



Desenvolvimento de Sistemas

Operações de carga massiva de dados

Introdução

No conhecimento **Conexão com banco de dados: bibliotecas e operações de conexão**, além de se verificar bibliotecas e operações de conexão, a manipulação de dados também é estudada, utilizando-se recursos de linguagem para manipulação de banco de dados, inserção, atualização, consulta e remoção de registros (JDBC). Também há a prática de listagem dos dados, trabalhando-se operações com seleções simples e estrutura no banco de dados.

Neste conhecimento, você estudará carga massiva de dados, iniciando pelo conceito desta. O que se pode chamar de carga massiva de dados? Qual é a sua importância? Quando podem ocorrer operações de carga massiva de dados? Você encontrará as respostas a essas questões nos próximos parágrafos.

Carga massiva de dados diz respeito a grandes conjuntos de dados que necessitam de processamento e armazenamento, tendo em vista velocidade, variedade e volume com que se consegue fazer isso.

Na atualidade, esse volume é muito grande. Diariamente, bilhões de informações são geradas, requerendo processamento e armazenamento. Para você ter uma ideia da quantidade de dados gerada diariamente, o Google processa por dia aproximadamente 3 bilhões de pesquisas no mundo todo, das quais aproximadamente 15% são inéditas. Além disso, a

busca do Google rastreia cerca de 20 bilhões de *sites* diariamente, armazenando em torno de 100 *petabytes* de informações estruturadas ou não estruturadas. Este material trabalhará com informações estruturadas, ou seja, informações que tenham um padrão que pode ser utilizado na leitura e extração de dados, como, por exemplo, arquivos de texto CSV (*comma-separated values*), TXT (*text*) ou XML(*extensible markup language*).


Uma outra questão a ser abordada a respeito de carga massiva de dados é a utilização de sistemas especialistas por empresas, os quais são CRM (Customer Relationship Management), SCM (Supply Chain Management), ERP (Enterprise Resource Planning) e outros mais. Para garantir o funcionamento desses sistemas, por exemplo, é primordial que os dados sejam compartilhados entre os sistemas por meio de integração.

Integração

A integração pode ser realizada a partir de diferentes métodos, entre eles os seguintes:

- API (application program interface) – Aglomerado de instruções que permite que sistemas se comuniquem entre si por meio de protocolos.
- Web services – Servidor que atende a requisições realizadas por protocolos HTTP (hypertext transfer protocol) e HTTPS (hypertext transfer protocol secure), nos quais são disponibilizados métodos para ler, gravar e alterar informações.
- Planilhas – Arquivos CSV utilizados para fazer a integração de sistemas diferentes, por exemplo, a extração de um arquivo do sistema X e a importação para o sistema Y.
- Protocolo banco a banco – Refere-se à extração de dados do banco de dados e à integração em um banco diferente por meio de comandos SQL (structured query language).



Todos esses métodos podem ser (e são) utilizados para operações de  carga massiva de dados, quando, por exemplo, empresas trocam, implantam ou ainda integram sistemas.

É possível encontrar problemas com cargas massivas em aplicações?

Problemas ao realizar cargas massivas de dados em aplicações não são raros e ocorrem muitas vezes dentro das corporações, impactando diretamente a *performance*, pois, se a escolha da arquitetura não for correta, o tempo de carga pode aumentar significativamente. Vale ressaltar que, quando se fala em *performance*, é preciso considerar vários fatores, por exemplo, latência, tráfego, disponibilidade, escalabilidade e eficiência. Diante disso, é um enorme desafio definir maneiras eficazes de coletar e armazenar as informações, garantindo que todos os fatores mencionados sejam contemplados.

É de extrema importância que a disponibilidade de informações cresça cada vez mais, dado o grande número de dispositivos conectados e equipados com sensores capazes de obter dados, sejam estruturados ou não. Esses conjuntos de dados crescem exponencialmente e necessitam de bancos de dados preparados para recebê-los, de acordo com seu volume e com sua desestruturação, ou ainda quanto a estarem refinados ou não esses dados. Diante disso, a área da computação é desafiada pelas questões ligadas a como manipular e armazenar esses dados e ainda realizar o processamento das consultas deles.

Agora, você praticará operações de carga massiva de dados e, para isso, aprenderá a ler arquivos com a extensão **.csv**, criará uma aplicação que lerá esses arquivos e aplicará os dados no banco de dados.

Atualmente, ao realizar integrações ou ainda carga de dados, é muito comum a utilização de arquivos com a extensão **.csv**, pois esse tipo de arquivo tem uma estrutura fácil de entender e sua manipulação torna-se simples em

plataformas de desenvolvimento. Além disso, operações de carga massiva de dados sempre trazem preocupação em relação a como executá-las com maior desempenho e menor consumo de recursos, o que se torna um grande desafio a ser alcançado.

Em seguida, há um exemplo de código de uma aplicação em Java que faz a leitura de arquivos CSV. Confira nos comentários onde se explica que foi utilizado o **File** para se obter o ficheiro e a leitura foi feita por meio da **Scanner**, e apresentou-se o resultado da leitura onde os dados são separados pelo *pipe* (**|**):

```
import java.io.*;
import java.util.Scanner;

public class leitura {

    public static void main(String[] args) throws FileNotFoundException {

        // para se obter o ficheiro (arquivo com seus dados), utilizou-se aqui o File e a leitura será feita pelo Scanner
        File getCSVFiles = new File("C:/Users/Priscila/Desktop/java/arquivoLeitura/arquivoLeitura/src/arquivo.csv");
        // utilizou-se aqui a classe Scanner de Java
        Scanner sc = new Scanner(getCSVFiles);
        //utilizou-se o ponto e vírgula como delimitador no código, tal qual ocorre no arquivo csv que se tem
        sc.useDelimiter(";");
        // aqui se tem a condição e se acrescentará o pipe ( | ) a cada informação lida deixando ela organizada até que não haja mais dados a serem lidos e apresentados
        while (sc.hasNext())
        {
            System.out.print(sc.next() + " | ");
        }
        sc.close();
    }
}
```

É possível executar o código utilizando um arquivo CSV com os seguintes dados:

```
name;cpf;age;phone;address
caio;123456789;20;1145223643;Avenida Paulista
vinicius;147852369;18;1125253625;Avenida Manoel
sandra;963258741;30;1174587858;Rua Teixeira
regina;125478522;40;1145254536;Rua Fernando
fernando;785245563;42;1145253669;Rua Pereira
augusto;456123014;50;1125363633;Avenida Paulinia
maria;456123789;10;1125455525;Avenida Nossa Senhora
caio;123456789;20;1145223643;Avenida Paulista
vinicius;147852369;18;1125253625;Avenida Manoel
sandra;963258741;30;1174587858;Rua Teixeira
regina;125478522;40;1145254536;Rua Fernando
fernando;785245563;42;1145253669;Rua Pereira
augusto;456123014;50;1125363633;Avenida Paulinia
maria;456123789;10;1125455525;Avenida Nossa Senhora
```

A partir da execução, haverá o seguinte resultado:

```
name | cpf | age | phone | address
caio | 123456789 | 20 | 1145223643 | Avenida Paulista
vinicius | 147852369 | 18 | 1125253625 | Avenida Manoel
sandra | 963258741 | 30 | 1174587858 | Rua Teixeira
regina | 125478522 | 40 | 1145254536 | Rua Fernando
fernando | 785245563 | 42 | 1145253669 | Rua Pereira
augusto | 456123014 | 50 | 1125363633 | Avenida Paulinia
maria | 456123789 | 10 | 1125455525 | Avenida Nossa Senhora
caio | 123456789 | 20 | 1145223643 | Avenida Paulista
vinicius | 147852369 | 18 | 1125253625 | Avenida Manoel
sandra | 963258741 | 30 | 1174587858 | Rua Teixeira
regina | 125478522 | 40 | 1145254536 | Rua Fernando
fernando | 785245563 | 42 | 1145253669 | Rua Pereira
augusto | 456123014 | 50 | 1125363633 | Avenida Paulinia
maria | 456123789 | 10 | 1125455525 | Avenida Nossa Senhora
PS C:\Users\Priscila\Desktop\java\arquivoLeitura\arquivoLeitura>
```

Agora que você já sabe como fazer a leitura do arquivo CSV, é possível verificar como implementar uma aplicação, utilizando JDBC, para fazer não somente a leitura dos dados, mas também a inserção no banco de dados.

Note que, no exemplo apresentado a seguir, o banco de dados já foi criado e será devidamente populado de acordo com o arquivo CSV que já se tem no exemplo. Ele será lido, armazenado e inserido no banco de dados “Pessoa”, cujas colunas são respectivamente: “name”, “cpf”, “age”, “phone”, “address”.

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class LerManipular {

    // Nome do driver JDBC e a url do banco de dados em questão
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost:3306/teste";
    // Credenciais do referentes ao banco de dados
    static final String USER = "root";
    static final String PASS = "";

    public static void main(String[] args) throws ClassNotFoundException, SQLException, FileNotFoundException {

        // Conexão com o banco de dados inicializada como null
        Connection conn = null;
        // Statement é uma interface utilizada para executar instruções SQL no banco de dados inicializada como null
        Statement stmt = null;

        try {
            // local onde está o arquivo que será lido pela aplicação
            FileInputStream arquivo = new FileInputStream("C:/Users/Priscila/Desktop/java/arquivoLeitura/arquivoLeitura/src/arquivo.csv");
            // classe voltada para a manipulação de caracteres
            InputStreamReader input = new InputStreamReader(arquivo);
            // classe que faz a alocação de memória temporária dos elementos que ainda não foram consumidos
            BufferedReader br = new BufferedReader(input);
            // declaração da variável: linha
            String linha;
            // classe voltada para conexão com o banco de dados mysql
            Class.forName("com.mysql.jdbc.Driver");

            //STEP 3: Abertura da conexão utilizando os dados que já temos
            // mensagem apresentada ao usuário dizendo que a conexão com o banco de dados está iniciando
            System.out.println("Connecting to a selected database...");
            //conexão que receberá as credenciais e será iniciada com o banco de dados
```

```

        conn = DriverManager.getConnection(DB_URL, USER, PASS);
        //mensagem apresentada ao usuário informando que a conexão foi realizada com sucesso
        System.out.println("Connected database successfully...");
        // neste trecho do código ele inicia a execução pegando o que tem na string linha, separando a informação a partir do ponto e vírgula e armazenando no array de acordo com o campo em questão até que não encontre nada na variável.
        do {
            linha = br.readLine();

            if (linha != null) {

                String[] campo = linha.split(";");

                String name = campo[0];
                String cpf = campo[1];
                String age = campo[2];
                String phone = campo[3];
                String address = campo[4];
                //inicializa a interface statement que irá executar o SQL desejado
                stmt = conn.createStatement();
                //instrução sql que irá inserir os dados no banco de dados de acordo com o que estiver guardado no array
                String sql = "insert into pessoa(name,cpf,age,phone,address) values(\"" + name + "\",\"" + cpf + "\",\"" + age + "\",\"" + phone + "\",\"" + address + "\")";
                //executa o update
                stmt.executeUpdate(sql);
                //encerra o statement
                stmt.close();
            }
        } while (linha != null);
        //trata a exceção se ela vir a ocorrer
    } catch (IOException e) {
        //apresenta a mensagem ao usuário caso dê algum erro ao ler o arquivo
        System.out.println("Erro ao ler Arquivo");
    }
}
}

```

Após a execução do código, o que se tem no banco de dados correspondente são todos os registros que estavam no arquivo CSV.

Um dos fatores de maior importância para as aplicações é o desempenho do banco de dados, pois é por meio desse desempenho e dessa *performance* que são determinados tanto o sucesso quanto o fracasso da aplicação.

Nos dias atuais, grande parte das aplicações usam JPA (Java Persistence API) para manipulação da base de dados, porém JDBC (Java Database Connectivity) ainda tem grande utilização em *softwares* comerciais, sendo praticada até mesmo pela JPA. Isso se deve ao fato de o *driver* JDBC geralmente apresentar maior *performance* em operações de banco de dados do que JPA, já que opera diretamente com instruções SQL (diferente do JPA, que depende de outras classes intermediárias nessas operações).

Os bancos de dados já têm seus próprios *drivers* JDBC, e ainda assim outros *drivers* alternativos estão à disposição para a maior parte das bases de dados mais populares, como uma forma de ofertar melhor *performance*.

Quando se fala em *performance*, também é necessário abordar as conexões com bases de dados. A conexão precisa de recursos da base de dados, por exemplo, alocação de memória adicional para cada *prepared statement* utilizado pelo *driver* JDBC. Dessa forma, pode ser que o desempenho seja afetado se o servidor da aplicação tiver muitas conexões abertas.

Um recurso que é muito utilizado para *pools* de conexão é ter uma conexão para cada *thread* na aplicação. Em servidores de aplicação, é preciso aplicar o mesmo tamanho para o *pool* de *thread* e para o *pool* de conexão. Já para aplicações *stand-alone*, o tamanho do *pool* de conexões tem como base o número de *threads* que a aplicação cria. Para casos normais, isso trará melhor desempenho e, dessa forma, nenhuma *thread* no programa precisará esperar uma conexão com a base de dados estar disponível. Além disso, geralmente haverá recursos suficientes no banco de dados para tratar da carga imposta pelo aplicativo.

Além das questões relacionadas ao número de conexões, também deve haver preocupação a respeito das consultas que são realizadas em tabelas com milhares de registros. Quando consultas trazem um número muito alto de linhas, uma das opções é a exibição dos resultados com rolamento vertical na tela. Porém, quando se trata de milhares de linhas, isso acarreta consumo de memória da aplicação, e a opção nesse caso é distribuir o resultado da consulta de forma que caiba na tela, sendo cada uma das partes chamadas de página. Assim, o resultado será exibido em uma página por vez e o usuário tem a opção de avançar ou retroceder; esse processo é chamado de paginação.

A paginação pode ser realizada quando é feita uma requisição de consulta e tem como finalidade filtrar a quantidade de registros que se deseja buscar. Por exemplo, se você estiver fazendo uma consulta em base de dados, além da utilização de filtros da *query* (consulta), você pode restringir a quantidade de registros que almeja retornar em um intervalo de páginas.

Ainda sobre o conceito de paginação, é necessário explicar que ela pode ser processada na aplicação, no servidor das aplicações ou ainda no servidor SQL Server, utilizando assim menos memória da aplicação e reduzindo o tráfego na rede.

***Multithreads* aplicados às situações de operações com carga massiva de dados**

A plataforma Java tem disponíveis diferentes APIs que podem ser utilizadas para o paralelismo, desde as versões iniciais, e estão em constante evolução, trazendo recursos novos e *frameworks*, que têm alto nível e auxiliam na programação.

Esse processamento paralelo, ou, ainda, que ocorre em concorrência, baseia-se em *hardware* com multiprocessador, ou seja, que tem muitos núcleos de processamento, o que não era muito comum em Java. Hoje em dia, essas arquiteturas estão amplamente difundidas.

O conceito de *multithreading* pode ser entendido como uma evolução de multitarefa em nível de processo, ou seja, os sistemas têm a capacidade de executar várias tarefas ou processos ao mesmo tempo, nos quais compartilham recursos com a CPU (unidade central de processamento).

Mas o que a aplicação *multithread* faz?

A *multithread* permite que o *software* divida as tarefas em pedaços de código independentes e que consigam ser executados em paralelo, denominados *threads*. A partir disso, as tarefas podem ser realizadas paralelamente, caso existam vários núcleos, assim como é apresentado no exemplo da figura 1 a seguir.

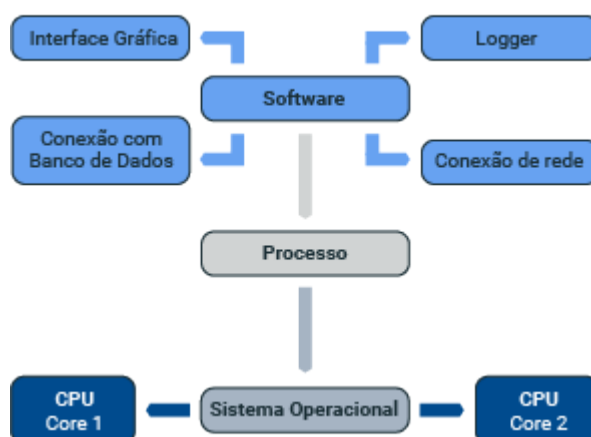


Figura 1 – Processo realizando várias tarefas

Fonte: Senac EAD (2022)

Diante disso, é possível notar que muitos benefícios estão presentes com a utilização desse recurso e o que mais se destaca é o ganho de *performance*. Também é importante destacar a utilização mais eficiente da CPU. Após compreender a importância das *multithreads*, veja o que são as *threads* e como utilizá-las na subdivisão das tarefas da aplicação.

Threads



Em Java, *threads* são a única forma de concorrência suportada. Pode-se dizer que são pedaços ou trechos de código que atuam de forma independente da sequência de execução principal.

Qual é o diferencial das *threads*? Os processos de *software* não dividem o mesmo espaço de memória, mas as *threads* sim.

E qual é a vantagem? Permite que dados e informações sejam compartilhados no *software*.



Veja como isso funciona nos bastidores.

Cada um dos objetos de *thread* tem identificador único, nome, ordem de prioridade, estado e estruturas que são utilizadas pela JVM (Java Virtual Machine), bem como pelo sistema operacional, em que seu contexto é salvo enquanto a *thread* permanece em pausa por quem a escalonou. Ainda no

mesmo contexto, o escalonador pode pausar e conceder espaço e tempo para que outra *thread* seja executada. As *threads* que têm prioridade mais alta, têm mais tempo para processar e são escalonadas com maior frequência do que outras.

Encerramento



Neste conhecimento, você aprendeu como implementar a leitura de arquivos CSV, além de implementar uma aplicação na qual os dados são lidos e inseridos ao banco de dados. Outros tópicos relevantes também foram estudados, por exemplo, como lidar com muitas conexões, soluções aplicáveis a consultas que tragam tabelas com milhares de registros por meio da paginação. Ainda, você conheceu um pouco as *multithreads* e como elas se aplicam a operações de carga massiva de dados.

Bons estudos e até breve!