Banco de Dados Oracle

Introdução a Banco de Dados e SQL

Versão	Data	Autor	Descrição
3.0	11/2/2020	Matheus Boesing	Apostila 3.0 completa.

Sumário

Objetivos deste Módulo

Ao final deste módulo, objetiva-se que o aluno adquira:

Conhecimentos:

- Discutir origem, conceitos fundamentais e implementação de bancos de dados.
- Entender elementos básicos relacionados a bancos de dados e diferentes estruturas
- Entender completamente Sintaxe SQL, operações aritméticas e demais funções
- Entender administração elementar de objetos Oracle
- Discutir estruturas avançadas em SQL, tais como JOINs e operações de conjunto

Habilidades:

- Consultar dados armazenados no Banco de Dados Oracle
- Criar e manter objetos do Banco de Dados, assim como, armazenar, recuperar e manipular dados
- Restringir e ordenar dados, utilizando cláusulas, como WHERE, strings de caractere e datas e operadores (BETWEEN, IN, LIKE, AND, OR...)
- Exibir dados a partir de múltiplas tabelas
- Utilizar funções de banco de dados
- Criar e gerenciar tabelas
- Controlar transações
- Implementar constraints e trabalhar com visões
- Trabalhar com sub-consultas e operadores SET
- Entender funções single row, funções de conversão e expressões de condição

1. Resgate Conceitual e Principais Componentes

Aforismos a parte, o conceito de banco de dados é simples e sua origem e necessidade são elementares: Armazenamento seguro de dados. Evidentemente a definição formalizou-se ao longo do tempo e adquiriu definições mais completas e modernas considerando as crescentes necessidades e constante evolução das tecnologias a fim de atendê-las.

A fim de iniciarmos este entendimento conceitual, basta que analisemos um antepassado longínquo dos bancos de dados modernos. O Arquivo de Escritório:

Se por um lado todos nós conhecemos algum destes, talvez não tenhamos refletido sobre alguns problemas nesta tecnologia de armazenamento de arquivos. Aqui citam-se algumas das limitações óbvias desta solução:

- Necessidade de inserção ordenada (tempo operacional)
- Não existem garantias de organização (embora haja expectativas de)
- A seleção é um processo simples, porém não há garantias de completude, uma vez que a organização não é garantida.

Existem alguns outros itens, não diretamente relacionados a manipulação dos arquivos, mas a administração dos mesmos:

- Se o arquivo não possuir chave, não há segurança.
- Ainda assim, a segurança é empírica, uma vez que a chave pode estar em lugar conhecido.
- Desastres naturais como inundações, fogo e umidade podem comprometer a integridade física dos dados muito facilmente
- Exposição natural a pragas como traças é um risco em potencial.
- O tempo de armazenamento invariavelmente degrada a qualidade do dado armazenado.

Do ponto de vista de desempenho contudo, há também um item bastante relevante:

- Apenas uma pessoa pode manipular o arquivo e apenas uma pessoa pode manipular cada pasta individualmente.

A partir da década de 60, ficou claro que uma solução em meio digital seria brevemente uma realidade para este tipo de problema, especialmente para sistemas de larga escala e para empresas de maior significância, a ponto de custear um Data Center. Como sabemos hoje, esta expectativa se confirmou e brevemente evoluiu de tal modo que quaisquer sistemas digitais ao nosso alcance do nosso bolso a possuem soluções de armazenamento digital e até remoto.

As décadas seguintes foram de bastante pesquisa na área de modo a obter a mais eficiente maneira para armazenamento destes dados de maneira digital e, principalmente, para localização e apresentação destes dados de maneira otimizada. Neste tocante, em 1970 Edgar F. Codd, reconhecido estudioso da área e engenheiro da IBM, publicou o artigo "*A Relational Model of Data for Large Shared Data Banks*".

Neste artigo, Codd apresentou uma maneira diferente de armazenar e organizar os dados baseados em conceitos da Álgebra Relacional, campo da matemática para operações em conjuntos de dados homogêneos e heterogêneos, do mesmo modo que estabeleceu regras para deduplicação de dados e distribuição hierárquica as quais passou-se a chamar de normalização de dados, ou ainda, "Formas Normais", conceitos até hoje estudados no que se refere a modelagem de bancos de dados. Veremos alguns destes conceitos a seguir.

Operações da Álgebra Relacional

A tabela abaixo apresenta as principais operações propostas por Codd, oriundas da Álgebra Relacional.

Símbolo	*Operação*	*Sintaxe*	*Tipo*
σ	Seleção / Restrição	$\sigma_{\text{condição}}$ (Relação)	Primitiva
π	Projeção	$\pi_{\text{expressões}}$ (Relação)	Primitiva
∪	União	Relação1 ∪ Relação2	Primitiva
∩	Intersecção	Relação1 ∩ Relação2	Adicional
-	Diferença de conjuntos	Relação1 - Relação2	Primitiva
x	Produto cartesiano	Relação1 x Relação2	Primitiva
|x|	Junção	Relação1 |x| Relação2	Adicional
÷	Divisão	Relação1 ÷ Relação2	Adicional
ρ	Renomeação	ρ_{nome} (Relação)	Primitiva
←	Atribuição	variável ← Relação	Adicional

Com estas operações, utilizadas até os dias atuais conforme se apresenta nos capítulos a seguir, é possível obter-se virtualmente quaisquer representações de dados desejadas baseadas em um conjunto de dados existente. São portanto, a base utilizada para obter-se dados de um banco de dados, processo conhecido como "CONSULTA" ou, no inglês, "QUERY". Abaixo alguns exemplos formais de CONSULTAS:

(i) $R1 \leftarrow \pi_{\text{cpf, nome, sexo}}(\text{PESSOA})$

(ii) $R1 \leftarrow \pi_{\text{cpf, nome, sexo}}(\sigma_{\text{sexo='M' OR sexo='F'}}(\text{PESSOA}))$

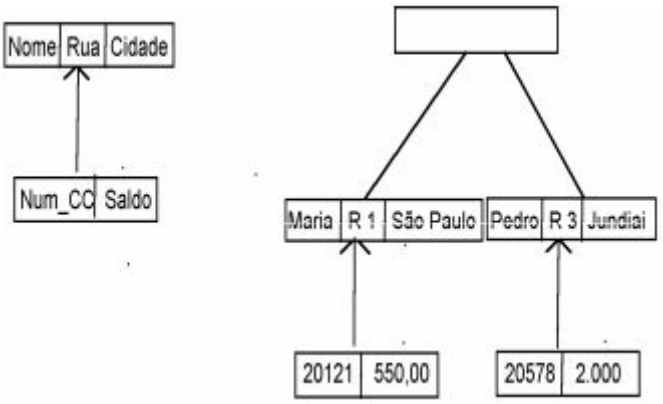
(iii) $R1 \leftarrow (\pi_{\text{cpf, nome, sexo}}(\sigma_{\text{sexo='M'}}(\text{PESSOA})) \cup (\pi_{\text{cpf, nome}}(\sigma_{\text{sexo='F'}}(\text{PESSOA}))$

Restava ainda definir qual seria a formalização para que tais operações fossem realizadas. Isto foi também proposto por Codd, inicialmente denominando tal linguagem como "SEQUEL", acrônimo para "*Structured English Query Language*" (Linguagem de Consulta Estruturada em Inglês). A evolução natural é o que hoje conhecemos por ***SQL - Structure Query Language***.

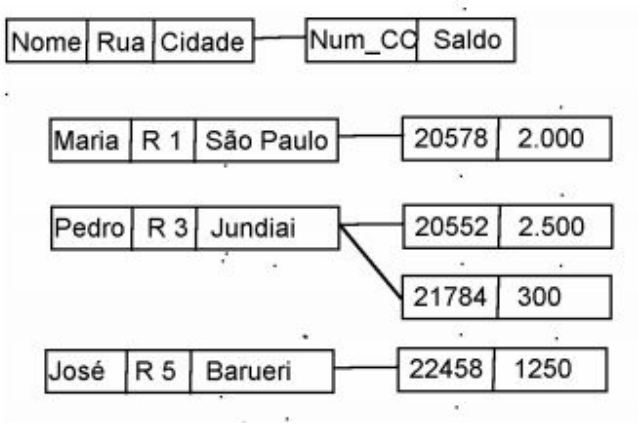
Banco de Dados Relacional

O SQL veio mais tarde a se tornar um padrão ANSI (*American National Standards Institute*), descrevendo linguagem e comandos base aceitos por qualquer implementação de banco de dados relacional. O padrão não limita, contudo, que linguagens tenham sintaxes específicas como de fato muitas possuem.

Cabe ressaltar, neste contexto, que a combinação da proposta solução de armazenamento (Banco de Dados) com o modelo matemático baseado em álgebra relacional criou a denominação "B*anco de Dados Relacional*", na época rivalizando com modelos hierárquicos (IMS/IBM):

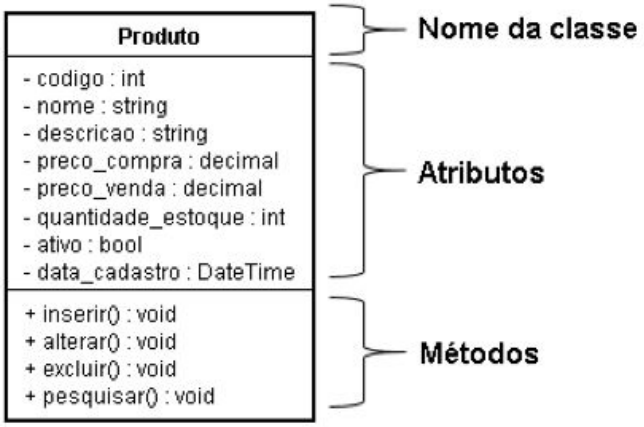


E modelos em rede (IDMS baseado em CODASYL):



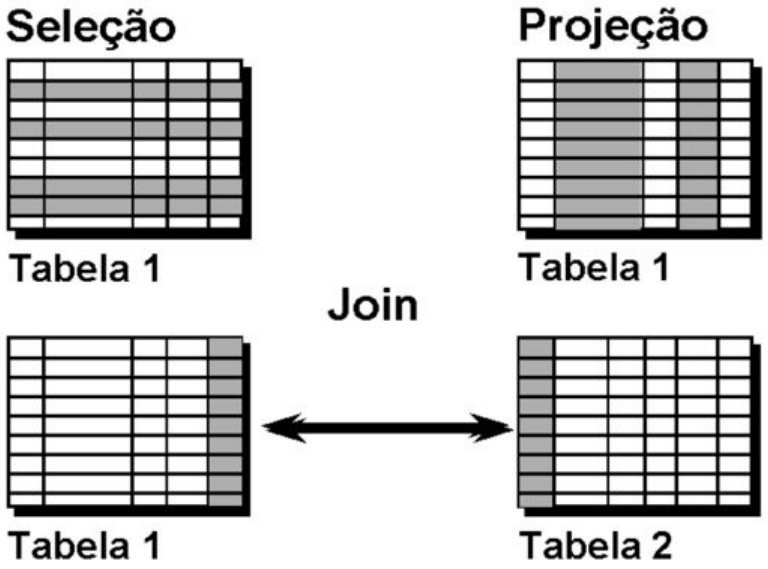
Um pouco mais tarde, na década de 80, em consequência a difusão de conceitos de Orientação a Objeto, surgiram o primeiros bancos de dados orientados a objeto (exemplo de entidade a seguir), porém a necessidade de eficiência no armazenamento logo fez com que os conceitos relacionais voltassem à tona.

Deste modo, apesar de essencialmente relacionais, bancos de dados como o Oracle Database denominam-se atualmente como Bancos de Dados Objeto-Relacionais, uma vez que conceitos como o herança, polimorfismo, encapsulamento e abstração estão presentes através de estruturas específicas, a serem estudadas no decorrer dos próximos capítulos.


Produto
- codigo : int
- nome : string
- descricao : string
- preco_compra : decimal
- preco_venda : decimal
- quantidade_estoque : int
- ativo : bool
- data_cadastro : DateTime
+ inserir() : void
+ alterar() : void
+ excluir() : void
+ pesquisar() : void
Nome da classe
Atributos
Métodos

Disposição e Modelagem de Dados - Modelo Relacional

Codd propôs em seu trabalho a organização dos dados em estruturas pré-definidas com colunas e linhas em formas de matrizes (conceito também da matemática), denominadas TABELAS ou ENTIDADES. As colunas seriam propriedades comuns com dados homogêneos e cada linha seria uma instância ou indivíduo pertencente ao grupo. De modo complementar, dados de outra natureza (ou com outras propriedades) seriam organizados em outras TABELAS, de modo a manter as tabelas enxutas e concisas. Assim, as operações relacionais descritas por Codd na tabela da página anterior teriam efeitos sobre conjuntos de dados denominados TABELAS.



Existem relacionamentos possíveis entre tabelas, uma vez que dados diferentes possuem relações. Alguns exemplos:

- O Aluno A estuda na Escola EXEMPLO1
- O Aluno B estuda na Escola EXEMPLO2

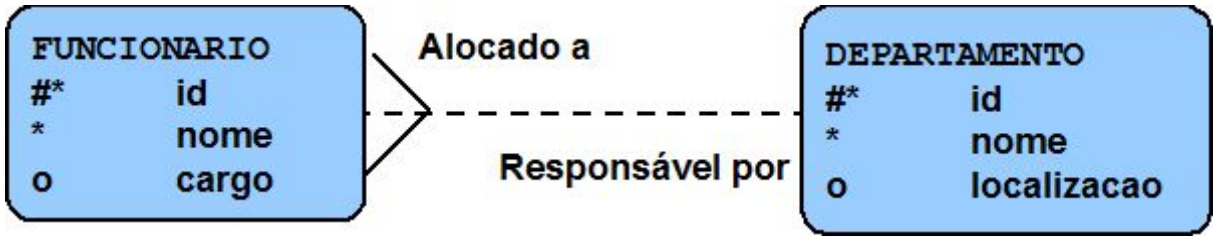
Neste contexto, poderíamos assumir duas TABELAS uma de ALUNOS e outra de ESCOLA, onde o relacionamento entre elas seria "ESTUDA".

Para que possamos aprofundar nesses conceitos, é interessante que abordamos o Modelo de Entidade-Relacionamento, também conhecido pelos acrônimos MER, DER (Diagrama de Entidade-Relacionamento) e, o mais comum: "Modelo E-R".

Modelo Entidade-Relacionamento

Um modelo entidade-relacionamento (ER) é uma ilustração de várias entidades (ou tabelas) que representam um conjunto de dados e as relações entre elas. Um modelo ER é comumente derivado de especificações empresariais ou narrativas e construído durante a fase de análise do ciclo de vida do desenvolvimento de sistemas, conforme discutido a seguir. Uma vez que os modelos ER especificam abstrações de modelos de dados, esta fase de desenvolvimento costuma ser única. Embora as empresas possam mudar suas atividades, o tipo de informação tende a permanecer constante. Portanto, as estruturas de dados também tendem a ser constantes.

Abaixo encontra-se um exemplo de modelo simples com duas tabelas (FUNCIONARIO e DEPARTAMENTO), com uma relação "responsável por".



Componentes Chaves de um ER:

- **Entidade:** Tabela ou repositório contendo todas as dimensões e informações sobre um mesmo conjunto de dados. Exemplos: contratos, clientes, departamentos, empregados e pedidos.

- **Atributo:** Propriedade de uma entidade. Por exemplo, para a entidade de pessoas, os atributos podem ser nome, a data de nascimento, email, entre outros. Cada um dos atributos pode ser obrigatório ou opcional.

- **Relacionamento:** Associação nomeada entre entidades mostrando a obrigatoriedade e grau. Exemplos: clientes e contratos; empregados e departamentos; pedidos e itens.

Convenções do Modelo Entidade-Relacionamento

Entidades: Para representar uma entidade em um modelo, utiliza-se as seguintes convenções:

- Caixa com qualquer dimensão.
- Nome da entidade no plural e em maiúsculo.
- Apenas uma Palavra ou Duas (se necessário apenas, e separadas por "_")

Atributos: Para representar um atributo em um modelo, siga as seguintes convenções:

- Utilize nomes no singular em minúsculo.
- Marque os atributos que compõe o identificador único principal da entidade com um símbolo "#"
- Marque atributos obrigatórios, ou valores que devem ser conhecidos, com um asterisco: "*"
- Valores opcionais não são marcados.

Relacionamentos: Cada lado da relação contém:

- Um nome. Por exemplo: **ensinado por** ou **designado para**
- Uma obrigatoriedade: **deve ser** ou **pode ser**
- Um grau: ambos **um e somente um** ou um dos lados **um ou mais Nota.**

Símbolo	**Descrição o relacionamento**
Linha pontilhada	Elemento opcional indicando “pode ser”
Linha sólida	Elemento obrigatório indicando “deve ser”
Pé de “galinha”	Elemento de grau indicando “um ou mais”
Linha simples	Elemento de grau indicando “um e somente um”

Identificadores Únicos: Um identificador único (UID) é qualquer combinação de atributos ou relações, ou ambos, que servem para distinguir ocorrências distintas de uma mesma entidade. Cada ocorrência da entidade deve ser exclusivamente identificada.

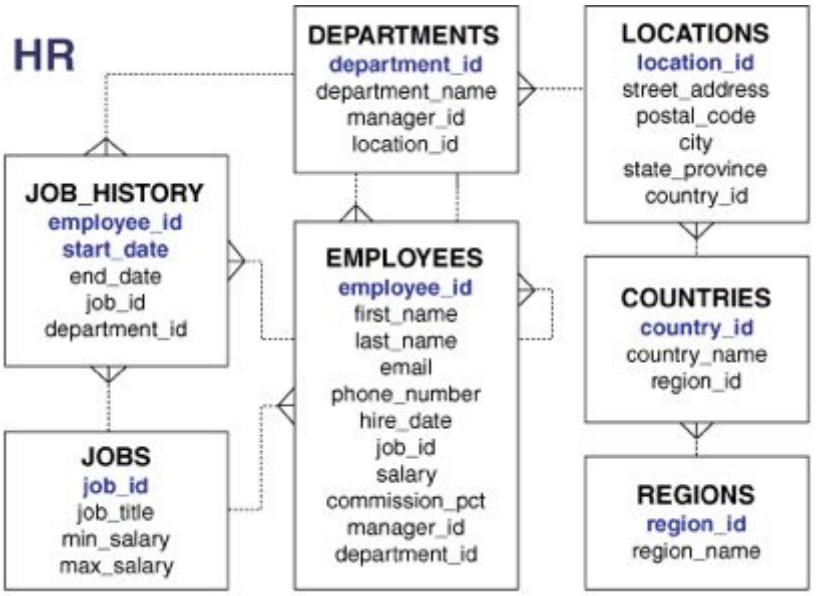
- Marque cada atributo que é parte do UID com o símbolo: #
- Marque UIDs secundários com o símbolo entre parênteses: (#)

Schemas

Na medida que o modelo ER adquire completude (tabelas, entidades, relacionamentos), é comum que seja denominado ESQUEMA ou no inglês "SCHEMA" de dados. O conceito de schema diferencia-se minimamente em cada tecnologia, porém o significado é sempre semelhante: Sistema, Módulo, ou Porção de Aplicação auto-contida e independente. Um banco de dados pode ter um ou mais schemas.

Por exemplo, um banco de dados de ERP pode possuir os schemas: HR (*Human Resources* / Recursos Humanos), SALES (Vendas), etc, todos partes de um mesmo sistema, porém segmentos separados e virtualmente dependentes entre si.

E como se parece um schema?



No modelo acima observa-se tabelas, relacionamentos e relacionamentos.

Normalização de Dados

Um tópico importante, embora muitas vezes massivo apesar de empírico, é o processo conhecido como Normalização de Dados, parte integrante da proposta inicial de Edgar F. Codd, conforme supracitado, a fim de eliminar duplicidade de dados e agregar velocidade e estrutura ao modelo.

A normalização de dados consiste em cinco regras, que recebem o nome de formas normais. Em seu trabalho original, Codd definiu três dessas formas, mas existem hoje outras formas normais geralmente aceitas. Essas correspondem a um conjunto de regras de simplificação e adequação de tabelas. Diz-se que a tabela do banco de dados relacional está numa certa forma normal quando satisfaz as condições exigentes.

Cada forma normal listada abaixo representa uma condição mais forte das que a precedem na lista. Para a maioria dos efeitos práticos, considera-se que as bases de dados estão normalizadas se aderirem à terceira forma normal.

Inicialmente, são definidos todos os atributos que estão relacionados a uma entidade principal, atribuindo uma chave primária. Feito isso, partimos para a análise da tabela de acordo com as formas normais a seguir:

- **Primeira Forma Normal (ou 1FN)**: Nesta forma os atributos precisam ser atômicos, o que significa que as tabelas não podem ter valores repetidos e nem atributos possuindo mais de um valor.

  Exemplo: CLIENTE = {ID + ENDEREÇO + TELEFONES}. Porém, uma pessoa poderá ter mais de um número de telefone, sendo assim o atributo "TELEFONES" é multivalorado.

  Para normalizar, é necessário:

  1. Identificar a chave primária e também a coluna que possui dados repetidos (nesse exemplo "TELEFONES") e removê-los;
  2. Construir uma outra tabela com o atributo em questão, no caso "TELEFONES". Mas não se esquecendo de fazer uma relação entre as duas tabelas: CLIENTE = {ID + ENDEREÇO} e TELEFONE (nova tabela) = {CLIENTE_ID (chave estrangeira) + TELEFONE}.

- **Segunda Forma Normal (ou 2FN)**: Primeiramente, para estar na 2FN é preciso estar também na 1FN. 2FN define que os atributos regulares, ou seja, os não chave, devem depender unicamente da chave primária da tabela. Assim como as colunas da tabela que não são dependentes dessa chave devem ser removidas da tabela principal e cria-se uma nova tabela utilizando esses dados.

  Exemplo: PROFESSOR_CURSO = {ID_PROF + ID_CURSO + SALARIO + DESCRICAO_CURSO} Como podemos observar, o atributo "DESCRICAO_CURSO" não depende unicamente da chave primária "ID_PROF", mas sim somente da chave "ID_CURSO".

  Para normalizar, é necessário:

  1. Identificar os dados não dependentes da chave primária (nesse exemplo

"DESCRICAO_CURSO") e removê-los;
2. Construir uma nova tabela com os dados em questão: PROFESSOR_CURSO = {ID_PROF + ID_CURSO + SALARIO} e CURSOS (nova tabela) = {ID_CURSO + DESCRICAO_CURSO}.

- **Terceira Forma Normal (ou 3FN)**: Assim como para estar na 2FN é preciso estar na 1FN, para estar na 3FN é preciso estar também na 2FN. 3FN define que todos os atributos dessa tabela devem ser funcionalmente independentes uns dos outros, ao mesmo tempo que devem ser dependentes exclusivamente da chave primária da tabela. 3NF foi projetada para melhorar o desempenho de processamento dos banco de dados e minimizar os custos de armazenamento.

  Exemplo: FUNCIONARIO = {ID + NOME + VALOR_SALARIO + VALOR_FGTS}. Como sabemos o valor do FGTS é proporcional ao salário, logo o atributo normal "VALOR_FGTS" é dependente do também atributo normal "VALOR_SALARIO".

  Para normalizar, é necessário:
  1. Identificar os dados dependentes de outros (nesse exemplo "VALOR_FGTS");
  2. Removê-los da tabela. Esses atributos poderiam ser definitivamente excluídos -- e deixando para a camada de negócio a responsabilidade pelo seu cálculo -- ou até ser movidos para uma nova tabela e referenciar a principal ("FUNCIONARIO").

A partir das 3 formas normais propostas por Codd, outras derivaram, em geral relacionadas a situações e cenários bastante específicos. São elas:

- Forma Normal de Boyce-Codd (ou BCNF): requer que não exista nenhuma dependência funcional não trivial de atributos em algo mais do que um superconjunto de uma chave candidata. Neste estágio, todos os atributos são dependentes de uma chave, de uma chave inteira e de nada mais que uma chave (excluindo dependências triviais, como A → A);

- Quarta Forma Normal (ou 4FN) requer que não exista nenhuma dependência multi-valorada não trivial de conjuntos de atributo em algo mais de que um superconjunto de uma chave candidata;

- Quinta Forma Normal (ou 5FN ou PJ/NF) requer que não exista dependências de joins (associações) não triviais que não venham de restrições chave;

- Domain-Key Normal Form (ou DK/NF) requer que todas as restrições sigam os domínios e restrições chave.

Existem ainda estudiosos que defendem a existência de outras formas normais, porém a sua exploração torna este estudo demasiadamente denso para o objetivo chave deste material. A imagem abaixo, retirada da Wikipédia, apresenta uma tabela e os temos para que uma pesquisa adicional seja realizada se assim for de seu interesse.

- UNF: Unnormalized form
- 1NF: First normal form
- 2NF: Second normal form
- 3NF: Third normal form
- EKNF: Elementary key normal form
- BCNF: Boyce–Codd normal form
- 4NF: Fourth normal form
- ETNF: Essential tuple normal form
- 5NF: Fifth normal form
- DKNF: Domain-key normal form
- 6NF: Sixth normal form

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Primary key (no duplicate tuples)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No repeating groups	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic columns (cells have single value)	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No partial dependencies (values depend on the whole of every Candidate key)	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
No transitive dependencies (values depend only on Candidate keys)	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency involves either a superkey or an elementary key's subkey	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	N/A
No redundancy from any functional dependency	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	N/A
Every non-trivial, multi-value dependency has a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	N/A
A component of every explicit join dependency is a superkey[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
Every non-trivial join dependency is implied by a candidate key	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	N/A
Every constraint is a consequence of domain constraints and key constraints	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	N/A
Every join dependency is trivial	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

ACID

Outro conceito importante no que diz respeito ao comportamento de bancos de dados em geral é o **ACID** (acrônimo de **A**tomicidade, **C**onsistência, **I**solamento e **D**urabilidade - do inglês: *Atomicity, Consistency, Isolation, Durability*). No contexto de banco de dados, quaisquer ações ou sequência de ações de banco de dados que insiram, alterem ou removam dados devem satisfazer as propriedades ACID e, portanto, pode ser percebida como uma operação lógica única sobre os dados.

Por exemplo, uma transferência de fundos de uma conta bancária para outra, mesmo envolvendo múltiplas mudanças, como debitar uma conta e creditar outra, é uma transação única.

De um modo resumido:

- **Atomicidade**: Trata a ação como parte indivisível. A transação deve ter todas as suas operações executadas em caso de sucesso ou, em caso de falha, nenhum resultado de alguma operação refletido sobre o banco de dados. Em projeto de banco de dados relacional, atomicidade também se refere à Primeira Forma Normal, na qual uma relação não pode conter atributos multi valorados, ou seja, o domínio de um atributo deve conter apenas valores atômicos.

- **Consistência**: Qualquer ação deve sempre levar o banco de dados de um estado consistente a um outro estado consistente, ou seja, uma ação deve respeitar as regras de integridade dos dados (como unicidade de chaves, restrições de integridade lógica, etc.). Todos os dados escritos no banco de dados devem ser válidos de acordo com todas as regras definidas, incluindo restrições. Isso não garante a correção da transação de todas as maneiras que o sistema tenha sido modelado.

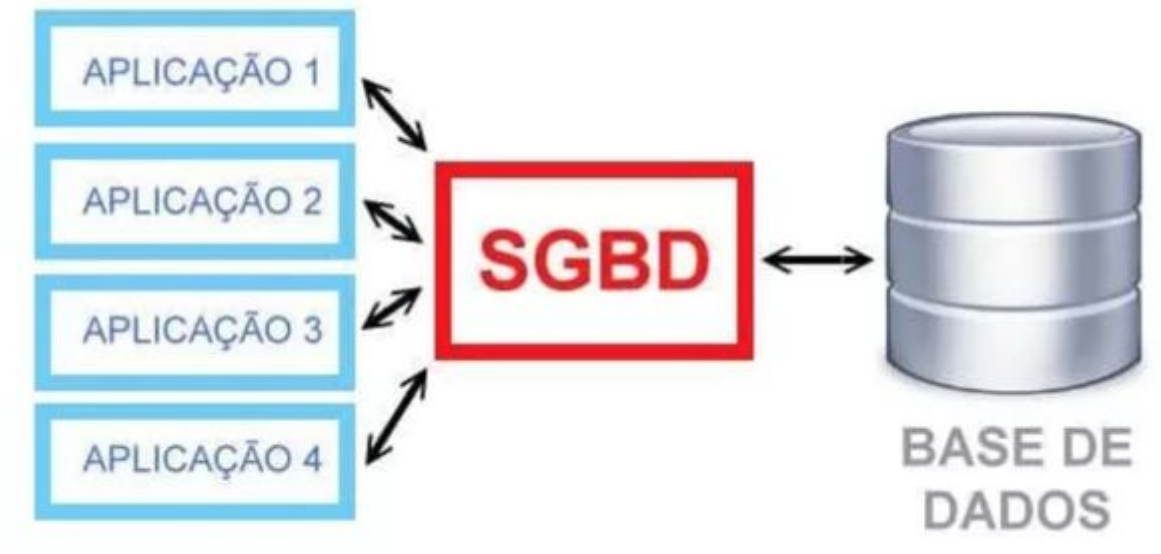
- **Isolamento**: Em sistemas multi usuários, várias transações podem acessar simultaneamente o mesmo dado (ou parte do dado) no banco de dados. Por exemplo, no mesmo instante é possível que um usuário tente alterar um registro e outro usuário esteja tentando ler este mesmo registro. O *isolamento* é um conjunto de técnicas que tentam evitar que transações paralelas interfiram umas nas outras, fazendo com que o resultado de várias transações em paralelo seja o mesmo resultado se as mesmas transações fossem executadas sequencialmente (uma após a outra). Operações exteriores a uma dada transação jamais verão esta transação em estados intermediários. Fornecer isolamento é o objetivo principal do controle de concorrência. Dependendo do método de controle de concorrência, os efeitos de uma transação incompleta podem não ser visíveis para outra transação.

- **Durabilidade**: Os efeitos de uma operação devem persistir no banco de dados mesmo em casos de quedas de energia, travamentos ou erros. Garante que os dados estarão disponíveis em definitivo. Em um banco de dados relacional, por exemplo, quando um grupo de instruções SQL é executado, os resultados precisam ser armazenados permanentemente (mesmo que o banco de dados falhe imediatamente depois). Para se defender contra a perda de energia, as transações (ou seus efeitos) devem ser registradas em uma memória não volátil.

Sistema Gerenciador de Banco de Dados (SGBD)

Ciente dos vários conceitos mencionados especialmente relacionados a ACID, gerenciamentos de entidades (tabelas) e relacionamentos, "entendimento" e execução de instruções SQL, qual o componente responsável por realizar toda esta gestão e controle?

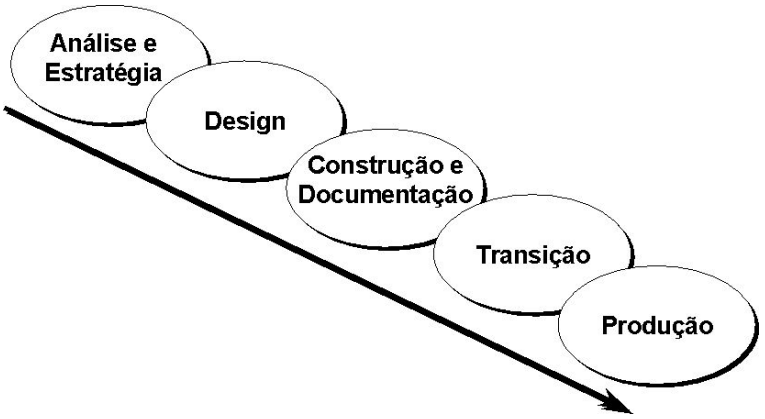
Na realidade se trata de um conjunto de programas que funcionam em conjunto a este conjunto se dá o nome de Sistema de Gerenciamento de Banco de Dados (SGBD) (em inglês, *Data Base Management System - DBMS*). Seu principal objetivo é retirar da aplicação cliente a responsabilidade de gerenciar o acesso, a persistência, a manipulação e a organização dos dados. O SGBD disponibiliza uma interface para que seus clientes possam incluir, alterar ou consultar dados previamente armazenados. Em bancos de dados relacionais esta interface comumente é a própria linguagem SQL.



Processo de Desenvolvimento de Software

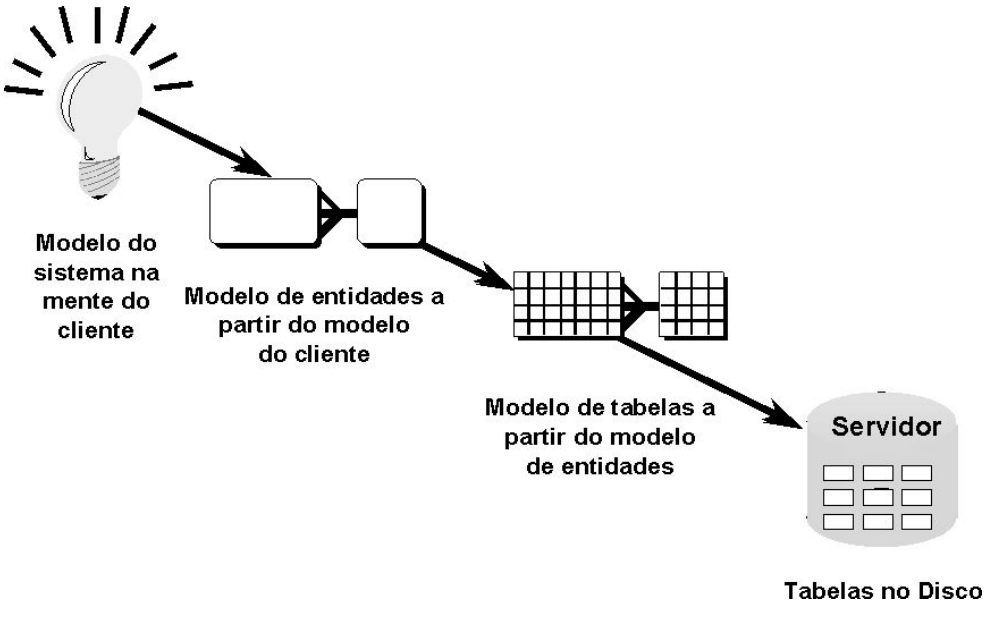
Uma outra perspectiva para análise destas informações, um tanto quanto mais mercadológica e menos teórica, permite-nos observar o projeto de banco de dados, ou elaboração do Modelo Entidade-Relacionamento a partir da perspectiva linear (embora nem sempre o seja).

O diagrama abaixo apresenta uma simplificação do ciclo de desenvolvimento de uma solução.



Neste contexto, a imagem abaixo ilustra claramente como o processo de concepção de um modelo de dados, onde as principais fases são:

- Descrição do problema e/ou sistema a ser resolvido/implementado
- Entendimento e levantamento de dados relacionados ao modelo
- Modelagem de dados em entidades e relacionamentos (normalização do modelo)
- A partir de Modelo E-R, geração de instruções SQL para definição de estruturas
- Criação das mesmas no SGBD através de instruções SQL.

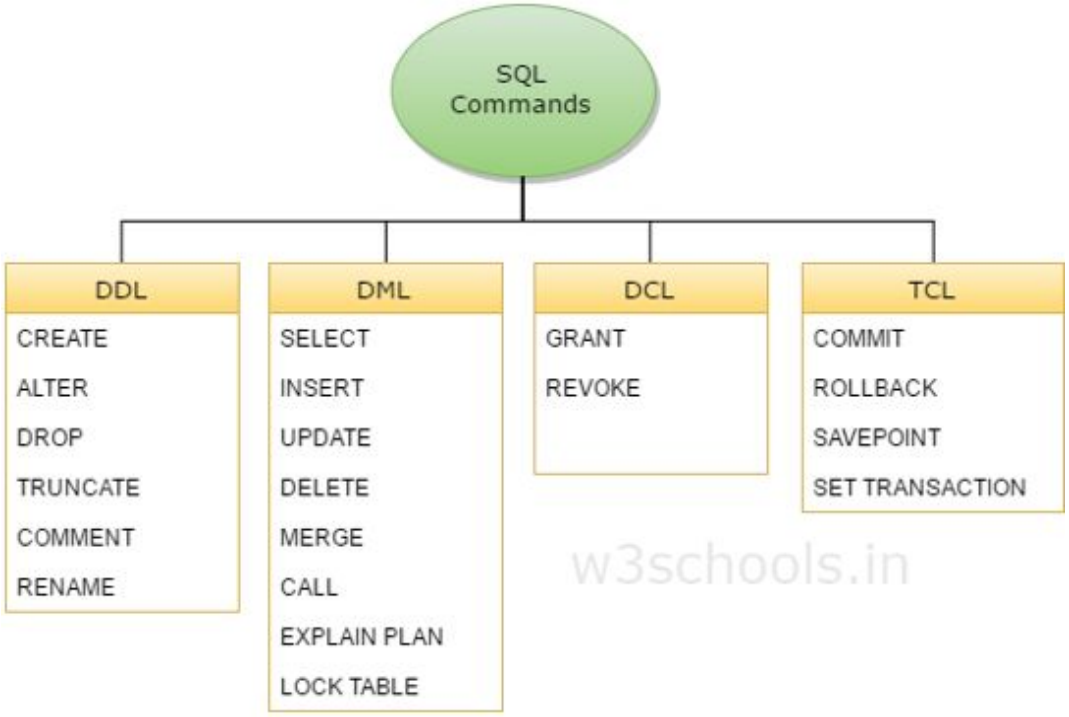


2. Introdução ao SQL

Conforme descrito no último capítulo, o SQL como conhecemos atualmente é resultado da evolução do SEQUEL, linguagem proposta por Edgar F. Codd em seu trabalho onde propôs a primeira concepção de banco de dados relacional.

O SQL adquiriu público muito rapidamente devido sua facilidade de compreensão e escrita. Ao mesmo passo, tornou-se padrão ANSI (*American National Standards Institute*), descrevendo linguagem e comandos base aceitos por qualquer implementação de banco de dados relacional (de qualquer fabricante). O padrão não limita, contudo, que linguagens tenham sintaxes específicas como de fato muitas possuem, como é o caso do Oracle, que será utilizado ao longo deste curso.

Ao tratarmos de SQL, convém tratarmos de conceitos como tipos de operações (DML, DDL, DCL, entre outros).



DDL x DML x DCL

Para que seja mais fácil classificar e administrar diferentes operações em bancos de dados através de SQL, foram criadas as seguintes categorias:

- **DDL (Data Definition Language ou Linguagem de definição de dados)** é um conjunto de comandos dentro do comando SQL usada para a definição das estruturas de dados, fornecendo as instruções que permitem a criação, modificação e remoção das tabelas, assim como criação de índices. Estas instruções SQL permitem definir a estrutura de uma base de dados, incluindo as linhas, colunas, tabelas, índices, e outros metadados. Entre os principais comandos DDL estão:
    - CREATE - Usado para criar uma nova tabela, uma visão de uma tabela, ou outro objeto em um banco de dados.
    - ALTER - Usado para modificar um objeto existente de um banco de dados, como uma tabela.
    - DROP - Usado para apagar toda uma tabela, uma visão de uma tabela ou outro objeto em um banco de dados.
    - REPLACE: Utilizado para renomear tabelas.
    - TRUNCATE: Utilizado para remover todas as linhas/blocos relacionados a uma tabela.

- **DML (Data Manipulation Language ou Linguagem de manipulação de dados)** é o grupo de comandos dentro da linguagem SQL utilizado para a recuperação, inclusão, remoção e modificação de informações em bancos de dados. Os principais comandos DML são:
    - INSERT: Inserção de Dados
    - UPDATE: Atualização de Dados
    - DELETE: Exclusão de Dados

- **DCL (Data Control Language ou Linguagem de controle de dados)**: é o grupo de comandos que permitem ao administrador de banco de dados controlar o acesso aos dados deste banco. Alguns exemplos de comandos DCL são:
    - GRANT: Permite dar permissões a um ou mais usuários e determinar as regras para tarefas determinadas;
    - REVOKE: Revoga permissões dadas por um GRANT.

  Algumas tarefas básicas que podemos conceder ou barrar permissões são:

    - CONNECT
    - SELECT
    - INSERT
    - UPDATE
    - DELETE

TCL e DQL

TCL e DQL são classificações secundárias, frequentemente não mencionadas ou agrupadas junto aos demais. Porém convém para o nosso estudo que estejam separadas, conforme:

- **TCL (Transaction Control Language ou Linguagem de controle de de transação)**: são comandos SQL relacionados ao controle de outras operações de SQL, inerentes ao conceito de transação de SQL, a ser descrito em mais detalhes a seguir. Por hora, convém saber:
    - COMMIT: Confirmação de Transação
    - ROLLBACK: Cancelamento de Transação
    - SAVEPOINT: Ponto de segurança em transação.

- **Data Query Language (DQL):** Consiste na consulta de dados. Muitas vezes caracterizada como um subconjunto de linguagem de manipulação. O comando é SELECT, usado para obter certos dados ou registros a partir de uma ou mais tabelas.

Sintaxe: DDLs

A sintaxe base de qualquer comando DDL segue o seguinte modelo:

<COMANDO DML> TIPO OBJETO NOME <CLÁUSULAS>

As cláusulas dependem do tipo de objeto. Usaremos objetos do tipo TABLE para ilustração da estrutura de SQLs do tipo DDL.

A criação de tabelas inclui a definição das colunas, por exemplo:

CREATE TABLE: (<COLUNAS>) <DEFINICOES FISICAS>

Um exemplo de criação de tabela:

```
CREATE TABLE exemplo(
          empno  NUMBER(5) PRIMARY KEY,
          ename  VARCHAR2(15) NOT NULL,
          ssn  NUMBER(9) ENCRYPT,
          sal    NUMBER(7,2),
     deptno  NUMBER(3) NOT NULL))
   TABLESPACE admin_tbs
   STORAGE ( INITIAL 50K);
```

Outro exemplo de criação de tabela, agora incluindo sintaxe para *constraint* do tipo chave estrangeira:

```
CREATE TABLE exemplo(
          empno  NUMBER(5) PRIMARY KEY,
          ename  VARCHAR2(15) NOT NULL,
          ssn  NUMBER(9) ENCRYPT,
          sal    NUMBER(7,2),
     deptno  NUMBER(3) NOT NULL)
CONSTRAINT admin_dept_fkey REFERENCES departments (department_id))
   TABLESPACE admin_tbs
   STORAGE ( INITIAL 50K);
```

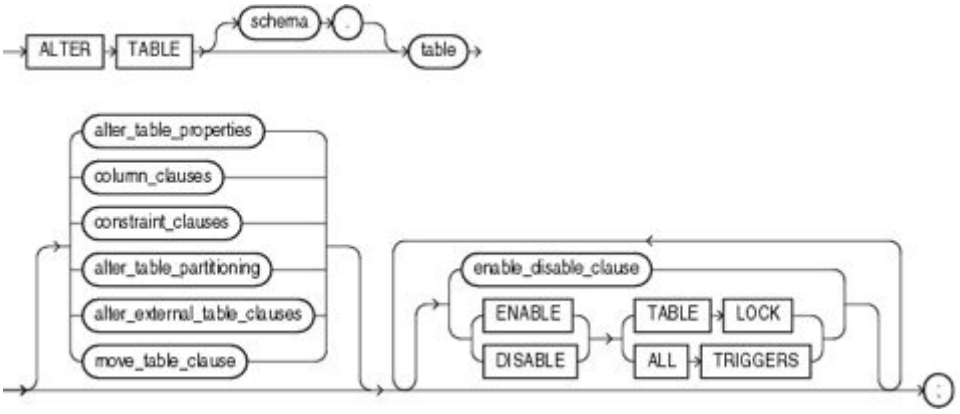
A sintaxe usada para remoção de tabelas segue a mesma estrutura:

```
DROP TABLE exemplo;
DROP TABLE exemplo CASCADE CONSTRAINTS;
DROP TABLE exemplo PURGE;
```

A sintaxe usada para alteração de tabelas também segue a mesma estrutura, porém possui algumas opções adicionais, uma vez que pode ser utilizada para alterar qualquer uma das definições feitas na criação da tabela. Alguns exemplos:

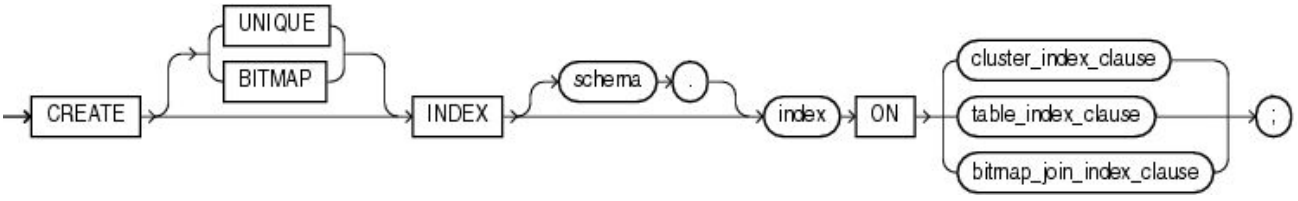
```
ALTER TABLE exemplo MODIFY ename varchar2(75);
ALTER TABLE exemplo ADD mothername varchar2(75);
```

A imagem abaixo apresenta um diagrama mais completo sobre as possíveis variações de comandos para ALTER TABLE:

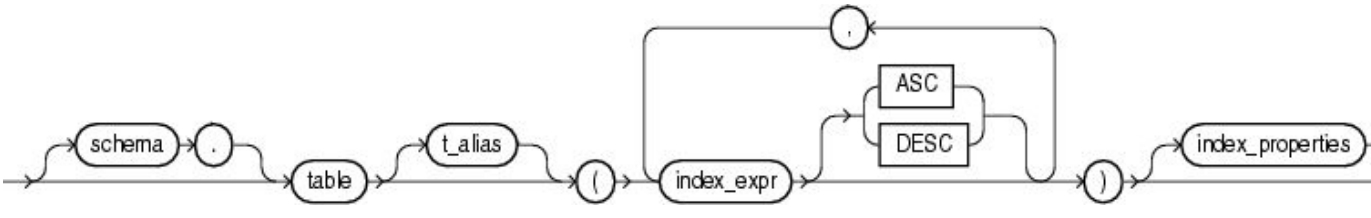


Da mesma forma, estruturas semelhantes são utilizadas para tipos de objetos diferentes. Ilustrativamente, a sintaxe para criação de índices:

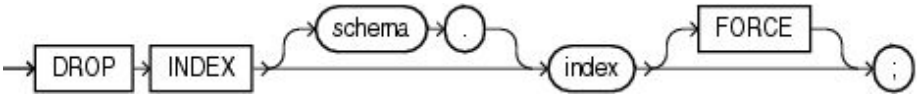
Criação de Índices simples:



Cláusulas para Índices de tabelas:



Remoção de Índices:



Sintaxe: DMLs

Conforme já citado, comandos de DML tratam essencialmente da manipulação de dados (inserção, alteração ou remoção) de estruturas criadas e mantidas através dos DMLs, essencialmente, tabelas. São eles:

- **INSERT**: Inserir dados em uma tabela. Exemplo:

```
INSERT INTO PESSOAS VALUES (64283467, 'ANA BEATRIZ');
```

- **UPDATE**: Atualizar dados em uma tabela. Exemplo:

```
UPDATE PESSOAS SET NOME='JOSÉ ARMANDO' WHERE ID= 64283467;
```

- **DELETE**: Remover dados de uma tabela. Exemplo:

```
DELETE FROM PESSOAS WHERE ID= 64283467;
```

- **MERGE**: Juntar dados entre duas ou mais tabelas de acordo com condições de junção estabelecidas. Exemplo:

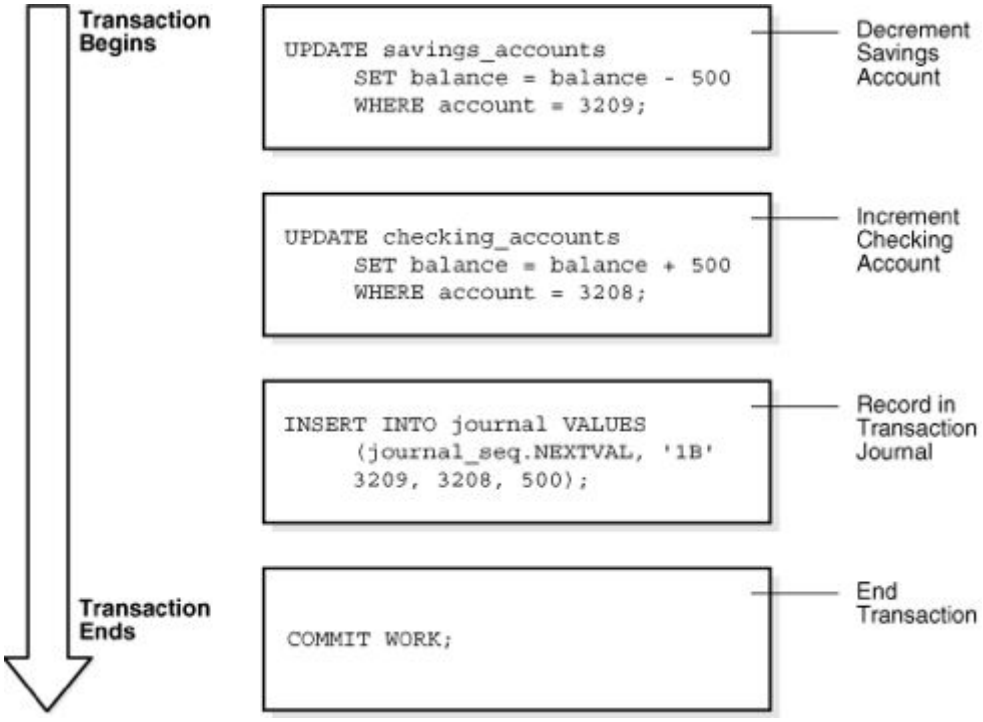
```
MERGE INTO  customers_backup bkup
USING       customers cust
ON         (bkup.cust_id = cust.cust_id)
WHEN MATCHED THEN
UPDATE SET
   bkup.cust_name = cust.cust_name ,
   bkup.cust_surfing_package = cust.cust_surfing_package
WHEN NOT MATCHED THEN
INSERT                        VALUES(cust.cust_id,cust.cust_name,
cust.cust_surfing_package)
```

Transação SQL

Uma transação simboliza uma unidade de trabalho executada SGBD, e tratada de maneira coerente e confiável, independente de outras transações. Em outras palavras, a transação é uma unidade lógica ou de trabalho, composta por uma ou mais operações (ou comandos SQL). Uma transação de banco de dados, por definição deve respeitar ACID.

Do ponto de vista prático, é comum que realizemos mais de uma operação DML para que o sistema saia de um estado consistente para outro. Por este motivo, operações DML são o único tipo de comandos no SGBD Oracle que precisam de confirmação, através do comando COMMIT ou de cancelamento, através do comando ROLLBACK (estes comandos serão abordados em mais detalhe no próximo subtítulo).

Este mecanismo permite que uma série de operações seja tratada como um bloco único de operação lógica, sendo cancelado completamente em caso de alguma falha ou comportamento inesperado com a operação ou confirmado em caso de sucesso do procedimento ou unidade de trabalho como um todo. Este mecanismo garante também a consistência e integridade lógica do sistema, evitando problemas com leituras em pontos intermediários do procedimento, causando, entre outros, problemas como *phantom read* (leitura fantasma).



Revisando o conceito de ACID a partir da perspectiva de transações:

- **Atomicidade**: Todas as ações que compõem a unidade de trabalho da transação são indivisíveis e o sucesso ou cancelamento afeta todas as operações neste conjunto.

- **Consistência**: Todas as regras e restrições definidas no banco de dados devem ser obedecidas. Relacionamentos por chaves estrangeiras, checagem de valores para campos restritos ou únicos devem

ser obedecidos para que uma transação possa ser completada com sucesso.

- **Isolamento**: Cada transação funciona completamente à parte de outras estações. Todas as operações são parte de uma transação única. O princípio é que nenhuma outra transação, operando no mesmo sistema, possa interferir no funcionamento da transação corrente(é um mecanismo de controle). Outras transações não podem visualizar os resultados parciais das operações de uma transação em andamento (ainda em respeito à propriedade da atomicidade).

- **Durabilidade**: Significa que os resultados de uma transação são permanentes e podem ser desfeitos somente por uma transação subsequente.Por exemplo: todos os dados e status relativos a uma transação devem ser armazenados num repositório permanente, não sendo passíveis de falha por uma falha de hardware.

Controles de Transação (TCL)

Conforme supracitado, o controle de transações é feito através de instruções SQL. Conforme já observamos, estes comandos são classificados como TCL (*Transaction Control Language* ou Linguagem de Controle de Transação). São eles:

- **COMMIT**: Confirmação de transação.

```
COMMIT;
```

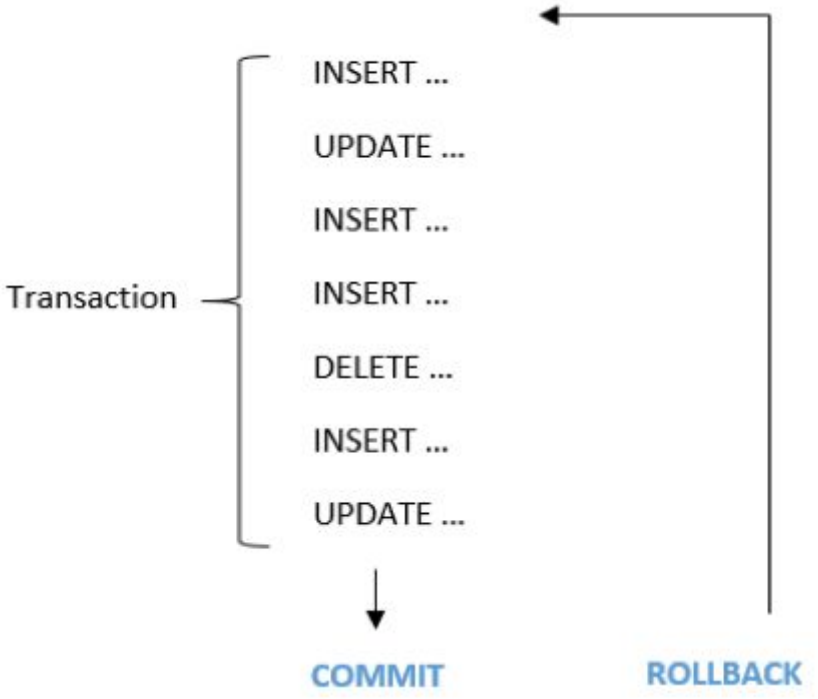
- **ROLLBACK**: Cancelamento de transação.

```
ROLLBACK;
```

- **SAVEPOINT**: Pontos de segurança durante a transação para rollback se necessário. Exemplo:

```
SAVEPOINT Antes_Merge;
ROLLBACK TO Antes_Merge;
```

A imagem abaixo ilustra o comportamento de TCLs combinado com DMLs:



Controles de Acesso e Permissionamento Básico (DCL)

Do mesmo modo, outra classificação relevante de comandos SQL são os DCLs (*Data Control Language* ou Linguagem de Controle de Dados), embora o termo seja muito mais relacionado ao *acesso* a dados, porém não restrito.

A entender. Um objeto ao ser criado no banco de dados pode apenas ser manipulado pelo schema ao qual pertence. Para que qualquer operação seja realizada a partir de demais schemas, estes devem possuir permissões para tal (permissão de consulta, alteração de dados, inserção de dados, deleção de dados).

Existem duas categorias de permissões:

- **Permissões Sobre Objetos**: Permissões para ações específicas em objetos específicos, como permissões de leitura, inserção, deleção, alteração de dados ou execução (em caso de procedimentos), sempre especificando o objeto ao qual o privilégio se refere. Exemplos:

```
SELECT ON SCHEMA.PESSOAS TO ALUNO1;
DELETE ON SCHEMA.PESSOAS TO ALUNO2;
EXECUTE ON SCHEMA.PROCEDURE1 TO ALUNO2;
```

- **Permissões de Sistema**: Permissões mais genéricas sobre classes de objetos ou operações em nível de sistema. Não há especificação de objetos e sim de ações e/ou classes de objetos. Exemplos:

```
SELECT ANY TABLE TO ALUNO1;
UPDATE ANY TABLE TO ALUNO1;
EXECUTE ANY PROCEDURE TO ALUNO1;
```

Que podem ser concedidas para:

- **Usuários**: Usuários ou schemas, especificamente (ALUNO1, ALUNO2). Conforme os exemplos acima.

- **Roles**: Traduzido como "papeis", representa uma classe de usuários que pode ser utilizada para agrupamento de privilégios. Estas roles podem ser também concedidas a usuários.

Estas permissões são concedidas ou revogadas através dos comandos:

- **GRANT**: Concede permissão. Exemplos:

```
GRANT SELECT ANY TABLE TO ALUNO1;
GRANT UPDATE ON SCHEMA.PESSOAS TO ALUNO1;
```

- **REVOKE**: Revoga permissão. Exemplos:

```
REVOKE SELECT ANY TABLE FROM ALUNO1;
REVOKE UPDATE ON SCHEMA.PESSOAS FROM ALUNO1;
REVOKE DBA FROM ALUNO1;
```

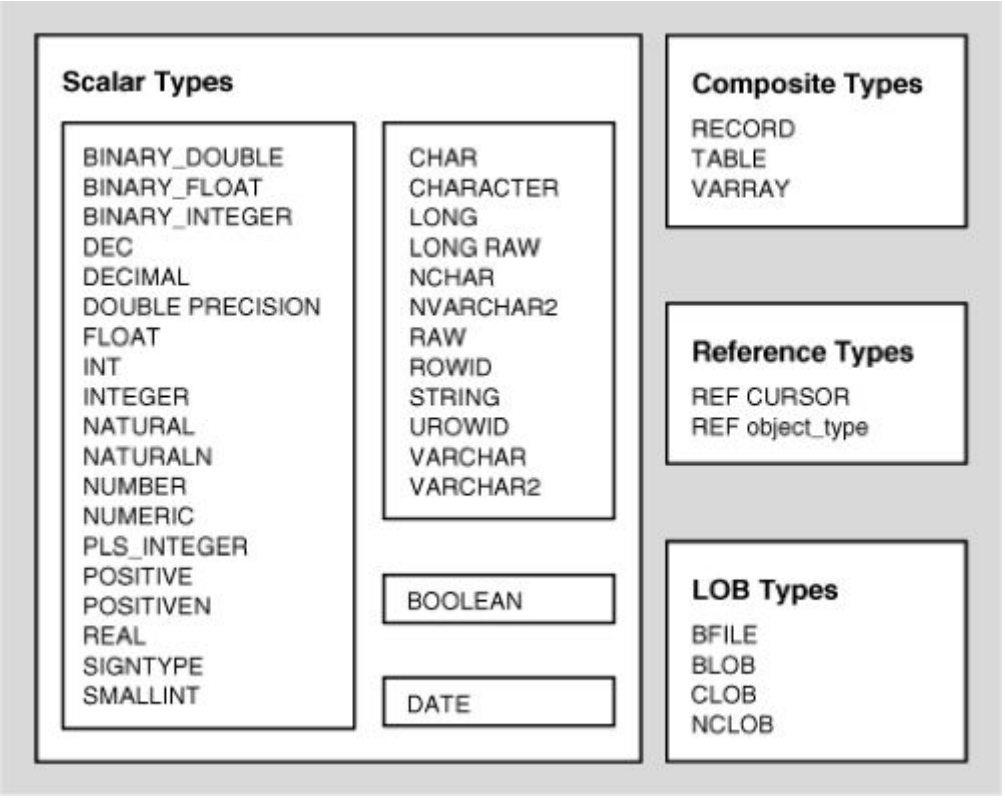
Tipos de Dados e Armazenamento

Outra perspectiva relevante sobre a evolução no que diz respeito a gestão dos dados está na maneira com que estes dados são armazenados e administrados fisicamente. Nas primeiras soluções de bancos de dados os dados eram todos armazenados contiguamente, todos do tipo alfanumérico e com tamanho pré estabelecido, conforme a ilustração abaixo. Alterações de tamanhos de colunas muitas vezes demandavam movimentação de todos os dados da tabela.

```
"E],003715,4,153,09061987,0140000.00,"IRENE HIRSH    ],1,085.00,2,066.00,3,088.00,4,125.00
"P],003715,01152000,01162000,00101,0005000.00,0007000.00,150.00,200.00,133.00,075.00,055.00,066.00,077.00
"P],003715,02152000,02162000,00102,0003000.00,0008000.00,120.00,180.00,120.00,065.00,044.00,075.00,055.00
"P],003715,03152000,03162000,00103,0005000.00,0009000.00,130.00,170.00,110.00,055.00,033.00,065.00,066.00
"P],003715,04152000,04162000,00104,0003000.00,0010000.00,140.00,160.00,100.00,045.00,056.00,075.00,065.00
"E],003941,2,165,03111959,0167000.00,"ANNE FAHEY     ],1,099.00,2,066.00,3,088.00,4,125.00
"P],003941,01152000,01162000,00105,0003000.00,0010000.00,140.00,160.00,100.00,045.00,056.00,075.00,065.00
"P],003941,02152000,02162000,00106,0003000.00,0010000.00,140.00,160.00,100.00,045.00,056.00,075.00,065.00
"P],003941,03152000,03162000,00107,0003000.00,0010000.00,140,00.160.00,100.00,045.00,056.00,075.00,065.00
"E],001939,2,265,09281988,0213000.00,"EMILY WELLMET   ],1,077.00,2,066.00,3,088.00,4,125.00
"P],001939,01152000,01162000,00108,0003000.00,0010000.00,140.00,160.00,100.00,045.00,056.00,075.00,065.00
"P],001939,02152000,02162000,00109,0003000.00,0010000.00,140.00,160.00,100.00,045.00,056.00,075.00,065.00
```

Uma evolução natural passou a ser a organização destas informações em estruturas de blocos independentes acessados fisicamente através de ponteiros e offsets. Do ponto de vista de organização de dados, contudo, a definição de tipos de dados diferentes permitiu uma melhora significativa no desempenho de algumas operações e, de quebra, agregou bastante flexibilidade as soluções existentes.

Bancos de Dados modernos possuem uma gama extremamente elevada de tipos de objetos, na sua maioria destinados a resolver problemas bastante específicos, conforme a tabela abaixo.



E cada um deles possui uma elevada variedade de especificações, conforme ilustra a tabela abaixo usada para um tipo de dado bastante simples, talvez o mais simples a disposição:

Input Data	Specified As	Stored As
7,456,123.89	`NUMBER`	7456123.89
7,456,123.89	`NUMBER(*,1)`	7456123.9
7,456,123.89	`NUMBER(9)`	7456124
7,456,123.89	`NUMBER(9,2)`	7456123.89
7,456,123.89	`NUMBER(9,1)`	7456123.9
7,456,123.89	`NUMBER(6)`	(not accepted, exceeds precision)
7,456,123.89	`NUMBER(7,-2)`	7456100

Desta forma, por hora, vamos utilizar exclusivamente aos principais tipos de dados, em geral os únicos utilizados pela maior parte das soluções de mercado:

- **NUMBER**: Números.
- **VARCHAR2(XX)**: Cadeias de caracteres alfanuméricos, onde XX é o tamanho do campo.
- **CHAR (X)**: Caracteres quando o tamanho é conhecido.
- **DATE**: Datas.

Sintaxe Base: SELECT

A sintaxe base de um comando SELECT é composta pela seguinte estrutura base:

SELECT

PROJEÇÃO / COLUNAS / ATRIBUTOS / EXPRESSÕES -> O que eu quero ver.

FROM

ENTIDADE / TABELA / [TABELAS / SUBQUERIES] -> Origem dos dados.

[**WHERE**

SELEÇÃO / FILTROS / CONDIÇÕES / "REQUISITOS"]

[**GROUP BY**

TODAS AS DIMENSÕES DA PROJEÇÃO QUE NÃO POSSUEM FUNÇÃO DE AGREGAÇÃO]

[**ORDER BY**
DIMENSÕES DA PROJEÇÃO E ORDERM DE ORDENAÇÃO]

As cláusulas obrigatórias mínimas são "**SELECT**" e "**FROM**". Imediatamente após o SELECT, informa-se a lista de colunas (ou projeção, dentro da álgebra relacional) que se procura, enquanto após o FROM se especifica a origem destes dados (em geral, a tabela ou entidade).

Após a cláusula opcional "**WHERE**" são especificados os filtros (restrições ou ainda seleção, conforme a nomenclatura da álgebra relacional). A elaboração destes filtros serão objeto de estudo dos tópicos que seguem este capítulo.

De maneira semelhante, a cláusula "**GROUP BY**" é utilizada para agrupamento de informações em caso de operações de agregação na projeção. Este tópico será clarificado nos capítulos que seguem.

A cláusula "**ORDER BY**", por sua vez, é utilizada para especificar a ordem de colunas ou propriedades da projeção (ou não) a serem utilizadas para ordenação dos dados apresentados.

SELECT: Projetando Coluna ou Listas de Colunas

Conforme descrito tópico anterior, a projeção consiste nas colunas, propriedades ou dimensões de uma tabela. Neste caso, é possível projetar (ou popularmente dito "selecionar", embora o termo esteja algebricamente incorreto) apenas as colunas que se deseja. Para exemplificar, vamos considerar a tabela abaixo:

ALUNOS			
NOME	*NASCIMENTO*	*SEXO*	*EMAIL*
ALUNO1 DA SILVA	30/09/1990	M	aluno1@targettrust.com.br
ALUNA2 SANTOS	27/08/1991	F	aluna2@targettrust.com.br

Suponhamos que seja necessário obter apenas o nome de cada aluno:

```
SELECT NOME FROM ALUNOS;
```

Agora suponhamos que seja necessário obter apenas o nome e o email de cada aluno. A sintaxe para separação de colunas na projeção é apenas a adição de vírgula, como em uma lista tradicional.

```
SELECT NOME, EMAIL FROM ALUNOS;
```

SELECT: Projetando Todas as Colunas

Suponhamos que para o mesmo exemplo, seja necessário listar todas as informações disponíveis na tabela alunos. É claro que poderíamos listar todas colunas, mas existe uma alternativa mais fácil, que é o operador *****, conforme o exemplo a seguir:

```
SELECT * FROM ALUNOS;
```

Exibindo a Estrutura de Tabelas

Suponhamos que a estrutura da tabela alunos seja desconhecida, de qual maneira poderíamos listar as colunas desta tabela? Evidentemente o "SELECT *" traria esta informação, mas traria também todos os registros desta tabela, que poderiam ser milhares. Todas as ferramentas que suportam Oracle possuem a seguinte ferramenta (que não é um SQL mas) ajuda muito na elaboração de comandos SQL:

```
DESCRIBE ALUNOS;
DESC ALUNOS;
```

Este comando descreve a estrutura da tabela bem como os tipos de dados de cada coluna.

Utilizando Alias de Colunas

Uma técnica muito comum para melhor formatação de resultados, utilização em subqueries (a serem apresentadas mais adiante) e principalmente quando funções são utilizadas é a utilização de *aliases* (ou apelidos) para colunas.

De acordo com o padrão ANSI, há sempre um *AS* (tradução literal seria "como" entre a coluna ou projeção e seu apelido (alias), conforme exemplo:

```
SELECT NOME AS NOME_COMPLETO FROM ALUNOS;
```

A sintaxe Oracle, contudo, permite que o "AS" seja opcional, de modo que o comando acima e o comando abaixo possui o mesmo efeito em bancos de dados Oracle:

```
SELECT NOME NOME_COMPLETO FROM ALUNOS;
```

Ambas as implementações permitem ainda a utilização de strings (texto) entre aspas duplas para especificação de aliases *case sensitive* (diferenciação de maiúsculas e minúsculas) e espaçamentos, conforme exemplos:

```
SELECT NOME AS "Nome Completo" FROM ALUNOS;
SELECT NOME "Nome Completo" FROM ALUNOS;
```

Para um exemplo mais completo:

```
SELECT NOME AS "Nome Completo", EMAIL "Email" FROM ALUNOS;
```

Utilizando Alias de Tabelas

De modo similar, é comum que necessitemos de colocar um alias ao nome dos objetos, sobretudo em pesquisas compostas ou quando subqueries são utilizadas (ambos a serem explorados na sequência). A sintaxe é semelhante a de aliases para colunas, conforme exemplos abaixo.

```
SELECT NOME FROM ALUNOS AL;
SELECT NOME FROM ALUNOS AS "Todos os Alunos";
```

Ambos as sintaxes estão corretas para sintaxe Oracle. A implementação de SQL ANSI pressupõe o uso da cláusula AS, da mesmo modo que para aliasing de colunas.

Prefixação de Schemas

Frequentemente necessita-se referenciar objetos pertencentes a outros schemas. Existem alguns artifícios como utilização de sinônimos e current_schema de sessão, porém a solução imediatamente prototipada para este tipo de problema é a prefixação de objetos.

Digamos, por exemplo, que a tabela ALUNOS pertença ao schema ESCOLA, e você está conectado no banco de dados com uma credencial de consulta apenas. A melhor maneira de acessar os objetos do schema ESCOLA é pré fixá-los com o nome do esquema que o possui e ponto. Neste caso, com "ESCOLA.". Por exemplo:

```
SELECT NOME, EMAIL FROM ESCOLA.ALUNOS;
```

Neste caso, a utilização de aliasing pode se tornar interessante, conforme o exemplo:

```
SELECT NOME, EMAIL FROM ESCOLA.ALUNOS AL;
```

Prefixação de Colunas

Da mesma maneira, esta é uma técnica comum quando queries compostas ou subqueries com aliases de colunas são utilizadas. Se faz necessário em situações onde duas tabelas (ou subqueries) possuem coluna de mesmo nome. Pode-se utilizar o nome da tabela ou o seu alias para prefixação, conforme exemplos a seguir:

```
SELECT ALUNOS.NOME, ALUNOS.EMAIL FROM ALUNOS;
```

```
SELECT AL.NOME, AL.EMAIL FROM ESCOLA.ALUNOS AL;
```

SELECT: Projetando uma String

Caso se queira projetar apenas uma string, a sintaxe SQL para tal é o uso da string entre aspas simples ('), conforme o exemplo abaixo.

```
SELECT 'Esta é uma string' FROM ALUNOS;
```

Evidentemente para este caso, a tabela não necessariamente seria ALUNOS, mas qualquer tabela do banco de dados. O uso de uma tabela em específico faz sentido, contudo, em caso a string seja utilizada em combinação com alguma coluna desta tabela, conforme descreve-se no próximo item.

Funções de Concatenação

Ainda no segmento de soluções interessantes, uma das primeiras funções tipicamente apresentadas ao estudarmos funções em consultas SQL é a função de concatenação, que no Oracle possui uma sintaxe in-fixada. A função de concatenação no Oracle é composta de "||" (dois pipelines). Esta função permite juntar dados em formato de string e retorná-las como se fosse uma só coluna (ou projeção). De modo que a projeção (ou coluna) consiste na realidade do resultado da operação de concatenação.

Veja o exemplo para melhor entendimento:

```
SELECT 'O email de '||NOME||' é '||EMAIL FROM ALUNOS;
```

Neste exemplo a função de concatenação é utilizada 3 vezes, intercalando dados com strings. Este é um exemplo onde o uso de aliases de colunas passa a ser interessante, por exemplo:

```
SELECT 'O email de '||NOME||' é '||EMAIL AS "Emails dos Alunos"
        FROM ALUNOS;
```

Neste contexto, considerando a tabela:

ALUNOS			
NOME	*NASCIMENTO*	*SEXO*	*EMAIL*
ALUNO1 DA SILVA	30/09/1990	M	aluno1@targettrust.com.br
ALUNA2 SANTOS	27/08/1991	F	aluna2@targettrust.com.br

O retorno da consulta seria:

```
SQL> SELECT 'O email de '||NOME||' é '||EMAIL AS "Emails dos Alunos" FROM ALUNOS;

Emails dos Alunos
-----------------------------
O email de ALUNO1 DA SILVA é aluno1@targettrust.com.br
O email de ALUNA2 SANTOS é aluna2@targettrust.com.br
```

Eliminação de Linhas Duplicadas

Agora, suponhamos que a tabela ALUNOS tenha o seguinte conteúdo:

ALUNOS			
NOME	*NASCIMENTO*	*SEXO*	*EMAIL*
ALUNO1 DA SILVA	30/09/1990	M	aluno1@targettrust.com.br
ALUNA2 SANTOS	27/08/1991	F	aluna2@targettrust.com.br
ALUNA2 SANTOS	27/08/1991	F	aluna2@targettrust.com.br

Existem evidentemente registros duplicados, nas linhas 2 e 3. Existe uma maneira simples para eliminação de tuplas (registros) duplicadas, trata-e da cláusula DISTINCT, a ser usada imediatamente antes após o SELECT e antes da projeção. O efeito será aplicado apenas porém a todas as colunas utilizadas na projeção. Exemplo:

```
SQL> SELECT NOME, EMAIL FROM ALUNOS;

NOME            EMAIL
--------------- -------------
ALUNO1 DA SILVA aluno1@targettrust.com.br
ALUNA2 SANTOS   aluna2@targettrust.com.br
ALUNA2 SANTOS   aluna2@targettrust.com.br


SQL> SELECT DISTINCT NOME, EMAIL FROM ALUNOS;

NOME            EMAIL
--------------- -------------
ALUNO1 DA SILVA aluno1@targettrust.com.br
ALUNA2 SANTOS   aluna2@targettrust.com.br
```

Operações e Expressões Aritméticas em SQL

Assim como a função de concatenação, operações aritméticas podem ser realizadas com notação in-fixada na projeção, são exemplos já utilizando aliasing para demonstração:

```
SELECT 10/2 "Multiplicação", 10*2 "Divisão",
       10+2 "Soma", 10-2 "Subtração"
 FROM ALUNOS;
```

O resultado esperado:

```
Multiplicação Divisão Soma Subtração
------------- ------- ---- ---------
           20       5   12         8
```

Evidentemente estas operações podem ser realizadas sobre as colunas da tabela, desde que ambas as colunas sejam de tipo numérico (ou data para soma e subtração), de modo que operações aritméticas façam sentido. Estas operações também podem ser encadeadas formando uma expressão, como na matemática que conhecemos. São alguns outros exemplos de operações e expressões aritméticas:

```
SELECT 5-2 from DUAL;
SELECT 5*2 from DUAL;
SELECT 5/2 from DUAL;
SELECT 1+2*3 from DUAL;
SELECT (1+2)*3 from DUAL;
SELECT IDADE+ID_ALUNO from ALUNOS;
```

Precedência dos Operadores

Também respeitando as regras de precedência da matemática, em caso de expressões aritméticas a precedência seguida é:

Operador	Significado	Ordem de Precedência
*	Multiplicação	1
/	Divisão	2
+	Soma	3
-	Subtração	4

Deste modo, a expressão abaixo resulta em "18":

```
SELECT 10*3/2+5-2 FROM ALUNOS;
```

Precedência utilizando Parênteses

Da mesma forma, o uso de parêntesis pode ser feito para identificação de precedência, assim como na matemática convencional.

Neste cenário, o resultado da expressão abaixo é "6".

```
SELECT (10*3)/(2+(5-2)) FROM ALUNOS;
```

Funções de Agregação em SQL

Antes que hajam desdobramentos sobre a sintaxe do SELECT, ainda tratando especificamente da projeção, existe uma classe funções relevante. São as funções de agregação. Estas funções operam sobre todos os registros da tabela (ou conforme filtro, de acordo com configuração da projeção), retornando resultados sobre todo o conjunto (ou subconjunto) de tuplas. Os exemplos abaixo certamente elucidarão a esta definição. Suponha uma tabela com todas as notas de todos os alunos.

- Quantidade de tuplas (linhas):

```
SELECT COUNT(*) from NOTAS;
```

- Quantidade de Notas (mesmo efeito prático do anterior):

```
SELECT COUNT(NOTA) from NOTAS;
```

- Maior Nota (nota máxima):

```
SELECT MAX(NOTA) from NOTAS;
```

  Menor Nota (nota mínima):

```
SELECT MIN(NOTA) from NOTAS;
```

- Média das notas (função *average* - média aritmética):

```
SELECT AVG(NOTA) from NOTAS;
```

- Soma das notas:

```
SELECT SUM(NOTA) from NOTAS;
```

Existem, evidentemente, uma gama enorme de funções de agregação, virtualmente quaisquer funções analíticas sobre conjuntos que se possa imaginar (MEDIANA, SIN, COS, etc), todas porém com o mesmo mecanismo e com notação pré-fixada, conforme exemplos acima.

3. Filtros e Tratamentos de Dados em SQL

Filtros em SQL (Cláusula WHERE)

Conforme descrito anteriormente, a cláusula WHERE é utilizada para realizar filtros (ou seleção de acordo com a álgebra relacional). Em linhas gerais, o entendimento é bastante simples: após o WHERE são feitas afirmações ou testes lógicos de verdadeiro-ou-falso. Apenas as tuplas (linhas da tabela) que satisfazem a condição (retornam verdadeiro) são selecionadas.

Os tópicos que seguem possuem mais exemplos a fim de elucidar este conceito.

Filtros com Operadores de Comparação Simples (=, >, <, <>)

Conforme supracitado, as condições dispostas na cláusula WHERE retornam sempre algum valor booleano, de modo que evidentemente operadores de comparação são utilizados com elevada frequência.

São eles:

- **Igualdade** (=): Retorna verdadeiro para as tuplas onde o valor da coluna de comparação for idêntico ao valor de comparação. Exemplo:

```
SELECT * FROM NOTAS WHERE ALUNO_ID=1;
SELECT NOME "Alunos Nota 10!" FROM NOTA WHERE NOTA=10;
SELECT NOME "Alunos em G2" FROM NOTA WHERE NOTA=6;
```

- **Maior** (>): Retorna verdadeiro para as tuplas onde o valor da coluna de comparação for maior ao valor de comparação. Exemplo:

```
SELECT NOME "Aprovados" FROM NOTA WHERE NOTA>7;
```

- **Menor** (<): Retorna verdadeiro para as tuplas onde o valor da coluna de comparação for menor ao valor de comparação. Exemplo:

```
SELECT NOME "Reprovados" FROM NOTA WHERE NOTA<5;
```

- **Diferente** (<>): Retorna verdadeiro para as tuplas onde o valor da coluna de comparação for maior e menor (ou seja, diferente) que valor de comparação. Diferente de outras linguagens, a sintaxe "!=" não é aceita para SQLs. Exemplo:

```
SELECT NOME "Aprovados ou Reprovados" FROM NOTA WHERE NOTA<>6;
```

Filtros com Operadores Lógicos (AND, OR, BETWEEN, NOT, IS)

Operadores lógicos são também elementos frequentes. São eles:

- **E (AND):** Utilizado para adicionar condições que devem ser satisfeitas em conjunto com a anterior (ou conjunto anterior). Exemplo:

```
SELECT NOME "Jovens Talentos" FROM NOTA
 WHERE NOTA=10 AND NASCIMENTO>'01/01/1990';
```

- **OU (OR):** Utilizado para adicionar condições opcionais. Ou seja, se ao menos uma das condições for satisfeita, a tupla é retornada pela consulta. Exemplo:

```
SELECT NOME "Jovens OU Talentos" FROM NOTA
 WHERE NOTA=10 OR NASCIMENTO>'01/01/1990';
```

- **É (IS):** Este é um operador diferente, utilizado para comparação de valores lógicos, em situações bastante específicas. No exemplo abaixo, estamos procurando o valor lógico "Nulo". Uma vez que o mesmo não possui valor em si, e apenas significado lógico, a cláusula **IS** é necessária. Exemplo:

```
SELECT NOME "Ausentes" FROM NOTA WHERE NOTA IS NULL;
```

- **Não/Negação (NOT)**: Operador de negação que tem efeito contrário sobre resultados lógico. Com o uso de NOT é possível implementar lógicas como OU EXCLUSIVO (XOR). Alguns exemplos:

```
SELECT NOME "Presentes" FROM NOTA WHERE NOTA IS NOT NULL;
SELECT NOME "Alunos Normais" FROM NOTA WHERE NOT NOTA=10
```

- **Intervalo (BETWEEN ... AND ...)**: Utilizado em cenários onde o valor pesquisado está em um intervalo. Exemplo:

```
SELECT NOME "Alunos em G2" FROM NOTA WHERE NOTA BETWEEN 5 AND 7;
```

Filtros com Operadores LIKE e IN List

Outros operadores bastante comuns são os operadores de "semelhança" e lista, conforme demonstrado abaixo

- **Semelhança (LIKE)**: Utilizado em cenários onde o se conhece parte do valor pesquisado. Neste caso, em geral usado para tipos string (char ou varchar2), logo o valor de comparação deve ser especificado entre aspas simples (string). Outro fator importante são os caracteres coringa. São Eles:

    - **Percentil (%):** Qualquer caractere, quantidade ilimitada.
    - **Underline (_)**: Apenas um caractere.

    Exemplos:

```
SELECT * FROM ALUNOS WHERE NOME LIKE 'AN%';
SELECT * FROM ALUNOS WHERE NOME LIKE '%N%';
SELECT * FROM ALUNOS WHERE NOME LIKE '%N';
SELECT * FROM ALUNOS WHERE NOME LIKE '%N_';
SELECT * FROM ALUNOS WHERE NOME LIKE 'A_';
```

- **Em (IN)**: Utilizado para especificar lista de valores. Exemplos:

```
SELECT * FROM NOTAS WHERE NOTA IN (9,10);
SELECT NOME, NOTA FROM NOTAS WHERE NOME IN ('ANA','ANDRE');
```

Limitando as Linhas Selecionadas (TOP-N)

Uma sintaxe classicamente utilizada para retornar apenas uma amostragem utiliza da cláusula ROWNUM (Número de linha), conforme exemplo abaixo, trazendo apenas as primeiras 5 notas (sem ordenação):

```
SELECT NOME, NOTA FROM NOTAS WHERE ROWNUM<6;
```

Um dos problemas clássicos desta implementação é o uso de ordenação, uma vez que a ordenação é realizada após o filtro, de modo que apenas estas linhas estão ordenadas, porém isto não garante que seja ordenação em relação ao conjunto completo de tuplas da tabela (ou origem dos dados, afinal).

Para resolução deste problema, na versão **Oracle 12c** (2013) foi lançada a sintaxe "FETCH FIRST <#> ROWS ONLY", conforme exemplo abaixo, que se mostra vantajoso em vários aspectos, inclusive performance.

```
SELECT NOME, NOTA FROM NOTAS WHERE FETCH FIRST 5 ROWS ONLY;
```

Existem outras maneiras de obter-se ordenação em TOP-N, porém estas não serão abordadas neste módulo devido densidade de implementação. De modo a citá-las, contudo: Inline View com ROWNUM, WITH Clause com ROWNUM, e funções: RANK, DENSE_RANK, ROW_NUMBER, PERCENT_RANK, NTILE.

Agrupamento (Cláusula GROUP BY)

Conforme já mencionado, o uso da cláusula GROUP BY é geralmente feito em situações onde existem funções de agregação na projeção e tenciona-se realizar o agrupamento por alguma outra propriedade também na projeção. Por exemplo, quantos alunos nascidos em cada ano:

```
SELECT COUNT(*) "Alunos", ANO_NASCIMENTO FROM ALUNOS
  GROUP BY ANO_NASCIMENTO;
```

Cabe ressaltar que todos as colunas ou valores projetados que não estiverem sofrendo operação de agregação devem ser listados na cláusula GROUP BY.

Alguns exemplos:

```
SELECT COUNT(*), JOB_ID FROM HR.EMPLOYEES GROUP BY JOB_ID;
SELECT SUM(SALARY), JOB_ID FROM HR.EMPLOYEES GROUP BY JOB_ID;
SELECT COUNT(*),REGION_ID FROM HR.COUNTRIES GROUP BY REGION_ID;
```

Ordenação (Cláusula ORDER BY)

De forma semelhante, a cláusula ORDER BY é utilizada para ordenação dos resultados com base nas colunas ou propriedades da projeção. Por definição a ordenação é feita de modo ascendente (crescente), mas pode ser realizado de modo descendente (decrescente) se utilizada a cláusula DESC. De modo complementar, a cláusula ASC pode ser utilizada para forçar a ordenação ascendente.

Existem algumas opções de valores que podem ser utilizados para ordenação são eles:

- Por Nome da Coluna:

```
SELECT * FROM NOTAS ORDER BY NOTA;
SELECT * FROM NOTAS ORDER BY NOTA DESC;
SELECT * FROM NOTAS ORDER BY NOTA ASC;
```

- Por Alias:

```
SELECT NOME, IDADE PRIMAVERAS FROM ALUNOS ORDER BY PRIMAVERAS;
```

- Por Número da Coluna:

```
SELECT NOME, NOTA FROM NOTAS ORDER BY 2;
```

Funções Numéricas

De maneira similar a funções de agregação, existem algumas funções especificamente utilizadas sobre valores numéricos cuja menção é importante. Essencialmente, funções de arredondamento ou truncamento são sobretudo importantes. São elas:

- **Truncamento (TRUNC)**: Função que, sem número de casas decimais especificadas trunca o valor informado números para um número inteiro. Conforme o exemplo abaixo, realizando truncamento do número 10.127:

  ```
  TRUNC(10.127) = 10
  ```

  Caso o número de casas decimais seja especificado, estas são respeitadas, mas o truncamento ainda assim é feito.

  ```
  TRUNC(10.127,2) = 10.12
  ```

- **Arredondamento (ROUND)**: Função que, sem número de casas decimais especificadas arredonda o valor informado números para um número inteiro. Conforme o exemplo abaixo, realizando arredondamento do número 10.127:

  ```
  ROUND(10.127,2) = 10
  ```

O mecanismo é semelhante ao de truncamento. Neste caso porém, não é realizado truncamento simples, como demonstrado no exemplo a seguir:

```
ROUND(10.127,2) = 10.13
```

Ainda no tocante a funções numéricas, todas as funções de agregação citadas anteriormente também podem ser classificadas nesta categoria, como por exemplo: AVG, MAX, MIN, MEDIANA, SUM, SIN, COS, entre outros.

Funções em Strings

De maneira semelhante, existe uma porção de funções aplicáveis sobre strings. As principais e mais comumente usadas são:

- **SUBSTR(String, POSICÃO INICIAL[, QUANTAS POSIÇÕES])**: Utilizada para obtenção de substring, retornando a substring.
- **INSTR(String, Substring [,POSICÃO INICIAL [,APARIÇÃO])**: Utilizada para obtenção da posição de uma substring na String.
- **CHR(ID TABELA ASCII)**: Conversão de ID na tabela ASCII para string.
- **ASCII(String)**: Obtenção de ID da tabela ASCII relacionado a uma string.
- **REPLACE(String, Substring [,Substituição])**: Substituição de uma substring por outra em uma string.
- **UPPER(String)**: Conversão para maiúsculas.
- **LOWER(String)**: Conversão para minúsculas.
- **INITCAP(String)**: Conversão para Capital (primeira maiúscula).
- **TRIM(String)**: Remoção de espaços vazios.

Existe uma porção de outras funções que podem ser utilizadas em casos bastante específicos. São elas:

ASCII	INSTR4	REGEXP_INSTR
ASCIISTR	INSTRB	REGEXP_REPLACE
CHR	INSTRC	REGEXP_SUBSTR
COMPOSE	LENGTH	REPLACE
CONCAT	LENGTH2	RPAD
Concat with ||	LENGTH4	RTRIM
CONVERT	LENGTHB	SOUNDEX
DECOMPOSE	LENGTHC	SUBSTR
DUMP	LOWER	TRANSLATE
INITCAP	LPAD	TRIM
INSTR	LTRIM	UPPER
INSTR2	NCHR	VSIZE

Funções Máscaras e de Conversão de Tipos de Dados

De modo semelhante existem algumas funções podem ser utilizadas para apresentação ou conversão de tipos de dados. Uma das conversões mais comuns é entre string e campos do tipo data, tanto para que estes possam ser armazenados (TO_DATE), quando para que possam ser lidos adequadamente (TO_CHAR). Exemplos:

```
TO_DATE('10/02/2020','dd/mm/yyyy')

TO_CHAR(sysdate,'dd/mm/yyyy hh24')
```

Existe uma extensa variedade de opções para máscaras durante a conversão entre tipos DATE e CHAR, conforme a tabela abaixo:

Parameter	Explanation
YEAR	Year, spelled out
YYYY	4-digit year
YYY YY Y	Last 3, 2, or 1 digit(s) of year.
IYY IY I	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	4-digit year based on the ISO standard
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).
MM	Month (01-12; JAN = 01).
MON	Abbreviated name of month.
MONTH	Name of month, padded with blanks to length of 9 characters.
RM	Roman numeral month (I-XII; JAN = I).
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
D	Day of week (1-7).
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DY	Abbreviated name of day.
J	Julian day; the number of days since January 1, 4712 BC.
HH	Hour of day (1-12).
HH12	Hour of day (1-12).
HH24	Hour of day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
FF	Fractional seconds.

Do mesmo modo, outras conversões podem ser realizadas, como por exemplo:

```
TO_NUMBER('01')

CAST('ABC' AS CLOB)
```

A tabela a seguir apresenta algumas informações considerando compatibilidade para conversões entre tipos de dados:

TO	FROM					
	char, varchar2	number	datetime / interval	raw	rowid, urowid	nchar, nvarchar2
char, varchar2	X	X	X	X	X	
number	X	X				
datetime / interval	X		X			
raw	X			X		
rowid, urowid	X				X	
nchar, nvarchar2		X	X	X	X	X

Consultas Condicionais (CASE .. WHEN .. THEN .. ELSE..)

De modo a agregar um pouco mais de complexidade a construção, é possível realizar consultas com valores condicionais e trazer diferentes resultados baseados nestes. A sintaxe, descrita abaixo, prevê que, para uma expressão "e" se a condição 1 "c1" for satisfeita, o resultado "r1" é apresentado, se a condição "c2" for satisfeita, o resultado "r2", e assim sucessivamente para "cn" será retornado resultado "rn". Caso nenhuma das condições sejam satisfeitas (ELSE) o resultado "r_else" é apresentado.

```
CASE e
     WHEN c1 THEN
          r1
     WHEN c2 THEN
          r2
     WHEN cn THEN
          rn
     [ ELSE r_else ]
END
```

O exemplo abaixo apresenta o mecanismo de um modo mais prático, retornando o "STATUS SOCIAL" com base no salário do indivíduo:

```
SELECT first_name||' '||last_name vivente, CASE
     WHEN SALARY>15000 THEN 'Magnata'
     WHEN SALARY<500 THEN 'Estagiário'
     ELSE 'Trabalha pra Viver' STATUS_SOCIAL
     END
FROM HR.EMPLOYEES WHERE ROWNUM<11;
```

Expressões Regulares

O tópico que talvez tenha maior complexidade a ser apresentado neste módulo trata do uso de expressões regulares em SQL. Expressões regulares são ferramentas muito poderosas para especificação de máscaras de caracteres (como CPF, email, entre outros) em strings. Embora sua definição seja computacionalmente antiga e comum, sua compreensão pode não ser trivial para usuários de diferentes naturezas, que frequentemente são usuários de SQL devido sua linguagem tradicionalmente natural (inglês estruturado).

As principais funções SQL que utilizam do padrão de expressões regulares são:

- **REGEXP_LIKE (String, PADRÃO)**: Retorna TRUE ou FALSE.
- **REGEXP_REPLACE(String, PADRÃO, SUBSTITUIR)**: Retorna string substituída.
- **REGEXP_SUBSTR(String, Substr[,flag])**: Retorna TRUE ou FALSE.
- **REGEXP_INSTR(String, Substr)**: Retorna a posição da sring encontrada
- **REGEXP_COUNT(String, PADRÃO)**: Retorna o número de ocorrências do padrão na string.

Ao escrever expressões regulares, utiliza-se notação comum para posicionamento de cursores ou estrutura de expressão, conforme a tabela abaixo.

Caracter	Função/Significado
.	Qualquer caracter
[ ]	Qualquer caracter definido
[^]	Nenhum dos caracteres definidos
*	Nenhum ou vários caracteres
^	Verifica se o padrão está no início na linha/string a verificar
$	Verifica se o padrão está no fim na linha/string a verificar

Alguns exemplos:

```
REGEXP_LIKE(first_name, '^Ste(v|ph)en$')

REGEXP_INSTR(email, '\w+@\w+(\.\w+)+')
```

Em um caso prático: Uma consulta para inversão de nome e sobrenome:

```
SELECT nome “Nome Completo”,
  REGEXP_REPLACE(nome, '^(\S+)\s(\S+)\s(\S+)$', '\3, \1 \2’)
        AS ”sobrenome, nome"
    FROM PESSOAS;
```

Caracteres posix também são aceitos, conforme tabela:

POSIX	Non-standard	Perl	ASCII	Description
[:alnum:]			[A-Za-z0-9]	Alphanumeric characters
	[:word:]	\w	[A-Za-z0-9_]	Alphanumeric characters plus "_"
		\W	[^A-Za-z0-9_]	Non-word characters
[:alpha:]			[A-Za-z]	Alphabetic characters
[:blank:]			[\t]	Space and tab
[:cntrl:]			[\x00-\x1F\x7F]	Control characters
[:digit:]		\d	[0-9]	Digits
		\D	[^0-9]	Non-digits
[:graph:]			[\x21-\x7E]	Visible characters
[:lower:]			[a-z]	Lowercase letters
[:print:]			[\x20-\x7E]	Visible characters and spaces
[:punct:]			[-!"#$%&'()*+,./:;<=>?@[\\\]^_`{|}~]	Punctuation characters
[:space:]		\s	[\t\r\n\v\f]	Whitespace characters
		\S	[^ \t\r\n\v\f]	Non-whitespace characters
[:upper:]			[A-Z]	Uppercase letters
[:xdigit:]			[A-Fa-f0-9]	Hexadecimal digits

4. Outros Objetos e Scripts SQL

Neste capítulo, abordaremos alguns outros tipos de objetos bastante relevantes para a construção de um banco de dados, seja do ponto de vista de consistência e integridade, seja do ponto de vista estrutural e de apresentação dos dados.

Constraints

A principal maneira que o Oracle (assim como outros bancos de dados) possui para garantir a integridade dos dados e a consistência do modelo é através do uso de constraints (ou restrições). Existem essencialmente 2 tipos de restrições:

- **Não Nulo (NOT NULL)**: A tupla a qual a restrição se aplica não pode possuir valor nulo.
- **Unicidade (UNIQUE)**: A tupla a qual a restrição se aplica deve ser único em toda tabela.

Que são utilizados no modelo em 4 situações:

- **Chaves Únicas (Unique Keys)**: Apenas restrição UNIQUE.
- **Chaves Primárias (Primary Keys)**: Usado para identificadores de tabela, obrigatoriamente UNIQUE e NOT NULL.
- **Chaves Estrangeiras (Foreign Keys)**: Usado para referenciar relação com tabelas estrangeiras. Pode ser nula e não necessariamente precisa ser única, por definição.
- **Checagens (Checks)**: Utilizado para aplicar restrições do modelo relacionados ao conteúdo de colunas ou relação entre elas dentro da própria tabela. Possui a seguinte sintaxe:

```
CONSTRAINT constraint_name CHECK (condition) [DISABLE]
```

O exemplo abaixo implementa uma chave primária no ID do fornecedor, uma restrição de não-nulo no nome do fornecedor e um check adicional força com que os fornecedores sejam inseridos apenas em maiúsculas (através da condição usando função upper, vista anteriormente).

```
CREATE TABLE suppliers (
supplier_id numeric(4) PRIMARY KEY,
supplier_name varchar2(50) NOT NULL,
CONSTRAINT check_supplier_name CHECK (supplier_name = upper(supplier_name)));
```

É possível ainda alterar a tabela adicionando uma nova constraint, permitindo ainda que apenas os supplies IBM, NVIDIA e Microsoft sejam inseridos.

```
ALTER TABLE suppliers
ADD CONSTRAINT check_supplier_name CHECK (supplier_name IN ('IBM', 'Microsoft', 'NVIDIA'));
```

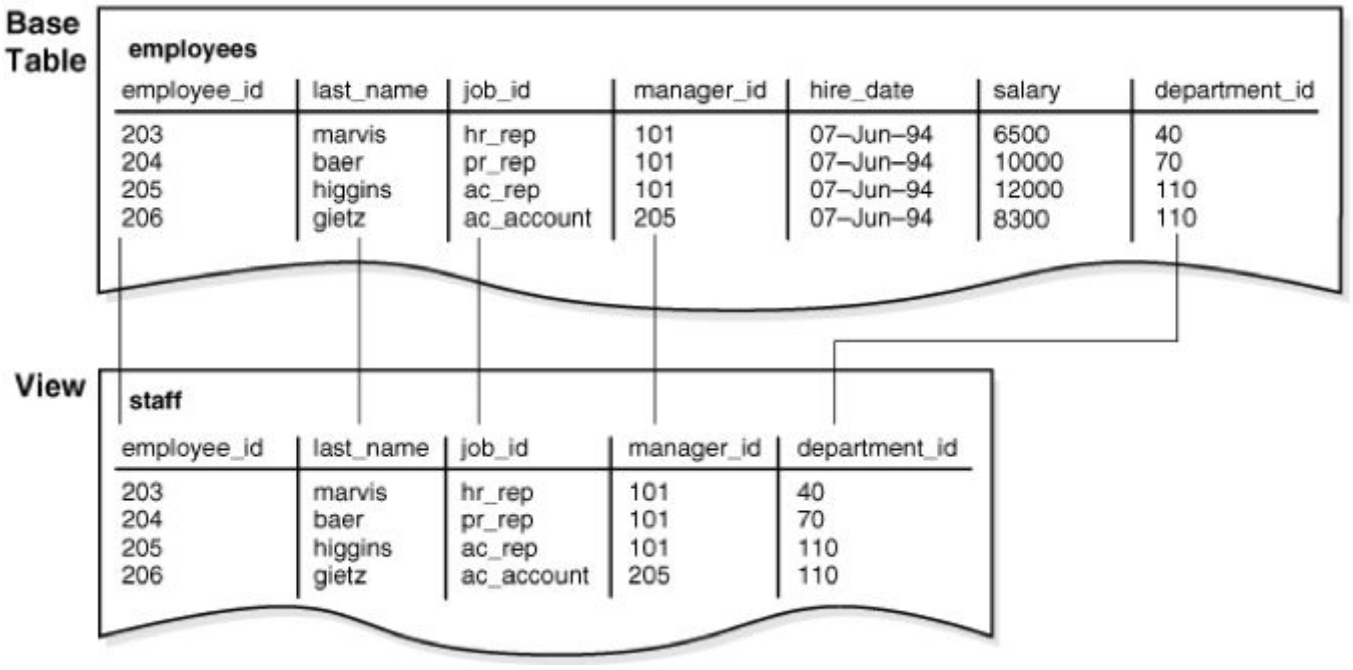
Adicionalmente, cabe ressaltar que as constraints podem ser removidas posteriormente ou simplesmente habilitadas ou desabilitadas temporariamente, conforme sintaxe abaixo.

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;

ALTER TABLE table_name DISABLE CONSTRAINT constraint_name;

ALTER TABLE table_name ENABLE CONSTRAINT constraint_name;
```

Visões (Views)

Outro tipo de objeto relevante, especialmente se considerando conceitos da orientação objeto, como abstração, é o uso de visões (ou views, do inglês). Uma view consiste, na prática, em um atalho para uma consulta em forma de objeto. Na prática, trata-se de um objeto com a consulta que nos é pretendida, muitas vezes a consulta realizada em algum relatório ou mesmo para exibição de informações em tela. Dessa forma, para alteração do conteúdo do relatório, basta-se alterar a consulta da view. São também um dispositivo de segurança, de modo que informações sensíveis podem ser excluídas da view e permissões de acesso para usuários externos podem ser concedidas apenas a view em si.

As views executam a consulta subjacente todas as vezes que são selecionadas, de modo que os dados são sempre retornados em tempo-real das suas origens. A imagem abaixo ilustra o funcionamento de uma view:



A sintaxe da criação de uma view é bastante simples:

```
CREATE [OR REPLACE] VIEW <NOME> AS <SELECT>
```

Exemplo prático:

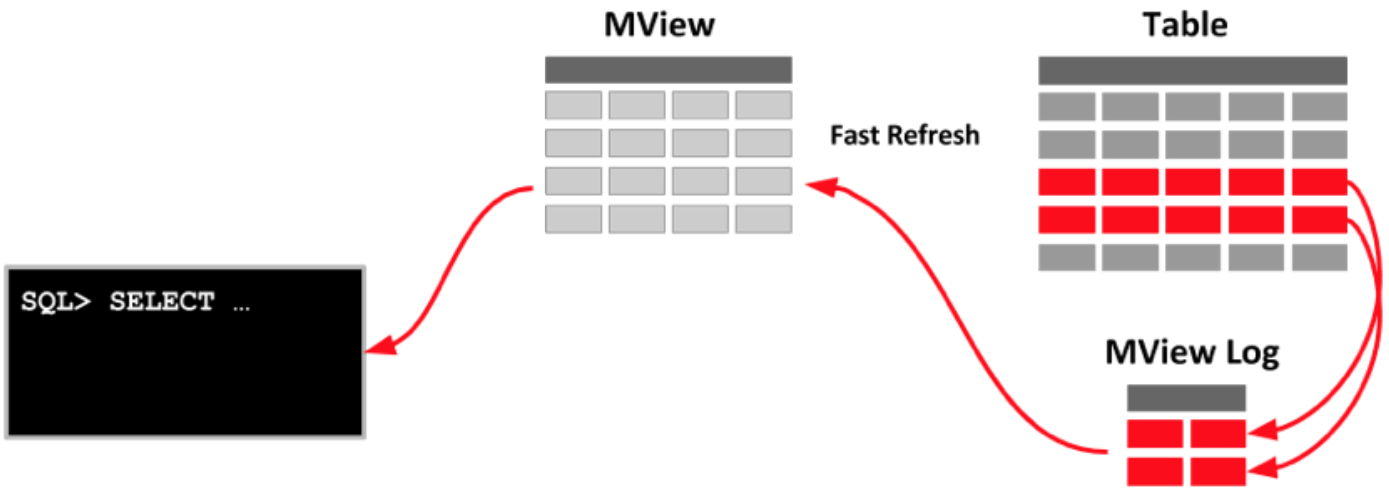
```
CREATE VIEW VW_ESTAGIARIOS AS
SELECT
  first_name,last_name
from
  hr.employees
where
 salary<500;
```

Visões Materializadas (Materialized Views)

As visões materializadas (Materialized Views, ou mviews) têm exatamente o mesmo comportamento lógico, a diferença está na estrutura física. Uma vez que as views muitas vezes podem conter consultas complexas com elevado tempo de execução, é possível "materializar" estes resultados em forma de tabela física. Deste modo o acesso a estes dados fica simplificado e potencialmente mais performático. O problema, porém, passa a estar relacionado a instantaneidade destes dados, uma vez que armazenados em uma tabela auxiliar e não sendo consultados diretamente da tabela de origem, diferenças podem ocorrer.

A solução para este problema é simples: atualizações da mview com base nas tabelas de origem podem ser feitas periodicamente ou de acordo com gatilhos, como alterações nas tabelas de origem. É possível, sob algumas condições, a utilização de mview logs, apenas contendo as alterações realizadas nas tabelas de origem para que se possa realizar uma atualização rápida (fast refresh).

A imagem abaixo ilustra este mecanismo.



Imagine que você tem uma consulta que demora 5 minutos porém vital ao seu sistema de modo que dezenas de pessoas executam a mesma consulta constantemente. Existe uma tolerância e os dados podem estar até 15 minutos atrasados. Ao transformar esta consulta em uma mview a consulta demorada é realizada apenas uma vez e as demais demorarão potencialmente milésimos de segundos.

A sintaxe para a criação de uma mview é muito semelhante a criação de uma view simples. Conforme:

```
CREATE [OR REPLACE] MATERIALIZED VIEW <NOME> AS <SELECT>
```

Ao final da consulta, pode ser adicionalmente informado modos de refresh, gatilhos e até intervalos. Uma sintaxe simplificada para tal seria:

```
REFRESH [FAST | COMPLETE | FORCE ]  ON [COMMIT | DEMAND ]
```

Um exemplo prático para a criação de uma visão materializada, com a mesma idéia do exemplo anterior, seria:

```
CREATE MATERIALIZED VIEW MVW_ESTAGIARIOS AS
SELECT
  first_name,last_name
from
  hr.employees
where
 salary<500;
```

E o comando para forçar a atualização da mesma, seria:

```
EXEC DBMS_MVIEW.refresh('MVW_ESTAGIARIOS');
```

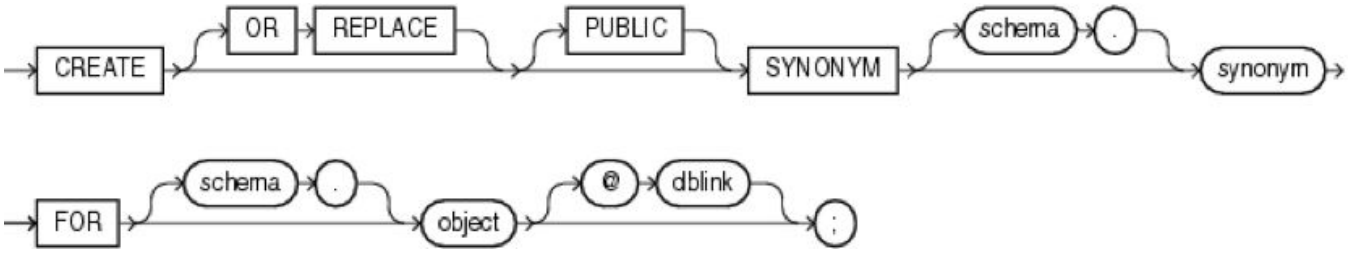
Sinônimos (Synonyms)

Outro tipo de objeto cuja utilização é bastante interessante do ponto de vista de abstração e qualidade de código é o uso de sinônimos. Eles atuam de forma parecida porém mais simples que as views. Enquanto as views armazenam uma consulta, o sinônimo é apenas um ponteiro para outro objeto. Um dos grandes benefícios é abstrair o uso de prefixação de objetos, ou menos de criar uma camada de abstração na aplicação de modo que possamos manter versões diferentes dos objetos reais e realizar chaveamentos através do uso de sinônimos.

Os sinônimos podem ser de duas categorias:

- **Públicos**: Válidos para qualquer schema/usuário no banco de dados.
- **Privados**: Válidos apenas para o schema que o possui.

A sintaxe para a criação de sinônimos é bastante simples, conforme diagrama abaixo:



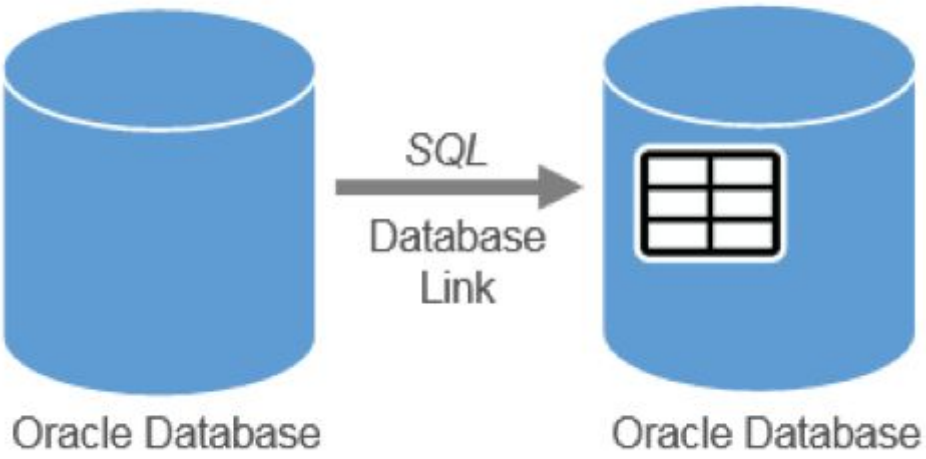
Alguns exemplos:

```
CREATE SYNONYM STAGS FOR MVW_ESTAGIARIOS;

CREATE PUBLIC SYNONYM ESCRITORIOS FOR HR.LOCATIONS;
```

Database Links

Ainda na linha de conectividade e abstração de acesso o Oracle possui outro tipo de objeto chamado Database Link. Como o nome supõe, este tipo de objeto pode ser utilizado para conectar 2 bancos de dados diferentes (diferentes servidores, datacenter, localizações. O uso deste tipo de objeto torna fácil a simples execução se sqls em outro database diretamente da camada de banco de dados, sem que aplicações necessitem conectar em ambos e realizar tratamentos de dados em nível de aplicação, podendo ocasionar em overload de rede e baixo desempenho.

A imagem abaixo ilustra de maneira bastante simplificada este mecanismo.



Cabe ressaltar ainda, que é possível a criação de database links para bancos de dados de outras tecnologias, como SQLServer, PostgreSQL, MySQL, IBM, Teradata e até sistemas de filas, como o MQSeries da IBM. Para tal, é utilizado componente adicional a ser configurado a nível de servidor, chamado Oracle Heterogeneous Services (OHS), também conhecido em versões mais antigas como Database Gateway.

A sintaxe para criação de database links é um pouco mais complexa, mas ainda assim bastante simples, conforme abaixo. No campo descritor de conexão é possível utilizar descrição completa da conexão, ou apenas alias de TNSNAMES, a ser estudado no segundo módulo desta formação.

```
CREATE DATABASE LINK <NOME>
CONNECT TO <USER_REMOTO> IDENTIFIED BY "<senha>"
USING '<descritor de conexão>';
```

A utilização de database links é feita através de sufixo com @ e o nome do database link, conforme:

```
SELECT <COLUNAS> FROM <OBJ>@DBLINK;
```

Um exemplo prático:

```
CREATE DATABASE LINK EXEMPLO
CONNECT TO HR IDENTIFIED BY "target"
USING 'PDB1';

SELECT * FROM HR.EMPLOYEES@EXEMPLO;
```

Um aspecto adicional interessante é a possibilidade de combinar sinônimos com database links, como por exemplo:

```
CREATE PUBLIC SYNONYM ESCRITORIOS FOR HR.LOCATIONS@EXEMPLO;

SELECT * FROM ESCRITORIOS;
```

Sequencias (Sequences)

As sequências, por sua vez, são tipos de objetos tradicionalmente peculiares ao Oracle. Elas tem uma função bastante específica e de certa forma simples: Retornar o próximo valor de uma sequência. Este tipo de objeto, porém, é especializado a este fim e, portanto, é capaz de realizar esta ação de maneira extremamente efetiva e performática, evitando quaisquer contenções nesta ação, se adequadamente configuradas.

Abaixo um exemplo de sintaxe de criação de uma sequence no Oracle. Como é possível perceber, podemos especificar, entre outros:

```
CREATE SEQUENCE schema_name.sequence_name
[INCREMENT BY interval]
[START WITH first_number]
[MAXVALUE max_value | NOMAXVALUE]
[MINVALUE min_value | NOMINVALUE]
[CYCLE | NOCYCLE]
[CACHE cache_size | NOCACHE]
[ORDER | NOORDER];
```

- **Interval**: Intervalo de incremento. O default é de 1 em 1.
- **Start with**: Número de início. O default é 1.
- **Max Value**: Valor máximo, caso se aplique ao caso. O Default é sem valor máximo.
- **Min Value**: Valor mínimo, caso se aplique ao caso. O Default é sem valor mínimo.
- **Cycle/NoCycle**: Caso a sequence deva reiniciar após atingir o valor máximo ou não.
- **Cache/NoCache**: Em caso de RAC (Cluster, a ser discutido em módulos futuros), se existe algum cache de números por instância.
- O**rder/NoOrder**: Diretamente relacionado ao anterior, especifica se os números a serem informados necessitam estar ordenados, em caso de uso de cache.

Auto-incremento Oracle (12c+)

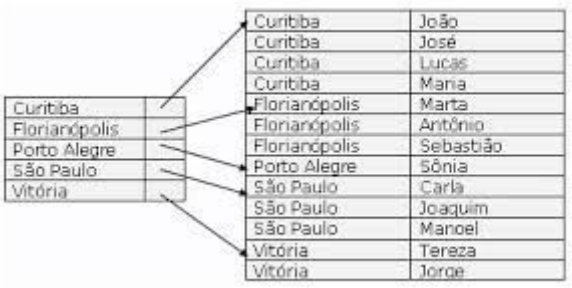
Um dos grandes motivos para o uso de sequences é o fato de o Oracle não possuir autoincremento. Fato este que não é mais verdade a partir da versão 12c. O exemplo abaixo ilustra como é feita a especificação de auto incremento na criação de uma tabela (12c+), utilizando a mesma definição de parâmetros que as sequences tradicionais:

```
     create table t1 (
         c1 NUMBER GENERATED ALWAYS as IDENTITY(START with 1
INCREMENT by 1),
         c2 VARCHAR2(10)
         );
```

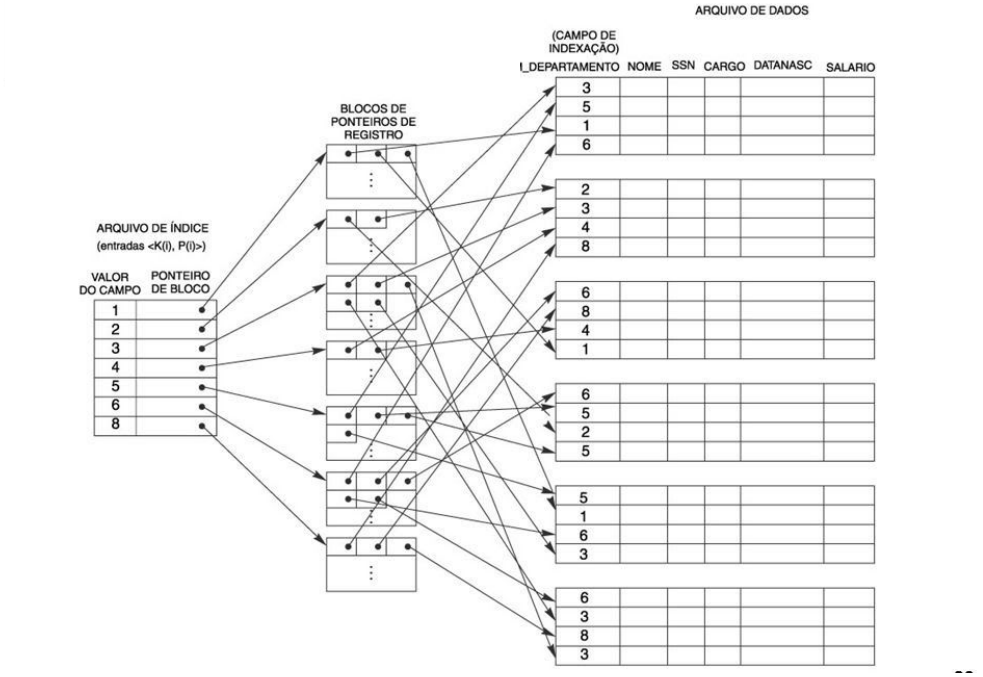
Índices (indexes) - Básico

Embora tenham uma finalidade simples, devido a importância deste tipo de recurso, especialmente em um banco de dados de ponta como Oracle Database, os índices adquiriram enorme variedade no que diz respeito às suas opções de configuração, tornando o tópico bastante complexo. Existem vários livros tratando exclusivamente de índices.

Como o próprio nome indica, índices são como sumários, listas com ponteiros para repositórios com maiores detalhes, o conceito clássico de ponteiros. Na imagem abaixo podemos observar uma ilustração simplificada de como um índice sobre as cidades de uma tabela de pessoas se comportaria:



Os índices, contudo, podem possuir vários níveis de ponteiros, conforme abaixo:



Estes e outros mecanismos existem para garantir o melhor desempenho de índices para diferentes situações e circunstâncias. De modo geral, no entanto, é possível simplificar a sintaxe:

```
CREATE INDEX <nome> ON <TABELA>(<COLUNA[S]>);
```

Construção de Scripts SQL (PROMPT, &, ACCEPT, DEFINE, SPOOL e Tratamento de Erros)

Um aspecto interessante relacionado a operação de um banco de dados é a criação de scripts sql. Scripts geralmente são utilizados para criação de estruturas (DDLs, DCLs) e população de dados (DMLs). Geralmente mudanças em produção são executadas a partir de scripts para atender fatores como repetibilidade, entre outros. Em miúdos, um script trata-se de um arquivo texto com os comandos SQLs a serem executados, geralmente salvos em extensão ".sql".

Para que tal, os clientes oficiais Oracle (SQL*Plus, SQLcl, SQLDeveloper) e a maior parte dos clientes de mercado (TOAD, PL/SQL Developer) suportaram algumas instruções e padrões como:

- **PROMPT**: Utilizado para escrever mensagem em tela.
- **SPOOL**: Gera arquivo de log com conteúdo apresentado em tela.
- **SPOOL OFF**: Desativa trace.
- **&:** Utilizado para especificar variável em tempo de execução.
- **DEFINE**: Utilizado para especificar no script o valor para uma variável antecipadamente.
- **ACCEPT**: Aceita valor em variável.
- **ACCEPT - HIDE**: Aceita valor mas mascara o valor apresentado em tela. Ex: senhas.
- **ACCEPT - DEFAULT**: Caso nenhum valor seja informado, possui valor default.
- **PAUSE**: Aguarda confirmação antes de dar seguimento à execução.
- **WHENEVER SQLERROR**: Tratamento de Erro. Em caso de erro se pode sair da execução (EXIT), desfazer transações (ROLLBACK) ou continuar a execução (CONTINUE).

Existem alguns comandos especificamente voltados a formatação do output. São eles:

- **FEEDBACK**: Ativa ou desativa o retorno de comandos SQL.
- **HEADER**: Ativa ou desativa apresentação de cabeçalho (nomes das colunas).
- **LINESIZE**: Tamanho da linha.
- **PAGES**: Quantidade de retornos por página.

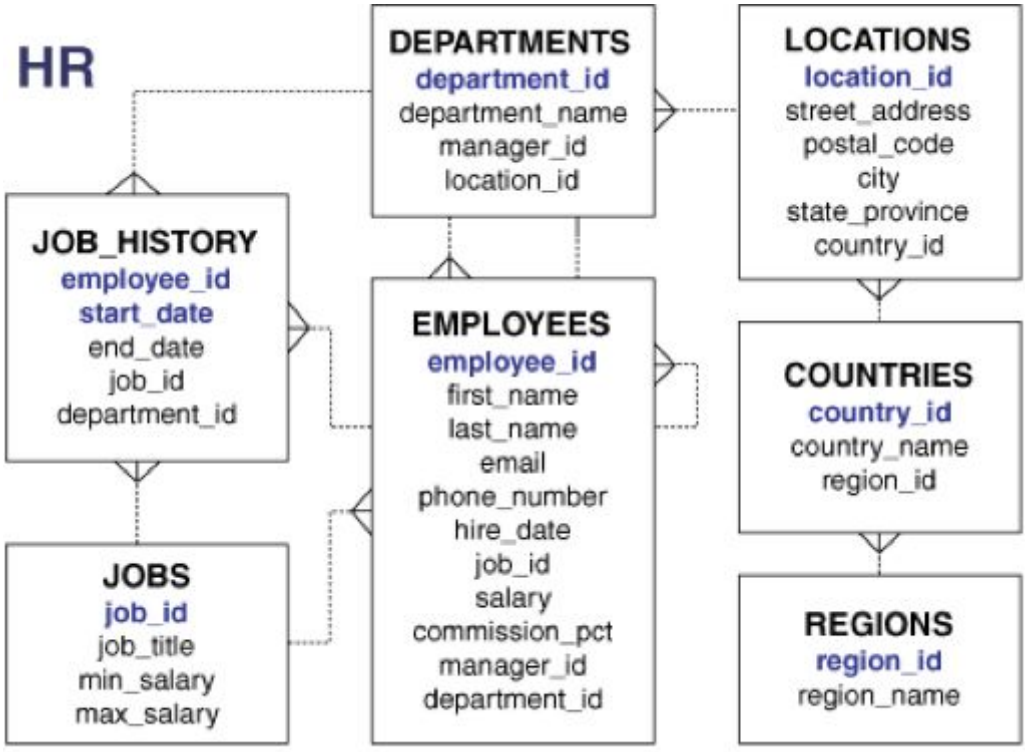
Abaixo exemplo de script contemplando todos estes itens.

```
SPOOL /temp/log_target.log
WHENEVER SQLERROR EXIT;
select 10-&desconto valor_final from dual;
select 20-&desconto valor_final from dual;
DEFINE DESCONTO=5
select 10-&desconto valor_final from dual;
ACCEPT DESCONTO NUMBER PROMPT "Desconto:"
select 10-&desconto valor_final from dual;
ACCEPT DESCONTO NUMBER PROMPT "Desconto:" HIDE
select 10-&desconto valor_final from dual;
ACCEPT DESCONTO NUMBER PROMPT "Desconto:" DEFAULT 5
select 10-&desconto valor_final from dual;
PAUSE
SPOOL OFF;
```

5. SQL Avançado

Muito frequentemente contudo, especialmente se o modelo de dados segue adequadamente as formas normais, requisitos de aplicação farão com que seja necessário obter dados de mais de uma origem ou tabela. É a partir deste ponto que a complexidade na escrita de consultas SQL passa a aumentar, devido a variedade de opções e técnicas para obter e usar estes dados.

Neste contexto, existem diversas técnicas que podem ser utilizadas, as quais abordaremos nos tópicos a seguir. Para próximos tópicos, os exemplos apresentados serão majoritariamente baseados nos schema HR, um dos schemas de exemplo fornecidos pela Oracle, me mencionado no primeiro capítulo desta apostila. Abaixo Modelo-ER do schema HR.

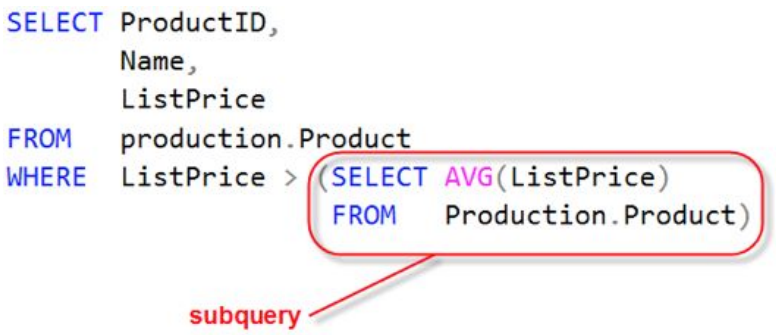


Sub Consultas

Sobretudo em situações onde buscamos informações de outras tabelas para serem usados como parâmetros ou listas de comparação. Nestes casos, em geral, as sub consultas possuem como valor de retorno:

- Um Valor (numérico, string, data, etc)
- Uma Lista de Valores
- Verdadeiro ou Falso (para cláusulas específicas, como o EXISTS, a serem discutidas a seguir)

O exemplo abaixo ilustra este mecanismo, onde o preço médio é obtido através de uma subconsulta para então ser utilizado como filtro na consulta inicial.



Outro exemplo, utilizando o Schema HR: Quem ganha mais que William Gietz?

```
SELECT first_name,last_name,salary
FROM hr.employees where salary > ( SELECT salary FROM hr.employees
                                   WHERE first_name='William' and
                                         last_name='Gietz');
```

Operações de Conjuntos com Mesmas Propriedades

De forma semelhante porém complementar, é possível que se queria realizar operações com conjuntos de dados de idêntica configuração ou propriedades. Estas operações têm efeito sobre o conjunto ou result set. São elas:

- **UNION**: União de ambos os conjuntos com remoção de duplicados
- **MINUS**: Conjunto A menos Conjunto B
- **INTERSECT**: Apenas elementos que estão simultaneamente em conjunto A e Conjunto B.
- **UNION ALL**: Conjunto A e Conjunto B sem remoção de duplicados.

A sintaxe para o uso destas operações é:

```
<SELECT1> [OPERADOR] <SELECT2>
```

Exemplo: Retorne todos os funcionários que não são estagiários.

```
SELECT first_name, last_name from HR.EMPLOYEES
MINUS
SELECT first_name, last_name from MVW_ESTAGICAS;
```

Da mesma forma como as demais construções SQL, parêntesis podem e devem ser aplicados para construções com mais de duas operações de conjuntos. Dessa forma, mais de uma operação de conjunto pode ser utilizada se necessário.

Consultas Compostas

Operações de conjunto, contudo, necessitam que as propriedades de ambos os conjuntos tenham a mesma configuração, ou seja, que os resultsets tenham as mesmas colunas. No entanto, os problemas normalmente são mais complexos que isso. Como proceder se desejamos resultados de dois conjuntos com configurações distintas?

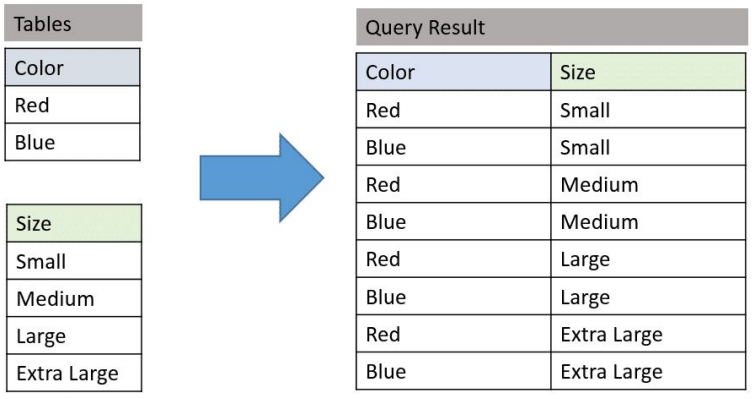
Neste caso, é possível simplesmente prefixar as colunas seja com o nome da tabela ou alias da tabela. Este é um requerimento para consultas compostas, a fim de eliminar ambiguidades (de qual tabela desejo retornar cada coluna).

Exemplo: Qual o primeiro nome e a função de cada funcionário?

```
SELECT e.first_name, j.job_title
FROM hr.employees e, hr.jobs j;
```

Produto Cartesiano

Este item tem possui um tópico separado dada a sua importância. A consulta acima não retornará o resultado desejado. Ou melhor, retornará o resultado desejado entre outras informações e não será possível distinguir qual o dado correto. A imagem abaixo ilustra o fenômeno de utilizando uma tabela de cores e outra de tamanhos.



Tables

Color
Red
Blue

Size
Small
Medium
Large
Extra Large

Query Result

Color	Size
Red	Small
Blue	Small
Red	Medium
Blue	Medium
Red	Large
Blue	Large
Red	Extra Large
Blue	Extra Large

Analize a consulta do item anterior. Ela retorna precisamente a combinação de todos os primeiros nomes com todas as funções cadastradas. Não há nenhum filtro ou especificação na consulta dizendo que se deseja APENAS a função de cada funcionário é desejada, como estabelece o problema inicial. Para tal, podemos recorrer ao modelo E-R do schema HR e observar que existe uma propriedade da tabela EMPLOYEES que permite identificar a função de cada empregado. Ela é, contudo, um ID para a tabela de Funções. A esta configuração chamados de relacionamento.

A consulta correta deve usar-se deste relacionamento no filtro para identificar apenas a função onde o ID é idêntico ao ID de cada funcionário. Conforme exemplo abaixo:

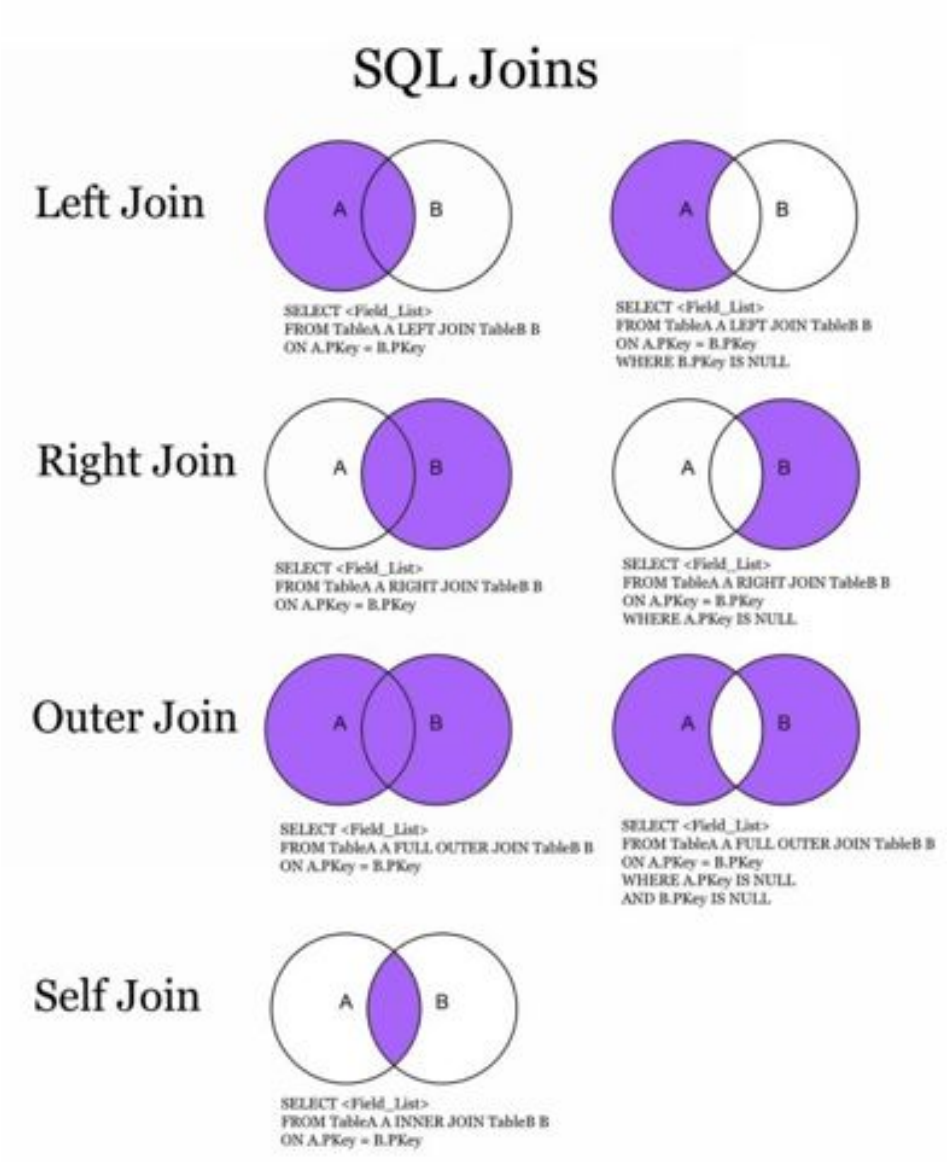
```
SELECT e.first_name, j.job_title
FROM hr.employees e, hr.jobs j
WHERE e.job_id=j.job_id;
```

SQL Joins

Como se pode observar na página anterior ao se referir em consultas compostas, é bastante provável que operações de conjuntos também sejam necessárias a fim de combinar informações de conjuntos distintos, porém com propriedades (colunas) diferentes. Para estes casos, temos os SQL Joins descritos formalmente no SQL ANSI.

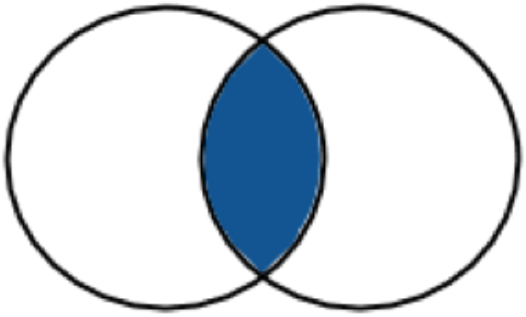
A ilustração abaixo identifica os principais joins com um exemplo simplificado. A seguir discutimos cada uma destas operações individualmente.

Cabe ressaltar que todas estas especificações ANSI possuem adicionalmente uma sintaxe própria do Oracle, conforme apresentado a seguir.



Join, Inner Join ou "Self Join"

O INNER JOIN ANSI, ou simplesmente JOIN, também conhecido como "Self Join", é o tipo de operação mais simples e com comportamento padrão, onde são retornados apenas os dados que tenham representatividade em ambos os conjuntos de dados. Ilustrativamente, os dados que possuem relação para ambos os conjuntos, conforme abaixo.



A sintaxe ANSI para esta operação é:

```
SELECT <PROJEÇÃO> FROM <TABELA1> JOIN <TABLE2>
ON <TABLE1>.<COLUNA> = <TABLE2>.<COLUNA>;
```

A parte em verde é necessária justamente para evitar o produto cartesiano e para que seja possível estabelecer qual a relação entre ambos os conjuntos.

Exemplo prático utilizando o schema HR: Quem é o chefe de cada departamento?

Exemplo sem a utilização da sintaxe JOIN:

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
FROM HR.DEPARTMENTS D, HR.EMPLOYEES E
WHERE D.MANAGER_ID = E.EMPLOYEE_ID
```

Exemplo utilizando a sintaxe INNER JOIN ANSI:

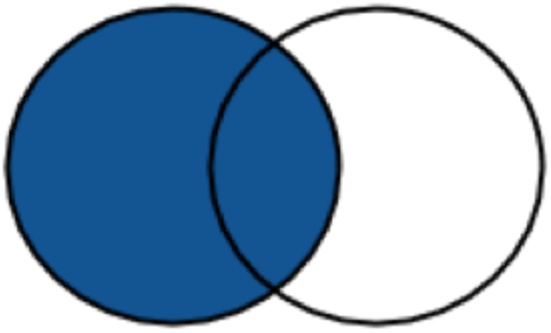
```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          INNER JOIN HR.EMPLOYEES E
     ON D.MANAGER_ID = E.EMPLOYEE_ID;
```

Exemplo utilizando a sintaxe JOIN Oracle:

```
SELECT D.DEPARTMENT_NAME, E.LAST_NAME
  FROM HR.DEPARTMENTS D
          JOIN HR.EMPLOYEES E
     ON D.MANAGER_ID = E.EMPLOYEE_ID
```

Left Outer Join

Por sua vez, o Left Join considera apenas o conjunto A, a esquerda ou primeiro apresentado na consulta, e apresenta os dados do conjunto B quando existem, mantendo a coluna como nula em caso de inexistência de relação. Conforme a ilustração abaixo.



A sintaxe ANSI para esta operação é:

```
SELECT <PROJEÇÃO> FROM <TABELA1> LEFT OUTER JOIN <TABLE2>
ON <TABLE1>.<COLUNA> = <TABLE2>.<COLUNA>;
```

Utilizando-se o mesmo exemplo: Quem é o chefe de cada departamento?

Exemplo utilizando a sintaxe LEFT OUTER JOIN ANSI:

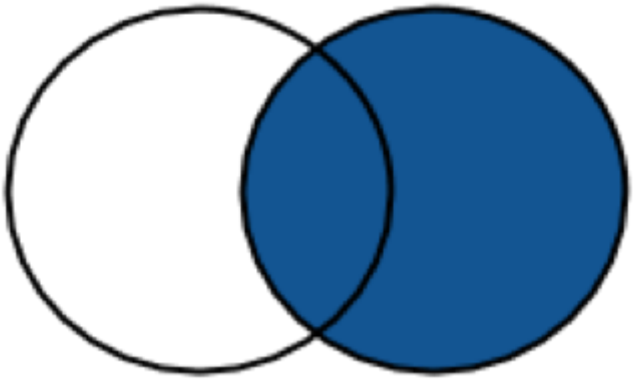
```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          LEFT OUTER JOIN HR.EMPLOYEES E
          ON D.MANAGER_ID = E.EMPLOYEE_ID
```

Exemplo utilizando a sintaxe LEFT OUTER JOIN Oracle:

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D, HR.EMPLOYEES E
  WHERE D.MANAGER_ID = E.EMPLOYEE_ID (+);
```

Right Outer Join

Já o Right Join consiste no oposto, considera apenas o conjunto B, a direita ou segundo apresentado na consulta, e apresenta os dados do conjunto A quando existem, mantendo a coluna como nula em caso de inexistência de relação. Conforme a ilustração abaixo.



A sintaxe ANSI para esta operação é:

```
SELECT <PROJEÇÃO> FROM <TABELA1> RIGHT OUTER JOIN <TABLE2>
ON <TABLE1>.<COLUNA> = <TABLE2>.<COLUNA>;
```

Para este exemplo, o problema precisa ser modificado semanticamente: Qual o departamento de qual cada um é chefe?
Obviamente sabemos quem nem todos são chefes de algum departamento).

Exemplo utilizando a sintaxe RIGHT OUTER JOIN ANSI:

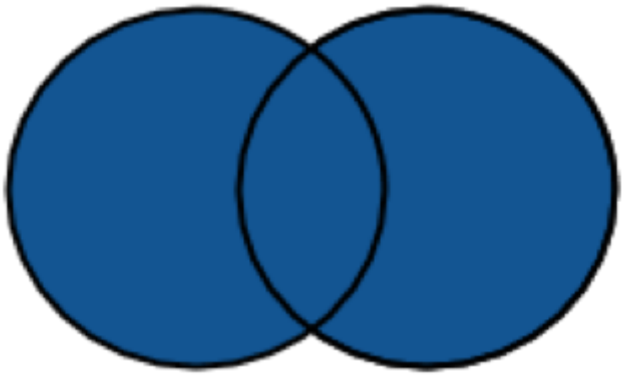
```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          RIGHT OUTER JOIN HR.EMPLOYEES E
     ON D.MANAGER_ID = E.EMPLOYEE_ID
```

Exemplo utilizando a sintaxe RIGHT OUTER JOIN Oracle:

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D, HR.EMPLOYEES E
  WHERE D.MANAGER_ID (+) = E.EMPLOYEE_ID;
```

Full Outer Join

De forma complementar, ao juntarmos ambas as operações acima (INNER + LEFT + RIGHT), eliminando-se as duplicadas, o resultado que obtemos é, de certa forma completo, apresentando todos os resultados de ambos os conjuntos, apresentando relações quando houver, mantendo a coluna como nula em caso de não correspondência, conforme ilustra a imagem abaixo.



A sintaxe ANSI para esta operação é:

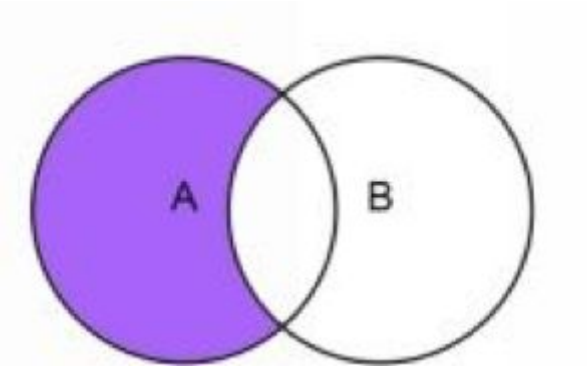
```
SELECT <PROJEÇÃO> FROM <TABELA1> FULL OUTER JOIN <TABLE2>
ON <TABLE1>.<COLUNA> = <TABLE2>.<COLUNA>;
```

Para este exemplo, o problema seria: Qual a relação entre departamentos e managers?

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          FULL OUTER JOIN HR.EMPLOYEES E
     ON D.MANAGER_ID = E.EMPLOYEE_ID
```

Left Anti Join

Retornando um passo atrás, é possível que desejemos um conjunto semelhante ao obtido através do LEFT OUTER JOIN, porém quem os dados que possuam relação. Ou seja, só aqueles do conjunto A que não possuam relação com o conjunto B, conforme ilustrado abaixo. A este tipo específico de Join convencionou-se chamar de ANTI JOIN, justamente pois os resultados apresentados são onde a relação NÃO é válida.



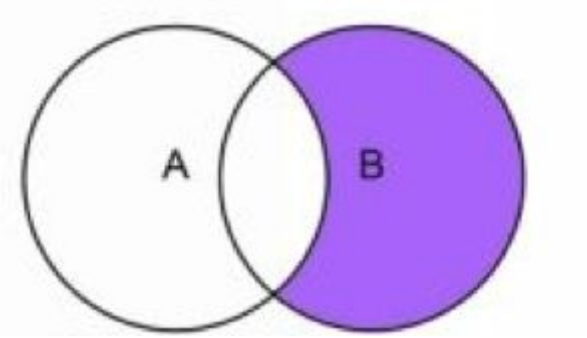
Este tipo de relação, contudo, não possui sintaxe própria, uma vez que deriva do LEFT OUTER JOIN, adicionando a cláusula de filtro para relação nula apenas. Neste caso um possível exemplo seria:

Quais os funcionários que não são chefes de nenhum departamento?

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          LEFT OUTER JOIN HR.EMPLOYEES E
     ON D.MANAGER_ID = E.EMPLOYEE_ID
     WHERE EMPLOYEE_ID IS NULL
```

Right Anti Join

Do mesmo modo, é possível que desejemos um conjunto semelhante ao obtido através do RIGHT OUTER JOIN, porém quem os dados que possuam relação. Ou seja, só aqueles do conjunto B que não possuam relação com o conjunto A, conforme ilustrado abaixo.



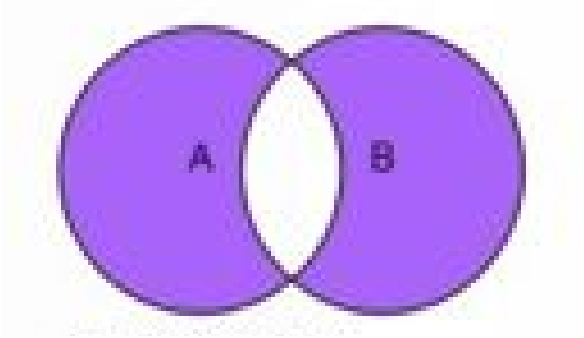
Do mesmo modo, esta relação não possui sintaxe própria, uma vez que deriva do RIGHT OUTER JOIN, adicionando a cláusula de filtro para relação nula apenas. Neste caso um possível exemplo seria:

Quais os departamentos que não possuem manager?

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          RIGHT OUTER JOIN HR.EMPLOYEES E
     ON D.MANAGER_ID = E.EMPLOYEE_ID
          WHERE D.MANAGER_ID IS NULL
```

Full Anti Join

A fim de complementar as possíveis variações, é possível que queiramos apresentar apenas as tuplas que não possuam relação, conforme ilustrado abaixo. Nota-se, adicionalmente, que esta é a operação inversa ao INNER JOIN.

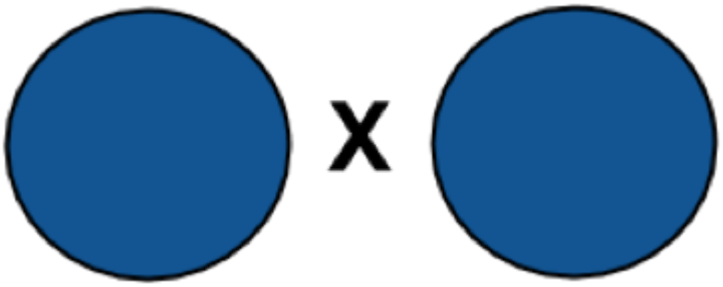


Semanticamente o problema poderia ser: Quais os funcionários que não são managers e os departamentos que não possuem managers?

```
SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
          FULL OUTER JOIN HR.EMPLOYEES E
      ON D.MANAGER_ID = E.EMPLOYEE_ID
          WHERE D.MANAGER_ID IS NULL
          OR E.MANAGER_ID IS NULL
```

Cross Join

Existe uma combinação de resultados que ainda não foi mencionado sintaticamente. Trata-se do CROSS JOIN, responsável justamente por realizar produtos cartesianos em situações onde este tipo de operação é necessária.



Suponhamos, por exemplo, que seja de nosso interesse produzir uma camiseta de cada tamanho em cada cor, conforme ilustrado abaixo:

Tables
Color
Red
Blue

Size
Small
Medium
Large
Extra Large

Query Result	
Color	Size
Red	Small
Blue	Small
Red	Medium
Blue	Medium
Red	Large
Blue	Large
Red	Extra Large
Blue	Extra Large

A sintaxe ANSI para esta operação é:

```
SELECT <PROJEÇÃO> FROM <TABELA1> CROSS JOIN <TABLE2>
ON <TABLE1>.<COLUNA> = <TABLE2>.<COLUNA>;
```

Um exemplo de SQL para este exemplo seria

```
SELECT COLORS.COLOR, SIZES.SIZE from COLORS CROSS JOIN
SIZES;
```

Natural Join

Do mesmo modo que os "Anti Joins" existem algumas outras variações que podem ser de auxílio na escrita de uma consulta, como o NATURAL JOIN. Esta cláusula não é prevista pelo padrão ANSI, mas é implementada pelo Oracle. Sintaticamente, ela elimina a necessidade da cláusula "N <TABLE1>.<COLUNA> = <TABLE2>.<COLUNA>" apresentada nos exemplos anteriores desde que haja apenas uma relação natural entre ambas as tabelas e as colunas utilizadas tenham o mesmo nome em ambas as tabelas.

Cabe ainda mencionar que o NATURAL JOIN pode ser utilizado em combinação com quaisquer Join ANSI mencionados até o momento (LEFT, RIGHT, FULL) e o default é INNER JOIN. A sintaxe deste Join é, portanto:

```
SELECT <PROJEÇÃO> FROM <TABELA1>
NATURAL [[LEFT|RIGHT|FULL] OUTER|INNER] JOIN <TABLE2>;
```

Um possível exemplo utilizando o schema HR: Qual o país de cada cidade?

```
SELECT CITY,COUNTRY_NAME
  FROM HR.LOCATIONS
          NATURAL JOIN HR.COUNTRIES;
```

Semi Join - Subquery (EXISTS)

É possível que hajam relações no entanto que não se tratem apenas de relação completa ou igualdade de IDs, e sim sobre a existência ou não de determinada condição. A este tipo de operação convencionou-se chamar de Semi Join, embora isto não tenha qualquer implicação do ponto de vista de padrão ANSI. Relações deste tipo são sempre marcadas pela existência de subquery/subconsulta e cláusula EXISTS, conforme exemplos abaixo:

Quais os departamentos que possuem funcionários ganhando mais de 2500?

```
SELECT * FROM departments
WHERE EXISTS
      (SELECT * FROM employees
            WHERE departments.department_id = employees.department_id
             AND employees.salary > 2500)
      ORDER BY department_name;
```

O oposto: Quais os departamentos que **NÃO** possuem funcionários ganhando mais de 2500?

Ou ainda: Quais os departamentos em que todos os funcionários ganham menos de 2500?

```
SELECT * FROM departments
WHERE NOT EXISTS
      (SELECT * FROM employees
            WHERE departments.department_id = employees.department_id
             AND employees.salary > 2500)
      ORDER BY department_name;
```

Fatoramento de Subquery (Cláusula WITH)

Do ponto de vista de construção de SQLs de modo um tanto quanto mais sofisticado, uma técnica bastante útil a fim de facilitar a leitura de um SQL é o faturamento da subquery com a cláusula WITH. Deste modo, uma ou mais subconsultas podem ser tratadas como um resultset nominal antes dos filtros a serem explicitados no final da consulta juntamente com a projeção final, conforme o exemplo:

Exemplo: Qual o menor salário de todas as áreas que possuem o salário mínimo menos que 5000 e cujo sobrenome do manager inicie com K?

```
WITH dpto_mgr AS (
  SELECT D.DEPARTMENT_NAME, E.LASTNAME
  FROM HR.DEPARTMENTS D
RIGHT OUTER JOIN HR.EMPLOYEES E ON D.MANAGER_ID = E.EMPLOYEE_ID)
SELECT department_name FROM dpto_mgr WHERE last_name LIKE 'K%
```

6. Tópicos Avançados em SQL

Inserts Compostos

Uma das atividades comuns a manipulação de dados e provavelmente realizada por você algumas vezes durante este módulo é a população de tabelas. através de comandos INSERT. Embora existam algumas ferramentas para cargas de dados de um banco de dados para outro (como o Datapump, a ser apresentado no próximo módulo), formas de realizar inserções em bloco é sempre uma pergunta recorrente.

O MySQL possua uma maneira consideravelmente produtiva para realização desta ação, conforme abaixo:

```
INSERT INTO TMP_DIM_EXCH_RT  (a,b,c,d,e,f,g)
VALUES
     (1, 1 109.49, 'USD', 'JPY' '28-AUG-2008', '28-AUG-2008'),
     (3, 1 1.05, 'USD', 'CAD' '28-AUG-2008', '28-AUG-2008'),
     (4, 1 .68, 'USD', 'EUR' '28-AUG-2008', '28-AUG-2008'),
     (6, 1 7.81, 'USD', 'HKD' '28-AUG-2008', '28-AUG-2008');
```

O Oracle, por sua vez, não possui tal sintaxe, sendo que as alternativas podem ser:

```
INSERT ALL
  INTO suppliers (supplier_id, name) VALUES (1000, 'IBM')
  INTO suppliers (supplier_id, name) VALUES (2000, 'Microsoft')
  INTO customers (customer_id, customer_name, city) VALUES
(999999, 'Anderson Construction', 'New York')
SELECT * FROM dual;
```

Ou ainda:

```
INSERT INTO people (person_id, given_name, family_name, title)
 WITH names AS (
SELECT 4, 'Ruth',    'Fox' family_name,        'Mrs'      FROM dual
UNION ALL
SELECT 5, 'Isabelle', 'Squirrel' family_name, 'Miss'    FROM dual
UNION ALL
SELECT 6, 'Justin',    'Frog' family_name,     'Master' FROM dual
UNION ALL
SELECT 7, 'Lisa',    'Owl' family_name,        'Dr'       FROM dual)
SELECT * FROM names WHERE  family_name LIKE 'F%'
```

SQL Loader

Uma alternativa plausível a este modelo é a utilização do SQL Loader (sqlldr), ferramenta provida pela Oracle para realizar leitura de arquivos texto e carregar esses arquivos diretamente ao banco de dados em formato de tabela através padrões estabelecidos em arquivo de configuração.

Veja abaixo este exemplo de arquivo texto em formato CSV:

```
cat employee.txt
100,Thomas,Sales,5000
200,Jason,Technology,5500
300,Mayla,Technology,7000
400,Nisha,Marketing,9500
500,Randy,Technology,6000
501,Ritu,Accounting,5400
```

E o arquivo de configuração abaixo especificando o caminho do arquivo texto, o nome da tabela destino, as colunas em ordem e o separador de valores (a vírgula neste caso):

```
cat example1.ctl
load data
 infile '/home/oracle/employee.txt'
 into table employee
 fields terminated by ","
 ( id, name, dept, salary )
```

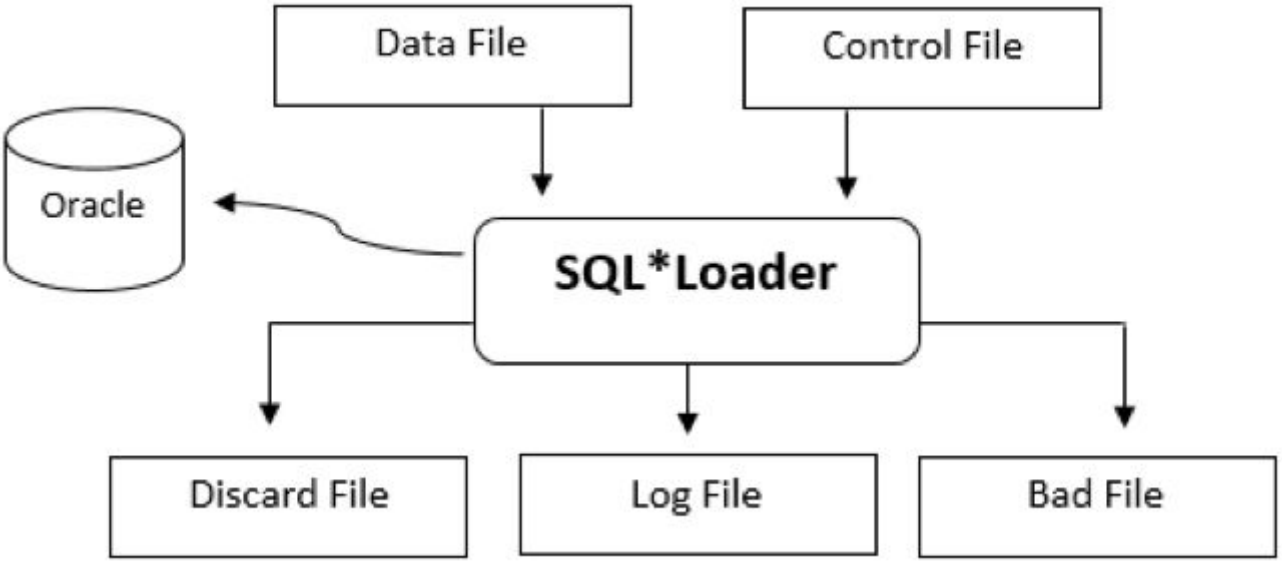
É preciso, evidentemente, que a tabela destinatária da carga de dados exista:

```
create table employee (
  id integer,
  name varchar2(10),
  dept varchar2(15),
  salary integer,
  hiredon date);
```

A sintaxe para a carga é a seguinte:

```
sqlldr hr/target@pdb1 control=/home/oracle/example1.ctl
```

Os dados do arquivo texto serão carregados da tabela EMPLOYEE. A ilustração abaixo demonstra o funcionamento do SQL Loader:



Existem evidentemente outras alternativas, especialmente considerando necessidades de integração direta com outros sistemas de dados ou mensageria que também podem realizar leituras em arquivos físicos, como external tables.

Mecanismos de Locks

Os mecanismos de lock ou bloqueio estão presentes no Oracle por razões óbvias: eles garante a ACID. O mecanismo básico de lock do Oracle é um dos mais sofisticados e eficientes dentre os bancos de dados de grande porte, frequentemente apontado como superior a bancos como SQL Server e DB2.

Existe uma complexidade bastante elevada neste tópicos e, assim como índices, existem livros inteiramente dedicados aos estudos sobre locks em Oracle, com tópicos complementares como níveis de isolamento de transação e fenômenos observáveis. Estes tópicos não serão foco de estudo de um curso de Introdução ao SQL.

No que diz respeito a eficiência, no entanto, de forma sintática, os outros dois SGBDs mencionados acima uma espécie de lock manager, responsável por garantir o acesso único aos dados, diferentemente do Oracle, onde esse controle é feito inteiramente no header (cabeçalho) do bloco onde o dado a ser locado reside. Uma estratégia inteligente, evitando a um possível ponto de gargalo em caso o banco de dados seja massivamente transacional, além de aumentar a escalabilidade do sistema como um todo.

Cabe ressaltar ainda que o Oracle possui alguns tipos de lock, separados por nível. São eles:

Tipo	**Nível**
Row Share (RS)	Tabela
Row Exclusive (RX)	Tabela
Share (S)	Tabela
Share Row Exclusive (SRX)	Tabela
Exclusive (X)	Tabela/Linha

Sobre os níveis de bloqueio:

Locks de Linha

Um lock do tipo linha é aplicado individualmente às linhas de uma tabela. Quando uma linha está bloqueada nenhuma outra sessão conseguirá alterá-la até que a transação que detém o lock chegue ao fim. No entanto, mesmo que uma determinada linha de uma tabela esteja bloqueada por uma determinada transação, outros processos poderão alterar as outras linhas dessa tabela que ainda não estejam bloqueadas.

Locks de Tabela

Esse tipo de lock é aplicado no nível tabela e pode ser utilizado para obter uma imagem consistente da tabela e assegurar que sua definição não mude enquanto transações ativas ainda existirem nela.

Sobre os tipos de bloqueio:

Row Share (RS)

Também é conhecido com subshare lock (SS), esse lock indica que a transação possui linhas bloqueadas exclusivamente, mas ainda não as alterou. É obtido através das instruções:

```
SELECT <PROJEÇÃO> FROM <TABLE> FOR UPDATE;

LOCK TABLE <TABLE> IN ROW SHARE MODE;
```

Um lock RS pode ser utilizado para garantir que a tabela não será bloqueada exclusivamente antes de sua transação alterá-la.

Row Exclusive (RX)

Também conhecido como subexclusive lock (SX), esse lock indica que a transação possui linhas bloquadas exclusivamente e já as alterou. É obtido através das instruções:

```
INSERT INTO <TABLE>(<COLUMNS>) VALUES (<VALUES>);

UPDATE <TABLE> ...;

DELETE FROM <TABLE> ...;

LOCK TABLE <TABLE> IN ROW EXCLUSIVE MODE;
```

Share (S)

Geralmente o lock S é utilizado para garantir uma imagem consistente da tabela ao longo da transação, isto é, ele garante que nenhum comando DML será permitido. É necessário utilizar essa abordagem, pois o nível de isolamento padrão do Oracle é Read Commit. Portanto, as alterações efetuadas por outras transações são imediatamente visíveis para todos os usuários logo após sofrerem commit. Sendo assim, sua transação pode ter várias versões da mesma tabela durante sua duração, o que às vezes não é interessante. Para evitar esse tipo de problema, o uso de um lock S ou iniciar uma transação com o isolation level definido para *Serializable*, podem ser as alternativas. Um lock Share S pode ser obtido explicitamente através da instrução:

```
LOCK TABLE <TABLE> IN SHARE MODE;
```

Um ponto de atenção, contudo, é: Quando a tabela está bloqueada em modo Shared, somente a transação que possui o lock pode executar DMLs. Por se tratar de um lock do tipo share, várias transações podem obtê-lo concorrentemente. Caso isso aconteça, nenhuma delas conseguirá efetuar alterações, pois uma estará bloqueando a outra, podendo ocasionar em um enfileiramento sequencial de locks que só pode ser desfeito na ordem correta.

Share Row Exclusive (SRX)

Esse tipo de lock é um pouco mais exclusivo que o S, pois somente uma transação pode obtê-lo por vez. Ele permite a outras transações obterem locks em linhas específicas, mas não permite a alteração delas. Esse tipo de lock é obtido através da instrução abaixo:

```
LOCK TABLE <TABLE> IN SHARE ROW EXCLUSIVE MODE;
```

Exclusive (X)

Esse é o mais restritivo lock existente, ele permite apenas que as outras sessões acessem a tabela através de instruções SELECTs, ou seja, nenhum tipo de lock é permitido. Esse lock também é o único lock que pode ser aplicado a linhas. Quando uma linha está bloqueada, ela está bloqueada exclusivamente e nenhuma transação conseguirá alterá-la até que a transação que mantém o lock chegue ao fim através de um rollback ou commit. Uma forma de realizar este tipo de bloqueio é através do comando abaixo.

```
LOCK TABLE <TABLE> IN EXCLUSIVE MODE;
```

Plano de Acessos de SQL

Do ponto de vista de desempenho, algo bastante importante é o entendimento do plano de execução de uma consulta.

Este aspecto será abordado com maiores detalhes no último módulo desta formação, contudo, há de de se mencionar:

- Todo SQL executado no SGBD para por três fases: Parse, Execute e Fetch.
- Durante o Parse, o SGBD não apenas válida a sintaxe e a semântica de um SQL como estima qual será o plano de de acesso a estes dados e quais os melhores algoritmos para eventuais joins e outras operações estudadas neste módulo.
- O resultado desta operação é o Plano de Execução do SQL, contendo as operações que serão realizadas para acesso a cada dado e as operações para junção e apresentação dos mesmos, bem como estimativas de custo para cada um destes passos.

O Plano de Execução de qualquer SQL já executado (ou não, neste caso uma estimativa de plano) podem ser obtidos e este é o primeiro nível de tuning a ser realizado sobre SQL.

A imagem abaixo possui um exemplo de exibição de um plano de execução de um SQL.

```
-------------------------------------------------------------------------------------
| Id  | Operation                          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |     1 |   141 |  3704   (1)| 00:00:45 |
|   1 |  SORT GROUP BY                     |     1 |   141 |  3704   (1)| 00:00:45 |
|   2 |   NESTED LOOPS                     |       |       |            |          |
|   3 |    NESTED LOOPS                    |     1 |   141 |  3703   (1)| 00:00:45 |
|   4 |     NESTED LOOPS                   |     4 |   396 |    10  (10)| 00:00:01 |
|   5 |      TABLE ACCESS BY INDEX ROWID   |     1 |    18 |     1   (0)| 00:00:01 |
|*  6 |       INDEX UNIQUE SCAN            |     1 |       |     0   (0)| 00:00:01 |
|   7 |      VIEW                          |     4 |   324 |     9  (12)| 00:00:01 |
|   8 |       SORT GROUP BY                |     4 |   228 |     9  (12)| 00:00:01 |
|   9 |        NESTED LOOPS                |    31 |  1767 |     8   (0)| 00:00:01 |
|  10 |         MERGE JOIN CARTESIAN       |   819 | 27027 |     8   (0)| 00:00:01 |
|*11 |          INDEX RANGE SCAN          |     1 |    10 |     2   (0)| 00:00:01 |
|  12 |          BUFFER SORT               |   819 | 18837 |     6   (0)| 00:00:01 |
|*13 |           TABLE ACCESS STORAGE FULL|   819 | 18837 |     6   (0)| 00:00:01 |
|*14 |         INDEX UNIQUE SCAN          |     1 |    24 |     0   (0)| 00:00:01 |
|*15 |      INDEX RANGE SCAN              |  1745 |       |   288   (0)| 00:00:04 |
|*16 |     TABLE ACCESS BY INDEX ROWID    |     1 |    42 |  1825   (1)| 00:00:22 |
-------------------------------------------------------------------------------------
```

Para mais informações, consulte a disponibilidade de turmas para o Módulo de Performance Tuning.