshold=10;enablePrepareOnFirstPreparedStatementCall=false;fips=false;socketTimeout=0;authentication=NotSpecified;authenticationScheme=nativeAuthentication;xopenStates=false;sendTimeAsDatetime=true;trustStoreType=JKS;trustServerCertificate=false;TransparentNetworkIPResolution=true;serverNameAsACE=false;sendStringParametersAsUnicode=true;selectMethod=direct;responseBuffering=adaptive;queryTimeout=-1;packetSize=8000;multiSubnetFailover=false;loginTimeout=15;lockTimeout=-1;lastUpdateCount=true;encrypt=false;disallowStatementPooling=true;databaseName=AdventureWorks2;columnEncryptionSetting=Disabled;applicationName=Microsoft JDBC Driver for SQL Server;applicationIntent=readwrite; (Microsoft SQL Server 14.0)
Successfully validated 2 migrations (execution time 00:00.121s)
Current version of schema [dbo]: 1
Migrating schema [dbo] to version 2 - Insere_Pessoas
Successfully applied 1 migration to schema [dbo] (execution time 00:00.081s)

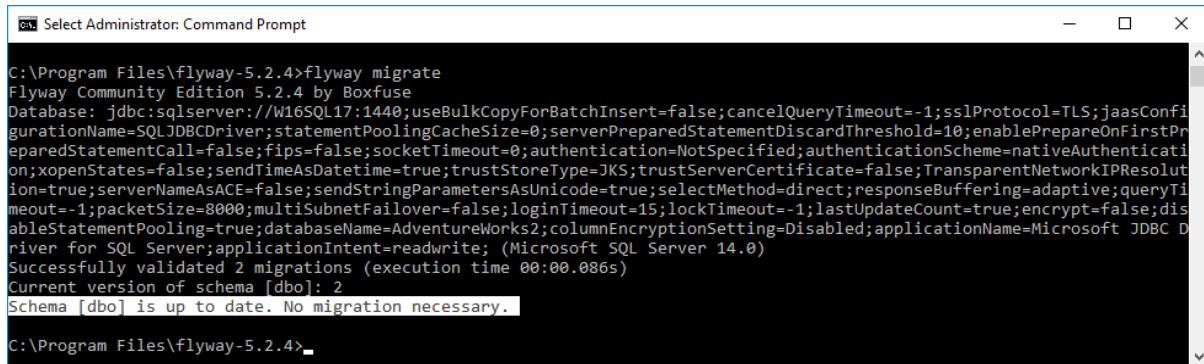
C:\Program Files\flyway-5.2.4>
```

**Fonte: Gustavo (2019).**

Se consultarmos a tabela no banco de dados, constataremos que os dados foram inseridos corretamente.

Para além disso, caso seja executado um comando de migrate, sem migrations novas a serem implantadas, o Flyway consegue identificar isso e retorna uma mensagem, como a mostrada abaixo:

**Figura 145 – Mensagem quando não há novas migrations.**



```
C:\Program Files\flyway-5.2.4>flyway migrate
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:sqlserver://W16SQL17:1440;useBulkCopyForBatchInsert=false;cancelQueryTimeout=-1;sslProtocol=TLS;jaasConfigurationName=SQLJDBCDriver;statementPoolingCacheSize=0;serverPreparedStatementDiscardThreshold=10;enablePrepareOnFirstPreparedStatementCall=false;fips=false;socketTimeout=0;authentication=NotSpecified;authenticationScheme=nativeAuthentication;xopenStates=false;sendTimeAsDatetime=true;trustStoreType=JKS;trustServerCertificate=false;TransparentNetworkIPResolution=true;serverNameAsACE=false;sendStringParametersAsUnicode=true;selectMethod=direct;responseBuffering=adaptive;queryTimeout=-1;packetSize=8000;multiSubnetFailover=false;loginTimeout=15;lockTimeout=-1;lastUpdateCount=true;encrypt=false;disallowStatementPooling=true;databaseName=AdventureWorks2;columnEncryptionSetting=Disabled;applicationName=Microsoft JDBC Driver for SQL Server;applicationIntent=readwrite; (Microsoft SQL Server 14.0)
Successfully validated 2 migrations (execution time 00:00:00.086s)
Current version of schema [dbo]: 2
Schema [dbo] is up to date. No migration necessary.

C:\Program Files\flyway-5.2.4>
```

**Fonte: Gustavo (2019).**

## Referências

---

AMERICAN National Standards Institute. *The SQL Standard – ISO/IEC 9075:2016.* Disponível em: <https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/>. Acesso em: 28 maio 2021.

BERNARDI, Diogo Alencar. *Técnicas de mapeamento objeto relacional.* Disponível em: <https://www.devmmedia.com.br/tecnicas-de-mapeamento-objeto-relacional-revista-sql-magazine-40/6980>. Acesso em: 28 maio 2021.

CELEPAR. *Guia para Mapeamento Objeto Relacional Metodologia Celepar.* Agosto 2009.

CHAMBERLIN, Donald D.; BOYCE, Raymond F. SEQUEL: *A Structured English Query Language.* Disponível em: <http://www.joakimdalby.dk/HTML/sequel.pdf>. Acesso em: 28 maio 2021.

CHEN, Peter. *Modelagem de Dados.* São Paulo: McGraw-Hill, Makron, 1990.

CHEN, Peter. *Peter Chen Home Page at Louisiana State University (LSU).* Disponível em: <https://www.csc.lsu.edu/~chen>. Acesso em: 28 maio 2021.

CHEN, Peter. *The Entity-Relationship Model - Toward a Unified View of Data.* Disponível em: <https://dspace.mit.edu/bitstream/handle/1721.1/47432/entityrelationshx00chen.pdf>. Acesso em: 28 maio 2021.

CODD, Edgar F. *A Relational Model of Data for Large Shared Data Banks.* Disponível em: <https://github.com/dmvaldman/library/blob/master/computer%20science/Codd%20-%20A%20Relational%20Model%20of%20Data%20for%20Large%20Shared%20Data%20Banks.pdf>. Acesso em: 28 maio 2021.

CODD, Edgar F. *The Relational Model for Database Management: Version 2.* United States Of America: Addison Wesley Publishing Company, 1990.

COUGO, Paulo. *Modelagem Conceitual e Projeto de Banco de Dados*. 14<sup>a</sup> reimpressão. Rio de Janeiro: Elsevier, 1997.

FLYWAY. Disponível em: <https://flywaydb.org/>. Acesso em: 28 maio 2021.

HILLS, Ted. *NOSQL and SQL Data Modeling: Bringing Together Data, Semantics, and Software*. Technics Publications, 2016.

IBM. *A History and Evaluation of System R*. Disponível em: <https://people.eecs.berkeley.edu/~brewer/cs262/SystemR.pdf>. Acesso em: 28 maio 2021.

KORTH, Henry F. *Sistema de bancos de dados*. 3<sup>a</sup> ed. São Paulo: McGraw-Hill, 1999.

MACHADO, Felipe Nery Rodrigues; ABREU, Maurício Pereira de. *Projeto de Banco de Dados: uma visão prática*. 13<sup>a</sup> ed. São Paulo: Érica, 1996.

MACORATTI.NET. *Node.js – Apresentando e usando o Sequelize* (acesso ao MySQL). Disponível em: [http://www.macoratti.net/17/01/node\\_sequelize1.html](http://www.macoratti.net/17/01/node_sequelize1.html). Acesso em: 28 maio 2021.

MICHAELIS. *Dicionário da Língua Portuguesa Brasileira*. Disponível em: <https://michaelis.uol.com.br/moderno-portugues>. Acesso em: 28 maio 2021.

MICROSOFT SQL Documentation. Disponível em: <https://docs.microsoft.com/en-us/sql/?view=sql-server-2017>. Acesso em: 28 maio 2021.

MICROSOFT SQL Server 2017. Disponível em: <https://www.microsoft.com/pt-br/sql-server/sql-server-2017>. Acesso em: 28 maio 2021.

MONQUEIRO, Julio Cesar Bessa. *Programação Orientada a Objetos: uma introdução*. Disponível em: <https://www.hardware.com.br/artigos/programacao-orientada-objetos>. Acesso em: 28 maio 2021.

NAVATHE, Shamkant B.; ELSMARI, Ramez. *Sistemas de Banco de Dados*. 4<sup>a</sup> edição. São Paulo: Pearson Addison Wesley, 2005.

NOSQL Database Org. Disponível em: <http://nosql-database.org/>. Acesso em: 28 maio 2021.

PAT RESEARCH. *Top 9 Object Databases*. Disponível em: <https://www.predictiveanalyticstoday.com/top-object-databases>. Acesso em: 28 maio 2021.

RODRIGUES, Joel. *Entity Framework*: Como fazer seu primeiro Mapeamento Objeto-Relacional. Disponível em: <https://www.devmedia.com.br/entity-framework-como-fazer-seu-primeiro-mapeamento-objeto-relacional/38756>. Acesso em: 28 maio 2021.

ROEBUCK, Kevin. *Object-Relational Mapping (Orm)*: High-Impact Strategies. Paperback, 2011.

SEQUELIZE. Disponível em: <http://docs.sequelizejs.com/>. Acesso em: 28 maio 2021.

SHALER, Sally; MELLOR, Stephen J. *Object Oriented Systems Analysis*: Modeling the World in Data. 1. ed. Prentice Hall, 1988.

SIMPLILEARN. *Introduction to NOSQL Databases Tutorial*. Disponível em: <https://www.simplilearn.com/introduction-to-nosql-databases-tutorial-video>. Acesso em: 28 maio 2021.

STROZZI, Carlo. *NoSQL: A non-SQL RDBMS*. Disponível em: [http://www.strozzi.it/cgi-bin/CSA/tw7/l/en\\_US/nosql/Home%20Page](http://www.strozzi.it/cgi-bin/CSA/tw7/l/en_US/nosql/Home%20Page). Acesso em: 28 maio 2021.

UNQL ORG. Disponível em: <http://unql.sqlite.org/index.html/wiki?name=UnQL>. Acesso em: 28 maio 2021.

UNQL QUERY LANGUAGE UNVEILED BY COUCHBASE AND SQLITE. Disponível em: <https://www.couchbase.com/press-releases/unql-query-language>. Acesso em: 28 maio 2021.

VERSIONSQL. Disponível em: <https://www.versionsql.com/>. Acesso em: 28 maio 2021.

