Explorando o SELECT

Explore as possibilidades do comando mais importante da linguagem SQL

Carlos Henrique Cantu – www.firebase.com.br

Para os que vieram de bancos *desktops* (ex: Clipper, dBase, Paradox, etc.) e nunca utilizaram a linguagem SQL, o primeiro contato com o comando SELECT pode parecer um tanto quanto assustador. Afinal, como um simples comando pode substituir as incontáveis linhas de comando e *loops* necessários para percorrer tabelas correlacionadas e recuperar suas informações?

O *select* é, com certeza, um dos comandos mais poderosos da SQL, e provavelmente o mais utilizado! Acostumados com sua forma mais básica de utilização, muitos usuários desconhecem o imenso poder "escondido" nesse comando através das inúmeras cláusulas e variações disponíveis.

Esse artigo será inteiramente dedicado ao comando SELECT. Analisaremos sua construção e a maioria das cláusulas disponíveis na sintaxe suportada pelo Firebird, apresentada abaixo:

```
<select statement> ::= <select expression> [FOR UPDATE] [WITH LOCK]

<select expression> ::=
<query specification> [UNION [{ALL | DISTINCT}] <query specification>]

<query specification> ::=
SELECT [FIRST <value>] [SKIP <value>] <select list>
FROM <table expression list>
WHERE <search condition>
GROUP BY <group value list>
HAVING <group condition>
PLAN <plan item list>
ORDER BY <sort value list>
ROWS <value> [TO <value>]

<table expression> ::=
<table name> | <joined table> | <derived table>

<joined table> ::=
{<cross join> | <qualified join>}

<cross join> ::=
<table expression> CROSS JOIN <table expression>

<qualified join> ::=
<table expression> [{INNER | {LEFT | RIGHT | FULL} [OUTER]}] JOIN <table expression>
ON <join condition>

<derived table> ::=
'(' <select expression> ')'
```

Ou resumidamente:

```
SELECT [FIRST (m)] [SKIP (n)] [[ALL] | DISTINCT]
<lista de colunas> [, [nome da coluna] | expressão |constante ] AS nome-alias]
FROM <tabela-procedure-view >
[{[[INNER] | [{LEFT | RIGHT | FULL} [OUTER]] JOIN}] <table-or-procedure-or-view>
 ON <join-conditions [{JOIN..]]
[WHERE <search-conditions>]
[GROUP BY <grouped-column-list>]
[HAVING <search-condition>]
[UNION <select-expression> [ALL]]
[PLAN <plan-expression>]
[ORDER BY <column-list>]
[ROWS <expr1> [TO <expr2>]]
[FOR UPDATE [OF col1 [,col2..]] [WITH LOCK]]
```

Vale lembrar que as cláusulas apresentadas entre colchetes são opcionais, e que a ordem em que elas aparecem no comando é importante.

Começaremos analisando as cláusulas que interferem no que será incluído ou excluído no resultado gerado pela instrução.

ALL

ALL significa que o *select* deverá retornar todas as linhas recuperadas. Esse é o comportamento padrão de um *select*, portanto sua menção explícita pode ser dispensada.

DISTINCT

A cláusula DISTINCT remove as linhas repetidas que foram recuperadas pelo *select*, portanto, se duas ou mais linhas apresentarem os mesmos valores nos campos recuperados, DISTINCT fará com que apenas uma delas seja apresentada no resultado.

Imagine uma tabela de clientes, onde você deseja obter uma lista das cidades onde os clientes estão localizados. O comando "*select cidade from clientes order by cidade*" provavelmente retornaria inúmeras linhas repetidas, pois vários clientes residem em uma mesma cidade. Se utilizarmos a cláusula *distinct*, todas as cidades serão listadas, mas sem repetição, ex: "*select **distinct** cidade from clientes order by cidade*".

Uma observação importante é que quando há um índice para o campo em que se deseja aplicar o DISTINCT, pode ser mais interessante usar o GROUP BY ao invés do DISTINCT, visto que o GROUP BY poderá usar esse índice para realizar o agrupamento, deixando a operação mais rápida, enquanto o DISTINCT não usa índice.

FIRST e SKIP

First determina quantas linhas devem ser retornadas do total de registros obtidos pelo *select*. O parâmetro *skip* permite informar quantas linhas deverão ser "puladas" antes de começar a retornar os registros. Veja alguns exemplos abaixo:

```
select first 5 codigo, nome,endereco from clientes order by codigo;
```

ID	NOME	ENDERECO
1	RODDY SLAVEN SOUTH	OSTEOPATHY
2	JULIE PITTARELLI	NORTH TORREY PINES ROAD
3	MELL SLOAN	BOX
4	GERARDO HALLETT	GASTON AVENUE
5	DANIELLE LEDFORD	HILLSIDE STREET

O comando acima retornaria os cinco primeiros clientes (considerando a ordenação por código – note o *order by*) do *resultset* obtido pelo *select*.

```
select first 5 skip 5 codigo,nome,endereco from clientes order by codigo;
```

ID	NOME	ENDERECO
6	CHUCK WOLINSKY	SETON DRIVE
7	PAUL BANDY	EAST PARK AVENUE
8	MARI REYNOLDS-COBB	MEDICAL BOULEVARD
9	SHAWN ABRAMS	SAWYER ROAD
10	ILKA CHANCE	NORTH ESPLANADE STREET

O comando anterior retornaria os próximos cinco clientes, descartando-se os cinco primeiros, considerando-se a ordenação pelo campo *codigo*.

Fica claro que um dos usos mais comuns das cláusulas FIRST e SKIP é o de paginar dados, sendo especialmente útil para quem desenvolve aplicações para *web*.

Note que é incorreto pensar que usando *first*/*skip* sempre resultará em um ganho de performance, se comparado ao mesmo comando sendo executado sem essas cláusulas. É óbvio que limitando o número de registros retornados, o tempo gasto para trafegar as informações do servidor para o cliente será menor. No entanto, dependendo do plano de acesso gerado pelo *select*, pode ser necessário que o Firebird processe toda a *query* antes de retornar a quantidade de registros desejada, o que acontece, por exemplo, quando o Firebird precisa fazer um SORT dos registros.

Dica

Sempre que utilizar FIRST e/ou SKIP, lembre-se de utilizar um critério de ordenação através da cláusula ORDER BY.

Colunas retornadas

Nos exemplos mostrados até agora, utilizamos o "*" (asterisco) para fazer com que o *select* retorne todas as colunas (campos) das tabelas acessadas, mas nem sempre isso é desejado. Na maioria das vezes, só nos interessa algumas colunas específicas. Podemos indicar ao *select* quais colunas devem ser retornadas (<lista de colunas>). Indo mais além, podemos retornar colunas em tempo de execução, resultado de fórmulas, expressões, UDFs, constantes ou mesmo parâmetros.

A seguir, veremos alguns exemplos de *selects* retornando informações variadas.

```
/* retorna a data e hora atual do sistema */
select current_timestamp
from rdb$database; -- rdb$database é uma tabela de sistema do FB

/* retorna apenas o código e o nome dos clientes */
select codigo, nome
from clientes order by codigo;

/* concatena o campo primeiro nome e sobrenome. O "order by 1" faz com que o resultado
seja ordenado pelo resultado dessa concatenção */
select primeiro_nome || sobrenome AS NOME_COMPLETO
from pessoas order by 1;

/* retorna o total de clientes cadastrados, chamando a coluna de resultado de TOTAL */
select count(*) AS TOTAL from clientes;

/* retorna o maior salario da tabela funcionarios */
select max(salario) as MAIOR_SALARIO from funcionarios;

/* retorna o nome completo de todas as pessoas que tenham o primeiro nome igual ao valor
do parâmetro :nome */
select primeiro_nome || sobrenome AS NOME_COMPLETO
from pessoas
where primeiro_nome = :nome
order by 1;
```

Exclusividade do Firebird 2.0

A partir do Firebird 2.0, é possível forçar a definição do tipo de dado de um parâmetro através do CAST, por exemplo:

*select * from pessoas where nome = cast(:nome as varchar(50));*

É interessante nomear as colunas resultantes de expressões ou constantes. Isso foi feito nos exemplos anteriores através do uso da cláusula AS. Caso não seja explicitamente nomeada, o Firebird atribuirá um nome automático, quando necessário.

Podemos definir uma coluna como sendo resultado de um outro *select*, chamado de *subquery*. O exemplo a seguir mostra um *select* que retornará a quantidade de *notas fiscais* já emitidas para cada cliente cadastrado:

```
select c.nome,
       (select count(*) from notas n
        where n.codcliente = c.codigo) AS total_notas
from clientes c;
```

NOME	TOTAL_NOTAS
RODDY SLAVEN	1
JULIE PITTARELLI	1
MELL SLOAN	0
GERARDO HALLETT	1
DANIELLE LEDFORD	1
...	

Importante

Colunas formadas pelo resultado de expressões, agregações, *subselects* ou por constantes são por natureza *readonly*, pois são "calculadas" em tempo de execução.

FROM *tabela-procedure-view-select*

É através do ***from*** que definimos a fonte principal de informações para o *select*. Podemos ter como fonte de dados para o *select* uma tabela, uma *view* ou mesmo uma *stored procedure selecionável*. O Firebird >= 2.0 permite que tenhamos um outro *select* como fonte de dados (chamado de **Tabelas Derivadas**).

Apensar de não recomendado, é possível indicar várias fontes de dados na cláusula *from*, separando-as por vírgula. No caso, o resultado obtido é um produto cartesiano dos registros dessas tabelas, exemplo:

```
select *
from produtos, categoria
```

O Firebird também permite que você faça o relacionamento entre as tabelas listadas no *from* através da cláusula *where* do *select* (que discutiremos mais a frente):

```
select *
from produtos p, categoria c
where p.categoria_id = c.id;
```

No exemplo acima, estaríamos obtendo uma lista composta por todos os registros das tabelas *produtos* e *categoria*, onde houvesse um relacionamento entre o campo *categ* da tabela *produtos* e o campo *id* da tabela *categoria*.

As letras “p” e “c” vistas logo após os nomes das tabelas definem seus “**apelidos**”. Isso facilita bastante a leitura e a clareza de um *select*, quando ele é muito extenso. Podemos indicar à qual tabela um campo referenciado pertence, utilizando o *alias* ou o nome da tabela seguido por “.” (ponto) e o nome do campo (como mostrado na cláusula where do exemplo anterior).

Importante!

Até a versão 1.5 do Firebird, o servidor permitia a utilização mista dos apelidos e nomes de tabelas na especificação das colunas, como no exemplo abaixo:

```
select c.codigo, cliente.nome
from clientes c;
```

A partir do Firebird 2.0, o comando acima gera um erro, pois não é mais permitido referenciar uma mesma tabela pelo seu apelido e pelo seu nome, em uma mesma query. Algumas variações do exemplo anterior, adaptadas para o Firebird 2, podem ser vistas a seguir:

```
select c.codigo, c.nome from clientes c;
select codigo, c.nome from clientes c;
select c.codigo, nome from clientes c;
select codigo, nome from clientes c;
select codigo, nome from clientes;
```

Em suma, se utilizarmos o apelido de uma tabela na definição de uma coluna do *select*, não podemos utilizar o nome dessa tabela na definição de uma outra coluna nesse mesmo *select*.

JOINs

O *join* faz relacionamentos entre tabelas em um mesmo *select*, permitindo retornar colunas obtidas de qualquer uma dessas tabelas, em um único *resultset*. É muito útil, por exemplo, para gerar dados para relatórios. Existem vários tipos de JOINs, a saber: INNER JOIN, OUTER JOINs (LEFT, RIGHT, FULL) e CROSSJOIN.

O JOIN é acompanhado da cláusula ON, que determina a expressão que relaciona as tabelas envolvidas. A informação vinda de cada tabela é chamada de *stream*. Veja o exemplo abaixo:

```
select  c.nome, n.nota_fiscal
from clientes c
join notas n on n.codcliente = c.codigo;
```

NOME	NOTA_FISCAL
ALISON ECHOLS	40001
KATHRYN MORGAN	40005
PAGE KIRK	40010
MARVIN HALLIWELL	40013
LAMAR SNYDER	40015

...

No exemplo anterior, o *select* irá retornar o(s) nome(s) do(s) cliente(s) e as notas fiscais já emitidas para cada um deles. O relacionamento foi criado ligando as tabelas *clientes* e *notas* através dos campos *codclient* e *codigo*.

INNER JOIN

O INNER JOIN (ou *exclusive join – junção exclusiva*) pode ser representado omitindo-se a cláusula INNER, ou seja, usando apenas JOIN (como no exemplo anterior). Ele fará o relacionamento entre os *streams* baseando-se na condição determinada pela expressão definida na cláusula ON. No exemplo anterior, o relacionamento é baseado no *código do cliente*. Por se tratar de um *inner join*, os clientes que nunca tiveram notas fiscais emitidas seriam descartados do resultado do *select*, ou seja, os registros serão incluídos no resultado do *select* somente se houver pelo menos uma ocorrência encontrada baseada na condição especificada pelo *join*.

OUTER JOINs

Um ***outer join*** retorna registros mesmo quando a condição especificada não é satisfeita, sendo que nesse caso, os campos pertencentes ao *stream* onde não houve ocorrência de registro serão retornados como NULL. É importante saber que o *outer join* é formado a partir de dois *streams* (direito e esquerdo). O *stream* esquerdo é o que precede o *join*. O *stream* direito, também chamado de argumento, refere-se ao *stream* especificado após o *join*. Em situações em que vários *joins* são utilizados, o argumento de um *join* pode ser o *stream* esquerdo de outro *join*, e assim por diante.

Existem três tipos de OUTER JOINs: LEFT, RIGHT e FULL. Em todos os casos, a palavra OUTER pode ser omitida na construção do comando (é opcional).

LEFT [OUTER] JOIN

O *left join* retorna todas as linhas do *stream* esquerdo, independente da condição especificada ter sido atendida. Veja o exemplo:

```
select c.nome, n.nota_fiscal
from clientes c
LEFT join notas n on n.codcliente = c.codigo;
```

```
NOME                NOTA_FISCAL
RODDY SLAVEN        55865
JULIE PITTARELLI    46147
MELL SLOAN          null
GERARDO HALLETT     51107
DANIELLE LEDFORD    43203
CHUCK WOLINSKY      55972
PAUL BANDY          46495
MARI REYNOLDS-COBB  null
...
```

O comando acima listará todos os clientes, independente de já possuírem uma nota fiscal emitida ou não. Quando não possuírem, a coluna *numero_nota* ficará nula.

RIGHT [OUTER] JOIN

Ao contrário do *left join*, o *right join* retorna todas as linhas do *argumento* do *join*, independente da condição especificada ter sido atendida. Veja o exemplo:

```
select c.nome, n.nota_fiscal
from clientes c
RIGHT join notas n on n.codcliente = c.codigo;
```

O comando acima listará todas as notas fiscais já emitidas, independente do cliente associado existir na tabela *clientes*. Quando o cliente não for encontrado, a coluna nome ficará nula.

***Nota**: Obviamente em um banco de dados bem projetado, fazendo uso de regras de integridade referencial, a situação de uma nota fiscal cadastrada não ter mais seu cliente relacionado na tabela de clientes nunca poderia existir, pois estaria quebrando a integridade referencial.*

FULL [OUTER] JOIN

Como o próprio nome sugere, o FULL OUTER JOIN é uma união do RIGHT e do LEFT JOIN, ou seja, registros de ambos os *streams* serão retornados, independente da condição ter sido satisfeita. Veja o exemplo:

```
select  c.nome, n.nota_fiscal
from clientes c
FULL join notas n on n.codcliente = c.codigo;
```

Nesse caso, todos os clientes seriam listados, bem como todas as notas fiscais cadastradas. Use com cuidado, pois pode gerar *resultsets* gigantescos e, portanto, causar problemas de performance.

CROSSJOIN

A versão 2.0 do Firebird introduziu a sintaxe para definição de *crossjoins*.
Um *crossjoin* retorna o produto cartesiano formado pelos registros das tabelas relacionadas. Tenha cuidado com sua utilização, pois pode retornar *resultsets* enormes (duas tabelas com 1.000 registros cada uma retornaria um *crossjoin* de 1.000.000 de registros!).

Dica

Até o Firebird 1.5, a única maneira de se fazer um *crossjoin* era utilizar a sintaxe de *junção exclusiva* SQL89, onde as tabelas eram listadas na cláusula FROM, por exemplo:

```
select t1.campoA, t2.campoB from tabela1 t1, tabela2 t2
```

O uso dessa sintaxe não é recomendado, pois dificulta o trabalho do otimizador do Firebird.

WHERE

Essa é a cláusula que permite definir condições para filtragem dos dados nas tabelas envolvidas, a fim de obter o resultado esperado. As condições de filtragem são montadas com a ajuda de *operadores* e *predicados*. É importante sabermos quais os operadores aceitos pelo Firebird, e qual a precedência de cada um deles.

GROUP BY

A cláusula GROUP BY define por quais colunas o resultado do *select* será agrupado. O uso do *group by* torna-se obrigatório quando o *select* faz uso de funções de agregação (ex: *sum, max, min*, etc.) em alguma de suas colunas. A seguir temos alguns exemplos de agrupamento:

```
select c.cidade, count(*)
from clientes c
group by c.cidade
order by 2 desc

CIDADE                COUNT
OVERGAARD             16
PISCATAWAY            15
GALENA                14
LA GRANGE             14
...
```

O resultado do comando anterior é uma lista de cidades e o número de clientes cadastrados para cada uma dessas cidades, ordenada de forma decrescente pelo número de clientes. Poderíamos referenciar o agrupamento pela posição da(s) coluna(s) no *select*, ou seja, `group by 1`, onde 1 indica a primeira coluna do *select*, que no caso é a coluna *cidade*. Caso o agrupamento envolva mais de um campo, basta listá-los no *group by* separados por vírgula. A **tabela 2** mostra as funções de agregação disponíveis no Firebird, bem como seu significado (*ver capítulo sobre Funções Internas*).

Função	Significado
MAX	Retorna o maior valor
MIN	Retorna o menor valor
COUNT	Retorna o número de registros encontrados
AVG	Retorna a média dos valores

SUM	Retorna a soma dos valores

Tabela 2. Funções internas de agregação

Dica

Desde o Firebird 2.0, é permitido usar os **apelidos** definidos para os campos, nas cláusulas *group by* ou *order by*, por exemplo:

```
select c.cidade as cid, count(*) as total
from clientes c
group by cid
order by total desc
```

O *group by* também aceita como critério de agrupamento, expressões/fórmulas arbitrárias.

HAVING

O HAVING funciona de forma semelhante ao WHERE, mas trabalha com condições baseadas em agregação. Considerando o exemplo anterior, suponha que você queira obter uma lista de cidades com menos de cinco clientes cadastrados, a fim de realizar um trabalho de ampliação das vendas naquela região. A maneira mais fácil de ser obter o resultado é usando a cláusula *having*:

```
select c.cidade, count(*)
from clientes c
group by c.cidade
having count(*) < 5
order by 2 desc

CIDADE      COUNT
L'ANSE      4
BASIN       4
BAY CITY    4
BALLINGER   4
...
```

UNION [ALL | DISTINCT]

A cláusula UNION permite juntar vários *selects* em um único *resultset*. Todos os *selects* envolvidos devem retornar a mesma quantidade de colunas, na mesma ordem, com tipos e tamanhos equivalentes. Os *selects* podem referenciar a mesma tabela, ou tabelas diferentes. Quando houver a necessidade de compatibilizar o tipo de dado de algum campo referenciado, podemos usar a função CAST ou mesmo UDFs para converter o dado no tipo desejado.

Dica

O Firebird >= 2.0 tenta compatibilizar automaticamente os tipos de dados retornados pelos campos equivalentes nos diversos *selects* que estão sendo unidos, dispensando-se na maioria das vezes o uso do CAST.

Supondo a existência de uma tabela de *clientes* e outra de *fornecedores*, com estruturas semelhantes, e que desejássemos retornar uma lista completa formada por todos os clientes e fornecedores cadastrados. Para isso, podemos usar o comando abaixo:

```
select c.nome from clientes c
UNION
select f.nome from fornec f
```

Note que o comportamento padrão do UNION é de **excluir os registros duplicados**, portanto, no exemplo acima, caso tenhamos um cliente e um fornecedor com o mesmo nome, apenas uma ocorrência estaria representada no *resultset* final. Isso envolve um processamento extra do servidor. Para termos um resultado completo, incluindo registros repetidos, devemos substituir o UNION por **UNION ALL**.

Nota

Para se adequar ao padrão SQL 99, o Firebird 2.0 permite o uso da cláusula UNION DISTINCT como sinônimo da cláusula UNION.

Nas versões 1.x do Firebird, era responsabilidade do desenvolvedor, compatibilizar os tipos de dados das colunas retornadas pelos *selects* envolvidos em uma UNION. Geralmente isso implicava no uso de diversos CASTs para converter os tipos de dados.

O Firebird ficou mais esperto a partir da versão 2.0, e procura fazer essa compatibilização automaticamente, seguindo as seguintes regras:

Considere que DTS é o conjunto de tipos de dados que será analisado para se determinar qual o tipo de dado final:

1. Todos os tipos de dados em DTS devem ser comparáveis entre si.
2. Caso:
    a. Se algum dos tipos em DTS for *string*, então:
        i. Se algum dos tipos em DTS for uma *string* de tamanho variável, o resultado será um *varchar* com tamanho máximo igual ao maior tamanho das colunas/dados envolvidos; caso contrário o resultado será um *char* de tamanho fixo definido pela maior coluna ou dado envolvido.
        ii. O *charset/collation* utilizado será o do primeiro tipo *string* encontrado no DTS.
    b. Se todos os tipos de dados do DTS forem numéricos exatos, então o resultado será um numérico exato com escala e precisão igual ao valor máximo da escala e precisão encontrada no DTS. **Nota**: A checagem de estouros (*overflow*) é feita somente em *runtime*. O desenvolvedor deve tomar os cuidados necessários para que a agregação não resulte em um *overflow*.
    c. Se qualquer um dos tipos de dados do DTS for um numérico aproximado, então todo tipo de dado desse DTS deverá ser numérico, caso contrário um erro será gerado.
    d. Se algum tipo de dado do DTS for relacionado à data/hora, então cada tipo de dado do DTS deverá ser do mesmo tipo (date/time/timestamp).
    e. Se um dos tipos do DTS for *blob*, então todos os tipos deverão ser *blob* e ter o mesmo *subtype*.

Nota

Unions poderão ser utilizadas em *subqueries* que façam parte de pesquisas ALL, ANY e IN, ou mesmo em comandos de inserção (INSERT) que utilizem *selects* como fonte de dados a ser inserida.

PLAN

Quando enviamos um comando SELECT para o servidor Firebird, o otimizador interno analisa as tabelas envolvidas, procurando pela existência de índices que possam ajudar na pesquisa dos dados e relacionamentos, verificando a cardinalidade das tabelas, seletividade dos índices e muitos outros fatores. O resultado do otimizador é um "**plano**" de recuperação dessas informações, de forma a obter o resultado no menor intervalo de tempo possível.

Algumas vezes, devido à complexidade das *queries* ou mesmo a fatores imprevisíveis, o otimizador acaba escolhendo um plano inadequado. Nesses casos, o desenvolvedor pode forçar o Firebird a utilizar um determinado plano na recuperação das informações, através da cláusula PLAN.

Não é recomendável forçar o plano de acesso, pois inúmeros fatores influenciam na decisão do otimizador, portanto, um plano que se mostra eficaz em um determinado momento, pode deixar de ser eficiente após alguns meses de utilização do banco de dados, pois os dados se modificam e geralmente a quantidade de informações de algumas tabelas tende a crescer consideravelmente, enquanto a de outras não. O otimizador levará em consideração as estatísticas/estatísticas mais recentes do BD para chegar ao plano de acesso mais adequado, enquanto um plano forçado agirá sempre da mesma forma.

Esse livro não tem a pretensão de entrar em detalhes no assunto "planos de acesso". Mais informações podem ser obtidas na internet.

Apenas para ilustração, abaixo vemos um select e o respectivo plano de acesso gerado pelo otimizador:

```
select c.nome, n.id
from clientes c
join notas n on n.codcliente = c.codigo
order by c.nome, n.id

PLAN SORT (JOIN (N NATURAL, C INDEX (PK_CLIENTES)))
```

ORDER BY

Com certeza essa é uma das cláusulas mais utilizadas em *selects*! Ela define a forma que o resultado do *select* será ordenado. Note que as colunas referenciadas pelo *order by* não precisam estar presentes no *resultset* do *select*.

Através das cláusulas **DESC**ENDING ou **ASC**ENDING (padrão), podemos definir se a ordenação será feita de forma crescente ou decrescente. O exemplo abaixo ordena os clientes por cidade (de forma decrescente), sub-ordenados pelo nome (de forma crescente).

```
select c.cidade, c.nome, c.telefone
from clientes c
order by c.cidade desc, c.nome asc
```

```
CIDADE       NOME                TELEFONE
ZUNI         AUBREY STROLLO      1(561)745-3159
ZUNI         EDGAR HUGHES        1(799)450-8586
ZUNI         JUDITH LYON         1(917)106-6523
ZUNI         KAITLIN HOOD        1(200)324-3114
ZUNI         NIGEL HEDRICK       1(186)517-1160
ZION         ROGER DIAMANDIS     1(557)500-5470
ZION         WINDER CASBERGUE    1(580)337-1867
ZEPHYRHILLS  ANTHEA HESTER       1(530)220-4911
ZEPHYRHILLS  BRADFORD DAVIS      1(782)787-7746
ZEPHYRHILLS  CHRISTINE SHANOR    1(185)993-6186
...
```

Também podemos indicar que *collation* desejamos utilizar para determinada ordenação, especificando-o como no exemplo abaixo:

```
select c.cidade, c.nome, c.telefone
from clientes c
order by c.cidade collate win_ptbr desc, c.nome collate pxw_intl850 asc
```

Para saber mais sobre os *collates* e seu efeito sobre as ordenações, verifique o capítulo que trata desse assunto.

ROWS

ROWS permite que definamos o intervalo de registros que devem ser retornados no resultset. Ela se comporta de forma semelhante ao FIRST/SKIP. Consulte o capítulo dedicado ao FIRST, SKIP e ROWS para saber mais.

FOR UPDATE

Quando requisitamos informações ao Firebird através de um *select*, essas informações nos são retornadas em lotes (pacotes) através da rede. Com isso, ganhamos performance, pois menos pacotes são necessários para enviar o *resultset* completo. A quantidade de "registros" trafegados em cada pacote de rede não é fixa e, portanto, nunca sabemos exatamente quantos registros serão retornados durante o próximo *fetch* de informações.

Há situações (raras) onde pode ser necessário o controle minucioso dos registros trafegados. Especificando-se o FOR UPDATE no *select* fará com que o Firebird envie apenas um registro a cada *fetch* efetuado. Sendo assim, se cinqüenta *fetches* foram executados, é sinal de que exatamente 50 registros foram enviados para o cliente. Use essa opção com cuidado, pois com grandes *resultsets*, o *fetch* total dos registros para o cliente pode ser tornar muito demorado. Recomenda-se utilizá-la somente quando o número de registros retornados for baixo.

WITH LOCK

O Firebird trabalha com um esquema otimista de controle de concorrência. Isso significa que ele presume que a possibilidade de várias transações realizarem uma alteração em um mesmo registro ao mesmo tempo é remota. A cláusula ***with lock*** permite que nós façamos com que o Firebird trave os registros lidos pelo *select*, fazendo com que somente a transação que executou o *select* possa alterar o conteúdo desses registros.

Esse recurso é também chamado de **travamento pessimista**, e deve ser usado com extrema cautela, pois pode gerar muitos *deadlocks*, dependendo do número de acessos simultâneos.

Note que o *with lock* trava apenas os registros que o *select* já recuperou. A maioria das aplicações de bancos de dados recupera os registros em blocos, para não saturar a rede. Sendo assim, só teremos certeza de que todos os registros do *select* foram travados quando o *fetch* do último registro for executado. Enquanto isso não acontecer, teremos uma situação em que parte dos registros está travada e parte ainda não está.

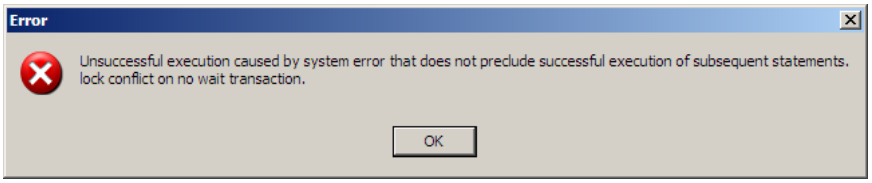
Se houver necessidade de travar todos os registros de uma só vez, um *fetch all* é necessário. Mais uma vez, tome muito cuidado com o uso dessa opção.

Para se ter um maior controle sobre o travamento dos registros, é comum utilizar o *with lock* em conjunto com o *for update*. No entanto, isso só é recomendável quando a quantidade de registros retornados for pequena.

A seguir está um exemplo de *select* com *with lock* que travará os 10 primeiros registros da tabela clientes:

```
select  first 10 *
from clientes
order by codigo
with lock;
```

Caso uma segunda transação execute o mesmo comando, enquanto a primeira transação estiver ativa, obteremos uma mensagem a da figura abaixo:



O erro está nos avisando de que os registros que tentamos acessar estão bloqueados por outra transação.

Conclusão

O *select* é um dos comandos mais poderosos da linguagem SQL. Saber explorar todos os seus recursos é fator importantíssimo para o correto desenvolvimento dos sistemas, obtenção correta das informações com a melhor performance possível.

Esse artigo foi extraído do livro “Firebird 2”, de minha autoria, e revisado em 07-maio-2021. Autor: Carlos Henrique Cantu