2) Utilizando SQL para selecionar, filtrar e agrupar registros

2.1) A linguagem SQL

Origem: Wikipédia, a enciclopédia livre.

Structured Query Language, ou **Linguagem de Consulta Estruturada** ou **SQL**, é uma linguagem de pesquisa declarativa para banco de dados relacional (base de dados relacional). Muitas das características originais do SQL foram inspiradas na álgebra relacional.

O SQL foi desenvolvido originalmente no início dos anos 70 nos laboratórios da IBM em San Jose, dentro do projeto System R, que tinha por objetivo demonstrar a viabilidade da implementação do modelo relacional proposto por E. F. Codd. O nome original da linguagem era **SEQUEL**, acrônimo para **"Structured English Query Language"** (**Linguagem de Consulta Estruturada em Inglês**) [1], vindo daí o fato de, até hoje, a sigla, em inglês, ser comumente pronunciada "síquel" ao invés de "és-kiú-él", letra a letra. No entanto, em português, a pronúncia mais corrente é a letra a letra: "ése-quê-éle".

A linguagem SQL é um grande padrão de banco de dados. Isto decorre da sua simplicidade e facilidade de uso. Ela se diferencia de outras linguagens de consulta a banco de dados no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele. Ela é uma linguagem declarativa em oposição a outras linguagens procedurais. Isto reduz o ciclo de aprendizado daqueles que se iniciam na linguagem.

Embora o SQL tenha sido originalmente criado pela IBM, rapidamente surgiram vários "dialectos" desenvolvidos por outros produtores. Essa expansão levou à necessidade de ser criado e adaptado um padrão para a linguagem. Esta tarefa foi realizada pela American National Standards Institute (ANSI) em 1986 e ISO em 1987.

O SQL foi revisto em 1992 e a esta versão foi dado o nome de SQL-92. Foi revisto novamente em 1999 e 2003 para se tornar SQL:1999 (SQL3) e SQL:2003, respectivamente. O SQL:1999 usa expressões regulares de emparelhamento, *queries* recursivas e gatilhos (*triggers*). Também foi feita uma adição controversa de tipos não-escalados e algumas características de orientação a objeto. O SQL:2003 introduz características relacionadas ao XML, seqüências padronizadas e colunas com valores de auto-generalização (inclusive colunas-identidade).

Tal como dito anteriormente, o SQL, embora padronizado pela ANSI e ISO, possui muitas

variações e extensões produzidos pelos diferentes fabricantes de sistemas gerenciadores de bases de dados. Tipicamente a linguagem pode ser migrada de plataforma para plataforma sem mudanças estruturais principais.

Outra aproximação é permitir para código de idioma procedural ser embutido e interagir com o banco de dados. Por exemplo, o Oracle e outros incluem Java na base de dados, enquanto o PostgreSQL permite que funções sejam escritas em Perl, Tcl, ou C, entre outras linguagens.

2.2) Principais Palavras-Chave e Identificadores

DML - Linguagem de Manipulação de Dados
Primeiro há os elementos da DML (Data Manipulation Language - Linguagem de Manipulação de Dados). A DML é um subconjunto da linguagem usada para selecionar, inserir, atualizar e apagar dados.

- **SELECT** é o comumente mais usado do DML, comanda e permite ao usuário especificar uma query como uma descrição do resultado desejado. A questão não especifica como os resultados deveriam ser localizados.
- **INSERT** é usada para somar uma fila (formalmente uma tupla) a uma tabela existente.
- **UPDATE** para mudar os valores de dados em uma fila de tabela existente.
- **DELETE** permite remover filas existentes de uma tabela.

DDL - Linguagem de Definição de Dados

O segundo grupo é a DDL (Data Definition Language - Linguagem de Definição de Dados). Uma DDL permite ao usuário definir tabelas novas e elementos associados. A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL.

Os comandos básicos da DDL são poucos

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

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

outros comandos DDL:

- **ALTER** TABLE
- CREATE INDEX
- ALTER INDEX
- DROP INDEX
- CREATE VIEW
- DROP VIEW

DCL - Linguagem de Controle de Dados

O terceiro grupo é o DCL (Data Control Language - Linguagem de Controle de Dados). DCL controla os aspectos de autorização de dados e licenças de usuários para controlar quem tem acesso para ver ou manipular dados dentro do banco de dados.

Duas palavras-chaves da DCL:

- **GRANT** - autoriza ao usuário executar ou setar operações.
- **REVOKE** - remove ou restringe a capacidade de um usuário de executar operações.

DTL - Linguagem de Transação de Dados

- **BEGIN WORK** (ou START TRANSACTION, dependendo do dialeto SQL) pode ser usado para marcar o começo de uma transação de banco de dados que pode ser completada ou não.
- **COMMIT** envia todos os dados das mudanças permanentemente.
- **ROLLBACK** faz com que as mudanças nos dados existentes desde que o último COMMIT ou ROLLBACK sejam descartadas.

COMMIT e ROLLBACK interagem com áreas de controle como transação e locação. Ambos terminam qualquer transação aberta e liberam qualquer cadeado ligado a dados. Na ausência de um BEGIN WORK ou uma declaração semelhante, a semântica de SQL é dependente da implementação.

outros comandos DCL:

- ALTER PASSWORD
- CREATE SYNONYM

DQL - Linguagem de Consulta de Dados

Embora tenha apenas um comando a DQL é a parte da SQL mais utilizada. O comando SELECT é composta de várias cláusulas e opções, possibilitando elaborar consultas das mais simples as mais elaboradas.

Fonte: http://pt.wikipedia.org/wiki/Sql

Palavras-chaves: http://pgdocptbr.sourceforge.net/pg80/sql-keywords-appendix.html e

http://www.postgresql.org/docs/8.3/interactive/sql-keywords-appendix.html

Tutoriais de SQL Online - http://sqlcourse.com/ http://sqlzoo.net/ e

http://www.cfxweb.net/modules.php?name=News&file=article&sid=161

2.3) Trabalhando e Manipulando Valores Nulos

Em SQL NULL é para valores inexistentes. Regra geral: NULL se propaga, o que significa que com quem NULL se combina o resultado será um NULL.
NULL não é zero, não é string vazia nem string de comprimento zero.

Um exemplo: num cadastro de alunos, para o aluno que ainda não se conhece a nota, não é correto usar zero para sua nota, mas sim NULL.
Não se pode efetuar cálculos de expressões onde um dos elementos é NULL.

COMPARANDO NULLs
NOT NULL com NULL -- Unknown
NULL com NULL -- Unknown

CONVERSÃO DE/PARA NULL
NULLIF() e COALESCE()

NULLIF(valor1, valor2)

NULLIF – Retorna NULL se, e somente se, valor1 e valor2 forem iguais, caso contrário retorna valor1.

Algo como:
if (valor1 == valor2){
then NULL
else valor1;

Retorna valor1 somente quando valor1 == valor2.

COALESCE – retorna o primeiro de seus argumentos que não for NULL. Só retorna NULL quando todos os seus argumentos forem NULL.

Uso: mudar o valor padrão cujo valor seja NULL.

create table nulos(nulo int, nulo2 int, nulo3 int);

insert into nulos values (1,null,null);

select coalesce(nulo, nulo2, nulo3) from nulos; - - Retorna 1, valor do campo nulo;

select coalesce(nulo2, nulo3) from nulos; - - Retorna NULL, pois ambos são NULL.

GREATEST - Retorna o maior valor de uma lista - SELECT GREATEST(1,4,6,8,2); - - 8

LEAST - Retorna o menor valor de uma lista.

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.
Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

CONCATENANDO NULLs
A regra é: NULL se propaga. Qualquer que concatene com NULL gerará NULL.

STRING || NULL -- NULL

Usos:
- Como valor default para campos que futuramente receberão valor.
- Valor default para campos que poderão ser sempre inexistentes.

2.4) Utilizando Comentários

Ao criar scripts SQL a serem importados pelo PostgreSQL, podemos utilizar dois tipos de comentários:

-- Este é um comentário padrão SQL e de uma única linha

/* Este é um comentário

oriundo da linguagem C

e para múltiplas linhas e que também é aceito

nos scripts SQL */

2.5) Entendendo os Tipos de Dados

Relação completa de tipos de dados da versão 8.3.1:

8.1. Numeric Types
  8.1.1. Integer Types
  8.1.2. Arbitrary Precision Numbers
  8.1.3. Floating-Point Types
  8.1.4. Serial Types
8.2. Monetary Types
8.3. Character Types
8.4. Binary Data Types
8.5. Date/Time Types
  8.5.1. Date/Time Input
  8.5.2. Date/Time Output
  8.5.3. Time Zones
  8.5.4. Internals

Mais detalhes em: http://www.postgresql.org/docs/8.3/interactive/datatype.html

Tipos de Dados Mais Comuns

Numéricos			
Tipo	**Tamanho**	**Apelido**	**Faixa**
smallint (INT2)	2 bytes	inteiro pequeno	-32768 a +32767
integer (INT ou INT4)	4 bytes	inteiro	-2147483648 até +2147483647
bigint (INT8)	8 bytes	inteiro longo	-9223372036854775808 a +9223372036854775807
numeric (p,e)			tamanho variável, precisão especificada pelo usuário. Exato e sem limite
decimal (p,e)			e – escala (casas decimais) p – precisão (total de dígitos, inclusive estala)
real (float)	4 bytes	ponto flutuante	precisão variável, não exato e precisão de 6 dígitos
double precision	8 bytes	dupla precisão	precisão variável, não exato e precisão de 15 dígitos
int (INT4)			mais indicado para índices de inteiros
serial	4 bytes	Inteiro autoinc	1 até 2147483647
bigserial	8 bytes	Inteiro longo autoinc	1 até 9223372036854775807
Caracteres			
character varying(n)		varchar(n)	comprimento variável, com limite
character(n)		char(n)	comprimento fixo, completa com brancos
text			comprimento variável e ilimitado
Desempenho semelhante para os tipos caractere.			
Data/Hora			
timestamp[(p)] [witout time zone]	8 bytes	data e hora sem zona	4713 AC a 5874897 DC
timestamp [(p)][with time zone]	8 bytes	data e hora com zona	4713 AC a 5874897 DC
interval	12 bytes	intervalo de tempo	178000000 anos a 178000000 anos
date	4 bytes	somente data	4713 AC até 32767 DC
time [(p)] [without time zone]	8 bytes	somente a hora	00:00:00.00 até 23:59:59.99
time [(p)] [with time zone]	8 bytes	somente a hora	00:00:00.00 até 23:59:59.99
[(p)] - é a precisão, que varia de 0 a 6 e o defualt é 2.			

Boleanos			
Tipo	**Tamanho**	**Apelido**	**Faixa**
TRUE		Representações:	't', 'true', 'y', 'yes' e '1'
FALSE		Representações:	'f', 'false', 'n', 'no', '0'
Apenas um dos dois estados. O terceiro estado, desconhecido, é representado pelo NULL.			

Exemplo de consulta com boolean:

```
CREATE TEMP TABLE teste1 (a boolean, b text);
INSERT INTO teste1 VALUES (TRUE, 'primeiro');
INSERT INTO teste1 VALUES (FALSE, 'segundo');
SELECT * FROM teste1;
```

Retorno:

```
 a |    b
---+---------
 t | primeiro
 f | segundo
```

```
SELECT * FROM teste1 WHERE a = TRUE;  -- Retorna 't'
SELECT * FROM teste1 WHERE a = t;  -- Erro
SELECT * FROM teste1 WHERE a = 't';     -- Retorna 't'
SELECT * FROM teste1 WHERE a = '1';        -- Retorna 't'
SELECT * FROM teste1 WHERE a = 'y';        -- Retorna 't'
SELECT * FROM teste1 WHERE a = '0';        -- Retorna 'f'
SELECT * FROM teste1 WHERE a = 'n';        -- Retorna 'f'
```

Alerta: a entrada pode ser: 1/0, t/f, true/false, TRUE/FALSE, mas o retorno será sempre t ou f.

Obs.: Para campos tipo data que permitam NULL, devemos prever isso na consulta SQL e passar NULL sem delimitadores e valores não NULL com delimitadores.

Obs2: Evite o tipo MONEY que está em obsolescência. Em seu lugar use NUMERIC. Prefira INT (INTEGER) em lugar de INT4, pois os primeiros são padrão SQL. Em geral evitar os nomes INT2, INT4 e INT8, que não são padrão. O INT8 ou bigint não é padrão SQL.
Obs3: Em índices utilize somente INT, evitando smallint e bigint, que nunca serão utilizados.

Tipos SQL Padrão
bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (com ou sem zona horária), timezone (com ou sem zona horária).

O tipo **NUMERIC** pode realizar cálculos exatos. **Recomendado para quantias monetárias** e outras quantidades onde a exatidão seja importante. Isso paga o preço de queda de desempenho comparado aos inteiros e flutuantes.

Pensando em portabilidade evita usar NUMERIC(12) e usar NUMERIC (12,0).

Alerta: A comparação de igualdade de dois valores de ponto flutuante pode funcionar conforme o esperado ou não.

O PostgreSQL trabalha com datas do calendário Juliano.
Trabalha com a faixa de meio dia de Janeiro de 4713 AC (ano bisexto, domingo de lua nova) até uma data bem distante no futuro. Leva em conta que o ano tem 365,2425 dias.

SERIAL
No PostgreSQL um campo criado do “tipo” SERIAL é internamente uma seqüência, inteiro positivo.

Os principais SGBDs utilizam alguma variação deste tipo de dados (auto-incremento). Serial é o "tipo" auto-incremento do PostgreSQL. Quando criamos um campo do tipo SERIAL ao inserir um novo registro na tabela com o comando INSERT omitimos o campo tipo SERIAL, pois ele será inserido automaticamente pelo PostgreSQL.

```
CREATE TABLE serial_teste (codigo SERIAL, nome VARCHAR(45));
INSERT INTO serial_teste (nome) VALUES ('Ribamar FS');
```

Obs.: A regra é nomear uma seqüência “serial_teste_codigo_seq”,ou seja, tabela_campo_seq.

```
select * from serial_teste_codigo_seq;
```

Esta consulta acima retorna muitas informações importantes sobre a seqüência criada: nome, valor inicial, incremento, valor final, maior e menor valor além de outras informações.

Veja que foi omitido o campo código mas o PostgreSQL irá atribuir para o mesmo o valor do próximo registro de código. Por default o primeiro valor de um serial é 1, mas se precisarmos começar com um valor diferente veja a solução abaixo:

Setando o Valor Inicial do Serial

```
create sequence produtos_codigo_seq start 9;
ALTER SEQUENCE tabela_campo_seq RESTART WITH 1000;
```

2.6) Utilizando Expressões e Constantes

Expressões SQL em: http://db.apache.org/derby/docs/dev/pt_BR/ref/rrefsqlj19433.html

Precedência dos operadores (decrescente)

Operador/Elemento	Associatividade	Descrição
.	esquerda	separador de nome de tabela/coluna
::	esquerda	conversão de tipo estilo PostgreSQL
[ ]	esquerda	seleção de elemento de matriz
-	direita	menos unário
^	esquerda	exponenciação
* / %	esquerda	multiplicação, divisão, módulo
+ -	esquerda	adição, subtração
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		teste de nulo
NOTNULL		teste de não nulo
(qualquer outro)	esquerda	os demais operadores nativos e os definidos pelo usuário
IN		membro de um conjunto
BETWEEN		contido em um intervalo
OVERLAPS		sobreposição de intervalo de tempo
LIKE ILIKE SIMILAR		correspondência de padrão em cadeia de caracteres
< >		menor que, maior que
=	direita	igualdade, atribuição
NOT	direita	negação lógica
AND	esquerda	conjunção lógica
OR	esquerda	disjunção lógica

Detalhes em: http://pgdocptbr.sourceforge.net/pg80/sql-syntax.html#SQL-PRECEDENCE

http://www.postgresql.org/docs/8.3/interactive/sql-syntax-lexical.html#SQL-PRECEDENCE-TABLE

2.7) Ocultando Linhas Duplicadas em uma Consulta (DISTINCT)

Cláusula *DISTINCT*

Se for especificada a cláusula DISTINCT, todas as linhas duplicadas serão removidas do conjunto de resultados (será mantida uma linha para cada grupo de duplicatas).
A cláusula ALL especifica o oposto: todas as linhas serão mantidas; este é o padrão.

DISTINCT ON (expressão [, ...]) preserva apenas a primeira linha de cada conjunto de linhas onde as expressões fornecidas forem iguais. As expressões em DISTINCT ON são interpretadas usando as mesmas regras da cláusula ORDER BY (veja acima). Deve ser observado que a "primeira linha" de cada conjunto é imprevisível, a menos que seja utilizado ORDER BY para garantir que a linha desejada apareça na frente. Por exemplo,

```
create temp table dup (c int);
insert into dup (c)  values (1);
insert into dup (c)  values (2);
insert into dup (c)  values (1);
insert into dup (c)  values (2);
insert into dup (c)  values (1);
select * from dup;
select distinct (c) from dup;
```

retorna o relatório de condição climática mais recente para cada local, mas se não tivesse sido usado ORDER BY para obrigar a ordem descendente dos valores da data para cada local, teria sido obtido um relatório com datas imprevisíveis para cada local.

As expressões em DISTINCT ON devem corresponder às expressões mais à esquerda no ORDER BY. A cláusula ORDER BY normalmente contém expressões adicionais para determinar a precedência desejada das linhas dentro de cada grupo DISTINCT ON.

DISTINCT – Escrita logo após SELECT desconsidera os registros duplicados, retornando apenas registros exclusivos.

2.8) Limitando o Resultado do Select

Caso utilizemos a forma:

SELECT * FROM tabela;

Serão retornados todos os registros com todos os campos. Podemos filtrar tanto os registros quanto os campos que retornam da consulta. Também, ao invés de *, podemos digitar todos os campos da tabela.

Filtrando os Campos que Retornam

Para filtrar os campos, retornando apenas alguns deles, basta ao invés de * digitar apenas os campos que desejamos retornar, por exemplo:

SELECT cpf, nome FROM clientes;

Filtrando os Registros que Retornam

Para filtrar os registros temos várias cláusulas SQL, como WHERE, GROUP BY, LIMIT, HAVING, etc.

Exemplos:
SELECT nome FROM clientes WHERE email = 'ribafs@ribafs.net';
SELECT nome FROM clientes WHERE idade > 18;
SELECT nome FROM clientes WHERE idade < 21;
SELECT nome FROM clientes WHERE idade >= 18;
SELECT nome FROM clientes WHERE idade <= 21;
SELECT nome FROM clientes WHERE UPPER(estado) != 'CE';

select nome, (current_date – data_nasc)/365 as idade from clientes;

2.9) Utilizando o Comando Case

CASE é uma expressão condicional do SQL. Uma estrutura semelhante ao IF das linguagens de programação. Caso WHEN algo THEN isso. Vários WHEN podem vir num único CASE. Finaliza com END.

```
CASE WHEN condição THEN resultado
     WHEN condição2 THEN resultado
     [WHEN ...]
     [ELSE resultado]
END
```

Obs.: O que vem entre concletes é opcional.

Mais detalhes em: http://pgdocptbr.sourceforge.net/pg80/functions-conditional.html

William Leite Araújo na lista pgbr-geral:
Ambas as formas são válidas. Você pode usar o

```
    SELECT CASE WHEN [teste] THEN ... ELSE [saida] END;
ou
    SELECT CASE [coluna]
        WHEN [valor1] THEN resultado
        WHEN [valor2] THEN resultado
        ...
        ELSE [saida] END
```

Quando o teste é entre 2 valores, a primeira forma é a mais aplicável. Quando quer se diferenciar mais de um valor de uma mesma coluna, a segunda é a mais apropriada. Por exemplo:

```
SELECT CASE tipo_credido WHEN 'S' THEN 'Salário' WHEN 'P' THEN 'Pró-labore' WHEN 'D'
THEN 'Depósito' ELSE 'Outros' END as tipo_credito;
```

```
SELECT nome, CASE WHEN EXISTS (SELECT nome FROM clientes WHERE nome=c.nome)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM clientes c;
```

Trazer o nome sempre que existir o nome do cliente.

IN
SELECT nome, CASE WHEN nome IN (SELECT nome FROM clientes)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM clientes;

NOT IN e ANY/SOME
SELECT cpf_cliente, CASE WHEN cpf_cliente = ANY (SELECT cpf_cliente FROM pedidos)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM pedidos;

Trazer o CPF

Outros Exemplos:

```
create database dml;
\c dml
create table amigos(
codigo serial primary key,
nome char(45),
idade int
);

insert into amigos (nome, idade) values ('João Brito', 25);
insert into amigos (nome, idade) values ('Roberto', 35);
insert into amigos (nome, idade) values ('Antônio', 15);
insert into amigos (nome, idade) values ('Francisco Queiroz', 23);
insert into amigos (nome, idade) values ('Bernardo dos Santos', 21);
insert into amigos (nome, idade) values ('Francisca Pinto', 22);
insert into amigos (nome, idade) values ('Natanael', 55);

select nome,
idade,
case
when idade >= 21 then 'Adulto'
else 'Menor'
end as status
from amigos order by nome;
```

-- CASE WHEN cria uma coluna apenas para exibição

```
create table amigos2(
codigo serial primary key,
nome char(45),
estado char(2)
);


insert into amigos2 (nome, estado) values ('João Brito', 'CE');
insert into amigos2 (nome, estado) values ('Roberto', 'MA');
insert into amigos2 (nome, estado) values ('Antônio', 'CE');
insert into amigos2 (nome, estado) values ('Francisco Queiroz', 'PB');
insert into amigos2 (nome, estado) values ('Bernardo dos Santos', 'MA');
insert into amigos2 (nome, estado) values ('Francisca Pinto', 'SP');
insert into amigos2 (nome, estado) values ('Natanael', 'SP');

select nome,
estado,
case
when estado = 'PB' then 'Fechado'
when estado = 'CE' or estado = 'SP' then 'Funcionando'
when estado = 'MA' then 'Funcionando a todo vapor'
else 'Menor'
end as status
from amigos2 order by nome;

create table notas(
nota numeric(4,2)
);

insert into notas(nota) values (4),
(6),
(3),
(10),
(6.5),
(7.3),
(7),
(8.8),
(9);

-- Mostrar cada nota junto com a menor nota, a maior nota, e a média de todas as notas.

SELECT nota,
(SELECT MIN(nota) FROM notas) AS menor,
(SELECT MAX(nota) FROM notas) AS maior,
```

(SELECT ROUND(AVG(nota),2) FROM notas) AS media
FROM notas;

2.10) Substituindo Valores Nulos para Formatar dados Retornados na Consulta

COALESCE – retorna o primeiro de seus argumentos que não for NULL. Só retorna NULL quando todos os seus argumentos forem NULL.

Uso: mudar o valor padrão cujo valor seja NULL.

```
create temp table nulos(
    nulo1 int,
    nulo2 int,
    nulo3 int
);

insert into nulos values (1,null,null);
select * from nulos;
select coalesce(nulo1, nulo2, nulo3) from nulos; - - Retorna 1, valor do campo nulo;
select coalesce(nulo2, nulo3) from nulos; - - Retorna NULL, pois ambos são NULL.
```

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.
Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

CONCATENANDO NULLs
A regra é: NULL se propaga. Qualquer que concatene com NULL gerará NULL.

STRING || NULL -- NULL

NULLIF
NULLIF(valor1, valor2)
A função NULLIF retorna o valor nulo se, e somente se, valor1 e valor2 forem iguais. Senão, retorna valor1. Pode ser utilizada para realizar a operação inversa do exemplo para COALESCE mostrado acima:

```
SELECT nullif(valor, '(nenhuma)') ...
```

Inserir nulo quando a cadeia de caracteres estiver vazia
Neste exemplo são utilizadas as funções NULLIF e TRIM para inserir o valor nulo na coluna da tabela quando a cadeia de caracteres passada como parâmetro para o comando INSERT preparado estiver vazia ou só contiver espaços.

```
CREATE TEMPORARY TABLE t (c1 SERIAL PRIMARY KEY, c2 TEXT);
PREPARE inserir (TEXT)
AS INSERT INTO t VALUES(DEFAULT, nullif(trim(' ' from $1),''));

 EXECUTE inserir('linha 1');
```

```
 EXECUTE inserir('');
 EXECUTE inserir('   ');
 EXECUTE inserir(NULL);

 \pset null nulo
 SELECT * FROM t;

 c1 |   c2
----+---------
  1 | linha 1
  2 | nulo
  3 | nulo
  4 | nulo
```

Mais detalhes em: http://pgdocptbr.sourceforge.net/pg80/functions-conditional.html

Desvendando o SELECT

O SQL (Structured Query Language) é uma linguagem que visa padronizar e facilitar o gerenciamento de informações em bancos de dados relacionais. No entanto, a padronização dos comandos sofre com as implementações proprietárias, ou seja, cada fabricante de banco de dados, embora em sua maioria implemente ao menos em parte o SQL ANSI, implementa também características próprias que o torna diferente do SQL padrão, criando vícios de programação que acabam por prender o desenvolvedor a esse ou aquele banco.

Isso ocorre com praticamente todos os SGBDs, sejam eles comerciais, "OpenSources" ou "Freewares". Nesse artigo, trataremos instruções que operam de maneira similar na maioria dos SGBDs, entre eles o *MySQL, PortgreSQL, Firebird* e *Oracle*.

O SQL é uma linguagem bastante simples, com instruções de alto nível, mas que também permite a escrita de códigos complexos.

Nesse artigo, estaremos abordando exclusivamente o comando **SELECT**. Essa instrução é sem dúvida uma das mais utilizadas do SQL, sendo responsável por realizar consultas na base de dados.

A seguir temos um resumo da sintaxe do SELECT:

```
SELECT [ DISTINCT | ALL ] campos FROM
tabela1 [, tabela n]
[ JOIN condição ]
[ WHERE condição ]
[ GROUP BY expressão ]
[ HAVING condição ]
[ ORDER BY expressão [ASC | DESC] ]
```

A instrução SELECT não se restringe somente a sintaxe apresentada, mas será esta sintaxe o objeto do nosso estudo. Com exceção da cláusula JOIN, toda cláusula deve aparecer somente uma única vez no comando, ou seja, não há como usar um WHERE duas vezes para o mesmo SELECT, ou dois ORDER BY. Analisando a sintaxe apresentada, temos: Tudo que aparece entre colchetes é opcional, ou seja, você pode ou não utilizar essas cláusulas, sem que isso gere qualquer tipo de erro.

Campos: São as colunas retornadas pela instrução. Pode ser empregado um coringa "*" quando se deseja recuperar todos os campos.

A maioria dos SGBDs exige que a cláusula FROM esteja presente no SELECT.

DISTINCT | ALL: Indica se o select deve descartar informações repetidas, ou se deve trazer todas as linhas encontradas. A cláusula ALL é o padrão, podendo ser omitida, e recupera todas as linhas/registros.

Tabela1 – Tabela n São exemplos de nomes de tabelas, sendo que quando mais de uma tabela forem especificadas, os nomes devem ser separados por vírgula. Podemos também designar apelidos para as tabelas (aliases), indicando-os logo após o nome de cada tabela. Os apelidos são válidos apenas para o select em questão.

Condição: Fator pelo qual a query irá filtrar os registros. Podemos utilizar operadores lógicos nas comparações como, por exemplo, OR (ou), AND (e), etc.

Expressão: São as informações pela qual a cláusula irá operar. Pode ser um campo, lista de campos, ou em alguns casos até mesmo uma condição.

Estaremos utilizando como exemplo neste artigo duas tabelas (pessoas e acessos), com a seguinte

estrutura:

```
PESSOAS       ACESSOS
ID INTEGER      ID INTEGER
UNAME CHAR(15)  DATA DATETIME
NOME CHAR(50)   PESSOA CHAR(15)
IDADE INTEGER   QTDE INTEGER
```

Cláusula FROM

Basicamente a cláusula FROM é utilizada para indicar de onde será extraída a informação retornada pela query, ou seja, de quais views, tabelas, ou em alguns bancos até mesmo stored procedures. Exemplo de utilização:

```
SELECT * FROM PESSOAS;
```

O * (asterisco) indica que desejamos receber todos os campos da tabela PESSOAS.

Podemos no lugar do * (asterisco) indicar o nome do campo desejado, ou nomes dos campos separados por vírgula.

Também é possível designar apelidos para as tabelas (alias), indicando-os logo após o nome da tabela. Os apelidos são válidos apenas para o select em questão, e são associados às colunas recuperadas, determinando portanto à qual fonte de dados a coluna está associada. Exemplo:

```
SELECT UNAME as "USER NAME"
FROM PESSOAS;
```

*Obs.: No exemplo anterior, a cláusula **AS** é opcional e poderia ser removida.*

A definição explícita de qual é a fonte de dados de uma determinada coluna tornase obrigatória quando mais de uma fonte de dados envolvida no select possuí campos com o mesmo nome. Abaixo mostramos um exemplo disso com duas tabelas:

```
SELECT pessoas.codigo,
vendedores.codigo,
vendedores.codpes,
vendedores.nome
FROM pessoas, vendedores
WHERE pessoas.codigo = vendedores.
codpes
ORDER BY vendedores.nome;
```

Nesse exemplo, observe que para toda coluna sendo recuperada, foi especificado o nome da tabela a quem ela pertence. Isso é necessário pois o campo CODIGO existe tanto na tabela PESSOAS como na VENDEDORES.

Recomendo que seja sempre especificado qual é a fonte de dados de cada coluna, pois torna o select mais legível.

Ainda no mesmo exemplo, podemos também atribuir apelidos para as tabelas, desta forma enxugando o código e tornando-o ainda mais legível.

Abaixo segue o mesmo exemplo anterior, só que usando apelidos (aliases):

```
SELECT PES.CODIGO,
VEN.CODIGO,
VEN.CODPES,
VEN.NOME
FROM PESSOAS PES, VENDEDORES VES
WHERE PES.CODIGO = VEN.CODPES
```

```
ORDER BY VEN.NOME;
```

Comentários

Comentários no SQL podem ser representados por

-- (dois sinais de menos, sem espaços) que definem que o restante da linha é um comentário, ou

/* e */ para comentar um trecho dentro do código SQL, que pode inclusive se estender por mais de uma linha.

Exemplo:

```
SELECT *
FROM PESSOAS -- Aqui é comentário
WHERE 1;
```

ou

```
SELECT *
FROM /* comentário */ PESSOAS
WHERE 1;
```

Dica

Os SGBDs citados nesse artigo permitem cálculos diretamente pelo SELECT, sem o uso da cláusula FROM, ou seja o uso do SELECT para obter resultado de cálculos aritméticos básicos. Como no exemplo abaixo:

```
SELECT 10 – 2;
10-2
8
```

Cláusula JOIN

A cláusula JOIN é empregada para permitir que um mesmo select recupere informações de mais de uma fonte de dados (tabelas, views, etc.). Em geral, as tabelas referenciadas possuem algum tipo de relacionamento entre elas, através de um ou mais campos que definam a ligação entre uma tabela e a outra (integridade referencial).

Há duas maneiras de implementar um join:

- A primeira é chamada de **non-ANSI** ou estilo ***theta***, que utiliza a cláusula WHERE para efetuar a junção de tabelas;
- A segunda é chamada de **ANSI Join,** e é baseada no uso da cláusula JOIN propriamente dita.

Simples ligação

Um exemplo de JOIN em estilo *ANSI*:

```
SELECT p.uname, p.nome, a.qtde
from PESSOAS p
CROSS JOIN ACESSOS a;
```

Um exemplo de JOIN em estilo *theta*:

```
SELECT p.uname, p.nome, a.qtde
from PESSOAS p, ACESSOS a;
```

Note que na chamada ANSI utilizamos CROSS JOIN, que é a sintaxe utilizada para recuperar todos os registros das tabelas ligadas, formando um produto cartesiano. É basicamente um INNER JOIN (citado adiante) sem condições.

Tipos de junções
Inner Joins

Somente as linhas/registros que satisfaçam a ligação determinada pelo JOIN serão recuperados pelo select, sendo assim, os registros que **não** se enquadram no relacionamento definido pelo join **não serão recuperados**.

Um exemplo de INNER JOIN em estilo ANSI:

```
SELECT p.uname,
p.nome,
a.qtde
from PESSOAS p
INNER JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

O mesmo JOIN em estilo theta:

```
SELECT p.uname,
p.nome,
a.qtde
from PESSOAS p, ACESSOS a WHERE p.uname = a.pessoa order by p.uname;
```

Left Joins
Através do uso do **LEFT**, todos os registros na tabela à esquerda da query serão listados, independente de terem ou não registros relacionados na tabela à direita. Nesse caso, as colunas relacionadas com a tabela da direita voltam nulos (NULL).

Um exemplo de uso LEFT JOIN:

```
SELECT p.uname,
p.nome,
a.pessoa,
a.qtde
from PESSOAS p
LEFT JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

No exemplo acima, todos os registros da tabela PESSOAS serão listados, independente de terem ou não registros associados na tabela ACESSOS. Caso não existam registros associados na tabela ACESSOS, os campos *a.pessoa* e *a.qtde* retornarão NULL.

Right joins
É o inverso do Left Join, ou seja, todos os registros da tabela à direita serão listados, independente de terem ou não registros relacionados na tabela à esquerda.

Um exemplo de uso RIGHT JOIN:

```
SELECT p.uname,
p.nome,
a.pessoa,
a.qtde
from pessoas p
RIGHT JOIN acessos a on p.uname=a.pessoas order by p.uname;
```

Ou seja, todos os registros da tabela ACESSOS serão listados, e caso não haja correspondentes na tabela PESSOAS, a query devolve NULL para os campos p.uname e p.nome.

Cláusula WHERE

A cláusula WHERE permite aplicar filtros sobre as informações vasculhadas pelo SELECT. É extremamente abrangente e permite gerar condições complexas de pesquisa. Os operadores aceitos pela cláusula WHERE são:
=, >, <, <>, >=, <= e **BETWEEN**.

Há também a possibilidade de se utilizar operadores de proximidade como o LIKE, que permite realizar buscas por apenas uma parte de um string. Pode-se negar uma comparação com o operador **NOT**, bem como utilizar operadores lógicos (**OR** ou **AND**).

Outros operadores suportados são:

• **IN**: verificar se um valor está contido em um grupo de valores;

• **EXISTS**: verifica se um valor existe no resultset retornado por um select;

Vale a pena lembrar que podemos utilizar parênteses, determinando a ordem das condições a serem aplicadas.

NULO ou NÃO NULO, eis a questão...
NULL determina um estado e não um valor, por isso deve ser tratado de maneira especial. Quando queremos saber se um campo é nulo, a comparação a ser feita é "campo **is null**" e não "campo **=** null", sendo que essa última sempre retornará FALSO.

Do mesmo modo, para determinar se um campo não é nulo, usamos "campo **is not null**".

Exemplos de utilização
Adiante temos exemplos de aplicação de WHEREs:

```
SELECT *
from PESSOAS
WHERE (IDADE >= 90);
```

Filtra todos os registros da tabela PESSOAS no qual o campo IDADE é maior ou igual a 90.

```
SELECT *
FROM PESSOAS
WHERE (IDADE >= 1 AND IDADE <= 17);

ou

SELECT *
FROM PESSOAS
WHERE IDADE BETWEEN 1 AND 17;
```

Lista todos os registros da tabela PESSOAS cujo conteúdo do campo IDADE seja maior e igual a 1 e menos e igual a 17.

```
SELECT *
from PESSOAS
WHERE NOME LIKE 'JO%';

ou

SELECT *
from PESSOAS
WHERE NOME STARTING WITH 'JO'; -- No Firebird
```

Retorna todos os campos e todos os registros da tabela pessoa que possuam as letras "JO" como iniciais do nome.

```
SELECT *
FROM PESSOAS
WHERE NOME LIKE '%AN%';

ou

SELECT *
FROM PESSOAS
WHERE NOME CONTAINING 'AN';
```

No caso anterior, retorna todos os registros que possuam no campo nome as letras AN, seja no começo, meio ou fim do campo.

Outro coringa que pode ser utilizado é o "_" (underline). Podemos utilizá-lo quando desejar-mos mascarar uma ou mais letras.

Por exemplo:

```
SELECT *
FROM PESSOAS
WHERE NOME LIKE "LUI_";
```

Nesse caso, obteremos todos os registros cujo campo NOME possui as três primeiras letras LUI, sendo que a quarta letra pode ser qualquer caractere.

```
SELECT *
FROM ACESSOS
WHERE QTDE IS NULL;
```

Retorna todos os campos e todos os registros onde o campo QTDE é NULL.

```
SELECT PE.UNAME,
PE.NOME
FROM PESSOAS PE
WHERE EXISTS(SELECT * from ACESSOS
WHERE PESSOA = PE.UNAME);
```

Retorna os campos UNAME e NOME de todos os registros da tabela PESSOAS que possuem registros na tabela ACESSOS. O uso da cláusula EXISTS é na verdade uma junção com uma subquerie (segundo select). Nesse caso, somente serão exibidos os registros da query que atendam a condição da subquery. Caso um registro na tabela pessoas não possua um registro correspondente na tabela ACESSOS, ele não será recuperado.

```
SELECT *
FROM PESSOAS
WHERE IDADE IN (32,24);
```

Serão retornados os registros cuja idade é 32 ou 24. Poderia ser implementada por um OR, mas quando a lista de campos na condição é extensa, o OR pode deixar a instrução muito longa e de difícil compreensão.

Dica

Pode-se utilizar o NOT em conjunto com praticamente todos os operadores de condição.Exemplo: NOT com o BETWEEN (NOT BETWEEN), com o LIKE (NOT LIKE), com o IN (NOT IN) e outras tantas maneiras. Dessa forma, as possibilidades de filtragem aumentam significativamente.

Cláusula GROUP BY

As cláusulas GROUP BY e HAVING são geralmente utilizadas quando utilizamos funcões de agrupamento. As principais funções de agrupamento são:

SUM Soma todos os valores da coluna informada
MAX Retorna o maior valor da coluna especificada.
MIN Retorna o menor valor da coluna informada.
COUNT Retorna o número de registros da tabela.
AVG Retorna a média aritmética dos valores da coluna informada.

Você Sabia?
Os registros cujo campo é NULL são desprezados quando se utiliza uma função de agregação.

Exemplos:

```
SELECT SUM(QTDE) FROM ACESSOS;
```

Query devolve a somatória do campo QTDE da tabela ACESSOS.

```
SELECT MAX(QTDE) FROM ACESSOS;
```

Query devolve o maior valor do campo QTDE da tabela ACESSOS.

```
SELECT NOME,
COUNT( AC.QTDE)
FROM PESSOAS PE, ACESSOS AC
WHERE PE.UNAME = AC.PESSOA
GROUP BY(AC.PESSOA);
```

Você Sabia?
As funções SUM e AVG somente podem ser utilizadas em campos com tipos numéricos.
As funções MAX e MIN podem ser utilizados em numéricos, data e com caracteres também, e COUNT em campos de qualquer tipo de dado.

Quando um mesmo select recupera tanto campos individuais e valores agrupados através das funções de agrupamento, é necessário utilizar o *group by* listando nele as colunas individuais, para que o select possa ser executado.

Cláusula HAVING
A cláusula HAVING aplica um filtro sobre o resultado de um GROUP BY e, portanto, é utilizada em conjunto com o mesmo. Ela não interfere no resultado obtido na Query, pois age após a totalização, ou seja, não altera resultados de totalizações, apenas filtra o que será exibido. Exemplo:

```
SELECT PESSOA, SUM(QTDE) AS TOTAG
FROM ACESSOS
GROUP BY PESSOA
HAVING TOTAG > 5;

ou

SELECT PESSOA, SUM(QTDE) AS TOTAG
FROM ACESSOS
GROUP BY PESSOA
HAVING SUM(QTDE) > 5;
```

A query acima agrupa os registros pelo campo PESSOA, depois soma os conteúdos do campo QTDE retornando-os com o alias TOTAG (por pessoa), e somente exibe os que possuem TOTAG maior que 5.

Cláusula ORDER BY
Como o próprio nome sugere, essa cláusula ordena informações obtidas em uma query, de forma [ASC]endente (menor para o maior, e da letra A para Z), que é o padrão e pode ser omitida, ou de maneira [DESC]endente. O NULL, por ser um estado e não um valor, aparece antes de todos na ordenação. Alguns

SGBDs permitem especificar se os nulls devem aparecer no início ou fim dos registros, como por exemplo o Firebird, com o uso das cláusulas **NULLS FIRST** e **NULLS LAST**.

Exemplo:

```
PESSOAS
ORDER BY PESSOA;
```

A query acima retorna os registros da tabela PESSOAS ordenados por PESSOA de maneira ascendente. Pode-se ordenar por "n" colunas, ou seja, fazer uma sub-ordenação. Isso é feito indicando os campos separados por vírgula, conforme a query abaixo:

```
SELECT *
FROM ACESSOS
ORDER BY PESSOA, QTDE;
```

Desta maneira, a query retornará o resultado ordenado por PESSOA e sub-ordenado pela QTDE.
Pode-se também montar queries onde se ordena de maneira ascendente uma coluna, e de maneira descendente outra. Exemplo:

```
SELECT *
FROM ACESSOS
ORDER BY PESSOA DESC,QTDE ASC;
```

A query retorna todos os registros da tabela ACESSOS ordenados pelo campo PESSOA de maneira descendente, e pelo campo QTDE na forma ascendente.

O ORDER BY também permite ordenações referenciando as colunas como números. Assim, a primeira coluna retornada no query é a 1, a segunda é a 2, e assim sucessivamente.

Desta forma, podemos escrever códigos conforme o abaixo:

```
SELECT CAMPO1,CAMPO2,CAMPO3,CAMPO4
FROM ACESSOS
ORDER BY 3;
```

No exemplo, a ordenação será através da coluna CAMPO3.

Dicas adicionais

Existem recursos adicionais que podem ser utilizados nos SELECTs. Geralmente são funções e variáveis que podem nos poupar bastante trabalho. Abaixo listo algumas funções interessantes, presentes na maioria dos bancos de dados, sejam de forma nativa, seja na forma de UDFs:

Funções ANSI para tratar Strings:

Lower(campo) - Devolve o conteúdo do campo em minúsculos

Upper(campo) - Devolve o conteúdo do campo em maiúsculos

Character_Length(campo) - Devolve o tamanho ocupado em bytes do campo.

Position(string1 IN string2) - Busca pelo String1 na String2. Se encontrar devolve o offset de posição, do contrário devolve 0 (zero). O LIKE internamente em alguns bancos utiliza essa função.

Concat(String1, String2, ..., StringN) - Devolve um String que é a concatenação dos strings informados.

Exemplo:

```
SELECT CONCAT( UNAME,'-',NOME ) AS
```

```
"NOME" FROM PESSOAS;
```

Funções para implementar rotinas de segurança no banco:

System_user() - Devolve o nome do usuário do sistema operacional para o servidor SQL.

Session_user() - Devolve um string com a autorização da sessão SQL atual.

User() - Devolve o nome do usuário ativo. No Firebird é CURRENT_USER.

Constantes de data e hora:

Current_Date - Devolve a data atual

Current_Time - Devolve a hora atual Existem otimizações que poderiam ser feitas nas tabelas utilizadas em nossos exemplos como, por exemplo, a implementação de índices. Em um próximo artigo iremos falar sobre otimizações do banco.

Diferenças entre SGBDs

Cada SGBD possui sintaxes próprias, que adicionam recursos e facilidades ao SELECT padrão.

Autor - Luiz Paulo de Oliveira Santos na DBFree Magazine 002