

Relacionamentos, otimizando N+1 e ferramentas ORM

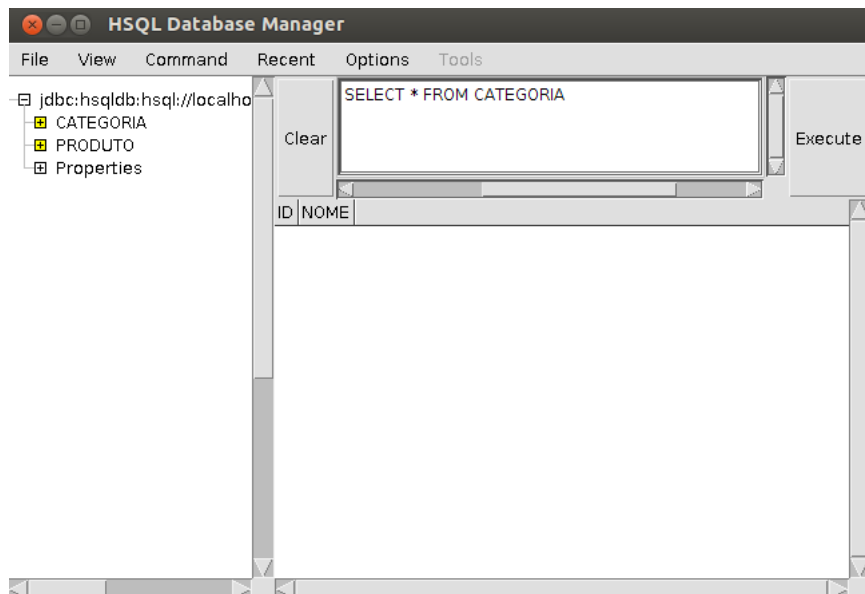
Transcrição

Vamos dar continuidade ao projeto utilizado no capítulo anterior. Caso não tenha criado ele, [importe este projeto no Eclipse](https://s3.amazonaws.com/caelum-online-public/JDBC/files/loja-virtual-cap6.zip) (<https://s3.amazonaws.com/caelum-online-public/JDBC/files/loja-virtual-cap6.zip>).

Vamos criar agora uma nova tabela para trabalharmos com relacionamentos entre ela e a tabela Produto. O nome dela será Categoria, contendo um id gerado automaticamente (chave primária) e o nome (string tradicional):

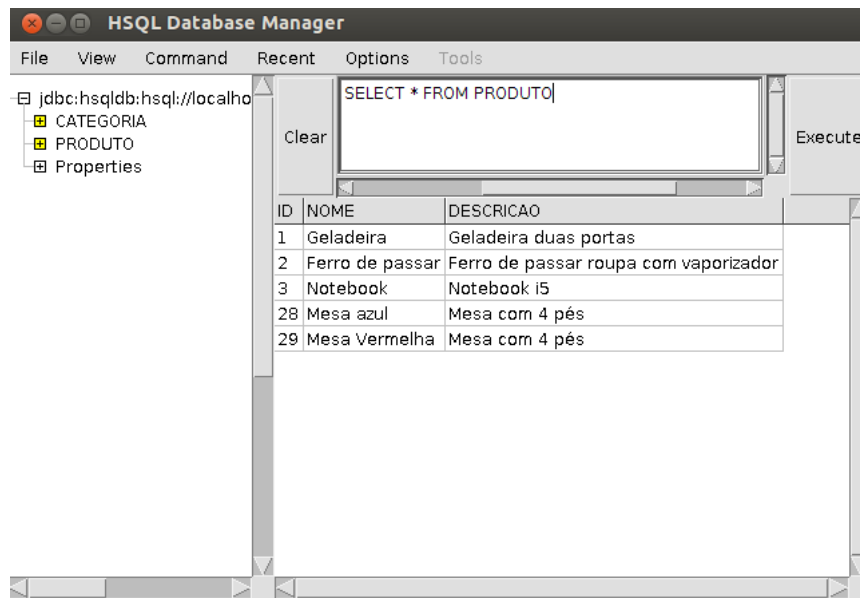
```
create table Categoria (id integer generated by default as identity primary key, nome varchar(255))
```

Ao atualizarmos a view do database manager conseguimos visualizar tanto a tabela Categoria quanto a Produto:



Agora vamos atualizar nossos produtos existentes para colocá-los em determinadas categorias. Para isso faremos primeiro um select para verificar quais produtos existem no banco:

```
select * from Produto
```

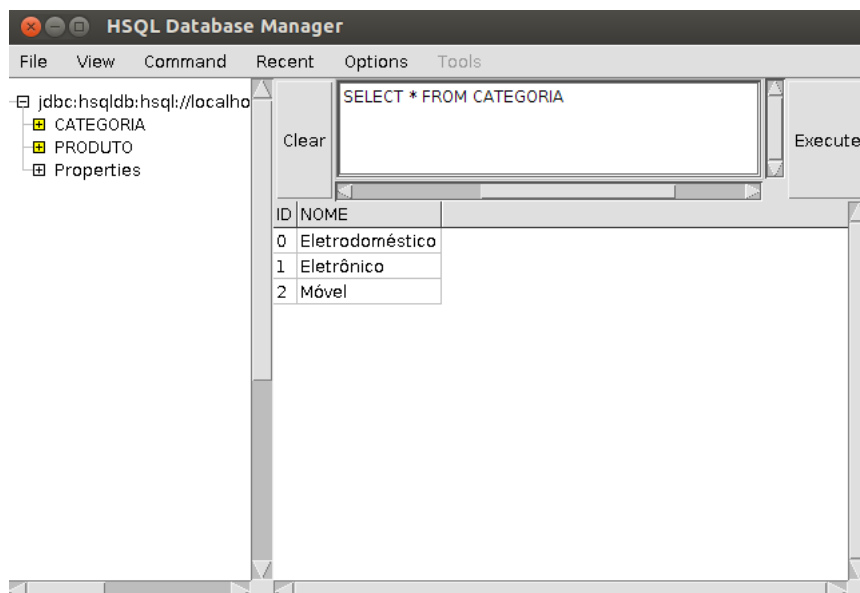


Temos uma geladeira, um ferro, um notebook e duas mesas. Criaremos três categorias para agrupar estes produtos: eletrodomésticos para a geladeira e o ferro; eletrônicos para o notebook e móveis para as mesas. Executaremos cada um dos sql a seguir sem os ponto e vírgula do fim da linha e um por vez:

```
insert into Categoria (id,nome) values (0, 'Eletrodoméstico');  
insert into Categoria (id,nome) values (1, 'Eletrônico');  
insert into Categoria (id,nome) values (2, 'Móvel');
```

Agora podemos conferir nossas categorias:

```
select * from Categoria;
```

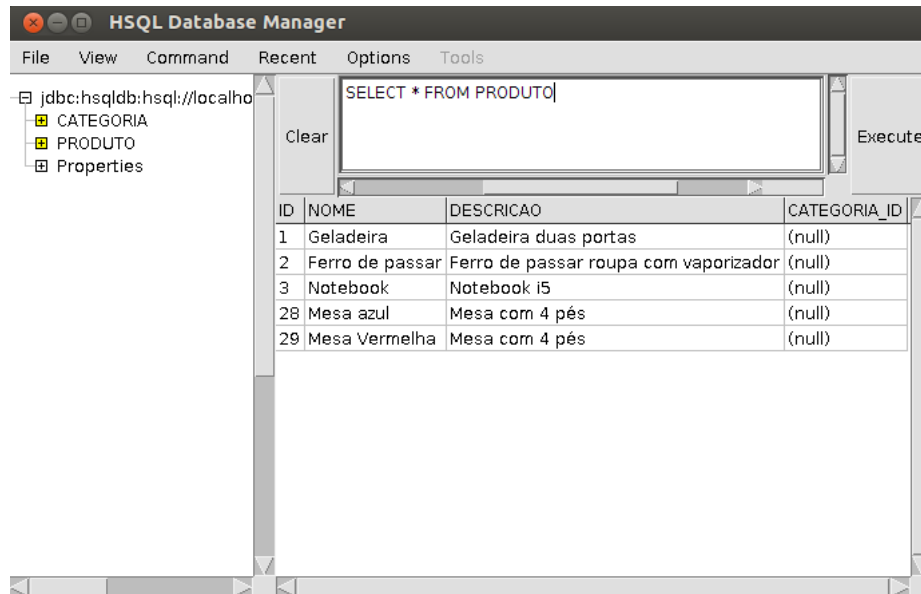


Precisamos encontrar alguma maneira de dizer que o produto 1 está na categoria 0, o produto 2 na categoria 0, o produto 3 na categoria 1 etc. Isto é, precisamos dizer o id da categoria para cada um dos nossos produtos, portanto adicionaremos o campo *categoria_id* dentro da tabela *Produto*:

```
alter table Produto add column categoria_id integer
```

E conferimos que a coluna foi criada com sucesso. Note que as categorias estão como null por padrão: no nosso caso não obrigamos nenhum produto a ter uma categoria.

```
select * from Produto
```



The screenshot shows the HSQL Database Manager interface. On the left, a tree view shows the database structure with 'jdbc:hsqldb:hsqldb://localhost' expanded, showing 'CATEGORIA' and 'PRODUTO' tables. The main window displays the SQL command 'SELECT * FROM PRODUTO' in the command editor. Below the editor, a table shows the results of the query. The table has four columns: ID, NOME, DESCRICAO, and CATEGORIA_ID. The data rows are as follows:

ID	NOME	DESCRICAO	CATEGORIA_ID
1	Geladeira	Geladeira duas portas	(null)
2	Ferro de passar	Ferro de passar roupa com vaporizador	(null)
3	Notebook	Notebook i5	(null)
28	Mesa azul	Mesa com 4 pés	(null)
29	Mesa Vermelha	Mesa com 4 pés	(null)

Marcaremos agora o produto 1 e 2 com a categoria de Eletrodoméstico (categoria 0):

```
update Produto set categoria_id=0 where id in (1,2)
```

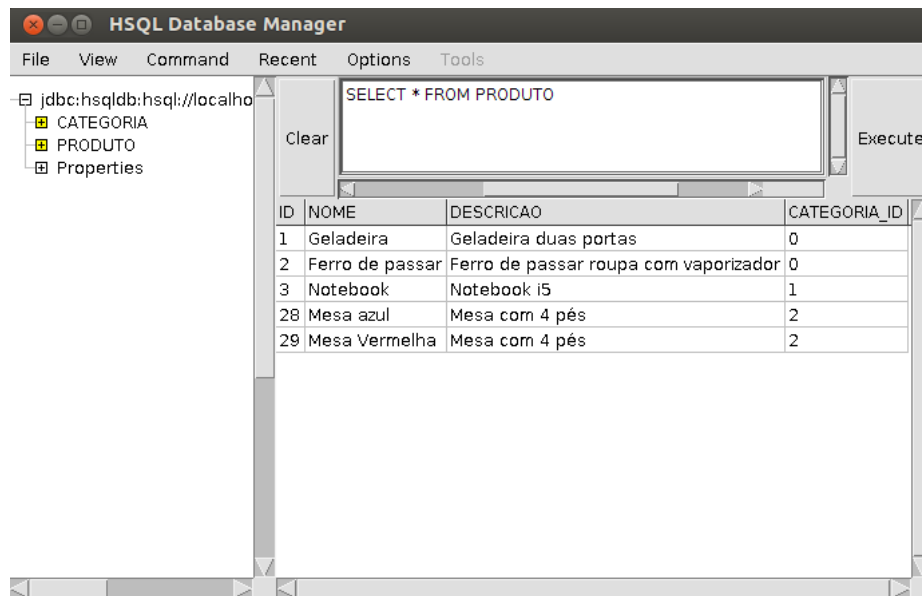
Marcamos o produto 3 com a categoria Eletrônico:

```
update Produto set categoria_id=1 where id in (3)
```

E marcamos os outros produtos com a categoria Móvel:

```
update Produto set categoria_id=2 where id>3
```

Conferimos novamente nossos produtos usando o select:



Agora atacaremos o código do modelo, nosso código Java. Analogamente ao que fizemos com a tabela Produto, criaremos agora uma classe chamada Categoria, com id e nome. Adicionamos o construtor com os dois argumentos:

```
package br.com.caelum.jdbc.modelo;

import java.util.ArrayList;
import java.util.List;

public class Categoria {

    private final Integer id;
    private final String nome;

    public Categoria(int id, String nome) {
        this.id = id;
        this.nome = nome;
    }
}
```

Criamos agora nosso teste TestaCategorias com o método main e a abertura de conexão do Java 7 que fecha automaticamente a mesma:

```
package br.com.caelum.jdbc;

import java.sql.Connection;
import java.sql.SQLException;

public class TestaCategorias {

    public static void main(String[] args) throws SQLException {
        try(Connection con = new ConnectionPool().getConnection()) {
        }
    }
}
```

Vamos agora instanciar o CategoriasDAO, que ainda não criamos, passando nossa conexão como argumento:

```
new CategoriasDAO(con);
```

E queremos invocar um método que traga uma lista de categorias:

```
List<Categoria> categorias = new CategoriasDAO(con).chamaAlgumMetodo();
```

Daremos um nome ao método que indica que ele traz a lista de categorias: lista:

```
List<Categoria> categorias = new CategoriasDAO(con).lista();
```

Agora que sabemos o que precisamos no nosso DAO, vamos criá-lo:

```
public class CategoriasDAO {  
  
    private final Connection con;  
  
    public CategoriasDAO(Connection con) {  
        this.con = con;  
    }  
  
    public List<Categoria> lista() throws SQLException {  
        return null;  
    }  
}
```

E precisamos implementar o método lista. Análogo ao nosso método lista da classe *ProdutosDAO* criaremos primeiro uma lista de categorias e retornaremos ela:

```
List<Categoria> categorias = new ArrayList<>();  
return categorias;
```

Agora executamos nossa query que seleciona todas as categorias:

```
List<Categoria> categorias = new ArrayList<>();  
  
String sql = "select * from Categoria";  
try(PreparedStatement stmt = con.prepareStatement(sql)) {  
    stmt.execute();  
}  
return categorias;
```

Para cada um dos resultados que encontrarmos no nosso *ResultSet* devemos instanciar uma nova *Categoria* e colocá-la na lista:

```
try(ResultSet rs = stmt.getResultSet()) {  
    while(rs.next()) {  
        int id = rs.getInt("id");
```

```

        String nome = rs.getString("nome");
        Categoria categoria = new Categoria(id, nome);
        categorias.add(categoria);
    }
}

```

Pronto. Criamos uma lista de categorias, executamos a query, para cada resultado adicionamos a categoria à lista e finalmente retornamos ela completa.

Voltamos à nossa classe TestaCategorias e adicionamos um laço para iterar por cada uma das categorias e imprimir seu nome:

```

for(Categoria categoria : categorias) {
    System.out.println(categoria.getNome());
}

```

Temos que adicionar o getter do nome da categoria, claro. Rodamos o programa e temos na saída a lista das categorias:



Mas nosso objetivo é trabalhar com o relacionamento entre uma categoria e diversos produtos. Vamos mostrar então os produtos que cada categoria tem? Dentro de nosso loop gostaríamos de fazer um outro loop, por todos os produtos daquela categoria, imprimindo o nome da categoria e o nome do produto:

```

for(Categoria categoria : categorias) {
    System.out.println(categoria.getNome());

    for(Produto produto : todosOsProdutosDestaCategoria) {
        System.out.println(categoria.getNome() + " - " + produto.getNome());
    }
}

```

Mas nós não temos *todosOsProdutosDestaCategoria*. Como podemos acessá-los? Vamos efetuar uma query no *ProdutosDAO*? Basta adicionarmos mais um método que recebe a categoria como parâmetro e traz todos os produtos daquela categoria.

Primeiro vamos analisar o nosso método lista:

```

public List<Produto> lista() throws SQLException {
    List<Produto> produtos = new ArrayList<>();
    String sql = "select * from Produto";

    try (PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.execute();
    }
}

```

```

        stmt.executeUpdate();
    }

    try (ResultSet rs = stmt.getResultSet()) {
        while (rs.next()) {
            int id = rs.getInt("id");
            String nome = rs.getString("nome");
            String descricao = rs.getString("descricao");
            Produto produto = new Produto(nome, descricao);
            produto.setId(id);
            produtos.add(produto);
        }
    }

    return produtos;
}

```

Repare que para facilitar nossa leitura, após executar o statement podemos extrair o código que busca o result set e transforma ele em produtos, adicionando na lista. Basta selecionarmos este pedaço de código e extrair um método (Refactor, Extract Method): `transformaResultadoEmProdutos`.

```

public List<Produto> lista() throws SQLException {
    List<Produto> produtos = new ArrayList<>();
    String sql = "select * from Produto";

    try (PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.execute();
        transformaResultadoEmProdutos(stmt, produtos);
    }
    return produtos;
}

private void transformaResultadoEmProdutos(PreparedStatement stmt, List<Produto> produtos)
    throws SQLException {
    try (ResultSet rs = stmt.getResultSet()) {
        while (rs.next()) {
            int id = rs.getInt("id");
            String nome = rs.getString("nome");
            String descricao = rs.getString("descricao");
            Produto produto = new Produto(nome, descricao);
            produto.setId(id);
            produtos.add(produto);
        }
    }
}

```

Tome cuidado com um possível bug do Eclipse que não passa o parâmetro Statement automaticamente. Neste caso, adicione o parâmetro você mesmo. Note como o código final fica mais simples de entender: um método é responsável por executar a query, enquanto o outro método é responsável por extrair seus resultados e adicionar os produtos equivalentes à uma lista.

Agora criaremos nosso outro método de listagem, algo como busca e recebe a categoria:

```

public List<Produto> busca(Categoria categoria) throws SQLException {
    List<Produto> produtos = new ArrayList<>();
    return produtos;
}

```

```
return produtos;  
}
```

Nossa query difere muito pouco da lista completa, bastando passar o id de nossa categoria:

```
String sql = "select * from Produto where categoria_id = ?";
```

Portanto basta setarmos esse primeiro parâmetro e executarmos o statement, parseando o ResultSet. Como já extraímos o método *transformaResultadoEmProdutos*, podemos utilizá-lo aqui novamente:

```
String sql = "select * from Produto where categoria_id = ?";  
  
try (PreparedStatement stmt = con.prepareStatement(sql)) {  
    stmt.setInt(1, categoria.getId());  
    stmt.execute();  
  
    transformaResultadoEmProdutos(stmt, produtos);  
}
```

Adicionamos também o getter do id da categoria. Pronto. Voltamos à nossa classe de teste e adicionamos a invocação ao método de busca do ProdutosDAO:

```
try(Connection con = new ConnectionPool().getConnection()) {  
    List<Categoria> categorias = new CategoriasDAO(con).listaComProdutos();  
    for(Categoria categoria : categorias) {  
        System.out.println(categoria.getNome());  
  
        for(Produto produto : new ProdutosDAO().busca(categoria)) {  
            System.out.println(categoria.getNome() + " - " + produto.getNome());  
        }  
    }  
}
```

Agora basta executarmos nosso programa para ter o resultado que desejávamos:



```
<terminated> TestaCategorias [Java Application] /home/lais/Downloads/jdk1.7.0_21/bin/java (May 28, 2013 6:20:47 PM)  
Eletrônico  
Eletrônico - Notebook  
Móvel  
Móvel - Mesa azul  
Móvel - Mesa Vermelha  
Eletrodoméstico  
Eletrodoméstico - Geladeira  
Eletrodoméstico - Ferro de passar
```

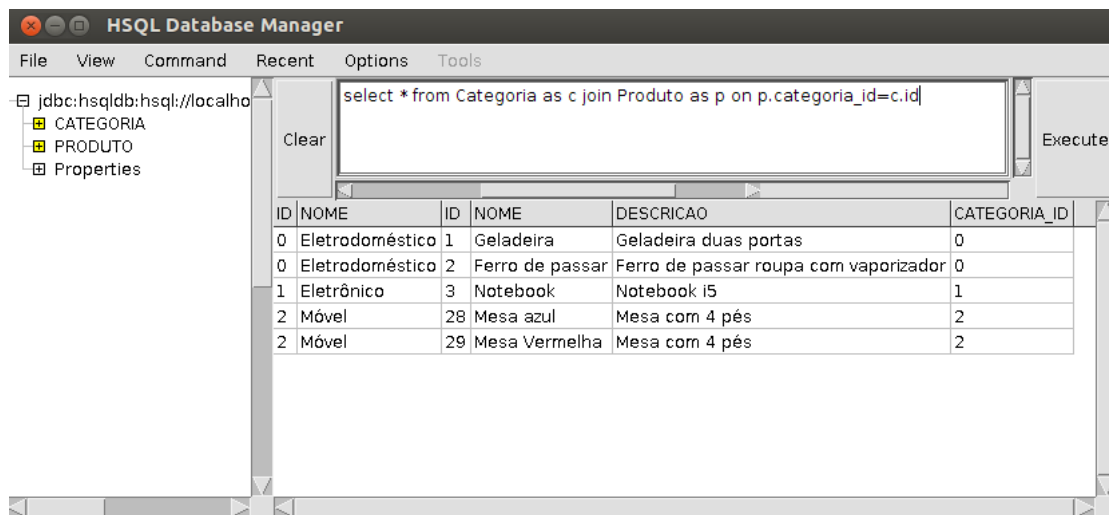
Mas quantas queries executamos? Uma? Duas? Quatro. Note que executamos uma primeira query para trazer todas as categorias. Depois disso executamos uma query para cada categoria, isto é: mais 3 queries. Se nosso sistema possui 1000 categorias, executaríamos 1001 queries. Estamos executando N+1 queries, onde N é o número de elementos retornados pela

primeira pesquisa. Isto é muito ruim, uma vez que cada pesquisa tem que ir e voltar de um sistema remoto, serializando dados etc.

Como podemos evitar executar uma query para cada categoria? Poderíamos trazer todos os produtos de uma única vez, através de um join:

```
select * from Categoria as c join Produto as p on p.categoria_id=c.id
```

Pronto. Com um único join trouxemos todos os dados da categoria e dos produtos que desejamos. Executamos essa query no banco de dados e temos:



The screenshot shows the HSQL Database Manager interface. On the left, a tree view shows the database structure with 'jdb:hsqldb:hsqldb://localhost' expanded, showing 'CATEGORIA' and 'PRODUTO'. The main window displays the SQL query: `select * from Categoria as c join Produto as p on p.categoria_id=c.id`. Below the query, the results are shown in a table with 6 columns: ID, NOME, ID, NOME, DESCRICAO, and CATEGORIA_ID. The data is as follows:

ID	NOME	ID	NOME	DESCRICAO	CATEGORIA_ID
0	Eletrodoméstico	1	Geladeira	Geladeira duas portas	0
0	Eletrodoméstico	2	Ferro de passar	Ferro de passar roupa com vaporizador	0
1	Eletrônico	3	Notebook	Notebook i5	1
2	Móvel	28	Mesa azul	Mesa com 4 pés	2
2	Móvel	29	Mesa Vermelha	Mesa com 4 pés	2

O join traz os valores da categoria uma vez para cada produto que ela possui. Isto é bom para os produtos, mas não tão bom para as categorias. Além disso temos o problema dos nomes dos campos: existe um conflito entre os nomes das duas tabelas. Resolvemos o problema dos nomes com aliases:

```
select c.id as c_id, c.nome as c_nome, p.id as p_id, p.nome as p_nome, p.descricao as p_descricao fr
```

Pronto. Executando novamente temos que cada campo tem um nome distinto: o prefixo "p" é usado para os campos do produto e o prefixo "c" para os campos da categoria.

Agora precisamos implementar um novo método de listagem que já traga as categorias e os produtos de uma só vez. Chamemos este método de `listaComProdutos` em nosso `CategoriasDAO`:

```
public List<Categoria> listaComProdutos() throws SQLException {
    List<Categoria> categorias = new ArrayList<>();
    String sql = "select c.id as c_id, c.nome as c_nome, p.id as p_id, p.nome as p_nome, p.descr
    try(PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.execute();
        try(ResultSet rs = stmt.getResultSet()) {
            while(rs.next()) {
            }
        }
    }
    return categorias;
}
```

O que devemos fazer para cada linha que o banco traz de volta para nós? Criar uma nova categoria. Perfeito, dentro do laço `while` do `ResultSet` fazemos:

```
int id = rs.getInt("c_id");
String nome = rs.getString("c_nome");
Categoria categoria = new Categoria(id, nome);
categorias.add(categoria);
```

Alteramos nosso método de teste para utilizar o novo método `listaComProdutos`:

```
public static void main(String[] args) throws SQLException {
    try(Connection con = new ConnectionPool().getConnection()) {
        List<Categoria> categorias = new CategoriasDAO(con).listaComProdutos();
        for(Categoria categoria : categorias) {
            System.out.println(categoria.getNome());
        }
    }
}
```

Executamos novamente o programa e temos um resultado estranho, mas compreensível:



Acontece que quando executamos uma query com `join` já havíamos visto que a query trazia 5 linhas: uma para cada produto, junto com os dados da categoria. Se para cada linha retornada instanciarmos uma nova categoria temos um problema pois 5 categorias serão instanciadas.

Precisamos de alguma maneira lembrar qual foi a última categoria instanciada e evitar que algum problema aconteça. Vamos então definir uma variável que representa a última categoria instanciada até agora:

```
List<Categoria> categorias = new ArrayList<>();
Categoria ultima = null;
```

No nosso laço, somente adicionaremos a categoria caso estejamos na primeira linha dos resultados (isto é, `ultima` tem valor nulo) ou caso a categoria tenha trocado (o nome esteja diferente):

```
while(rs.next()) {
    int id = rs.getInt("c_id");
    String nome = rs.getString("c_nome");
    if(ultima==null || !ultima.getNome().equals(nome)) {
        Categoria categoria = new Categoria(id, nome);
        categorias.add(categoria);
        ultima = categoria;
    }
}
```

```
        ultima = categoria;  
    }  
}
```

Pronto. Agora instanciamos uma categoria somente quando entre uma linha e outra o nome da categoria mudar. Rodamos o programa novamente e verificamos o resultado:



Mas ainda precisamos carregar os produtos. Para isso, dentro do nosso laço devemos instanciar um novo produto, lembremos os campos dos produtos:

```
int idDoProduto = rs.getInt("p_id");  
String nomeDoProduto = rs.getString("p_nome");  
String descricaoDoProduto = rs.getString("p_descricao");
```

Instanciamos ele:

```
Produto p = new Produto(nomeDoProduto, descricaoDoProduto);  
p.setId(idDoProduto);
```

E adicionamos ele na categoria atual, a variável ultima:

```
ultima.adiciona(p);
```

Nosso código dentro do laço se resume então a carregar o id e o nome da categoria, verificar se a categoria mudou e em caso positivo trocá-la, criar um produto e adicionar o produto na categoria:

```
while(rs.next()) {  
    int id = rs.getInt("c_id");  
    String nome = rs.getString("c_nome");  
    if(ultima==null || !ultima.getNome().equals(nome)) {  
        Categoria categoria = new Categoria(id, nome);  
        categorias.add(categoria);  
        ultima = categoria;  
    }  
    int idDoProduto = rs.getInt("p_id");  
    String nomeDoProduto = rs.getString("p_nome");  
    String descricaoDoProduto = rs.getString("p_descricao");  
    Produto p = new Produto(nomeDoProduto, descricaoDoProduto);  
    p.setId(idDoProduto);  
    ultima.adiciona(p);  
}
```

Mas ainda não existe o método adiciona em uma *Categoria* que recebe um *Produto*. Vamos na primeira classe para incluir uma lista de produtos:

```
public class Categoria {  
  
    private final Integer id;  
    private final String nome;  
    private final List<Produto> produtos = new ArrayList<>();  
    // resto da classe  
}
```

E precisamos adicionar o método adiciona:

```
public void adiciona(Produto p) {  
    produtos.add(p);  
}
```

Agora sim nosso DAO traz todas as categorias e produtos. Mas precisamos iterar por estes produtos e para isso criarmos o getter dos produtos:

```
public List<Produto> getProdutos() {  
    return produtos;  
}
```

E alteramos nosso teste para utilizá-lo:

```
List<Categoria> categorias = new CategoriasDAO(con).listaComProdutos();  
for(Categoria categoria : categorias) {  
    System.out.println(categoria.getNome());  
  
    for(Produto produto : categoria.getProdutos()) {  
        System.out.println(categoria.getNome() + " - " + produto.getNome());  
    }  
}
```

Pronto, rodamos nossa aplicação e temos que com uma única query e um join trouxemos todas as categorias com todos os produtos, evitando o problema do N+1:



```
<terminated> TestaCategorias [Java Application] /home/lais/Downloads/jdk1.7.0_21/bin/java (May 28, 2013 6:52:07 PM)  
Eletrônico  
Eletrônico - Geladeira  
Eletrônico - Ferro de passar  
Eletrônico  
Eletrônico - Notebook  
Móvel  
Móvel - Mesa azul  
Móvel - Mesa Vermelha
```

O JDBC acaba nos dando controle total sobre o que desejamos fazer com nossas conexões e quando necessário podemos fazer a query que carrega somente as categorias ou a query que busca tanto elas quanto os produtos. Para evitar diversas dessas configurações padrões, e minimizar a repetição de código entre projetos e entre classes, surgiram diversas bibliotecas de mapeamento entre nossos modelos orientado a objetos e o mundo relacional. São as ferramentas de mapeamento objeto relacional (Object Relational Mapping: ORM). Entre elas destacam-se no mundo Java a implementação Hibernate e a especificação JPA.

Mesmo assim, [o problema do N+1 é muito famoso \(http://blog.caelum.com.br/os-7-habitos-dos-desenvolvedores-hibernate-e-jpa-altamente-eficazes/\)](http://blog.caelum.com.br/os-7-habitos-dos-desenvolvedores-hibernate-e-jpa-altamente-eficazes/) e até mesmo o mal uso de tais bibliotecas pode causar efeitos negativos em uma aplicação.

Faça a escolha da biblioteca que deseja utilizar (JDBC, Hibernate, JPA etc) de acordo com as necessidades de seu projeto e otimize as queries também de acordo com aquilo que você e sua equipe precisam.