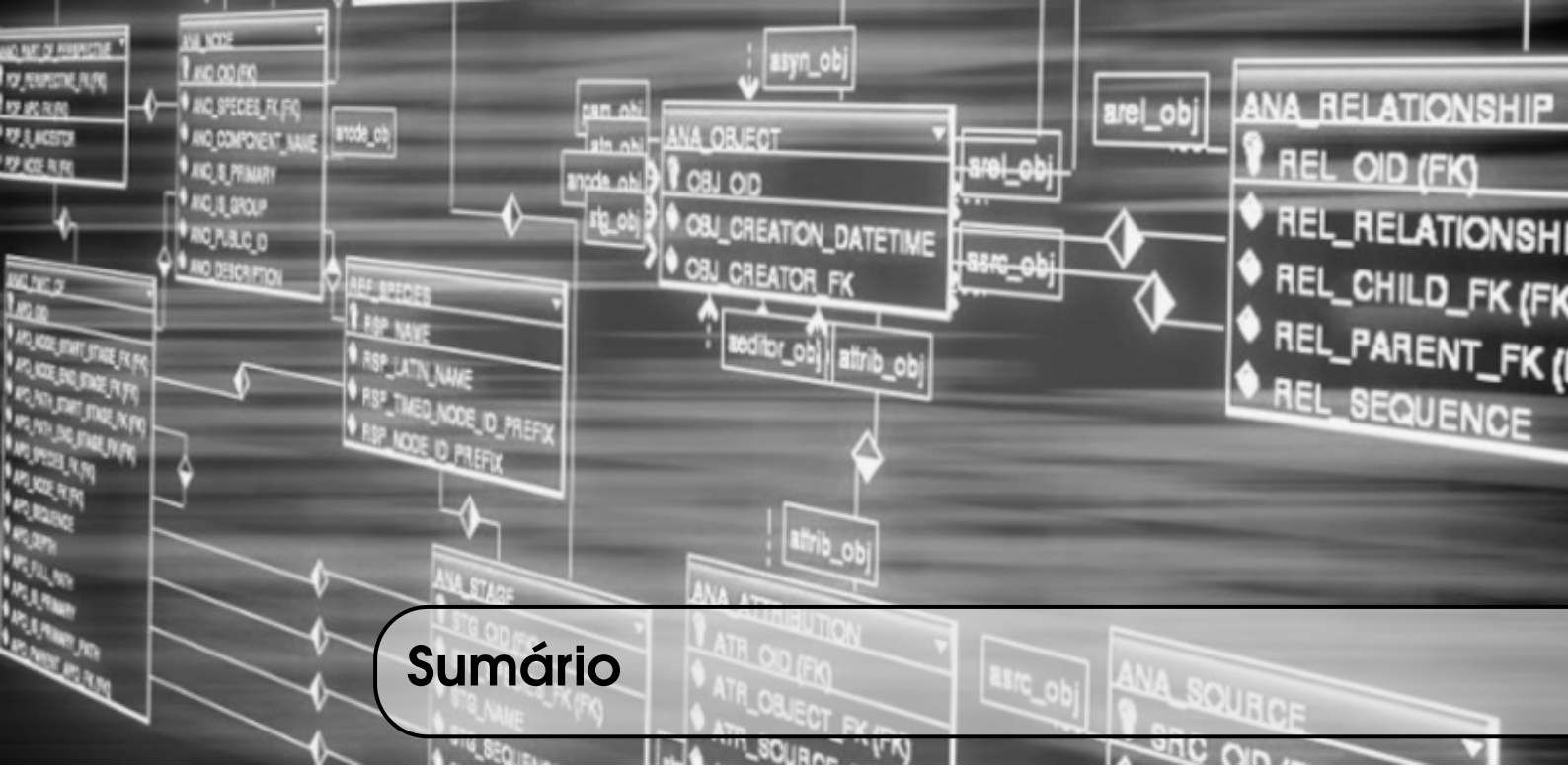
Structured Query Language

Newton Carlos Will



Sumário

1. Introdução

A linguagem SQL pode ser considerada um dos principais motivos para o sucesso dos bancos de dados relacionais comerciais. Como ela se tornou um padrão para esse tipo de banco de dados, os usuários ficaram menos preocupados com a migração de suas aplicações de outros tipos de sistemas de banco de dados - por exemplo, sistemas de rede e hierárquicos - para sistemas relacionais. Isso aconteceu porque, mesmo que os usuários estivesse insatisfeitos com o produto de SGBD relacional em particular que estavam usando, a conversão para outro produto SGBD relacional não seria tão cara ou demorada, pois os dois sistemas seguiam os mesmos padrões de linguagem. Na prática, é óbvio, existem muitas diferenças entre diversos pacotes de SGBD relacionais comerciais. Porém, se o usuário for cuidadoso em usar apenas dos recursos que fazem parte do padrão, e se os dois sistemas relacionais admitirem fielmente o padrão, então a conversão entre ambos deverá ser bastante simplificada. Outra vantagem de ter esse padrão é que os usuários podem escrever comandos em um programa de aplicação de banco de dados que pode acessar dados em dois ou mais SGBDs relacionais sem ter de mudar a sublinguagem de banco de dados (SQL) se os sistemas admitirem o padrão SQL.

1.1 Histórico

Em junho de 1970, E. F. Codd, membro do Laboratório de Pesquisa da IBM em San Jose, na Califórnia, publicou um trabalho intitulado “A Relational Model of Data for Large Shared Data Banks” (Um Modelo Relacional de Dados para Grandes Bancos de Dados Compartilhados), no jornal Association of Computer Machinery. Nesse trabalho, Codd estabeleceu príncipios sobre gerência de banco de dados, denominando-os com o termo relacional. Essa foi a base utilizada na criação de uma linguagem-padrão para manipular informações em Banco de Dados Relacionais. E essa linguagem é a SQL (Structured Query Language).

Inicialmente chamada de SEQUEL (Structured English Query Language), a linguagem SQL foi concebida e desenvolvida pela IBM, utilizando os conceitos de Codd. Em 1979, a Relational Software Inc., hoje Oracle Corporation, lançou a primeira versão comercial da linguagem SQL.

Atualmente, a SQL pode ser considerada um padrão para manipulação de dados em um banco de dados. Duas entidades, a ANSI (American National Standards Institute) e a ISO (International Standards Organization), vêm ao longo do tempo padronizando a linguagem SQL. O primeiro padrão (SQL-86) foi definido pela ANSI em 1986 e consistia basicamente na SQL da IBM, com poucas modificações.

Em 1987, a ISO adotou o mesmo padrão. Em 1989, surge uma nova versão (SQL-89), com significativas modificações. Essa versão é utilizada pelos bancos de dados atuais. Em 1992, houve uma nova versão aprimorando a anterior (SQL-92). Essa versão define as regras básicas para os bancos de dados relacionais. Em 1999 surge a SQL-99, conhecida também como SQL3, que define um modelo de banco de dados objeto-relacional.

Divide-se o padrão SQL-92 em quatro níveis: Entry (básico), Transational (em evolução), Intermediate (intermediário) e Full (completo). A maior parte dos bancos de dados utilizados atualmente atende ao nível básico. Mesmo existindo uma versão mais nova do padrão, a maior parte dos bancos de dados ainda utiliza, de forma básica, o padrão anterior. Alguns comandos, contudo, atingem os níveis intermediário e completo.

1.2 O que é SQL?

SQL (Structured Query Language) é um conjunto de comandos de manipulação de banco de dados utilizado para criar e manter a estrutura desse banco de dados, além de incluir, excluir, modificar e pesquisar informações nas tabelas dele. A linguagem SQL não é uma linguagem de programação autônoma; poderia ser chamada de "sublinguagem". Quando se escrevem aplicações para banco de dados, é necessário utilizar uma linguagem de programação tradicional (C, Java, Pascal, COBOL, etc...) e embutir comandos SQL para manipular os dados.

Em um modelo relacional, apenas uma tipo de estrutura de dados existe: a tabela. Novas tabelas são criadas com a junção ou combinação de outras tabelas. Utilizando apenas um comando SQL é possível pesquisar dados em diversas tabelas ou atualizar e excluir diversas linhas das mesmas.

A linguagem SQL não é procedural, logo é possível especificar o que deve ser feito, e não como deve ser feito. Dessa forma, um conjunto de linhas será atingido pelo comando e não cada uma das linhas, como é feito no ambiente procedural. Portanto, não é necessário entender o funcionamento interno do banco de dados e como e onde estão armazenados fisicamente os dados.

1.3 Modelo do Banco de Dados Utilizado nos Exemplos

Para os exemplos que se seguirão nos próximos capítulos, será utilizado, basicamente, o modelo relacional descrito na Figura 1.1, o qual apresenta o modelo para um banco de dados básico de uma locadora.

Os comandos SQL colocados neste documento seguem o padrão SQL-92, contendo alguns comandos específicos do MySQL, que são informados explicitamente.

Como referência ao diagrama da Figura 1.1, a nomenclatura é como segue:

- **(PK):** Primary Key - Chave Primária
- **(FK):** Foreign Key - Chave Estrangeira
- **NN:** Not Null - Campo não nulo

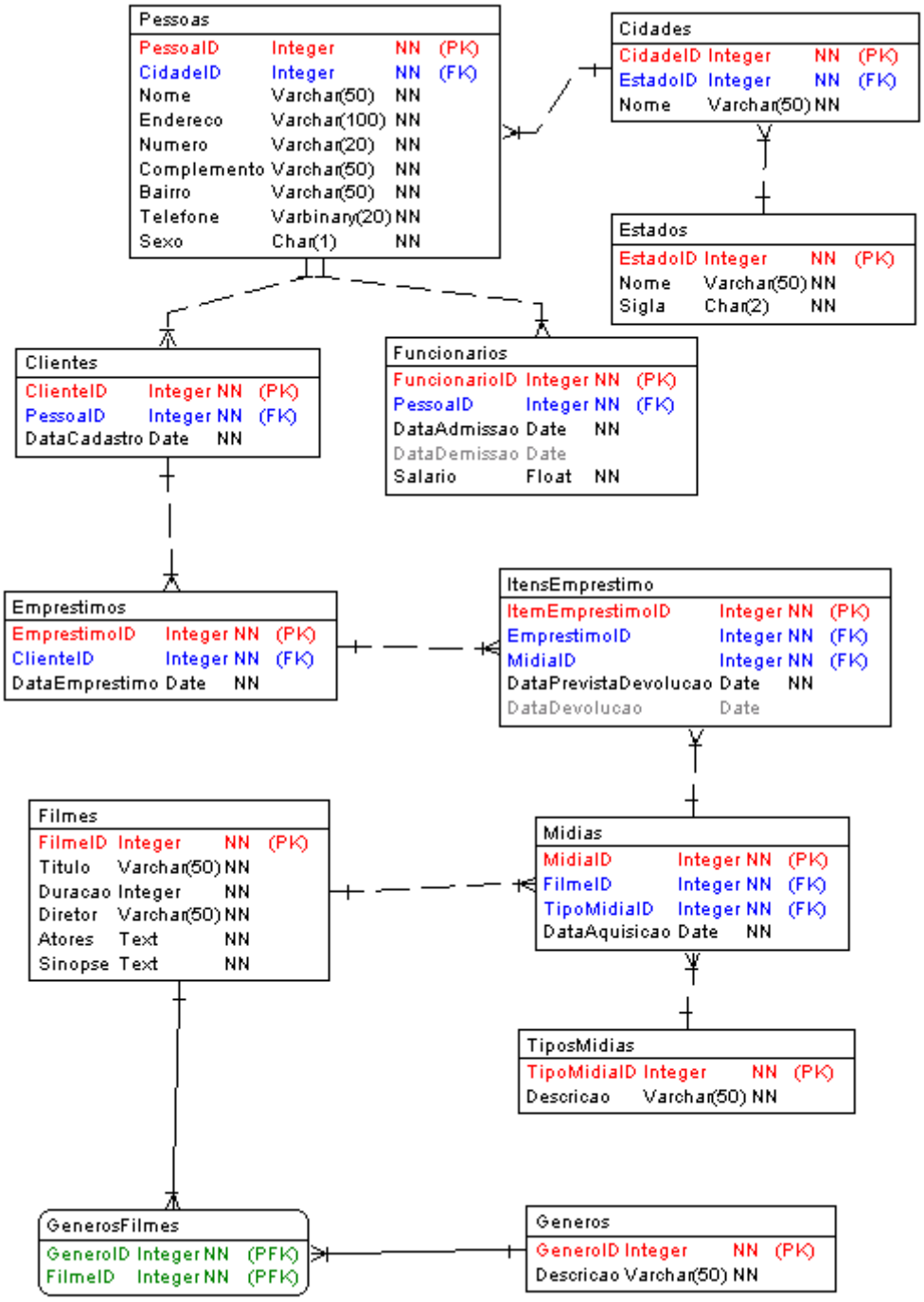
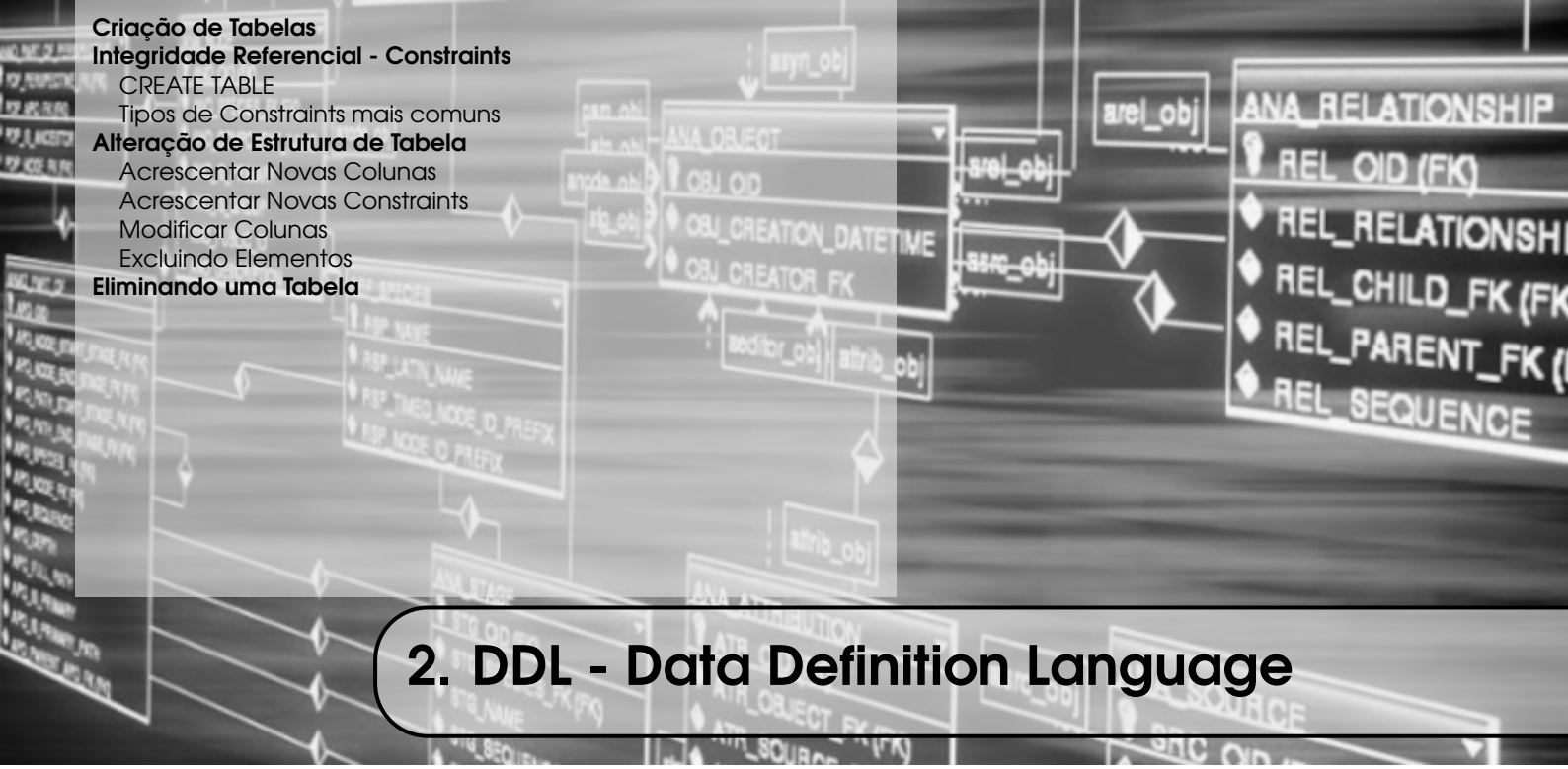


Figura 1.1: Modelo relacional de banco de dados para uma locadora, utilizado nos exemplos descritos nos próximos capítulos



2. DDL - Data Definition Language

A linguagem SQL se divide, basicamente, em duas partes: A DDL e a DML. A DDL (Data Definition Language ou Linguagem de Definição de Dados) permite ao utilizador definir tabelas novas e elementos associados. A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL.

Os comandos básicos da DDL são poucos:

- CREATE: cria um objeto (uma tabela, por exemplo) dentro da base de dados.
- DROP: apaga um objeto do banco de dados.

Alguns sistemas de banco de dados usam o comando ALTER, que permite ao usuário alterar um objeto, por exemplo, adicionando uma coluna a uma tabela existente.

2.1 Criação de Tabelas

Tabelas são as estruturas mais importantes de um banco de dados. Nas tabelas estará o conteúdo que representa cada objeto do mundo real, cuja importância para o funcionamento do sistema justifica a sua criação. As próprias tabelas criadas no banco de dados ficam armazenadas em tabelas internas do gerenciador de banco de dados. É o chamado Dicionário de Dados. Com a utilização do dicionário de dados, as informações das tabelas ficam sempre à disposição do usuário e permitem um controle maior do que está sendo armazenado.

O padrão SQL divide em três categorias as tabelas de um banco de dados:

1. Tabelas permanentes: ficam permanentemente armazenadas no banco de dados, a menos que sejam explicitamente excluídas.
2. Tabelas temporárias globais: utilizadas para armazenamento temporário de informações durante algum processamento específico, são eliminadas assim que for encerrado o acesso ao banco de dados. Possuem visibilidade em todo o banco de dados.
3. Tabelas temporárias locais: semelhante às globais, são visíveis apenas em um módulo específico em que foram criadas.

Note que as tabelas temporárias possuem um conceito diferente de Visões de banco de dados (VIEWS), que serão apresentadas adiante.

As tabelas que criaremos para armazenar as informações no nosso sistema serão tabelas permanentes. Para criá-las, utilizaremos o comando CREATE TABLE.

2.2 Integridade Referencial - Constraints

Constraints são regras agregadas a colunas ou tabelas. Assim, pode-se definir como obrigatório o preenchimento de uma coluna que tenha um valor-padrão quando uma linha for incluída na tabela ou quando aceitar apenas alguns valores predefinidos. No caso de regras aplicadas a tabelas, tem-se a definição de chaves primárias e estrangeiras.

2.2.1 CREATE TABLE

A sintaxe básica do comando é a seguinte:

```
CREATE TABLE nome-tabela(
        nome-coluna1 tipo-de-dado-coluna1 constraint-coluna1,
        nome-coluna2 tipo-de-dado-coluna2 constraint-coluna2,
        ...
        nome-colunan tipo-de-dado-colunan constraint-colunan,
        constraint-de-tabela
)
```

onde:

Argumento	Descrição
`nome-tabela`	Nome da tabela. Deve ser único para o usuário. Não pode coincidir com o nome de outros objetos do banco de dados de um mesmo usuário.
`nome-coluna`	Nome da coluna. Esse nome deve ser único e exclusivo na tabela.
`tipo-de-dado-coluna`	Tipo de dado que constará no campo.
`constraint-coluna`	Regras agregadas à coluna.
`constraint-de-tabela`	Regras agregadas à tabela inteira.

Exemplo:

```
CREATE TABLE Estados (
        EstadoID INTEGER NOT NULL,
        Nome VARCHAR(50) NOT NULL DEFAULT '',
        Sigla CHAR(2) NOT NULL DEFAULT '',
        CONSTRAINT pk_Estados_ID PRIMARY KEY(EstadoID)
)
```

2.2.2 Tipos de Constraints mais comuns

Serão analisados alguns tipos de constraint encontrados com mais frequência nos bancos de dados. As constraints variam muito de um banco de dados para outro.

DEFAULT

Serve para atribuir um conteúdo padrão a uma coluna da tabela, sempre que for incluída uma nova linha na tabela. Deve-se especificar a palavra-chave DEFAULT, seguida do conteúdo padrão. Como exemplo, na tabela Estados, define-se o valor padrão para o campo Nome da seguinte forma:

```
...
Nome VARCHAR(50) NOT NULL DEFAULT '',
...
```

onde especifica-se que, caso não seja informado o valor para o campo, será atribuída uma *string* vazia para o mesmo.

NOT NULL

Indica que o conteúdo de uma coluna não poderá ser *nulo*. Se cada coluna não tiver valor atribuído durante uma inclusão, terá seu valor *nulo* (caso não seja especificado o valor padrão).

Desta forma, ao tentar incluir uma coluna com essa restrição, que não apresenta valor, o banco de dados retornará uma mensagem de erro e não incluirá a linha. Imagine um caso em que não se possa incluir um estado se que seja preenchido o nome. A coluna Nome da tabela Estados poderia ser criada da seguinte forma:

```
1 ...
2 Nome VARCHAR(50) NOT NULL,
3 ...
```

Observe que, por não ter nenhum valor padrão definido para o campo, em caso de uma inclusão de registro onde não seja especificado o valor para o nome, este teria o valor *nulo*, mas a operação não será efetuada pela restrição de não conter valores nulos para o campo.

UNIQUE

Indica que não pode haver repetição no conteúdo da coluna. Isso é diferente do conceito de chave primária. A chave primária, além de não permitir repetição, não pode conter valores nulos. Ao especificar que uma coluna deve conter valores únicos, é indicado que todos os valores não nulos devem ser exclusivos. Deve-se acrescentar a cláusula UNIQUE para os campos que devam ter essa característica. Ainda na tabela Estados, pode-se especificar a Sigla como um campo de valor único na tabela, assim:

```
1 ...
2 Sigla CHAR(2) NOT NULL DEFAULT '' UNIQUE,
3 ...
```

Chave Primária

Chave primária é a coluna, ou grupo de colunas, que permite identificar um único registro na tabela. Para especificar que uma coluna ou grupo de colunas representa a chave primária de uma tabela, deve-se acrescentar a palavra-chave PRIMARY KEY seguida do nome da(s) coluna(s). Alguns bancos de dados permitem especificar a chave primária como constraint da coluna (quando a chave primária for apenas uma coluna) ou como constraint de tabela. Outros aceitam apenas como constraint de tabela. Para facilitar, aconselha-se utilizar a chave primária como constraint de tabela.

Como exemplo, pode-se ver a criação da chave primária para a tabela Estados

```
1 ...
2 CONSTRAINT pk_Estados_ID PRIMARY KEY(EstadoID)
3 ...
```

onde se define o nome da constraint como *pk_Estados_ID*. Esta definição pode ser simplificada para

```
1 ...
2 PRIMARY KEY(EstadoID)
3 ...
```

Chave Estrangeira

Chave estrangeira é o campo que estabelece o relacionamento entre duas tabelas. Assim, uma coluna, ou grupo de colunas, de uma tabela correspondente à mesma coluna, ou grupo de colunas, que é a chave primária de outra tabela. Dessa forma, deve-se especificar na tabela que contém a

chave estrangeira quais são essas colunas e à qual tabela está relacionada. Veja que o banco de dados irá verificar se todos os campos que fazer referência à tabela estão especificados.

Ao determinar esse tipo de relacionamento, fica garantida a integridade das informações. Os valores presentes na(s) coluna(s) definida(s) como chave estrangeira devem ter um correspondente na outra tabela, caso contrário o banco de dados deve retornar uma mensagem de erro. É normal que, ao tentar excluir um registro de uma tabela, o banco de dados verifique se há chaves estrangeiras relacionadas a este registro. Se houver, também será retornado um erro impedindo essa operação.

Infelizmente, muitos programadores não utilizam esses recursos dos bancos de dados. Se não forem definidas as chaves estrangeiras nas tabelas, apenas o programa controlará o que pode ou não ser excluído ou alterado, colocando em risco informações relevantes do sistema. Por exemplo, pode-se excluir uma obra, mesmo que haja mídias ou empréstimos relacionadas a ela. Dessa forma, perde-se a informação.

Para especificar uma chave estrangeira, utiliza-se a seguinte sintaxe:

```
1 CONSTRAINT nome-chave-estrangeira FOREIGN KEY (lista-de-colunas)
2 REFERENCES nome-tabela (lista-de-colunas)
3 ON UPDATE ação
4 ON DELETE ação
```

onde:

Opção	Descrição
`nome-chave-estrangeira`	Nome opcional da constraint.
`lista-de-colunas`	Lista de colunas da tabela que faz referência a outra tabela.
`nome-tabela`	Nome da tabela em que está a chave primária.
`ação`	Determina qual ação o banco de dados deve tomar quando for excluída ou alterada uma linha da tabela que contém referência a esta chave. Pode ser SET NULL (altera o conteúdo da coluna para Nulo, perdendo a referência, sem deixar valores inconsistentes), CASCADE (exclui ou altera todos os registros que se relacionam a eles), NO ACTION (em caso de alteração, não modifica os valores que se relacionam a eles) ou RESTRICT (não permite a exclusão quando se tem registros relacionados, sendo esta a ação padrão).

Como exemplo, na criação da tabela Cidades, cria-se a referência para a tabela Estados:

```
1  CREATE TABLE Cidades (
2      CidadeID INTEGER NOT NULL AUTO_INCREMENT,
3      EstadoID INTEGER NOT NULL,
4      Nome VARCHAR(50) NOT NULL DEFAULT '',
5      CONSTRAINT pk_Cidades_ID PRIMARY KEY(CidadeID),
6      CONSTRAINT fk_Estados_ID FOREIGN KEY(EstadoID)
7      REFERENCES Estados(EstadoID)
8      ON UPDATE CASCADE
9      ON DELETE RESTRICT
10 )
```

Outra forma de criar a chave estrangeira seria:

```
1 ...
2 FOREIGN KEY(EstadoID)
3 REFERENCES Estados
4 ...
```

Veja que não houve a necessidade de incluir a lista de colunas após o nome da tabela que contém a chave primária, pois esta foi definida na criação da tabela Estados. Também não há a necessidade de especificar o nome da constraint, nem as ações ON UPDATE e ON DELETE, as quais serão, por padrão, RESTRICT.

Utilizando esta simplificação, pode-se criar a tabela Pessoas da seguinte forma:

```
CREATE TABLE Pessoas (
    PessoaID INTEGER NOT NULL AUTO_INCREMENT,
    CidadeID INTEGER NOT NULL,
    Nome VARCHAR(50) NOT NULL DEFAULT '',
    Endereco VARCHAR(100) NOT NULL DEFAULT '',
    Complemento VARCHAR(50) NOT NULL DEFAULT '',
    Telefone VARCHAR(20) NOT NULL DEFAULT '',
    Sexo CHAR(1) NOT NULL DEFAULT '',
    PRIMARY KEY(PessoaID),
    FOREIGN KEY(CidadeID) REFERENCES Cidades
)
```

2.3 Alteração de Estrutura de Tabela

Para alterar a estrutura de uma tabela utiliza-se o comando ALTER TABLE. Com este comando pode-se acrescentar novas colunas e constraints à tabela, modificar colunas, excluir e alterar o nome dos elementos.

2.3.1 Acrescentar Novas Colunas

O comando utilizado para acrescentar novas colunas é muito semelhante ao da criação de colunas em uma tabela:

```
ALTER TABLE tabela
    ADD nome-coluna tipo-de-dado constraint-coluna [, nome-coluna tipo-de-
        dado constraint-coluna, ...]
```

Como exemplo, pode-se adicionar o campo e-mail na tabela Pessoas:

```
ALTER TABLE Pessoas
    ADD Email VARCHAR(80) UNIQUE
```

2.3.2 Acrescentar Novas Constraints

O comando utilizado para acrescentar novas constraints é muito semelhante ao da criação de constraints em uma tabela:

```
ALTER TABLE tabela
    ADD ( constraint )
```

Como exemplo, pode-se adicionar a chave primária na tabela Pessoas:

```
ALTER TABLE tabela
    ADD PRIMARY KEY (PessoaID)
```

Nesse caso, está sendo alterado uma constraint de tabela. Nota-se que a colocação do parênteses após a cláusula ADD é opcional. Uma constraint de coluna deve ser alterada utilizando o comando de modificação de colunas.

2.3.3 Modificar Colunas

Pode-se modificar qualquer característica de uma coluna, seja o tipo de dado (alguns bancos de dados requerem a ausência de conteúdo na coluna para fazer essa alteração), o tamanho (alguns só aceitam alterações para valores maiores que o definido) e as constraints:

```
1 ALTER TABLE tabela
2     ALTER (nome-coluna) SET (constraint-coluna)
```

Como exemplo:

```
1 ALTER TABLE Pessoas
2     ALTER Email SET NOT NULL
```

No MySQL, deve-se sempre especificar toda a estrutura do campo que está sendo modificada, incluindo seu tipo e suas constraints.

2.3.4 Excluindo Elementos

Pelo padrão SQL, deveria ser possível excluir colunas ou constraints de uma tabela. Alguns bancos de dados não permitem a exclusão de colunas. No MySQL, a cláusula de exclusão é o DROP, da seguinte forma:

```
1 ALTER TABLE tabela
2     DROP elemento
```

Como exemplo de exclusão de coluna, pode-se tomar a exclusão da coluna Sexo da tabela Pessoas:

```
1 ALTER TABLE Pessoas
2     DROP Sexo
```

Para excluir a constraint de chave primária da tabela:

```
1 ALTER TABLE Estados
2     DROP PRIMARY KEY
```

Já a exclusão da chave estrangeira é:

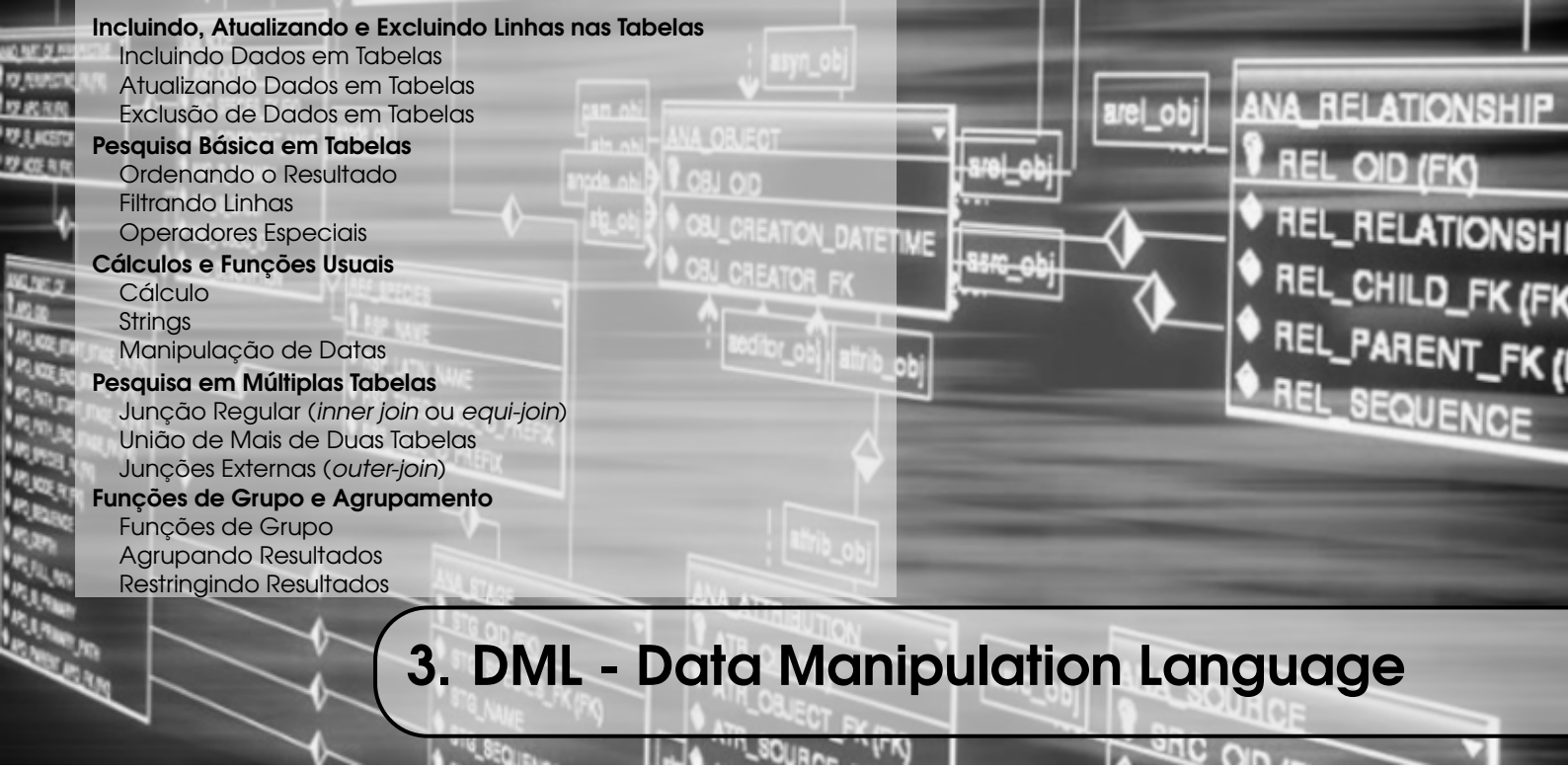
```
1 ALTER TABLE Cidades
2     DROP FOREIGN KEY pk_Cidades_ID
```

2.4 Eliminando uma Tabela

Para eliminar uma tabela do banco de dados, utiliza-se o comando DROP TABLE seguido do nome da tabela. Alguns bancos de dados somente permitirão a exclusão da tabela desde que esta não esteja relacionada a outras. Alguns deles podem ter cláusulas específicas para excluir a tabela, independentemente das constraints definidas.

Um exemplo para a exclusão da tabela Estados:

```
1 DROP TABLE Estados
```



3. DML - Data Manipulation Language

Linguagens de manipulação de dados são utilizadas para a recuperação, inclusão, remoção e modificação de informações em bancos de dados. Pode ser procedural, que especifica como os dados devem ser obtidos do banco; pode também ser declarativa (não procedural), em que os usuários não necessitam especificar o caminho de acesso, isto é, como os dados serão obtidos.

3.1 Incluindo, Atualizando e Excluindo Linhas nas Tabelas

Há três comandos utilizados para modificar dados em tabelas: INSERT, DELETE e UPDATE. Normalmente, um sistema desenvolvido para o usuário final "esconde" a complexidade desses comandos, mas o programador precisa conhecê-los para criar rotinas que cumpram esse papel.

3.1.1 Incluindo Dados em Tabelas

Para incluir linhas em tabelas, utiliza-se o comando:

```
1 INSERT INTO tabela(coluna1, coluna2, ..., colunan)
2     VALUES (valor-coluna1, valor-coluna2, ..., valor-colunan)
```

onde:

Cláusula	Descrição
`tabela`	Nome da tabela em que será efetuada a inclusão da linha
`coluna`	Lista de colunas que terão seus valores atribuídos
`valor-coluna`	Conteúdo que será atribuído a cada um dos campos. Estes valores devem corresponder (em quantidade e tipo de dado) à lista de colunas especificadas anteriormente

A lista de colunas é opcional no comando INSERT. Caso não seja especificado em quais colunas será a inclusão dos valores, assume-se a inclusão de valores em todas as colunas e a ordem de inclusão (lista de conteúdo) corresponderá àquela definida na criação da tabela. Se houver uma reorganização na estrutura da tabela ou se forem acrescentadas novas colunas, o comando INSERT retornará uma mensagem de erro. É natural que o comando ficará muito mais claro quando especificada a lista de colunas.

Valores numéricos não devem vir acompanhados de aspas simples ou apóstrofos. Já os valores alfanuméricos devem conter esse separador. Colunas do tipo data devem ser tratadas conforme a especificação do banco de dados (o MySQL trabalha com datas no formato AAAA-MM-DD).

Na inclusão de um registro no banco de dados, o gerenciador checará quaisquer restrições de integridade (constraints). Assim, chaves primárias e estrangeiras serão verificados no momento da inclusão. Se alguma restrição for violada, o registro não será incluído e uma mensagem de erro será enviada pelo banco de dados.

Colunas que não tiverem valores atribuídos terão conteúdo nulo, ou o valor especificado na contraint DEFAULT, quando esta for colocada para a coluna na criação da tabela.

Exemplos:

```
1 INSERT INTO Estados(EstadoID, Nome, Sigla)
2     VALUES (1, 'Paraná', 'PR')
```

```
1 INSERT INTO Funcionarios(FuncionarioID, PessoaID, DataAdmissao)
2     VALUES (1, 1, '2010-03-27')
```

3.1.2 Atualizando Dados em Tabelas

Uma vez que um registro esteja em uma tabela, pode-se querer alterar o conteúdo de uma ou mais colunas, ou até o conteúdo de uma coluna em diversos registros. Para isso, utiliza-se o comando UPDATE:

```
1 UPDATE tabela
2     SET coluna1 = valor-coluna1, coluna2 = valor-coluna2, ...
3     WHERE condição
```

onde:

Cláusula	Descrição
`tabela`	Nome da tabela em que será realizada a alteração
`coluna`	Coluna que terá seu valor atualizado
`valor-coluna`	Novo conteúdo para a coluna
`condição`	Condição que indicará o escopo (limite) de atualização das linhas

O valor da coluna deve respeitar o tipo de dado da coluna.

A condição expressa no comando servirá para definir quais linhas devem ser atualizadas. Assim, pode-se definir que várias linhas de uma mesma tabela sejam atualizadas ao mesmo tempo.

Caso a cláusula WHERE contenha a chave primária da tabela, somente uma linha da tabela será atualizada, pois a chave primária não admite repetições. Todas as condições de restrição de integridade (constraints) serão avaliadas pelo banco de dados, incluindo chaves primária e estrangeira.

Assim, para alterar o nome de uma pessoa no banco de dados, pode-se utilizar o seguinte comando, filtrando na cláusula WHERE a chave primária do registro da pessoa cujo nome será alterado:

```
1 UPDATE Pessoas
2     SET Nome = 'João'
3     WHERE PessoaID = 1
```

Também é possível alterar o conteúdo de uma coluna com base nela mesma. Aumenta-se o salário do funcionário 1 em 10%, multiplicando o próprio salário do funcionário por 1.1:

```
1 UPDATE Funcionarios
2     SET Salario = Salario * 1.1
3     WHERE FuncionarioID = 1
```

Se não for especificada a cláusula WHERE, então todos os registros serão afetados pelo comando UPDATE, Todos os funcionários terão um aumento de 10% em seus salários:

```
1 UPDATE Funcionarios
2     SET Salario = Salario * 1.1
```

3.1.3 Exclusão de Dados em Tabelas

Sempre que alguma informação deixar de ser relevante para ser armazenada, pode-se excluí-la de tabelas. O comando DELETE pode, da mesma forma que o comando UPDATE, afetar uma ou mais linhas de uma tabela. Mais uma vez, o escopo da exclusão será definido pela cláusula WHERE. A sintaxe é:

```
1 DELETE FROM tabela
2     WHERE condição
```

onde:

Cláusula	Descrição
`tabela`	Nome da tabela onde será excluída a linha
`condição`	Condição que indicará o escopo (limite) de exclusão de linhas na tabela

Se for especificado na cláusula WHERE a chave primária de uma tabela, somente uma linha da tabela será excluída. Se for outra condição, várias linhas poderão ser excluídas. Caso não seja especificada a cláusula WHERE, todas as linhas da tabela serão excluídas.

Outro ponto importante é que o gerenciador irá sempre checar a integridade das informações. Ao tentar excluir um registro de uma tabela que tenha "filhos" em outras tabelas (por exemplo, tentar excluir um estado que tenha cidades vinculadas a ele), será retornada uma mensagem de erro. Isso ocorre porque, de outra forma, a informação ficaria "perdida" na tabela dependente. Naturalmente isso somente ocorrerá quando essa função não tiver sido desabilitada na criação da tabela (veja Seção 2.2.2).

Neste primeiro exemplo, apenas a cidade com código 1 será excluída (desde que não haja relacionamentos com outras tabelas):

```
1 DELETE FROM Cidades
2     WHERE CidadeID = 1
```

Neste exemplo, todas as cidades do estado 1 serão excluídas:

```
1 DELETE FROM Cidades
2     WHERE EstadoID = 1
```

Neste último exemplo, todas as cidades serão excluídas:

```
1 DELETE FROM Cidades
```

3.2 Pesquisa Básica em Tabelas

Para extrair informações do banco de dados, utiliza-se o comando SELECT. Atrás dele, há uma extensão de possibilidades que vão desde a simples extração do conteúdo de todas as linhas e colunas de uma tabela até a união de diversas tabelas, cálculos, agrupamentos, além de ordenações e filtragens de linhas e colunas.

A sintaxe mais simples do comando SELECT é:

```
1 SELECT [DISTINCT | ALL] { * | coluna [, coluna, ... ] }
2     FROM tabela
```

onde:

Cláusula	Descrição
`DISTINCT`	Não mostra eventuais valores repetidos de colunas
`ALL`	Mostra todos os valores, mesmo que repetidos. Esse é o padrão se DISTINCT não for definido
`*`	Indica que devem ser mostradas todas as colunas da tabela
`coluna`	Lista de colunas que devem ser mostradas
`tabela`	Nome da tabela em que será realizada a busca

Para visualizar todas as linhas e colunas da tabela Pessoas, deve-se utilizar o seguinte comando:

```
1 SELECT * FROM Pessoas
```

Para filtrar apenas algumas colunas da tabela, como por exemplo, nome e telefone, utiliza-se:

```
1 SELECT Nome, Telefone FROM Pessoas
```

3.2.1 Ordenando o Resultado

Ao utilizar a SQL, deve-se utilizar a cláusula ORDER BY para determinar a ordem em que são mostradas as linhas de uma tabela. Portanto, para alterar a ordem, deverá ser acrescentado ao comando a cláusula ORDER BY seguida pela(s) coluna(s) que se deseja. Assim:

```
1 SELECT [DISTINCT | ALL] { * | coluna [, coluna, ... ] }
2     FROM tabela
3     ORDER BY coluna-ord [, coluna-ord, ... ] [ ASC | DESC ]
```

onde:

Cláusula	Descrição
`coluna-ord`	Lista de colunas na ordem de precedência de classificação
`ASC`	Efetua a ordenação de forma ascendente (forma padrão de ordenação)
`DESC`	Efetua a ordenação de forma descendente

Para listar as pessoas cadastradas em ordem alfabética, deve-se escrever o comando da seguinte forma:

```
1 SELECT Nome, Telefone
2     FROM Pessoas
3     ORDER BY Nome
```

Se for especificado mais de uma coluna para ser ordenada, o gerenciado primeiro ordenará pela primeira coluna e, caso houver valores repetidos nesta, será usado como critério de ordenação a segunda coluna, e assim será sucessivamente. Por exemplo, a sentença:

```
1 SELECT Bairro, Nome, Telefone
2     FROM Pessoas
3     ORDER BY Bairro, Nome
```

irá ordenar os registros pelo Bairro e, tendo registro de pessoas com o mesmo bairro, estas serão agrupadas posteriormente pelo nome.

3.2.2 Filtrando Linhas

Para filtrar linhas em uma pesquisa, utiliza-se a cláusula WHERE. Nesta, define-se uma expressão lógica (condição) que será avaliada e mostrará apenas as linhas que atendem ao critério estabelecido. A sintaxe é:

```
1 SELECT [DISTINCT | ALL] { * | coluna [, coluna, ... ] }
2     FROM tabela
3     WHERE condição
```

onde:

Cláusula	Descrição
`condição`	Condição que define o escopo (limite) de apresentação dos resultados

Sempre que a condição especificada for verdadeira, a linha será mostrada. Por outro lado, sempre que a condição testada na linha for falsa, então a linha não será mostrada.

Para definir a condição, se faz uso dos operadores relacionais: = (igual), < (menor que), <= (menor ou igual a), > (maior que), >= (maior ou igual a) e <> (diferente). Desta forma, para buscar os funcionários com salário maior que R$ 1000,00, utiliza-se o seguinte comando:

```
1 SELECT * FROM Funcionarios
2     WHERE Salario > 1000
```

Caso seja necessário ordenar este resultado pelo salário, utiliza-se a cláusula ORDER BY após a cláusula WHERE:

```
1 SELECT * FROM Funcionarios
2     WHERE Salario > 1000
3     ORDER BY Salario
```

Da mesma forma que pode-se comparar uma coluna com um valor, pode-se comparar uma coluna com outra. O funcionamento é exatamente igual. Note que será comparado o valor da coluna de uma linha com o valor da outra coluna na mesma linha. Por exemplo, para buscar os itens de empréstimos que foram devolvidos após a data prevista:

```
1 SELECT * FROM ItensEmprestimo
2     WHERE DataDevolucao > DataPrevistaDevolucao
```

Fazer comparações com conteúdos de linhas diferentes é possível, mas requer consultas mais complexas.

Sempre que se faz esse tipo de comparação, deve-se obedecer ao tipo de dado que se está comparando. Se estão sendo comparados valores numéricos, não deve-se colocar qualquer símbolo separador como cifrão ou vírgula, tampouco letras. O ponto decimal, contudo, deve ser especificado.

Comparações entre alfanuméricos devem estar entre apóstrofos. Colunas do tipo data devem seguir as regras do banco de dados.

Também pode-se utilizar os operadores lógicos (AND, OR e NOT) para combinar mais de uma condição. Por exemplo, para buscar as pessoas do sexo masculino que moram no bairro Centro, pode-se fazer a seguinte sentença:

```
1 SELECT * FROM Pessoas
2     WHERE Sexo = 'M' AND Bairro = 'Centro'
```

3.2.3 Operadores Especiais

Há alguns operadores que são utilizados para determinar melhor as linhas que se quer filtrar. São eles IS NULL, IS NOT NULL, BETWEEN, LIKE e IN.

IS NULL

Esse comando é utilizado para saber se o conteúdo de uma coluna é nulo. Por exemplo, sabendo-se que, se o item ainda não foi devolvido, o campo DataDevolucao permanece nulo, é possível pesquisar pelos itens ainda não devolvidos da seguinte forma:

```
1 SELECT * FROM ItensEmprestimo
2     WHERE DataDevolucao IS NULL
```

IS NOT NULL

Compara a negação do operador anterior. Somente aqueles que tiverem conteúdo atribuído serão mostrados. Para pesquisar os itens já devolvidos, ou seja, aqueles que têm a data de devolução preenchida, efetua-se a seguinte consulta:

```
1 SELECT * FROM ItensEmprestimo
2     WHERE DataDevolucao IS NOT NULL
```

BETWEEN

Esse operador serve para determinar um intervalo de busca. Assim, sempre que for preciso efetuar buscas que indiquem um intervalo de números ou datas pode-se utilizar o BETWEEN para simplificar a forma de escrever o comando.

Por exemplo, para buscar os empréstimos efetuados em janeiro de 2012:

```
1 SELECT * FROM Emprestimos
2     WHERE DataEmprestimo BETWEEN '2012-01-01' AND '2012-01-31'
```

Para busca os funcionários que ganham entre R$ 1000,00 e R$ 2000,00:

```
1 SELECT * FROM Funcionarios
2     WHERE Salario BETWEEN 1000 AND 2000
```

Caso seja utilizado o NOT, somente as linhas fora do intervalo serão apresentadas. Por exemplo, listar todos os funcionários exceto aqueles que ganham entre R$ 1000,00 e R$ 2000,00:

```
1 SELECT * FROM Funcionarios
2     WHERE Salario NOT BETWEEN 1000 AND 2000
```

LIKE

Com este operador, pode-se comparar cadeias de caracteres utilizando padrões de comparação (*wildcard*) para um ou mais caracteres. Normalmente, o caractere percentual (%) substitui zero, um ou mais caracteres e o *underline* (_) substitui um caractere.

Expressão	Explicação
`LIKE 'A%'`	Todas as palavras que iniciem com a letra A
`LIKE '%A'`	Todas as palavras que terminem com a letra A
`LIKE '%A%'`	Todas as palavras que tenham a letra A em qualquer posição
`LIKE 'A_'`	String de dois caracteres que tenham a primeira letra A e o segundo caractere seja qualquer outro
`LIKE '_A'`	String de dois caracteres cujo primeiro caractere seja qualquer um e a última letra seja A
`LIKE '_A_'`	String de três caracteres cuja segunda letra seja A, independentemente do primeiro ou do último caractere
`LIKE '%A_'`	Todos que tenham a letra A na penúltima posição e a última seja qualquer outro caractere
`LIKE '_A%'`	Todos que tenham a letra A na segunda posição e o primeiro caractere seja qualquer um

Utilizando a combinação desses caracteres especiais com o que se quer localizar, pode-se conseguir uma variedade muito grande de expressões. Veja na tabela a seguir algumas possíveis combinações.

No exemplo seguinte, busca-se todas as pessoas cujos nomes sejam iniciados com a letra R:

```
1 SELECT * FROM Pessoas
2     WHERE Nome LIKE 'R\%'
```

IN

Permite comparar o valor de uma coluna com um conjunto de valores. Normalmente, utiliza-se o IN para substituir uma série de comparações seguidas da cláusula OR. Neste exemplo, somente as pessoas que residem nas cidades 1, 3 ou 6 serão retornadas:

```
1 SELECT * FROM Pessoas
2     WHERE CidadeID IN (1, 3, 6)
```

Para buscar todas as pessoas, exceto as que residem nas cidades 1, 3 ou 6, efetua-se a pesquisa da seguinte forma:

```
1 SELECT * FROM Pessoas
2     WHERE CidadeID NOT IN (1, 3, 6)
```

3.3 Cálculos e Funções Usuais

Outra utilização bastante importante de SQL é fazer cálculos e totalizações de valores unitários, como será visto adiante.

3.3.1 Cálculo

Pode-se fazer cálculos ao realizar buscas no banco de dados, simplesmente aplicando um dos operadores aritméticos a colunas. Dessa forma, mesmo não tendo armazenado o valor total (preço unitário multiplicado pela quantidade) em um item de pedido, por exemplo, é possível realizar este cálculo. Como exemplo, pode-se verificar o salário dos funcionários se estes tiverem um reajuste de 5%, mostrando o seu salário atual multiplicado por 1.05:

```
1 SELECT FuncionarioID, Salario, (Salario * 1.05) FROM Funcionarios
```

Vale lembrar que o salário não foi alterado no banco de dados, apenas na apresentação final.

Os operadores aritméticos padrão do SQL são relativamente limitados, sendo eles:

Operador	Ação
+	Soma
-	Subtração
*	Multiplicação
/	Divisão

A precedência dos operadores é igual à da matemática, ou seja, multiplicação e divisão têm prioridade sobre soma e subtração. Se houver duas ou mais operações do mesmo grupo (multiplicação e divisão ou adição e subtração), a operação será realizada na ordem que aparecer. Dentro do mesmo grupo não há prioridade. Para alterar a prioridade, deve-se utilizar parênteses.

Dessa forma, o cálculo $15/5*3$ tem como resposta 9. O resultado seria 1 se fosse escrito como $15/(5*3)$.

3.3.2 Strings

A seguir, são apresentadas algumas funções que são utilizadas para o tratamento de strings em SQL.

POSITION / INSTR

O objetivo de ambas as funções é retornar a posição do caractere de busca na cadeia de caracteres de origem. Dependendo do banco de dados, encontra-se uma ou outra função (ou até mesmo as duas).

```
1 POSITION(destino IN origem)
2 INSTR(origem, destino [,início[, fim]])
```

Tipo	Descrição
`destino`	Cadeia de caracteres em que se quer pesquisar
`origem`	Cadeia de caracteres em que será realizada a pesquisa
`início`	Posição inicial de busca (se não especificada inicia na posição 1)
`fim`	Ocorrência que deve ser pesquisada (se não especificada, indica que é a primeira ocorrência)

```
1 SELECT POSITION('Fra' IN Nome) FROM CIDADES
```

```
1 SELECT INSTR(Nome, 'Fra') FROM CIDADES
```

CHARACTER_LENGTH / LENGTH

Retorna o número de caracteres contidos em uma cadeia de caracteres.

```
1 CHARACTER_LENGTH(string)
2 LENGTH(string)
```

```
1 SELECT CHARACTER_LENGTH(Nome) FROM CIDADES
```

```
1 SELECT LENGTH(Nome) FROM CIDADES
```

Concatenando Strings

Para concatenar cadeias de caracteres, utiliza-se dois *pipes* (||). Dessa forma, para trazer o nome do estado seguido de sua sigla, pode-se utilizar a seguinte sentença:

```
1 SELECT Nome || ' - ' || Sigla FROM ESTADOS
```

UPPER e LOWER

Quando são realizadas buscas alfanuméricas no banco de dados, deve-se notar que o conteúdo do campo será comparado, literalmente, com a cadeia de caracteres informada. Isso quer dizer que caso o que seja informado na busca seja composto por letras maiúsculas e o que está armazenado no banco de dados esteja com a primeira letra maiúscula e as demais minúsculas, não será retornada nenhuma linha.

Esta é uma das situações em que se faz necessário converter a cadeia de caracteres tendo todas as letras maiúsculas ou minúsculas. Isto pode ser feito utilizando as funções UPPER e LOWER, respectivamente.

```
1 UPPER( string )
2 LOWER( string )
```

Para retornar o nome dos filmes com todas as letras em maiúsculo

```
1 SELECT UPPER(TITULO) FROM FILMES
```

e em minúsculo

```
1 SELECT LOWER(TITULO) FROM FILMES
```

TRIM

O comando TRIM serve para retirar caracteres antes e/ou depois de outra cadeia de caracteres. Ela pode ser usada em qualquer cadeia de caracteres, mesmo que esta contenha números ou símbolos.

Sintaxe:

```
1 TRIM([{ BOTH | LEADING | TRAILING } caractere FROM] string
```

Exemplo de utilização:

Exemplo	Resultado	Explicação
`TRIM(' teste ')`	`'teste'`	Este é o padrão. Retira espaços antes e depois da palavra teste
`TRIM(BOTH '#' FROM '#teste#')`	`'teste'`	Retira o símbolo # que estava antes e depois (BOTH) da cadeia de caracteres
`TRIM(LEADING ' ' FROM ' teste ')`	`'teste '`	Retira o espaço que estava antes da palavra teste
`TRIM(TRAILING ' ' FROM ' teste ')`	`' teste'`	Retira o espaço que estava após a palavra teste

SUBSTRING

Esta função é requisito para SQL-92 no nível intermediário. Retorna uma parte de uma cadeia de caracteres.

```
1 SUBSTRING( string FROM início FOR número_caracteres )
```

Por exemplo, para retornar as três primeiras letras das cidades, pode-se executar a seguinte sentença:

```
1 SELECT SUBSTRING(NOME FROM 1 FOR 3) FROM CIDADES
```

REPLACE

Substitui uma cadeia inteira de caracteres por outra. A cadeia de caracteres que estiver em *pesquisa* será substituída pela cadeia de caracteres especificada em *substituição*. Não havendo uma ocorrência de toda a cadeia, nada será substituído.

```
1 REPLACE(caracteres, pesquisa, substituição)
```

3.3.3 Manipulação de Datas

Em colunas do tipo data, pode-se realizar uma série de cálculos e operações cronológicas. Pode-se calcular o número de dias entre duas datas, somar e subtrair dias, meses, etc.

O padrão SQL especifica quatro tipos de dados relacionados a data e hora:

Tipo	Descrição
DATE	Apenas data
TIME	Apenas hora
TIMESTAMP	Data e hora
INTERVAL	Intervalo entre os dois tipos de dados anteriores

O padrão SQL definiu algumas funções para determinar a data e/ou hora atuais do sistema, acrescentando CURRENT_ antes dos três primeiros tipos anteriores: CURRENT_DATE, CURRENT_TIME e CURRENT_TIMESTAMP. Estas funções podem ser usadas em consultas, inclusões e alterações. Por exemplo, para buscar os clientes cadastrados no dia de hoje

```
1 SELECT * FROM CLIENTES
2     WHERE DATACADASTRO = CURRENT_DATE
```

Operações Aritméticas com Datas

Uma coluna do tipo data é composta por seis elementos:

- YEAR (ano).
- MONTH (mês).
- DAY (dia).
- HOUR (hora).
- MINUTE (minuto).
- SECOND (segundo).

Desta forma, é possível realizar operações como encontrar o intervalo de dias entre duas datas:

```
1 SELECT CURRENT_DATE, DATACADASTRO, CURRENT_DATE - DATACADASTRO
2     FROM CLIENTES
```

Para adicionar dias a uma data, pode-se fazer da seguinte forma:

```
1 SELECT DATACADASTRO, DATACADASTRO + 7
2     FROM CLIENTES
```

ou

```
SELECT DATACADASTRO, DATACADASTRO + INTERVAL '7' DAY
    FROM CLIENTES
```

Já para adicionar meses

```
SELECT DATACADASTRO, DATACADASTRO + INTERVAL '2' MONTH
    FROM CLIENTES
```

EXTRACT

Essa função extrai e retorna um valor de um campo do tipo data. É possível extrair apenas o dia, o mês, o ano, a hora, o minuto ou o segundo.

```
SELECT EXTRACT(DAY FROM DATACADASTRO) FROM CLIENTES
```

```
SELECT EXTRACT(MONTH FROM DATACADASTRO) FROM CLIENTES
```

```
SELECT EXTRACT(YEAR FROM DATACADASTRO) FROM CLIENTES
```

3.4 Pesquisa em Múltiplas Tabelas

Na montagem de um modelo de dados, sempre tem-se diversas tabelas. Logo, é preciso vincular os dados destas tabelas para mostrar a informação necessária e de maneira correta. A isto é dado o nome de junção de tabelas (*join*).

A junção entre as tabelas se dá por meio das chaves primárias e estrangeiras. Essas chaves são, na representação física do modelo, as colunas que as tabelas têm em comum.

Existem diferentes formas de se fazer a junção entre tabelas relacionadas no banco de dados.

3.4.1 Junção Regular (*inner join* ou *equi-join*)

Denomina-se junção regular as junções que têm a cláusula WHERE unindo a chave primária à estrangeira das tabelas afetadas pelo comando SELECT. Exemplo:

```
SELECT PESSOAS.NOME, PESSOAS.SEXO, CLIENTES.DATACADASTRO
    FROM PESSOAS, CLIENTES
    WHERE PESSOAS.PESSOAID = CLIENTES.PESSOAID
```

Desta forma, é possível unir as informações dos clientes que estão contidas nas tabelas CLIENTES e PESSOAS.

O padrão SQL determina uma sintaxe alternativa para esse comando. Quando a chave primária e a chave estrangeira têm o mesmo nome em ambas as tabelas, é possível simplificar o comando utilizando:

```
SELECT PESSOAS.NOME, PESSOAS.SEXO, CLIENTES.DATACADASTRO
    FROM PESSOAS NATURAL JOIN CLIENTES
```

Isto significa que a ligação entre as duas tabelas se dará utilizando a igualdade de valores nos campos que têm o mesmo nome em ambas as tabelas (neste caso o campo PESSOAID).

Outras duas maneiras definidas no padrão SQL para realizar esse tipo de união de tabelas é determinar qual(is) coluna(s) utilizar na junção (cláusula USING) e, caso o nome das colunas não seja igual determinar quais são as colunas com a cláusula ON. Todos os comandos produzem o mesmo efeito. Exemplos:

```
SELECT PESSOAS.NOME, PESSOAS.SEXO, CLIENTES.DATACADASTRO
    FROM PESSOAS JOIN CLIENTES USING(PESSOAID)
```

```
SELECT PESSOAS.NOME, PESSOAS.SEXO, CLIENTES.DATACADASTRO
    FROM PESSOAS JOIN CLIENTES ON PESSOAS.PESSOAID = CLIENTES.PESSOAID
```

A cláusula JOIN é equivalente à cláusula INNER JOIN.

Apelidos em Tabelas

Para evitar que o comando fique extremamente extenso, é possível atribuir apelidos às tabelas utilizadas no comando SELECT. Deve-se fazer isso colocando o apelido após o nome da tabela na cláusula FROM. Dessa forma, o seguinte comando teria o mesmo efeito do comando anterior:

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM PESSOAS A, CLIENTES B
    WHERE A.PESSOAID = B.PESSOAID
```

3.4.2 União de Mais de Duas Tabelas

Frequentemente é necessário unir mais de duas tabelas para fornecer uma informação relevante do banco de dados. No nosso caso, suponha que se queira saber o nome do cliente, a sua data de cadastro e as datas em que ele fez empréstimos. Neste caso, é necessário utilizar dados de três tabelas para chegar a essa informação: PESSOAS, CLIENTES e EMPRESTIMOS. Assim, o comando fica:

```
SELECT A.NOME, B.DATACADASTRO, C.DATAEMPRESTIMO
    FROM PESSOAS A, CLIENTES B, EMPRESTIMOS C
    WHERE A.PESSOAID = B.PESSOAID AND B.CLIENTEID = C.CLIENTEID
```

Como se pode notar, a única coisa feita foi acrescentar o relacionamento entre todas as tabelas envolvidas no comando SELECT. Mais uma vez convém lembrar que, de acordo com o padrão SQL, o comando a seguir produziria o resultado igual:

```
SELECT A.NOME, B.DATACADASTRO, C.DATAEMPRESTIMO
    FROM PESSOAS A INNER JOIN CLIENTES B ON A.PESSOAID = B.PESSOAID
    INNER JOIN EMPRESTIMOS C ON B.CLIENTEID = C.CLIENTEID
```

Utilizando essa sintaxe, pode-se determinar e influenciar o tipo de recuperação de informações no banco de dados. No padrão SQL, a ordem em que coloca-se a sequência das tabelas na cláusula FROM determina quais tabelas serão pesquisadas primeiro. Logo, se forem colocadas as tabelas menores primeiro, a busca ficará mais rápida. Deixe as tabelas maiores, sempre que possível, para o final da cláusula.

3.4.3 Junções Externas (*outer-join*)

Quando uma linha não satisfaz a condição de uma junção entre as tabelas, a linha não será mostrada no resultado da busca. Isso acontece porque o banco de dados, não podendo estabelecer a relação entre as colunas que estão sendo unidas na busca, coloca NULL onde o dado não existe.

Define-se junção externa como aquela que inclui linhas no resultado da busca mesmo que não haja relação entre as duas tabelas que estão sendo unidas. Existem três tipos de junção externa.

Junção Externa à Esquerda (*Left Outer Join*)

Como o nome diz, a união pela esquerda incluirá linhas da primeira tabela na expressão de junção. Note que, caso se faça uma junção com chave primária de uma tabela com a respectiva chave estrangeira na outra e deixar a chave estrangeira à esquerda, será igual a fazer uma junção regular (*inner join*), uma vez que não haverá linhas opcionais na tabela da esquerda.

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM PESSOAS A LEFT OUTER JOIN CLIENTES B USING(PESSOAID)
```

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM PESSOAS A LEFT OUTER JOIN CLIENTES B ON A.PESSOAID = B.PESSOAID
```

As duas sentenças terão o mesmo resultado, trazendo todos os registros da tabela PESSOAS e a data de cadastro do cliente onde houver a relação, caso contrário, o campo virá como o valor NULL.

Junção Externa à Direita (*Right Outer Join*)

Realiza a união pela direita, ou seja, as linhas da segunda tabela serão incluídas na busca, mesmo sem haver registros correspondentes na primeira tabela. Da mesma forma que a junção à esquerda, se for colocada a tabela da chave estrangeira à direita da busca, o resultado será uma junção regular entre as tabelas.

Com a junção à direita, é possível obter o mesmo resultado que a junção à esquerda, apenas invertendo a ordem das tabelas na sentença:

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM CLIENTES B RIGHT OUTER JOIN PESSOAS A USING(PESSOAID)
```

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM CLIENTES B RIGHT OUTER JOIN PESSOAS A ON A.PESSOAID = B.PESSOAID
```

Junção Externa Total (*Full Outer Join*)

Realiza uma junção, independente de a coluna opcional estar à direita ou à esquerda. A junção externa total tem como resultado registros da primeira tabela que têm ou não relação com registros da segunda tabela, e vice-e-versa. Desta forma, obtém-se uma listagem de todos os registros da primeira tabela e todos os registros da segunda tabela, não importando se estes têm relação uns com os outros.

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM CLIENTES B FULL OUTER JOIN PESSOAS A USING(PESSOAID)
```

ou

```
SELECT A.NOME, A.SEXO, B.DATACADASTRO
    FROM CLIENTES B FULL OUTER JOIN PESSOAS A ON A.PESSOAID = B.PESSOAID
```

A cláusula *FULL OUTER JOIN* não está disponível no MySQL.

3.5 Funções de Grupo e Agrupamento

Funções de grupo operam conjuntos de linhas visando a fornecer um resultado para o grupo. Diferente das funções que trabalham uma linha de cada vez, estas utilizam grupos de linhas. Esses grupos podem ser construídos desde toda a tabela até subgrupos da tabela.

3.5.1 Funções de Grupo

Existem diversas funções de grupo que são implementadas pelo padrão SQL. Essas funções auxiliam a computar uma variedade de medidas baseadas em valores das colunas do banco de dados. As principais funções de grupo são:

Função	Ação
COUNT	Retorna o número de registros
SUM	Retorna o somatório do valor da coluna especificada
AVG	Retorna a média aritmética dos valores da coluna especificada
MIN	Retorna o menor valor da coluna de um grupo de registros
MAX	Retorna o maior valor da coluna de um grupo de registros
STDDEV	Retorna o desvio padrão da coluna
VARIANCE	Retorna a variância da coluna

COUNT

Diferentemente das outras funções de grupo, o COUNT retorna o número de registros que atende a uma determinada condição. Pode-se utilizá-lo com um valor fixado, como por exemplo 1, para retornar a quantidade de registros, independente de haver linhas com colunas nulas ou não.

```
1 SELECT COUNT(1) FROM ITENSEMPRESTIMO
```

Caso queira-se saber quantos registros existem e quais destes não têm valor nulo em determinada coluna, esta coluna é especificada entre parênteses. Por exemplo, quantos itens já foram devolvidos:

```
1 SELECT COUNT(DATADEVOLUCAO) FROM ITENSEMPRESTIMO
```

Uma outra forma interessante de utilizar o COUNT é acrescentando a ele uma cláusula DISTINCT. Se é preciso saber quantos clientes efetuaram empréstimos, basta executar a seguinte sentença

```
1 SELECT COUNT(DISTINCT CLIENTEID) FROM EMPRESTIMOS
```

e será retornado o número de clientes distintos, ou seja, sem repetições, que já efetuaram empréstimos.

SUM

Retorna o valor total de uma determinada coluna em um determinado grupo de registros. Assim, se é necessário saber o total do salário de todos os funcionários, tem-se:

```
1 SELECT SUM(SALARIO) FROM FUNCIONARIOS
```

Pode-se realizar outros cálculos baseados na somatória ou mesmo incluir outras colunas e operações no comando. Imaginando que se queira saber como ficariam os gastos após um aumento de 20% para todos os funcionários:

```
1 SELECT SUM(SALARIO) * 1.2 FROM FUNCIONARIOS
```

AVG

Extrai a média aritmética de um determinado grupo de registros. Para saber a média salarial dos funcionários da locadora, faz-se:

```
1 SELECT AVG(SALARIO) FROM FUNCIONARIOS
```

MIN

Retorna o menor valor de uma coluna em um grupo de registros. Pode-se utilizá-la para colunas do tipo data ou alfanuméricas. Para saber quando foi efetuado o primeiro empréstimo na locadora, basta executar:

```
1 SELECT MIN(DATAEMPRESTIMO) FROM EMPRESTIMOS
```

MAX

Retorna o maior valor de uma coluna em um grupo de registros. Igualmente ao MIN, pode-se utilizá-la para colunas do tipo data ou alfanuméricas. Para saber qual é o salário mais alto dentre os funcionários:

```
1 SELECT MAX(SALARIO) FROM FUNCIONARIOS
```

STDDEV

Retorna o desvio-padrão de uma determinada coluna. Para saber o desvio-padrão do salário dos funcionários:

```
1 SELECT STDDEV(SALARIO) FROM FUNCIONARIOS
```

VARIANCE

Retorna a variância de uma determinada coluna. Para saber a variância do salário dos funcionários:

```
1 SELECT VARIANCE(SALARIO) FROM FUNCIONARIOS
```

3.5.2 Agrupando Resultados

Uma característica muito importante do SQL é o poder de agrupar registros com base em valores de determinadas colunas. Dessa forma, não se está trabalhando na pesquisa em todas as linhas da tabela, como se fazia nos exemplos anteriores, mas sim em grupos menores. Para isso, utiliza-se as funções de grupo já mostradas, com a cláusula GROUP BY no comando SELECT. A cláusula GROUP BY deve vir antes da cláusula ORDER BY e depois do WHERE (se houver necessidade de utilizá-los).

Como exemplo, pode-se querer saber quantos empréstimos foram feitos por cada cliente. Deve-se realizar a busca na tabela EMPRESTIMOS e agrupar por CLIENTEID a quantidade (COUNT) de empréstimos.

```
1 SELECT CLIENTEID, COUNT(1)
2     FROM EMPRESTIMOS
3     GROUP BY CLIENTEID
```

Pode-se realizar mais de uma função de grupo dentro de um mesmo SELECT. Aqui, tem-se a quantidade de itens de cada empréstimo junto com a quantidade de itens devolvidos para este empréstimo:

```
1 SELECT EMPRESTIMOID, COUNT(1), COUNT(DATADEVOLUCAO)
2     FROM ITENSEMPRESTIMO
3     GROUP BY EMPRESTIMOID
```

3.5.3 Restringindo Resultados

Os resultados dos dados agrupados podem ser restritos. Até agora, nos exemplos toda a tabela era afetada pelo comando GROUP BY. Contudo, nem sempre isso é desejável. Há duas maneiras de fazer isso: uma é utilizar a cláusula WHERE em conjunto com o GROUP BY; a outra é utilizar HAVING.

Para saber o total de itens de cada empréstimo agrupados pelo EMPRESTIMOID, tendo este menor que 5 pode-se fazer da seguinte forma:

```
1 SELECT EMPRESTIMOID, COUNT(1)
2     FROM ITENSEMPRESTIMO
3     WHERE EMPRESTIMOID < 5
4     GROUP BY EMPRESTIMOID
```

ou pode-se utilizar a cláusula HAVING após o GROUP BY para obter o mesmo resultado:

```
1 SELECT EMPRESTIMOID, COUNT(1)
2     FROM ITENSEMPRESTIMO
3     GROUP BY EMPRESTIMOID
4     HAVING EMPRESTIMOID < 5
```

Há uma diferença básica entre os dois métodos. Ao utilizar a cláusula WHERE, as linhas são filtradas antes do agrupamento. Ao utilizar o HAVING, as linhas são filtradas depois do agrupamento. A única restrição é que a cláusula HAVING só pode utilizar as colunas que fazem parte do GROUP BY para filtrar os registros. Para o WHERE, isso não é necessário.

4. Referências

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson, 2011.

OLIVEIRA, Celso H. Poderoso de. **SQL: curso prático**. São Paulo, SP: Novatec, 2002.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de banco de dados**. 3. ed. São Paulo: Makron, 1999.