10) Filtrando dados nas tabelas

10.1) Filtrando dados com a cláusula where
10.2) Entendendo os operadores Like e Ilike
10.3) Utilizando o Operador Between
10.4) Utilizando o IN
10.5) Cuidados ao fazer Comparações com valores “NULL“
10.6) Utilizando a Cláusula Order By

10.1) Filtrando dados com a cláusula where

Rótulos de coluna

Podem ser atribuídos nomes para as entradas da lista de seleção para processamento posterior. Neste caso, "processamento posterior" é uma especificação opcional de classificação e o aplicativo cliente (por exemplo, os títulos das colunas para exibição). Por exemplo:

```
SELECT a AS valor, b + c AS soma FROM ...
```

Se nenhum nome de coluna de saída for especificado utilizando AS, o sistema atribui um nome padrão. Para referências a colunas simples, é o nome da coluna referenciada. Para chamadas de função, é o nome da função. Para expressões complexas o sistema gera um nome genérico.

> **Nota:** Aqui, o nome dado à coluna de saída é diferente do nome dado na cláusula FROM (consulte a Seção 7.2.1.2). Na verdade, este processo permite mudar o nome da mesma coluna duas vezes, mas o nome escolhido na lista de seleção é o passado adiante.

7.3.3. DISTINCT

Após a lista de seleção ser processada, a tabela resultante pode opcionalmente estar sujeita à remoção das linhas duplicadas. A palavra chave DISTINCT deve ser escrita logo após o SELECT para especificar esta funcionalidade:

```
SELECT DISTINCT lista_de_seleção ...
```

(Em vez de DISTINCT pode ser utilizada a palavra ALL para especificar o comportamento padrão de manter todas as linhas)

Como é óbvio, duas linhas são consideradas distintas quando têm pelo menos uma coluna diferente. Os valores nulos são considerados iguais nesta comparação.

Como alternativa, uma expressão arbitrária pode determinar quais linhas devem ser consideradas distintas:

```
SELECT DISTINCT ON (expressão [, expressão ...]) lista_de_seleção ...
```

Neste caso, expressão é uma expressão de valor arbitrária avaliada para todas as linhas. Um conjunto de linhas para as quais todas as expressões são iguais são consideradas duplicadas, e somente a primeira linha do conjunto é mantida na saída. Deve ser observado que a "primeira linha" de um conjunto é imprevisível, a não ser que a consulta seja ordenada por um número suficiente de colunas para garantir a ordem única das linhas que chegam no filtro DISTINCT (o processamento de DISTINCT ON ocorre após a ordenação do ORDER BY).

A cláusula DISTINCT ON não faz parte do padrão SQL, sendo algumas vezes considerada um estilo ruim devido à natureza potencialmente indeterminada de seus resultados. Utilizando-se adequadamente GROUP BY e subconsultas no FROM esta construção pode ser evitada, mas geralmente é a alternativa mais fácil.

A cláusula WHERE

A sintaxe da *Cláusula* WHERE é

```
WHERE condição_de_pesquisa
```

onde a condição_de_pesquisa é qualquer expressão de valor (consulte a Seção 4.2) que retorne um valor do tipo boolean.

Após o processamento da cláusula FROM ter sido feito, cada linha da tabela virtual derivada é verificada com relação à condição de pesquisa. Se o resultado da condição for verdade, a linha é mantida na tabela de saída, senão (ou seja, se o resultado for falso ou nulo) a linha é desprezada. Normalmente a condição de pesquisa faz referência a pelo menos uma coluna da tabela gerada pela cláusula FROM; embora isto não seja requerido, se não for assim a cláusula WHERE não terá utilidade.

> **Nota:** A condição de junção de uma junção interna pode ser escrita tanto na cláusula WHERE quanto na cláusula JOIN. Por exemplo, estas duas expressões de tabela são equivalentes:
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> e
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> ou talvez até mesmo
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Qual destas formas deve ser utilizada é principalmente uma questão de estilo. A sintaxe do JOIN na cláusula FROM provavelmente não é muito portável para outros sistemas gerenciadores de banco de dados SQL. Para as junções externas não existe escolha em nenhum caso: devem ser feitas na cláusula FROM. A cláusula ON/USING da junção externa *não* é equivalente à condição WHERE, porque determina a adição de linhas (para as linhas de entrada sem correspondência) assim como a remoção de linhas do resultado final.

Abaixo estão mostrados alguns exemplos de cláusulas WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

sendo que fdt é a tabela derivada da cláusula FROM. As linhas que não aderem à condição de pesquisa da cláusula WHERE são eliminadas de fdt. Deve ser observada a utilização de subconsultas escalares como expressões de valor. Assim como qualquer outra consulta, as subconsultas podem utilizar expressões de tabela complexas. Deve ser observado, também, como fdt é referenciada nas subconsultas. A qualificação de c1 como fdt.c1 somente será necessária se c1 também for o nome de uma coluna na tabela de entrada derivada da subconsulta. Entretanto, a qualificação do nome da coluna torna mais clara a consulta, mesmo quando não é necessária. Este exemplo mostra como o escopo do nome da coluna de uma consulta externa se estende às suas consultas internas.

As cláusulas GROUP BY e HAVING

Após passar pelo filtro WHERE, a tabela de entrada derivada pode estar sujeita ao agrupamento, utilizando a cláusula GROUP BY, e à eliminação de grupos de linhas, utilizando a cláusula HAVING.

```
SELECT lista_de_seleção
    FROM ...
    [WHERE ...]
    GROUP BY referência_a_coluna_de_agrupamento [,
referência_a_coluna_de_agrupamento]...
```

A *Cláusula* GROUP BY é utilizada para agrupar linhas da tabela que compartilham os mesmos valores em todas as colunas da lista. Em que ordem as colunas são listadas não faz diferença. O efeito é combinar cada conjunto de linhas que compartilham valores comuns em uma linha de grupo que representa todas as linhas do grupo. Isto é feito para eliminar redundância na saída, e/ou para calcular agregações aplicáveis a estes grupos. Por exemplo:

```
=> SELECT * FROM teste1;

 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 linhas)

=> SELECT x FROM teste1 GROUP BY x;
```

```
 x
---
 a
 b
 c
(3 linhas)
```

Na segunda consulta não poderia ser escrito SELECT * FROM teste1 GROUP BY x, porque não existe um valor único da coluna y que poderia ser associado com cada grupo. As colunas agrupadas podem ser referenciadas na lista de seleção, desde que possuam um valor único em cada grupo.

De modo geral, se uma tabela for agrupada as colunas que não são usadas nos agrupamentos não podem ser referenciadas, exceto nas expressões de agregação. Um exemplo de expressão de agregação é:

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x;

 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 linhas)
```

Aqui sum() é a função de agregação que calcula um valor único para o grupo todo. Mais informações sobre as funções de agregação disponíveis podem ser encontradas na Seção 9.15.

> **Dica:** Um agrupamento sem expressão de agregação computa, efetivamente, o conjunto de valores distintas na coluna. Também poderia ser obtido por meio da cláusula DISTINCT (consulte a Seção 7.3.3).

Abaixo está mostrado um outro exemplo: cálculo do total das vendas de cada produto (e não o total das vendas de todos os produtos).

```
SELECT cod_prod, p.nome, (sum(v.unidades) * p.preco) AS vendas
    FROM produtos p LEFT JOIN vendas v USING (cod_prod)
    GROUP BY cod_prod, p.nome, p.preco;
```

Neste exemplo, as colunas cod_prod, p.nome e p.preco devem estar na cláusula GROUP BY, porque são referenciadas na lista de seleção da consulta (dependendo da forma exata como a tabela produtos for definida, as colunas nome e preço podem ser totalmente dependentes da coluna cod_prod, tornando os agrupamentos adicionais teoricamente desnecessários, mas isto ainda não está implementado). A coluna v.unidades não precisa estar na lista do GROUP BY, porque é usada apenas na expressão de agregação (sum(...)), que representa as vendas do produto. Para cada produto, a consulta retorna uma linha sumarizando todas as vendas do produto.

No SQL estrito, a cláusula GROUP BY somente pode agrupar pelas colunas da tabela de origem, mas o PostgreSQL estende esta funcionalidade para permitir o GROUP BY agrupar pelas colunas da lista de seleção. O agrupamento por expressões de valor, em vez de nomes simples de colunas, também é permitido.

Se uma tabela for agrupada utilizando a cláusula GROUP BY, mas houver interesse em alguns grupos apenas, pode ser utilizada a cláusula HAVING, de forma parecida com a cláusula WHERE, para eliminar grupos da tabela agrupada. A sintaxe é:

```
SELECT lista_de_seleção FROM ... [WHERE ...] GROUP BY ... HAVING
expressão_booleana
```

As expressões na cláusula HAVING podem fazer referência tanto a expressões agrupadas quanto a não agrupadas (as quais necessariamente envolvem uma função de agregação).

Exemplo:

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x HAVING sum(y) > 3;

 x | sum
---+-----
 a |   4
 b |   5
(2 linhas)

=> SELECT x, sum(y) FROM teste1 GROUP BY x HAVING x < 'c';

 x | sum
---+-----
 a |   4
 b |   5
(2 linhas)
```

Agora vamos fazer um exemplo mais próximo da realidade:

```
SELECT cod_prod, p.nome, (sum(v.unidades) * (p.preco - p.custo)) AS lucro
    FROM produtos p LEFT JOIN vendas v USING (cod_prod)
    WHERE v.data > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY cod_prod, p.nome, p.preco, p.custo
    HAVING sum(p.preco * v.unidades) > 5000;
```

No exemplo acima, a cláusula WHERE está selecionando linhas por uma coluna que não é agrupada (a expressão somente é verdadeira para as vendas feitas nas quatro últimas semanas, enquanto a cláusula HAVING restringe a saída aos grupos com um total de vendas brutas acima de 5000. Deve ser observado que as expressões de agregação não precisam ser necessariamente as mesmas em todas as partes da consulta.

Exemplo 7-1. Utilização de HAVING sem GROUP BY no SELECT

O exemplo abaixo mostra a utilização da cláusula HAVING sem a cláusula GROUP BY no comando SELECT. É criada a tabela produtos e são inseridas cinco linhas. Quando a cláusula HAVING exige a presença de mais de cinco linhas na tabela, a consulta não retorna nenhuma linha. [4] [5]

```
=> create global temporary table produtos(codigo int, valor float);
=> insert into produtos values (1, 102);
=> insert into produtos values (2, 104);
=> insert into produtos values (3, 202);
=> insert into produtos values (4, 203);
=> insert into produtos values (5, 204);

=> select avg(valor) from produtos;

 avg
-----
 163
(1 linha)
```

```
=> select avg(valor) from produtos having count(*)>=5;

 avg
-----
 163
(1 linha)

=> select avg(valor) from produtos having count(*)=5;

 avg
-----
 163
(1 linha)

=> select avg(valor) from produtos having count(*)>5;

 avg
-----
(0 linhas)
```

10.2) Entendendo os operadores Like e Ilike

Correspondência com padrão

O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão: o operador LIKE tradicional do SQL; o operador mais recente SIMILAR TO (adicionado ao SQL:1999); e as expressões regulares no estilo POSIX. Além disso, também está disponível a função de correspondência com padrão substring, que utiliza expressões regulares tanto no estilo SIMILAR TO quanto no estilo POSIX.

> **Dica:** Havendo necessidade de correspondência com padrão acima destas, deve ser considerado o desenvolvimento de uma função definida pelo usuário em Perl ou Tcl.

9.7.1. LIKE

```
cadeia_de_caracteres LIKE padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT LIKE padrão [ESCAPE caractere_de_escape]
```

Cada padrão define um conjunto de cadeias de caracteres. A expressão LIKE retorna verdade se a cadeia_de_caracteres estiver contida no conjunto de cadeias de caracteres representado pelo padrão; como esperado, a expressão NOT LIKE retorna falso quando LIKE retorna verdade, e vice-versa, e a expressão equivalente é NOT (cadeia_de_caracteres LIKE padrão).

Quando o padrão não contém os caracteres percentagem ou sublinhado, o padrão representa apenas a própria cadeia de caracteres; neste caso LIKE atua como o operador igual. No padrão o caractere sublinhado (_) representa (corresponde a) qualquer um único caractere; o caractere percentagem (%) corresponde a qualquer cadeia com zero ou mais caracteres.

Alguns exemplos:

```
'abc' LIKE 'abc'    verdade
'abc' LIKE 'a%'     verdade
'abc' LIKE '_b_'    verdade
'abc' LIKE 'c'      falso
```

A correspondência com padrão LIKE sempre abrange toda a cadeia de caracteres. Para haver correspondência com o padrão em qualquer posição da cadeia de caracteres, o padrão deve começar e terminar pelo caractere percentagem.

Para corresponder ao próprio caractere sublinhado ou percentagem, sem corresponder a outros caracteres, estes caracteres devem ser precedidos pelo caractere de escape no padrão. O caractere de escape padrão é a contrabarra, mas pode ser definido um outro caractere através da cláusula ESCAPE. Para corresponder ao próprio caractere de escape, devem ser escritos dois caracteres de escape.

Deve ser observado que a contrabarra também possui significado especial nos literais cadeias de caracteres e, portanto, para escrever em uma constante um padrão contendo uma contrabarra devem ser escritas duas contrabarras no comando SQL. Assim sendo, para escrever um padrão que corresponda ao literal contrabarra é necessário escrever quatro contrabarras no comando, o que pode ser evitado escolhendo um caractere de escape diferente na cláusula ESCAPE; assim a contrabarra deixa de ser um caractere especial para o LIKE (Mas continua sendo especial para o analisador de literais cadeias de caracteres e, por isso, continuam sendo necessárias duas contrabarras).

Também é possível fazer com que nenhum caractere sirva de escape declarando ESCAPE ''. Esta declaração tem como efeito desativar o mecanismo de escape, tornando impossível anular o significado especial dos caracteres sublinhado e percentagem no padrão.

Alguns exemplos: (N. do T.)

```
-- Neste exemplo a contrabarra única é consumida pelo analisador de literais
-- cadeias de caracteres, e não anula o significado especial do _

=> SELECT tablename FROM pg_tables WHERE tablename LIKE '%g\_o%' ESCAPE '\\';

  tablename
--------------
 pg_group
 pg_operator
 pg_opclass
 pg_largeobject

-- Neste exemplo somente uma das duas contrabarras é consumida pelo analisador
-- de literais cadeias de caracteres e, portanto, a segunda contrabarra anula
-- o significado especial do _

=> SELECT tablename FROM pg_tables WHERE tablename LIKE '%g\\_o%' ESCAPE '\\';

 tablename
-----------
 pg_operator
 pg_opclass

-- No Oracle não são necessárias duas contrabarras, como mostrado abaixo:
```

```
SQL> SELECT view_name FROM all_views WHERE view_name LIKE 'ALL\_%' ESCAPE '\';
```

Pode ser utilizada a palavra chave ILIKE no lugar de LIKE para fazer a correspondência não diferenciar letras maiúsculas de minúsculas, conforme o idioma ativo. [1] Isto não faz parte do padrão SQL, sendo uma extensão do PostgreSQL.

O operador ~~ equivale ao LIKE, enquanto ~~* corresponde ao ILIKE. Também existem os operadores !~~ e !~~*, representando o NOT LIKE e o NOT ILIKE respectivamente. Todos estes operadores são específicos do PostgreSQL.

9.7.2. Expressões regulares do SIMILAR TO

```
cadeia_de_caracteres SIMILAR TO padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT SIMILAR TO padrão [ESCAPE caractere_de_escape]
```

O operador SIMILAR TO retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL. As expressões regulares do padrão SQL são um cruzamento curioso entre a notação do LIKE e a notação habitual das expressões regulares.

Da mesma forma que o LIKE, o operador SIMILAR TO somente é bem-sucedido quando o padrão corresponde a toda cadeia de caracteres; é diferente do praticado habitualmente nas expressões regulares, onde o padrão pode corresponder a qualquer parte da cadeia de caracteres. Também como o LIKE, o operador SIMILAR TO utiliza _ e % como caracteres curinga, representando qualquer um único caractere e qualquer cadeia de caracteres, respectivamente (são comparáveis ao . e ao .* das expressões regulares POSIX).

Além destas funcionalidades pegas emprestada do LIKE, o SIMILAR TO suporta os seguintes metacaracteres para correspondência com padrão pegos emprestado das expressões regulares POSIX:

- | representa alternância (uma das duas alternativas).
- * representa a repetição do item anterior zero ou mais vezes.
- + representa a repetição do item anterior uma ou mais vezes.
- Os parênteses () podem ser utilizados para agrupar itens em um único item lógico.
- A expressão de colchetes [...] especifica uma classe de caracteres, do mesmo modo que na expressão regular POSIX.

Deve ser observado que as repetições limitadas (? e {...}) não estão disponíveis, embora existam no POSIX. Além disso, o ponto (.) não é um metacaractere.

Da mesma forma que no LIKE, a contrabarra desativa o significado especial de qualquer um dos metacaracteres; ou pode ser especificado um caractere de escape diferente por meio da cláusula ESCAPE.

Alguns exemplos:

```
'abc' SIMILAR TO 'abc'      verdade
'abc' SIMILAR TO 'a'        falso
'abc' SIMILAR TO '%(b|d)%'  verdade
'abc' SIMILAR TO '(b|c)%'   falso
```

A função substring com três parâmetros, substring(cadeia_de_caracteres FROM padrão FOR caractere_de_escape), permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular SQL:1999. Assim como em SIMILAR TO, o padrão especificado deve corresponder a toda a cadeia de caracteres, senão a função falha e retorna nulo. Para indicar a parte do padrão que deve ser retornada em caso de sucesso, o padrão deve conter duas ocorrências do caractere de escape seguidas por aspas ("). É retornado o texto correspondente à parte do padrão entre estas marcas.

Alguns exemplos:

```
substring('foobar' FROM '%#"o_b#"%' FOR '#')    oob
substring('foobar' FROM '#"o_b#"%' FOR '#')     NULL
```

10.3) Utilizando o Operador Between

Operadores de comparação

Estão disponíveis os operadores de comparação habituais, conforme mostrado na Tabela 9-1.

Tabela 9-1. Operadores de comparação

Operador	Descrição
<	menor
>	maior
<=	menor ou igual
>=	maior ou igual
=	igual
<> ou !=	diferente

Nota: O operador != é convertido em <> no estágio de análise. Não é possível implementar os operadores != e <> realizando operações diferentes.

Os operadores de comparação estão disponíveis para todos os tipos de dado onde fazem sentido. Todos os operadores de comparação são operadores binários, que retornam valores do tipo boolean; expressões como 1 < 2 < 3 não são válidas (porque não existe o operador < para comparar um valor booleano com 3).

Além dos operadores de comparação, está disponível a construção especial BETWEEN.

```
a BETWEEN x AND y
```

equivale a

```
a >= x AND a <= y
```

Analogamente,

```
a NOT BETWEEN x AND y
```

equivale a

```
a < x OR a > y
```

Não existe diferença entre as duas formas, além dos ciclos de CPU necessários para reescrever a primeira forma na segunda internamente.

Para verificar se um valor é nulo ou não, são usadas as construções

```
expressão IS NULL
expressão IS NOT NULL
```

ou às construções equivalentes, mas fora do padrão,

```
expressão ISNULL
expressão NOTNULL
```

Não deve ser escrito expressão = NULL, porque NULL não é "igual a" NULL (O valor nulo representa um valor desconhecido, e não se pode saber se dois valores desconhecidos são iguais). Este comportamento está de acordo com o padrão SQL.

> **Dica:** Alguns aplicativos podem (incorretamente) esperar que expressão = NULL retorne verdade se o resultado da expressão for o valor nulo. É altamente recomendado que estes aplicativos sejam modificadas para ficarem em conformidade com o padrão SQL. Entretanto, se isto não puder ser feito, está disponível a variável de configuração transform_null_equals. Quando transform_null_equals está ativado, o PostgreSQL converte as cláusulas x = NULL em x IS NULL. Este foi o comportamento padrão do PostgreSQL nas versões de 6.5 a 7.1.

O resultado dos operadores de comparação comuns é nulo (significando "desconhecido"), quando algum dos operandos é nulo. Outra forma de fazer comparação é com a construção IS DISTINCT FROM:

```
expressão IS DISTINCT FROM expressão
```

Para expressões não-nulas é o mesmo que o operador <>. Entretanto, quando as duas expressões são nulas retorna falso, e quando apenas uma expressão é nula retorna verdade. Portanto, atua efetivamente como se nulo fosse um valor de dado normal, em vez de "desconhecido".

Os valores booleanos também podem ser testados utilizando as construções

```
expressão IS TRUE
expressão IS NOT TRUE
expressão IS FALSE
expressão IS NOT FALSE
expressão IS UNKNOWN
expressão IS NOT UNKNOWN
```

Estas formas sempre retornam verdade ou falso, e nunca o valor nulo, mesmo quando o operando é nulo. A entrada nula é tratada como o valor lógico "desconhecido". Deve ser observado que IS UNKNOWN e IS NOT UNKNOWN são efetivamente o mesmo que IS NULL e IS NOT NULL, respectivamente, exceto que a expressão de entrada deve ser do tipo booleana.

10.4) Utilizando o IN

IN

```
expressão IN (subconsulta)
```

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada uma linha igual na subconsulta. O resultado é "falso" se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Deve ser observado que, se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

```
construtor_de_linha IN (subconsulta)
```

O lado esquerdo desta forma do IN é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada uma linha igual na subconsulta. O resultado é "falso" se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Da maneira usual, os valores nulos nas linhas são combinados de acordo com com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do IN será nulo.

Exemplo 9-17. Utilização das cláusulas CASE e IN juntas

Este exemplo é idêntico ao Exemplo 9-16, só que utiliza a cláusula IN para executar a consulta, conforme mostrado abaixo. [2]

```
SELECT nome, CASE WHEN nome IN (SELECT nome FROM frutas)
                  THEN 'sim'
                  ELSE 'não'
             END AS fruta
FROM alimentos;
```

Abaixo está mostrado o resultado da execução do script.

```
   nome     | fruta
------------+-------
 maçã       | sim
 espinafre  | não
```

NOT IN

```
expressão NOT IN (subconsulta)
```

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado de NOT IN é "verdade" se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se for encontrada alguma linha igual.

Deve ser observado que se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção NOT IN será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

```
construtor_de_linha NOT IN (subconsulta)
```

O lado esquerdo desta forma do NOT IN é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se for encontrada alguma linha igual.

Da maneira usual, os valores nulos nas linhas são combinados de acordo com com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do NOT IN será nulo.

10.5) Cuidados ao fazer Comparações com valores “NULL“

OPERADOR NULL

NULLIF

Função que atualiza campos com valor NULL.

```
update produtos
   set codigo_fornecedor = NULLIF (codigo_fornecedor, 3);
```

Semelhante a:

```
if codigo_fornecedor = 3 then NULL
elseif codigo_fornecedor != 3 then  codigo_fornecedor= codigo_fornecedor
end
```

COALESCE

Função tipo CASE que testa valores diferentes de NULL e retorna o primeiro não NULL.

```
select codigo, nome, codigo_fornecedor
    coalesce (codigo_fornecedor, codigo)
    from produtos;
```

Similar a:

```
if exists (codigo_fornecedor) then retorne  codigo_fornecedor
elseif not exists (codigo_fornecedor) then retorne  codigo
end
```

Caso o primeiro for NULL procurará retornar o segundo. Se o segundo for NULL procurará retornar o terceiro até o último.

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 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.

10.6) Utilizando a Cláusula Order By

ORDER BY - Ordena o resultado da consulta por um ou mais campos em ordem ascendente (ASC, default) ou descendente (DESC).

Exemplos:
ORDER BY cliente; -- pelo cliente e ascendente
ORDER BY cliente DESC; -- descendente
ORDER BY cliente, quantidade; -- pelo cliente e sub ordenado pela quantidade
ORDER BY cliente DESC, quant ASC;

No exemplo ordenando por dois campos:

SELECT * FROM pedidos ORDER BY cliente, quantidade;

A saída ficaria algo como:

Antônio – 1
Antônio – 2
João - 1
Pedro - 1
Pedro - 2

Combinação de NULL com FALSE, TRUE, NULL e NOT NULL:

select null and true - gera null

select null and false - gera false

select null and null - gera null

select null and not null - gera null

Referência: Livro SQL Curso Prático - Celso Henrique Poderoso de Oliveira
(Muito bom livro sobre a linguagem SQL)

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**".

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

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

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

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**.